3D shapes reconstruction based on 2D views

Céline BUGAUD Marie-Hélène PAPADOPOULOS David MANCEL
Jean-Philippe BENTEJAC Julien PERRUCHE Marco TESSARI
Jérôme POUILLER

EPITA - October 2004

Contents

1 Neural Network approach for shape from shading problem **3**
1.1 The objective . . . 3
1.2 Defining the illumination model . . . 3
1.2.1 The diffuse model . . . 4
1.2.2 The specular model . . . 6
1.2.3 Hybrid model . . . 7
1.3 Using neural networks . . . 7
1.3.1 The FNN . . . 7
1.3.2 The RBF . . . 7
1.4 Putting all together . . . 7
1.4.1 The learning algorithm . . . 7

2 Interpolation approach **8**

3 Hybrid solution **9**

A Listing of PMC library **11**
A.1 pmc_macro.hh . . . 11
A.2 pmc_datatypes.hh . . . 11
A.3 pmc_activation.hh . . . 12
A.4 pmc_activation.hxx . . . 13
A.5 pmc_neuron.hh . . . 15
A.6 pmc_neuron.hxx . . . 16
A.7 pmc_neuron.cc . . . 17
A.8 pmc_layer.hh . . . 19
A.9 pmc_layer.hxx . . . 20
A.10 pmc_layer.cc . . . 20
A.11 pmc_network.hh . . . 22
A.12 pmc_network.hxx . . . 23
A.13 pmc_network.cc . . . 24
A.14 test.cc . . . 27

Bibliography **29**

Abstract

The aim of this paper is to study possibilities of 3D objects construction from 2D views using neural networks. The goal isn't to apply directly this paper in an industrial context. We do not have all constraint than in real application. It study specular approach based on one view and interpolation method and it try to find a solution.
Keywords: neural networks, shape from shading.

Introduction

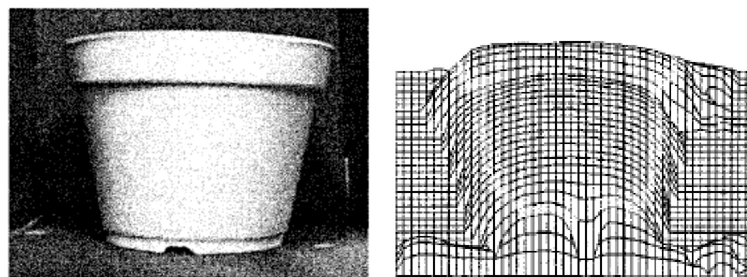
The variation in brightness due to change in surface oriantation across a surface can be used to extract shape of object. Extracting 3D shape information from one or multiple 2D shading images[1] has been one the classic research problems in low-level vision. In this paper, we try to use neural networks to solve problem. We studied two approachs: specular base model and interpolation model.

1 Neural Network approach for shape from shading problem

To reconstruct a 3D shape, you can use the shading information. The method **Shape From Shading**, or SFS, is the one explained below. The shading is **the brightness variation** from one point of an image to one of its neighbors[2].

1.1 The objective

The aim is to recover spatial information from a picture:



1.2 Defining the illumination model

The first thing needed is the illumination model. A lot of model exists like:

- ambient illumination model
- diffuse illumination model

[1] As know as the shape from shading problem

- specular illumination model
- ...

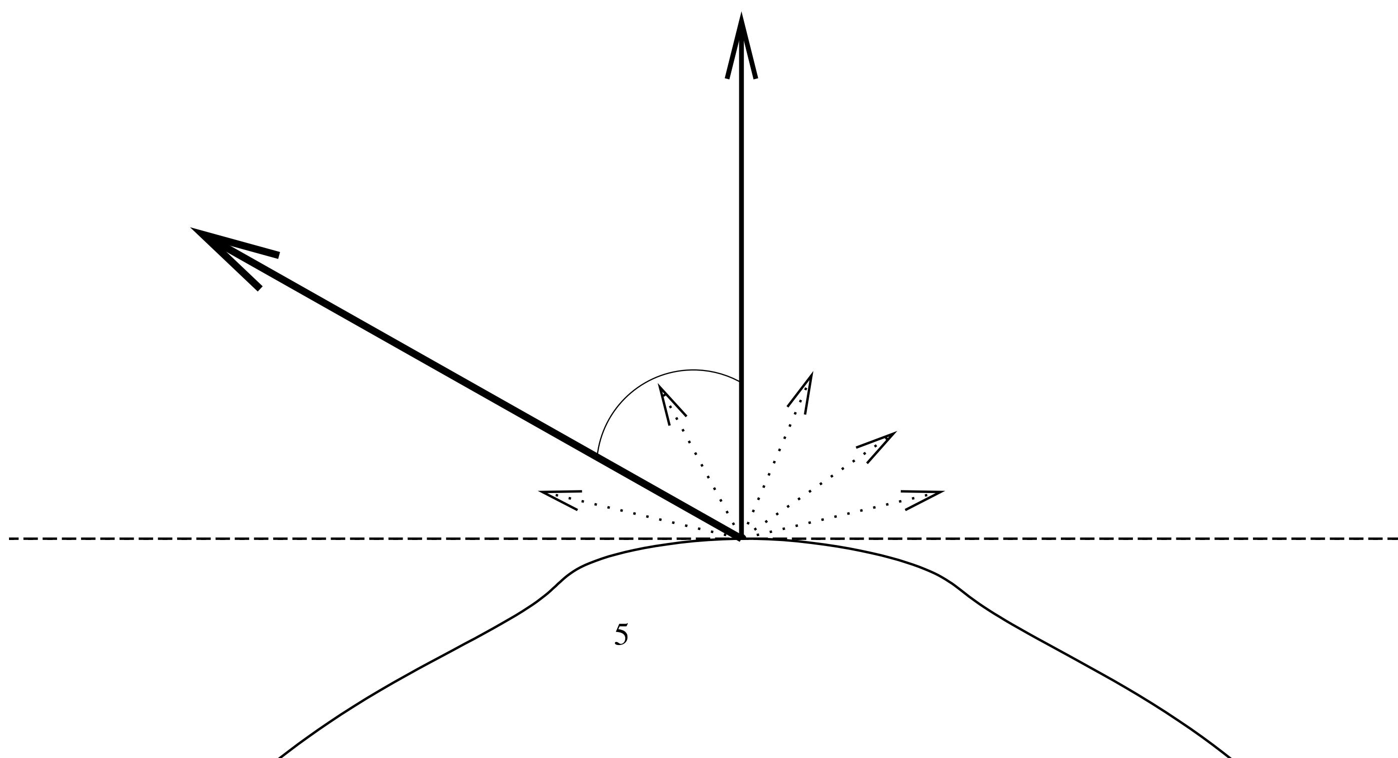
and a choice must be made.

