IM INTERACTIVE MULTIMODAL INFORMATION MANAGEMENT

MUCATAR

People Tracking and Activity Recognition using Multiple Cameras

White Paper IM2 IM2.WP.MUCATAR

D2 :

Sequential Monte Carlo Software - Description

Technical informations related to this report:

- Project title : MUCATAR
- Delivrable type : Public
- Deliverable number : D2
- Contractual Date of Delivery: July 2003
- Actual Date of Delivery: July 2003
- Deliverable title : Sequential Monte Carlo Software - Description
- Nature of the Deliverable: Software and software description
- Authors: Jean-Marc Odobez IDIAP
Daniel Gatica-Perez IDIAP
- Keyword list : Sequential Monte-Carlo - Software - Torch3.

Contents

1 Introduction **1**

2 Algorithmic description - Main elements of software **1**

2.1 Particle filtering 1

2.2 Basic elements of the software 2

3 Installation - An example of use **3**

3.1 Installation - Software organization 3

3.2 An example 3

4 Application to visual tracking - Conclusion **5**

1 Introduction

This report describes the software developped at IDIAP to implement the Sequential Monte-Carlo (SMC) filtering techniques. Generic and modular tools have been implemented. All the algorithmic parts that are application-independant are integrated into the Torch3 environment (www.torch.ch).

In this report, we first recall the basics of SMC filtering and describe the main elements of the software. In Section 3, we explain how to install the software, and provide an example of use in a simple filtering case. We then provide some indication on the implemented visual tracking algorithms and conclude.

2 Algorithmic description - Main elements of software

2.1 Particle filtering

Particle filtering is a technique for implementing a recursive Bayesian filter by Monte-Carlo simulations. The key idea is to represent the required density function by a set of random samples with associated weights. Let $c_{0:k} = \{c_l, l = 0, \ldots, k\}$ (resp. $z_{1:k} = \{z_l, l = 1, \ldots, k\}$) represents the sequence of states (resp. of observations) up to time k. Furthermore, let $\{c^i_{0:k}, w^i_k\}^{N_s}_{i=1}$ denote a set of weighted samples that characterizes the posterior probability density function (pdf) $\mathrm{p}(c_{0:k}|z_{0:k})$, where $\{c^i_{0:k}, i = 1, \ldots, N_s\}$ is a set of support points with associated weights w^i_k. The weights are normalized such that $\sum_i w^i_k = 1$. Then, a discrete approximation of the true posterior at time k is given by :

$$\mathrm{p}(c_{0:k}|z_{1:k}) \approx \sum_{i=1}^{N_s} w^i_k \delta(c_{0:k} - c^i_{0:k}) \,. \tag{1}$$

The weights are chosen using the principle of Importance Sampling (IS), and the goal of the particle filtering algorithm is the recursive propagation of the samples and estimation of the associated weights as each measurement is received sequentially. Under the three hypotheses commonly made to derive the standard particle filter :

1. The state sequence $c_{0:k}$ follows a first-order Markov chain model, characterized by the definition of the dynamics $\mathrm{p}(c_k|c_{k-1})$.
2. The observations $\{z_k\}$, given the sequence of states, are independent. This leads to $\mathrm{p}(z_{1:k}|c_{0:k}) = \prod_{i=1}^{k} \mathrm{p}(z_k|c_k)$, which requires the definition of the individual data-likelihood $\mathrm{p}(z_k|c_k)$;
3. The prior distribution $\mathrm{p}(x_{0:k})$ is employed as importance function. In this case, $q(c_k|c_{0:k-1}, z_{1:k}) = \mathrm{p}(c_k|c_{k-1})$.

we obtain the following recursive equation for the weight :

$$w^i_k \propto w^i_{k-1} \; \mathrm{p}(z_k|c^i_k). \tag{2}$$

It is known that importance sampling is usually inefficient in high-dimensionnal, which is the case of the state space $c_{0:k}$ as k increases. To solve this problem, an additional resampling step is necessary, whose effect is to eliminate the particles with low importance weights and to multiply particles having high weights, giving rise to more variety around the modes of the posterior after the next importance sampling step.