All these models can be combined to have a more realistic yet simple model.

1.2.1 The diffuse model

The most known diffuse model is the Lambert [2] model suppose that when a light heat a surface, the reflected light depends only on the angle between the light direction and the surface normal, and that the light is **equally** reflected in all direction.

[2] Johann Heinrich Lambert (26/08/1728 - 25/09/1777), has published is Photometria in 1760, the Lambertian diffuse lighting model.



The following formula is used in 2D:

$$R_{diffuse} = \eta \cdot \overrightarrow{N} \cdot \overrightarrow{L}$$

where
- η the albedo of the surface (measure of reflectivity)
- $\overrightarrow{N}$ surface normal
- $\overrightarrow{L}$ light source direction

In 3D, the formula becomes:

$$R_{diffuse}(p(x,y),q(x,y)) = \eta(n \cdot s^t), \forall(x,y)$$

with
- η the albedo
- n the normal norm
- $s^t = \begin{pmatrix} cos\tau \\ sin\sigma \cdot sin\tau \\ sin\sigma \cdot cos\sigma \end{pmatrix}$ light source direction
- $p(x,y) = \frac{\partial z}{\partial x}$ the depth variation on the x axis (surface gradient)
- $q(x,y) = \frac{\partial z}{\partial y}$ the depth variation on the y axis

In order to use it this model, the light source direction must be known, and this is not always the case.

1.2.2 The specular model

In addition to diffuse, there is the specular reflection model.

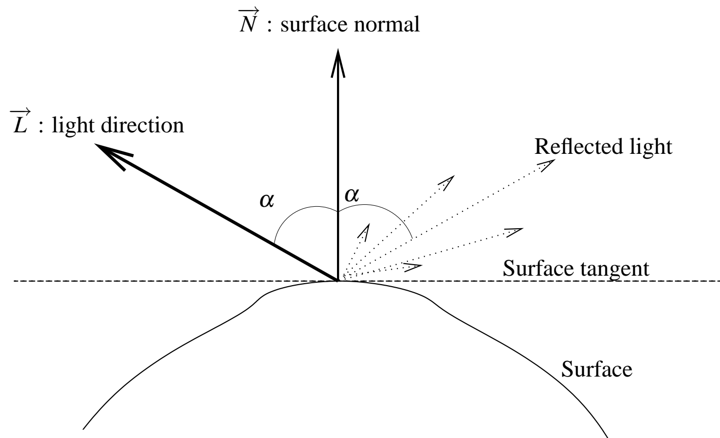


Figure 2: The specular model

This model does not work well for reality picture, but there is a better model derived from that one.

1.2.3 Hybrid model

Since none of the previous model is enough, an hybrid model is defined. It's a simple addition with a coefficient:

$$R_{hybrid} = (1 - w) \cdot R_{diffuse} + w \cdot R_{specular}$$

1.3 Using neural networks

Now that the illumination model is defined, the next step is to define how to use neural networks capabilities in the model.

1.3.1 The FNN

Based on the universal approximation of FNN (Forward Neural Network), a proper function F_{FNN} is defined as the diffuse component of the Lambertian model:

$$\begin{aligned} R_{diffuse}(p,q) &= F_{FNN}(a) \\ &= \varphi_a \left(v_0 + \sum_{k=1}^{N} v_k \cdot \varphi_a(w_k \cdot a_{i,j}) \right) \end{aligned}$$

where

w_k	weights vector connected between the input and the hidden layer
v_k and v_0	weights and the bias connected between hidden layer and the output layer
N	number of hidden unit

1.3.2 The RBF

RBFs are well designed for nonlinear separable problem and use a bell-shape Gaussian activation function.

1.4 Putting all together

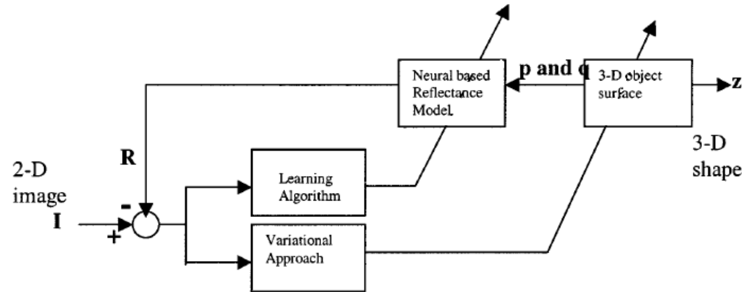


Figure 3: The hybrid neural network model

1.4.1 The learning algorithm

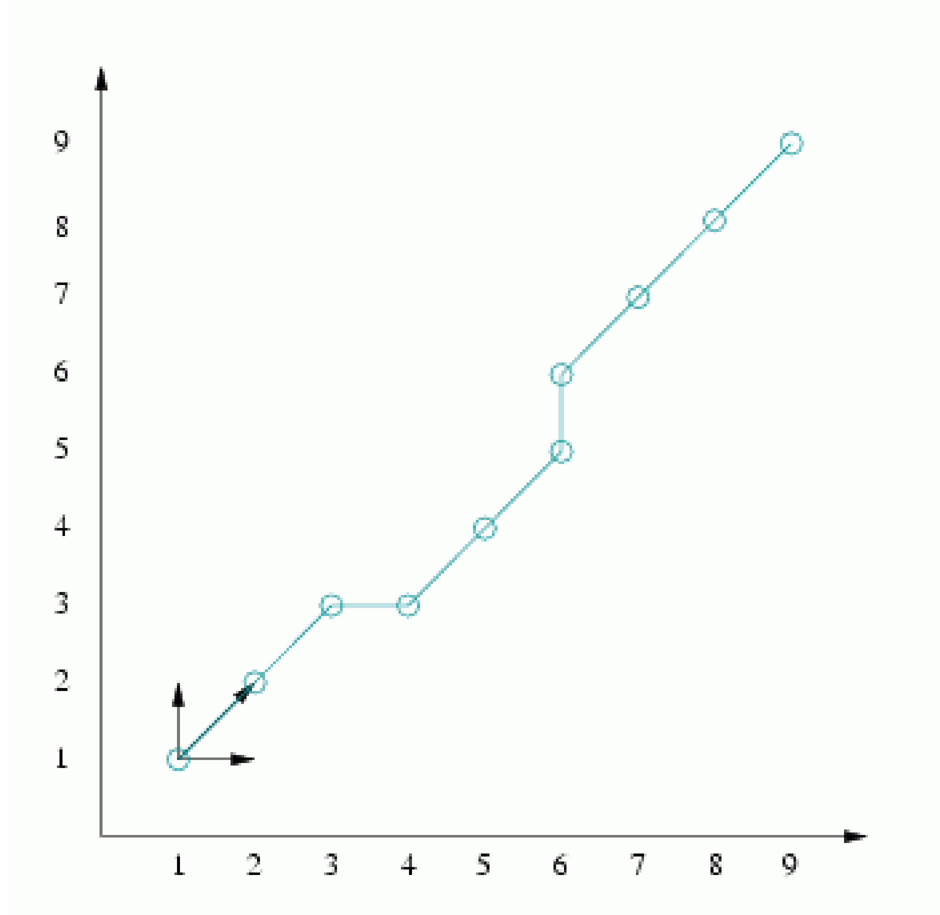
1. n ¡- 0

2. The surface gradient $p_{i,j}$ and $q_{i,j}$, and the depth $z_{i,j}$ receive 0.
3. The weights of the NN are randomly initialized
4. while the stop criterion is not meet do:
   (a) n ¡- $n+1$
   (b) for each (i,j) do
      i. Calculate $\frac{\partial R_{NN}(a_{i,j})}{\partial p}$ and $\frac{\partial R_{NN}(a_{i,j})}{\partial q}$
      ii. Estimate $p_{i,j}(n)$ and $q_{i,j}(n)$
      iii. Estimate $z_{i,j}(n)$
      iv. Compute the free parameters
      v. Find the error between $I_{i,j}$ and $R_{NN}(a_{i,j})$

2 Interpolation approach

Koch[8][7][6] studied reconstruction of 3D scenes from uncalibrated image sequences. His works has interested us because he have good results and they are appliqued in creation of 3D map of Sagalassos archaeological excavation site. First, his method try to interpolate the different views. To do this, it use dynamic programing. His algorithm is near DTW algorithm. It try to connect each point between views warping views.

Figure 4: Illustration of minimization of distance between to 1D view using dynamic programing. Each axes represent 1D view. We try to connect each point between two views.



Next, it estimate depth of point using stereo matching algorithms. Knowing camera position, it's easy to found depth of point. Finally, it refine estimation using the different viewpoints.

Koch has good results with this approach. Nevertheless, it do not use neural networks.

3 Hybrid solution

Specular approach use only one view of object. We want use informations given by series of views.

Interpolation approach use some characteristics of views we do not want to use and its approach don't use neuronal networks.

We try to make a solution using specular approach of [2] and interpolation approach of [6]. But this work is not yet finished

Conclusion

It should now find a method tow use specular model using picture interpolation. we have not got time to solve it and problem stay open.

Our work is far to be finish. Our solution is only theoretical. This paper is not completed without results, but we hope it could be used as pedagogical aims. We have add in appendices our perceptron implementation used for our tests.