Altogether, we obtain the standard particle filter that is displayed in Fig. 1.

1. Initialisation, k=0
   For $i = 1, \ldots, N_s$, sample $c_0^i \sim \mathrm{p}(c_0)$ and set $k = 1$

2. Importance sampling step
   - For $i = 1, \ldots, N_s$, sample $\tilde{c}_k^i \sim \mathrm{p}(c_k | c_{k-1}^i)$
   - For $i = 1, \ldots, N_s$, evaluate the importance weights : $\tilde{w}_k^i \propto w_{k-1}^i \; \mathrm{p}(z_k | \tilde{c}_k^i)$
   - Normalize the importance weights $\tilde{w}_k^i$

3. Selection step
   - Resample with replacement N_s particles $\{c_k^i, w_k^i = \frac{1}{N_s}\}$ from the sample set $\{\tilde{c}_k^i, \tilde{w}_k^i\}$
   - set $k = k + 1$ and go to step 2

Figure 1: The SIR algorithm.

2.2 Basic elements of the software

Analising the SIR algorithm (1), we can see that it is charcacterized by the following elements :

1. the definition of a state space with state elements c.
2. the definition of a particle distribution, characterised by a set of particles c^i and associated weights w^i
3. the definition of the dynamics $\mathrm{p}(c_k | c_{k-1})$. It is a conditional distribution from which we can sample from (see step 2).
4. the definition of the data-likelihood $\mathrm{p}(z_k | c_k)$. It is a conditional distribution that we can evaluate (see step 2).

Accordingly, we have defined similar C++ classes that match these elements :

1. a class for the state (**bf_RandomVariable**). It has a generic structure to contain the state data elements.
2. a particle distribution class, **bf_ParticleDistribution**, to handle a particle set, and being able to sample from it.
3. a generic set of classes to represent distributions (cf files bf_Distribution.h bf_Distribution.h) that are conditional or not, that can be sampled or evaluated. A variant of these have been defined to handle simple state variables characterized by a vector or reals (cf files bf_DistributionReal.h, bf_DistributionReal.h). These classes can be used to define the dynamical model and the data-likelihood term.

The algorithm itself is implemented in the particle filter class **bf_ParticleFilter**, whose header is given is printed in (2). It is easy to see that it requires the above mentioned elements to be defined at creation. Note that the implementation of the loop is performed

in another class (called **bf_Trainer**, although it does not train anything), and requires the definition of a data random variable providing data in sequential order.

To illustrate the use of these classes, we present the implementation of a basic example in the next section.

3 Installation - An example of use

In this section, we first describe the installation of the software and its structure. We then explain in detail an example of use of the software.

3.1 Installation - Software organization

The software is based on the Torch3 software package (www.torch.ch), though not much of it is used. It is available as a large archive (that already includes the Torch package). The image processing parts are based on the opencv library[1].

The main difference with respect to Torch is the addition of three directories in the base directory of the Torch package :

- the bayes_filter directory : it contains all the bayes generic filtering classes.
- the image_processing directory : it contains all the image related classes.
- the bayes_image directory : it contains the implementation of all the classes that specifically implement the bayes filtering classes for visual tracking.

An associated library is generated with each of these directoy.

To install the software, unzip it and untar it. It compiles and runs under both Linux and Unix. The package includes a Makefile, and a Makefile_options_Linux where you can change some options (select the g++ compiler, add libraries to compile your own program, etc). To compile the libraries (the Torch one and the new ones), you need to apply the following steps :

- make clean : remove the dependency files, the object files and the libraries for the current system.
- make depend : create the dependency files.
- make : compile the libraries.