A Listing of PMC library

A.1 pmc_macro.hh

```
#ifndef MACRO_HH
# define MACRO_HH

#ifdef FORALL
#undef FORALL
#endif // ! FORALL

# define FORALL(Type, Container, Ite)                                        \
  for (Type::iterator Ite = Container.begin();                               \
       Ite != Container.end();                                               \
       ++Ite)

#ifdef CFORALL
#undef CFORALL
#endif // ! FORALL

# define CFORALL(Type, Container, Ite)                                       \
  for (Type::const_iterator Ite = Container.begin();                         \
       Ite != Container.end();                                               \
       ++Ite)

#endif
```

A.2 pmc_datatypes.hh

```
#ifndef PMC_DATA_TYPES_HH
# define PMC_DATA_TYPES_HH

#include <iostream>
#include <vector>
#include "pmc_macro.hh"

namespace pmc
{

  class Neuron;
  class Layer;
  class Network;

  typedef double                                  data_type;
  typedef std::vector<data_type>                  datas_type;

  typedef Layer                                   layer_type;
  typedef std::vector<layer_type *>               layers_type;
  typedef std::vector<unsigned int>               desc_layers_type;

  typedef Neuron                                  neuron_type;
  typedef std::vector<neuron_type *>              neurons_type;

  typedef double                                  weight_type;
  typedef std::vector<weight_type>                weights_type;

  inline std::ostream& operator<<(std::ostream &ostr, const datas_type &v)
  {
    CFORALL(pmc::datas_type, v, i)
      ostr << (*i) << " ";
    return ostr;
```

```
  }
} // namespace pmc


#endif // !PMC_DATA_TYPES_HH

#define BIAIS 1.0
```

A.3 pmc_activation.hh

```
#ifndef PMC_ACTIVATION_HH
# define PMC_ACTIVATION_HH

# include "pmc_datatypes.hh"

namespace pmc
{
  class ActivationFunc
  {
  public:
    ActivationFunc(const std::string &name);
    virtual ~ActivationFunc();
    virtual data_type normal(data_type in, double theta) const = 0;
    virtual data_type derivate(data_type in, double theta) const = 0;
    inline std::ostream &print(std::ostream &ostr) const;
  protected:
    const std::string name_;
  };

  // \f f_{act} (x, \Theta) = \tanh (x - \Theta) \f
  class AFTanh : public ActivationFunc
  {
  public:
    inline AFTanh::AFTanh();
    inline data_type normal(data_type in, double theta) const;
    inline data_type derivate(data_type in, double theta) const;
  };

  /// \f f_{act} (x, \Theta) = \frac{1}{1 + e^{-(x - \Theta)}} \f
  class AFLog : public ActivationFunc
  {
  public:
    inline AFLog::AFLog();
    inline virtual data_type normal(data_type in, double theta) const;
    inline virtual data_type derivate(data_type in, double theta) const;
  };

  /// \f f_{act} (x, \Theta) = x - \Theta \f
  class AFLinear : public ActivationFunc
  {
  public:
    inline AFLinear::AFLinear();
    inline data_type normal(data_type in, double theta) const;
    inline data_type derivate(data_type in, double theta) const;
  };


  /** \f f_{act} (x, \Theta) = \left\{ \begin{array}{cl}1.0 & x \geq
      \Theta\\-1.0 & x < \Theta\end{array}\right \f
   */
  class AFBinary : public ActivationFunc
  {
```

```
  public:
    inline AFBinary::AFBinary();
    inline data_type normal(data_type in, double theta) const;
    inline data_type derivate(data_type in, double theta) const;
  };

  /** \f f_{act} (x, \Theta) = 3(AFLog.normal(x, \Theta) - 1/2) \f
   */
  class AFLogNormalized : public AFLog
  {
  public:
    inline AFLogNormalized::AFLogNormalized();
    inline data_type normal(data_type in, double theta) const;
    inline data_type derivate(data_type in, double theta) const;
  };


  inline std::ostream& operator<< (std::ostream &ostr,
                                   const ActivationFunc &func);


} // namespace pmc

# include "pmc_activation.hxx"

#endif // ! PMC_ACTIVATION_HH
```

A.4 pmc_activation.hxx

```
namespace pmc
{
  inline ActivationFunc::ActivationFunc(const std::string &name) : name_(name)
  {
  }

  inline ActivationFunc::~ActivationFunc()
  {
  }

  inline std::ostream &ActivationFunc::print(std::ostream &ostr) const
  {
    ostr << name_;
    return ostr;
  }

  // Tanh

  inline AFTanh::AFTanh() : ActivationFunc("tanh")
  {
  }

  inline data_type AFTanh::normal(data_type in, double theta) const
  {
    return (tanh (in - theta));
  }

  inline data_type AFTanh::derivate(data_type in, double theta) const
  {
    return (1.0 - pow (tanh (in - theta), 2.0));
  }

  // Log
```

```
inline AFLog::AFLog() : ActivationFunc("log")
{
}

inline data_type AFLog::normal(data_type in, double theta) const
{
  return (1.0 / (1.0 + pow (M_E, theta - in)));
}

inline data_type AFLog::derivate(data_type in, double theta) const
{
  double      e_val;

  e_val = pow (M_E, theta - in);

  return (e_val / pow (e_val + 1.0, 2.0));
}


// Normalized Log

inline AFLogNormalized::AFLogNormalized() : AFLog()
{
}

inline data_type AFLogNormalized::normal(data_type in, double theta) const
{
  return 3.0 * (AFLog::normal(in, theta) - 0.5);
}

inline data_type AFLogNormalized::derivate(data_type in, double theta) const
{
  return 0.75 - (pow(normal(in, theta), 2) / 3);
}

// Linear


inline AFLinear::AFLinear() : ActivationFunc("linear")
{
}

inline data_type AFLinear::normal(data_type in, double theta) const
{
  return (in - theta);
}

inline data_type AFLinear::derivate(data_type in, double theta) const
{
  return (1.0);
}

// Binary


inline AFBinary::AFBinary() : ActivationFunc("linear")
{
}

inline data_type AFBinary::normal(data_type in, double theta) const
{
  if (in >= theta)
    return (1.0);
```

```
    return (-1.0);
  }

  inline data_type AFBinary::derivate(data_type in, double theta) const
  {
    return (1.0);
  }


  inline std::ostream& operator<< (std::ostream &ostr,
                                    const ActivationFunc &func)
  {
    return func.print(ostr);
  }
} // namespace pmc
```

A.5 pmc_neuron.hh

```
#ifndef PMC_NEURON_HH
# define PMC_NEURON_HH

# include <vector>
# include <iostream>
# include "pmc_datatypes.hh"
# include "pmc_activation.hh"

namespace pmc
{

  class Neuron
  {
  public:
    Neuron(unsigned int id, unsigned int nb_weights,
           const ActivationFunc *phi);
    ~Neuron();

    std::istream        &load(std::istream &istr);
    std::ostream        &print(std::ostream &ostr) const;
    inline std::ostream &save(std::ostream &ostr) const;

    data_type           operator()(const datas_type &in);

    void                deltaEval(const neurons_type &succ);
    void                deltaEval(data_type optimal, double tolerance);
    inline void         deltaSet(double delta);
    inline double       deltaGet() const;

    inline void         thetaSet(double theta);
    inline double       thetaGet() const;

    inline double       weightGet(unsigned int i) const;
    void                weightsUpdate(double espilon);
    void                weightsModifReset();
    void                weightsModifEval(const neurons_type &prev,
                                         bool socastic);
    void                weightsModifEval(const datas_type &in,
                                         bool stocastic);


    inline void         activationFunctionSet(const ActivationFunc *phi);

    inline data_type    inputGet() const;
```

```
    inline data_type     outputGet() const;


  private:
    unsigned int                id_;
    weights_type                weights_;
    weights_type                weights_diff_;

    const ActivationFunc        *phi_;

    data_type                   input_;
    data_type                   output_;

    double                      delta_;
    double                      theta_;
  };

  inline std::istream& operator>> (std::istream &istr, Neuron &neuron);
  inline std::ostream& operator<< (std::ostream &ostr, const Neuron &neuron);

} // namespace pmc

#include "pmc_neuron.hxx"

#endif // !PMC_NERON_HH
```

A.6 pmc_neuron.hxx

```
#include "pmc_neuron.hh"

namespace pmc
{

  inline void   Neuron::activationFunctionSet(const ActivationFunc *phi)
  {
    phi_ = phi;
  }

  inline void   Neuron::deltaSet(double delta)
  {
    delta_ = delta;
  }

  inline double Neuron::deltaGet() const
  {
    return delta_;
  }

  inline void   Neuron::thetaSet(double theta)
  {
    theta_ = theta;
  }

  inline double Neuron::thetaGet() const
  {
    return theta_;
  }

  inline double Neuron::inputGet() const
  {
    return input_;
  }
```

```
  inline double Neuron::outputGet() const
  {
    return output_;
  }

  inline std::ostream &Neuron::save(std::ostream &ostr) const
  {
    return print(ostr);
  }

  inline double Neuron::weightGet(unsigned int i) const
  {
    return weights_[i];
  }

  inline std::istream& operator>> (std::istream &istr, Neuron &neuron)
  {
    return neuron.load(istr);
  }

  inline std::ostream& operator<< (std::ostream &ostr, const Neuron &neuron)
  {
    return neuron.print(ostr);
  }

} // namespace pmc
```

A.7 pmc_neuron.cc

```
#include <cassert>
#include "pmc_macro.hh"
#include "pmc_neuron.hh"

namespace pmc
{

  Neuron::Neuron(unsigned int id, unsigned int nb_weights,
                 const ActivationFunc *phi)
    : id_(id), weights_(nb_weights, 1), weights_diff_(nb_weights, 0.0),
      phi_(phi), delta_(rand() / (double) RAND_MAX - 0.5), theta_(0)
  {
    for (weights_type::iterator w = weights_.begin();
         w != weights_.end(); ++w) {
      *w = rand() / (double) RAND_MAX - 0.5;
    }

  }

  Neuron::~Neuron()
  {
  }

  std::istream &Neuron::load(std::istream &istr)
  {
    FORALL(weights_type, weights_, i)
      istr >> *i;
    return istr;
  }

  std::ostream &Neuron::print(std::ostream &ostr) const
  {
    ostr << "delta= " << delta_ << "\t|";
    weights_type::const_iterator wdiff = weights_diff_.begin();
```

```
  CFORALL(weights_type, weights_, i)
    ostr << " " << *i << " (" << *(wdiff++) << ")";
  ostr << " |";
  return ostr;
}

data_type     Neuron::operator()(const datas_type &in)
{
  input_ = 0;

  datas_type::const_iterator v = in.begin();
  CFORALL(weights_type, weights_, w)
    input_ += (*w) * (*(v++));
  output_ = phi_->normal(input_, theta_);
  return output_;
}

void  Neuron::deltaEval(const neurons_type &succ)
{
  delta_ = 0.0;
  CFORALL(neurons_type, succ, i)
    delta_ += (*i)->weightGet(id_) * (*i)->deltaGet();
  delta_ *= phi_->derivate(input_, theta_);
}

void  Neuron::deltaEval(data_type optimal, double tolerance)
{
  data_type diff = output_ - optimal;

  delta_ = phi_->derivate(input_, theta_);
  if ((tolerance != 0.0) && (diff >= (-1.0 * tolerance)) &&
      (diff <= tolerance))
    diff = 0.0;
  delta_ *= diff;
}

void  Neuron::weightsModifEval(const neurons_type &prev, bool stocastic)
{
  weights_type::iterator wdiff = weights_diff_.begin();

  CFORALL(neurons_type, prev, p)
    if (stocastic)
      *(wdiff++) = (*p)->outputGet() * delta_;
    else
      *(wdiff++) += (*p)->outputGet() * delta_;
}

void  Neuron::weightsModifEval(const datas_type &in, bool stocastic)
{
  weights_type::iterator wdiff = weights_diff_.begin();
  CFORALL(datas_type, in, i)
    if (stocastic)
      *(wdiff++) = (*i) * delta_;
    else
      *(wdiff++) += (*i) * delta_;

}

void  Neuron::weightsUpdate(double epsilon)
{
  weights_type::const_iterator wdiff = weights_diff_.begin();
  FORALL(weights_type, weights_, w)
    *w = *w - epsilon * (*wdiff++);
```

```
  }

  void  Neuron::weightsModifReset()
  {
    FORALL(weights_type, weights_diff_, w)
      *w = 0.0;
  }

} // namespace pmc
```