3.2 An example

To illustrate the programing, let us consider a one dimensional example. In this example, the dynamics is given by :

$$x_k = x_{k-1} + (x_{k-1} - x_{k-2}) + \nu$$

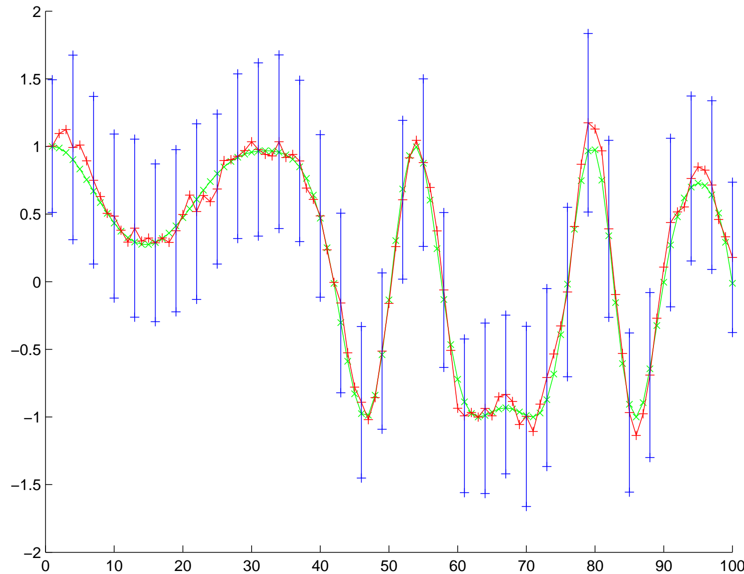
where ν is a random gaussian noise with variance σ^2_{dyn}. Thus, we need an augmented state $c_k = (x_k, x_{k-1})$ and the "full" dynamic is given by :

$$c_k = Ac_{k-1} + B\nu$$

[1] freely available at http://www.intel.com/research/mrl/research/opencv/

```
#ifndef BF_PARTICLEFILTER_INC
#define BF_PARTICLEFILTER_INC

#include "general.h"
#include "bf_RandomVariable.h"
#include "bf_ParticleDistribution.h"
#include "bf_Distribution.h"
#include "bf_PredictModel.h"
#include "bf_BayesFilter.h"

namespace Torch {

//-----

/** Implements the basic particle filter (condensation)

    @author Jean-Marc Odobez (Jean-Marc.Odobez@idiap.ch)
    @author Daniel Gatica-Perez (gatica@idiap.ch)
*/

  class bf_ParticleFilter : public bf_BayesFilter
  {

    public:

    //members

    bf_ParticleDistribution *m_pParticleSet;
    bf_ParticleDistribution *m_pParticleSetAux;
    bf_SamplePredictModel   *m_pDynamicModel;
    bf_EvalCondDist         *m_pObservationLikelihood;

    //member functions

    //-----
    bf_ParticleFilter(bf_ParticleDistribution *pPrior,
                      bf_ParticleDistribution *pAux,
                      bf_SamplePredictModel *pDynMod,
                      bf_EvalCondDist *pObsLike);

    ///////////////////////////////////////////
    // functions inherited from class BayesianFilter
    ///////////////////////////////////////////
    //-----
    virtual void init(void);

    //-----
    // Prediction step of the algorithm
    virtual void predict(void);

    //-----
    // implements that update step of the algorithm
    virtual void observe(bf_RandomVariable *pData);

    //-----
    // may be needed by more general particle filters
    virtual void update(void);

    //-----
    // performs one iteration of the algorithm
    virtual void iterate(bf_RandomVariable *pData);

    ////////////////////////////////////////////
    //specific member functions of particle filter
    ////////////////////////////////////////////

    // to exchange the particle distribution
    virtual void exchange(void);

    // implement a resampling step, if necessary
    virtual void resample(void);

    //-----
    virtual ~bf_ParticleFilter();

  };
}

#endif
```

Figure 2: The particle filter class **bf_ParticleFilter**.



(a) particle filtering

Figure 3: Example of particle filtering : 100 particles. In green, the observations. In red, the mean of the samples. In blue, plus/minus one standard deviation.

with :

$$A = \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

The likelihood is classically given by :

$$z_k = x_k + \eta$$

where η is a random gaussian noise with variance σ^2_{like}, leading to a likelihood function :

$$\text{p}(z_k|c_k) \propto \exp\left(-\frac{(x_k - z_k)^2}{2\sigma^2_{like}}\right)$$

The figure 4, 5 and (6) display how the example was coded using the software. Figure 3.2 shows a plot of running the executable.