A.8 pmc_layer.hh

```
#ifndef PMC_LAYER_HH
# define PMC_LAYER_HH

# include <vector>
# include <iostream>
# include "pmc_datatypes.hh"
# include "pmc_neuron.hh"
# include "pmc_activation.hh"

namespace pmc
{

  class Layer
  {
  public:
    Layer(unsigned int size, unsigned int previous_size,
          const ActivationFunc *phi);
    ~Layer();

    std::istream &load(std::istream &istr);
    std::ostream &print(std::ostream &ostr) const;
    std::ostream &save(std::ostream &ostr) const;

    inline unsigned int size() const;
    inline const neurons_type &neuronsGet() const;

    void        operator()(const datas_type &in, datas_type &out);

    void        deltasEval(Layer &succ);
    void        deltasEval(const datas_type &optimal, double tolerance);
    void        deltasReset();
    void        postprocess(Layer &succ, double epsilon, bool stocastic);
    void        postprocess(const datas_type  &in, double espilon,
                            bool stocastic);
    void        weightsUpdate(double epsilon);
    void        weightsModifReset();


    inline void activationFunctionSet(const ActivationFunc *phi);

  private:
    neurons_type                        neurons_;
  };

  inline std::istream& operator>> (std::istream &istr, Layer &layer);
  inline std::ostream& operator<< (std::ostream &ostr, const Layer &layer);

} // namespace pmc

# include "pmc_layer.hxx"
```

```
#endif // ! PMC_LAYER_HH
```

A.9 pmc_layer.hxx

```
#include "pmc_layer.hh"

namespace pmc
{

  inline void Layer::activationFunctionSet(const ActivationFunc *phi)
  {
    FORALL(neurons_type, neurons_, i)
      (*i)->activationFunctionSet(phi);
  }

  inline void Layer::deltasReset()
  {
    FORALL(neurons_type, neurons_, i)
      (*i)->deltaSet(0.0);
  }

  inline unsigned int Layer::size() const
  {
    return neurons_.size();
  }

  inline const neurons_type &Layer::neuronsGet() const
  {
    return neurons_;
  }

  inline std::istream& operator>> (std::istream &istr, Layer &layer)
  {
    return layer.load(istr);
  }

  inline std::ostream& operator<< (std::ostream &ostr, const Layer &layer)
  {
    return layer.print(ostr);
  }

} // namespace pmc
```

A.10 pmc_layer.cc

```
#include <cassert>
#include "pmc_macro.hh"
#include "pmc_layer.hh"

namespace pmc
{

Layer::Layer(unsigned int size, unsigned int previous_size,
             const ActivationFunc *phi)
  : neurons_(size)
{
  unsigned int id = 0;
  FORALL(neurons_type, neurons_, i)
    (*i) = new neuron_type(id++, previous_size, phi);
}
```

```
Layer::~Layer()
{
  FORALL(neurons_type, neurons_, i)
    delete (*i);
}

std::istream &Layer::load(std::istream &istr)
{
  FORALL(neurons_type, neurons_, i)
    istr >> **i;
  return istr;
}

std::ostream &Layer::print(std::ostream &ostr) const
{
  unsigned int id = 0;
  CFORALL(neurons_type, neurons_, i)
    ostr << "  w" << id++ << ": " << **i << std::endl;
  return ostr;
}

std::ostream &Layer::save(std::ostream &ostr) const
{
  CFORALL(neurons_type, neurons_, i)
    ostr << **i << std::endl;
  ostr << std::endl;
  return ostr;
}
void    Layer::operator()(const datas_type &in, datas_type &out)
{
  out.clear();
  CFORALL(neurons_type, neurons_, i)
  {
    //Biais
    data_type t = BIAIS;
    if (i != neurons_.begin())
      t = (**i)(in);
    out.push_back(t);
  }
}

void    Layer::deltasEval(Layer &succ)
{
  neurons_type successors = succ.neuronsGet();
  FORALL(neurons_type, neurons_, i)
    //Biais
    if (i != neurons_.begin())
      (*i)->deltaEval(successors);
}

void    Layer::deltasEval(const datas_type &optimal, double tolerance)
{
  assert(neurons_.size() == optimal.size() + 1);
  datas_type::const_iterator o = optimal.begin();

  FORALL(neurons_type, neurons_, i)
    //Biais
    if (i != neurons_.begin())
      (*i)->deltaEval(*(o++), tolerance);
}

void    Layer::postprocess(Layer &prev, double epsilon, bool stocastic)
{
```

```
  neurons_type prevNeurons = prev.neuronsGet();
  FORALL(neurons_type, neurons_, i)
    // Biais
    if (i != neurons_.begin())
      (*i)->weightsModifEval(prevNeurons, stocastic);
}

void    Layer::postprocess(const datas_type &in, double epsilon,
                           bool stocastic)
{
  FORALL(neurons_type, neurons_, i)
    //Biais
    if (i != neurons_.begin())
      (*i)->weightsModifEval(in, stocastic);
}

void    Layer::weightsUpdate(double espilon)
{
  FORALL(neurons_type, neurons_, i)
    //Biais
    if (i != neurons_.begin())
      (*i)->weightsUpdate(espilon);
}

void    Layer::weightsModifReset()
{
  FORALL(neurons_type, neurons_, i)
    //Biais
    if (i != neurons_.begin())
      (*i)->weightsModifReset();
}

} // namespace pmc
```