4 Application to visual tracking - Conclusion

The same methodology has been applied to the visual tracking case. We have developped trackers that are based on histogram likelihoods, contour likelihoods, the product of both (see deliverable D1). Figure 7 shows as an example the options available with the histogram tracker.

It handles different color representations as well as different transformation. The software has been made modular so that each component (likelihood, sequential monte-carlo method, dynamics) can be changed easily.

```
#include <stdio.h>

#include "bf_DynamicModel.h"

#include "bf_MixedParticleDistribution.h"
#include "bf_DataRandomVariable.h"
#include "bf_ParticleFilter.h"
#include "bf_Trainer.h"

using namespace Torch;


/*
  [xk+1  xk] = A [xk  xk-1] + B nu

  A = [2 -1; 1 0] (A=Fx in implementation)   B = [1]

  nu = noise : variance sigma2;
*/

class MyPredictor : public bf_LinearPredictModelReal {

  public:
    bf_RandomGenerator   rng;

  public:
    MyPredictor(float sigma2=1.) :
    bf_LinearPredictModelReal(2,1,&rng) {

    m_pRng=&rng;

    Fx.ptr[0][0]=2; Fx.ptr[0][1]=-1;
    Fx.ptr[1][0]=1; Fx.ptr[1][1]=0;

    G.ptr[0][0]=1;

    q.ptr[0]=sigma2;

  }

  ~MyPredictor(){  }

};


//==========================================================
// likelihood ( p(yk| [xk xk-1]) proportional to exp(-(yk-xk)^2 / (2*sigma2))
//   yk = data, x = [xk xk-1] = state

class MyLikelihood : virtual public bf_EvalCondDistReal {

  public:
    real m_rCoef;

  MyLikelihood(real sigma2) : bf_EvalCondDistReal() {
    m_rCoef = 1./(2*sigma2);
  }

  // computing P ( X | Y ) : here first element of state is
  virtual inline real evaluateConditionalReal(real *pX, real *pY){
    return exp(-m_rCoef*(*pX-*pY)*(*pX-*pY));
  }

  ~MyLikelihood(){}

};
```

Figure 4: Classes to implement the particle filter example. The prediction model and the likelihood model.

```
#include <stdio.h>

#include "EasyExampleClasses1.h"
#include "bf_MixedParticleDistribution.h"
#include "bf_DataRandomVariable.h"
#include "bf_Trainer.h"

using namespace Torch;

//=========================================================
// My Data should be read from a file. Here we use a model to
// generate them
//
class MyData : virtual public bf_DataRandomVariable {

  public :
    real observation,w,wv,iter;

  MyData(){ w=0.08; iter=0; addToData(&observation); }

  inline virtual void initData(){ wv=w*(1+0.9*cos(iter*0.1));     observation=(real)(cos(wv*iter));  }

  inline virtual void nextData(){ iter++; wv=w*(1+0.9*cos(iter*0.1));     observation=(real)(cos(wv*iter));  }

  inline virtual bool dataAvailable(){ return true; }

  ~MyData(){}

};
//=========================================================

class MyTracker : public bf_Trainer {
  //-- members

 public:
  // for the particles
  bf_RandomGenerator              *m_pRng1;
  bf_RandomGenerator              *m_pRng2;
  bf_MixedParticleDistribution    *m_pPosteriorDistribution;
  bf_MixedParticleDistribution    *m_pAuxDistribution;

  // for the dynamical model
  MyPredictor                     *m_pDynamics;

  // for the likelihood
  MyLikelihood                    *m_pLikelihood;

  // pointers to filter and data are inherited from the Trainer
  // nevertheless, here we declare a particle filter
  bf_ParticleFilter               *m_pParticleFilter;

  MyTracker(bf_DataRandomVariable *pZ,int NberOfParticles,
          real sigma2Dyn=1.,real sigma2Like=1.) :
    bf_Trainer(NULL,pZ) {

    m_pRng1=new bf_RandomGenerator(3);    m_pRng2=new bf_RandomGenerator(2);

    int  StateSize=2;
    m_pPosteriorDistribution=new bf_MixedParticleDistribution(NberOfParticles,1,StateSize,0,m_pRng1);
    m_pAuxDistribution=new bf_MixedParticleDistribution(NberOfParticles,1,StateSize,0,m_pRng2);

    m_pDynamics=new MyPredictor(sigma2Dyn);
    m_pLikelihood=new MyLikelihood(sigma2Like);
    m_pParticleFilter =new bf_ParticleFilter(m_pPosteriorDistribution,m_pAuxDistribution,
                                             m_pDynamics,m_pLikelihood);
    m_pBayesFilter = m_pParticleFilter;
  }
 //--------
  void  init(long N=-1){
    bf_Trainer::init(N);

    real *initstate=new real [2];
    // initialization with the first data obervation available
    initstate[0]=m_pDataInput->m_cData.nodes[0][0];
    initstate[1]=m_pDataInput->m_cData.nodes[0][0];
    m_pPosteriorDistribution->setAllSample(initstate,0);
  }

 //--------
  ~MyTracker(){
    delete m_pDynamics;    delete m_pRng1;    delete m_pRng2;
    delete m_pPosteriorDistribution;    delete m_pAuxDistribution;
    delete m_pParticleFilter;    delete m_pLikelihood;
  }
};
```