A.11 pmc_network.hh

```
#ifndef PMC_NETWORK_HH
# define PMC_NETWORK_HH

# include <vector>
# include "pmc_layer.hh"
# include "pmc_activation.hh"
# include "pmc_datatypes.hh"

namespace pmc
{

class Network
{
public:
  Network(unsigned int input_size, unsigned int output_size,
          desc_layers_type layers, double epsilon,
          const ActivationFunc *phi);
  Network(const char *filename, double epsilon, const ActivationFunc *phi);
  ~Network();

  inline std::istream &load(std::istream &istr);
  inline std::ostream &print(std::ostream &ostr) const;
  std::ostream &save(std::ostream &ostr) const;

  const datas_type &operator()(const datas_type &in);

  void          weightsUpdate();
```

```
  void          weightsModifReset();

  void          learn(const datas_type &in,
                      const datas_type &optimal,
                      bool stocastic,
                      double tolerance = 0);


  // FIXME: Not yet the good formaula.
  double        errorGet(const datas_type &optimal) const;

  inline void   activationFunctionSet(const ActivationFunc *phi);

private:
  void          layersInit_(desc_layers_type &layers,
                            const ActivationFunc *phi);
  inline void   deltasReset_();

  void          backpropagate_(const datas_type &optimal, double tolerance);
  void          postprocess_(bool stocastic);


private:

  layers_type           *layers_;

  unsigned int          input_size_;
  unsigned int          output_size_;
  double                epsilon_;

  datas_type            output_;
  datas_type            input_;

};

inline std::istream& operator>> (std::istream &istr, pmc::Network &net);
inline std::ostream& operator<< (std::ostream &ostr, const pmc::Network &net);

} // namespace pmc

#include "pmc_network.hxx"

#endif /* !PMC_NETWORK */
```

A.12 pmc_network.hxx

```
#include <cassert>
#include "pmc_network.hh"

namespace pmc
{

  inline void Network::activationFunctionSet(const ActivationFunc *phi)
  {
    FORALL(layers_type, (*layers_), i)
      (*i)->activationFunctionSet(phi);
  }

  inline void Network::deltasReset_()
  {
    FORALL(layers_type, (*layers_), i)
      (*i)->deltasReset();
  }
```

```
  inline std::istream &Network::load(std::istream &istr)
  {
    FORALL(layers_type, (*layers_), i)
      istr >> **i;
    return istr;
  }

  inline std::ostream &Network::print(std::ostream &ostr) const
  {
    unsigned int id = 0;
    CFORALL(layers_type, (*layers_), i)
    {
      ostr << "Layer " << id++ << std::endl;
      ostr << **i;
    }
    return ostr;
  }

inline std::istream& operator>> (std::istream &istr, pmc::Network &net)
{
  return net.load(istr);
}

inline std::ostream& operator<< (std::ostream &ostr, const pmc::Network &net)
{
  return net.print(ostr);
}

} // namespace pmc
```

A.13 pmc_network.cc

```
#include <cassert>
#include <vector>
#include <fstream>
#include "pmc_macro.hh"
#include "pmc_network.hh"

namespace pmc
{

  Network::Network(unsigned int input_size, unsigned int output_size,
                   desc_layers_type layers, double epsilon,
                   const ActivationFunc *phi)
    : input_size_(input_size), output_size_(output_size),
      epsilon_(epsilon)
  {
    assert(input_size_);
    assert(output_size_);

    // Je ne suis pas d'accord avec cet assert, on peut avoir un PMC sans
    // couche intermediaire
    //assert(layers.size());

    //biais
    input_size_++;
    output_size_++;
    FORALL(desc_layers_type, layers, i)
      (*i)++;

    layers.push_back(output_size_ );
    layersInit_(layers, phi);
```

```
  assert(layers_);
}

Network::Network(const char *filename, double epsilon,
                 const ActivationFunc *phi)
  : input_size_(0), output_size_(0), epsilon_(epsilon)
{
  std::ifstream file(filename);

  try
  {
    // Read number of layers.
    unsigned int layers = 0;
    file >> layers;
    if (layers < 2)
      throw 1;

    // Read input size.
    file >> input_size_;
    if (!input_size_)
      throw 2;

    // Read each layer size.
    desc_layers_type sizes(layers - 2);
    FORALL(desc_layers_type, sizes, i)
    {
      if (!file.good())
        throw 3;
      file >> *i;
    }

    // Read output size.
    file >> output_size_;
    if (!output_size_)
      throw 4;
    sizes.push_back(output_size_);

    // Build Network.
    layersInit_(sizes, phi);
    load(file);
  }
  catch(...)
  {
    file.close();
    std::cerr << "Network (loader): Error while loading file: '"
              << filename << "'" << std::endl;
    exit(0);
  }
}

void Network::layersInit_(desc_layers_type &layers, const ActivationFunc *phi)
{
  layers_ = new layers_type(layers.size());
  layers_type::iterator l = layers_->begin();

      std::cout << "layers : " << layers[0] << std::endl;
  unsigned int previous_size = input_size_;

  CFORALL(desc_layers_type, layers, i)
  {
    std::cout << "previous : " << previous_size << std::endl;
    *(l++) = new layer_type(*i, previous_size, phi);
```

```
    previous_size = *i;
    }
}

Network::~Network()
{
  if (layers_)
  {
    FORALL(layers_type, (*layers_), i)
      delete *i;
    delete layers_;
  }
}

std::ostream &Network::save(std::ostream &ostr) const
{
  ostr << layers_->size() + 1 << "\n"
       << input_size_ << " ";
  CFORALL(layers_type, (*layers_), i)
    ostr << (*i)->size() << " ";
  ostr << "\n" << std::endl;
  CFORALL(layers_type, (*layers_), i)
    (*i)->save(ostr);
  return ostr;
}

const datas_type &Network::operator()(const datas_type & in)
{

  assert(in.size()+1 == input_size_);

  input_ = in;
  input_.insert(input_.begin(), BIAIS);
  datas_type input = input_;

  CFORALL(layers_type, (*layers_), i)
  {
    (**i)(input, output_);
    input = output_;
  }
  assert(output_.size() == output_size_);
  // Virer le Biais
  output_.erase(output_.begin());
  return output_;
}

void Network::backpropagate_(const datas_type &optimal, double tolerance)
{
  assert(optimal.size()+1 == output_size_);

  layers_type::reverse_iterator i(layers_->end());
  layers_type::reverse_iterator end(layers_->begin());

  layer_type *succ = 0;

  (*i)->deltasEval(optimal, tolerance);
  for (succ = *(i++); i != end; succ = *(i++))
    (*i)->deltasEval(*succ);
}

void Network::postprocess_(bool stocastic)
{
  layer_type *prev = 0;
```

```
    FORALL(layers_type, (*layers_), i)
    {
      if (prev == 0)
        (*i)->postprocess(input_, epsilon_, stocastic);
      else
        (*i)->postprocess(*prev, epsilon_, stocastic);
      prev = (*i);
    }
  }

  void Network::weightsUpdate()
  {
    FORALL(layers_type, (*layers_), i)
      (*i)->weightsUpdate(epsilon_);
  }

  void  Network::weightsModifReset()
  {
    FORALL(layers_type, (*layers_), i)
      (*i)->weightsModifReset();
  }


  double Network::errorGet(const datas_type &optimal) const
  {
    double error = 0;
    datas_type::const_iterator j = output_.begin();
    CFORALL(datas_type, optimal, i)
      {
        error += ((*j) - (*i)) * ((*j) - (*i));
        ++j;
    }
    return error;
  }

  void Network::learn(const datas_type &in, const datas_type &optimal,
                      bool stocastic, double tolerance)
  {
    this->operator()(in);
//     std::cout << "in: "<< in[0] << ", " <<in[1]<< std::endl;
//     std::cout << "out : " << output_[0] << ", " <<output_[1]<< std::endl;
//     std::cout << "optimal : " << optimal[0] << ", " <<optimal[1]<< std::endl;
    backpropagate_(optimal, tolerance);
    postprocess_(stocastic);
  }

} // namespace pmc
```

A.14 test.cc

```
#include "pmc_datatypes.hh"
#include "pmc_network.hh"
#include "pmc_activation.hh"

using namespace pmc;

int main()
{
  pmc::AFLog f;
  pmc::Network net("sample.in", 0.2,  &f);
  std::cout << net << std::endl;
  //net.save(std::cout);
```

```
  pmc::datas_type in;
  in.push_back(-1);
  in.push_back(1);
  pmc::datas_type ref;
  ref.push_back(0.2);
  ref.push_back(0.7);

  std::cout << net(in) << std::endl;

  for (unsigned int i = 0; i < 1000; ++i)
    net.learn(in, ref, 0);

  std::cout << net << std::endl;
  std::cout << net(in) << std::endl;


  return 0;
}
```

References

[1] M. BROWN, T. DRUMMOND, AND R. CIPOLLA, *3d model acquisition by tracking 2d wire-frames*, BMVC, (2000).

[2] CHO AND T. CHOW, *Neural computation approach for developing a 3-d shape reconstruction model*, IEEE Transactions on neural networks, (2001).

[3] Y.-J. DANIEL, *Cours de réseaux de neuronnes*, 2004. Le cours de Réseau de Neuronne d'EPITA.

[4] M. DESCOTEAUX, *Learning shape-from-shading by network models*, (2002).

[5] T. JIANG, *A neural network approach to shape from shading*, (1999).

[6] R. KOCH, M. POLLEFEYS, AND L. V. GOOL, *Realistics 3d scene modeling from uncalibrated image sequences*.

[7] ——, *Realistics surface reconstruction of 3d scenes from uncalibrated image sequences*, (2000).

[8] R. KOCH, M. POLLEFEYS, B. HEIGL, L. V. GOOL, AND H. NIEMANN, *Calibration of hand-held camera sequences for plenoptic modeling*, (1999).

[9] L. OISEL, *Reconstruction 3D de scène complexe à partir de séquences video non calibrée : estimation et maillage d'un champ de disparité*, PhD thesis, Université de Renne I, november 1998.

[10] E. STEINGBACH, B. GIROD, P. EISERT, AND A. BETZ, *3-d object reconstruction using spatially extended voxels and multi-hypothesis voxel coloring*, icpr, (2000).

[11] R. SZELISKI AND P. GOLLAND, *Stereo matching with transparency and matting*, ijcv, (1998).

[12] G. ZENG, S. PARIS, M. LHUILLIER, AND L. QUAN, *Study of volumetric methods for face reconstruction*, iac, (2003).