Figure 5: Classes to implement the particle filter example. The data class and the tracker itself

```
#include "EasyExampleClasses2.h"

int main (int argc, char **argv)
{

  int iter,NbIter=100,NbParticles=100;

  MyData        md;
  real          sigma2Dyn=0.5,sigma2Like=0.5;
  MyTracker     mt(&md, NbParticles, sigma2Dyn, sigma2Like);

  mt.init();

  // filtering loop
  iter=0;
  while(iter<NbIter){

    // perform iteration with current data
    mt.iterate();

    // DISPLAY SOME RESULT
    bf_MixedParticleDistribution *pCurrent=(bf_MixedParticleDistribution *)mt.m_pParticleFilter->m_pParticleSet;

    real  mean[2],variance[2];

    // Getting mean and variance
    pCurrent->getMeanVariance(mean,variance,0);

    // output observation, mean and variance
    printf("%f \t%f \t%f \n",md.observation,mean[0],variance[0]);

    // switch to next data
    mt.nextData();

    iter++;
  }

}
```

Figure 6: The main file for the example.

```
Tracking a square region based on color histograms and particle filter
#
# usage: Linux_OPT_FLOAT/TrackH [options] <OneImageName> <First> <Last>
#

Arguments:
  <OneImageName>     ->  Name of one of the image in the sequence (<string>)
  <First>            ->  number of the first image to process (<int>)
  <Last>             ->  number of the last image to process (<int>)

General options:
  -step <int>        ->  step between two image number [1]
  -transform <int>   ->  transformation to track :
 1(translation+scale) 2(trans+scalex+scaley) others (translation) [1]
  -imagedir <string> ->  directory of the input images []

Particle Filter options:
  -nbsamples <int>   ->  number of particle samples [500]

Histogram likelihood options:
  -colormodel <int>  ->  color model (1 = HSV, 0 = RGB) [1]
  -colormodel <int>  ->  number of bands to keep (for histograming)
 in the chosen model [3]
  -nbbin <int>       ->  number of bins in the histogram [8]
  -lambdah <real>    ->  lambda coefficient in exponential distribution [19.5312]
  -partitionh <real> ->  partition function of the exponential distribution [1]

Initial box options:
  -botrightco <int>  ->  column number of bottom right initial box corner [-1]
  -botrightli <int>  ->  line number of bottom right initial box corner [-1]
  -upleftco <int>    ->  column number of bottom right initial box corner [-1]
  -upleftli <int>    ->  column number of bottom right initial box corner [-1]
  -splitheight <int> ->  number of box split of the height [1]
  -splitwidth <int>  ->  number of box split of the width [1]

Dynamics options:
  -trans_std <real>  ->  noise standart deviation (translation components) [2]
  -affine_std <real> ->  noise standart deviation (affine components) [0.01]
```

Figure 7: Example of options of the histogram tracker.