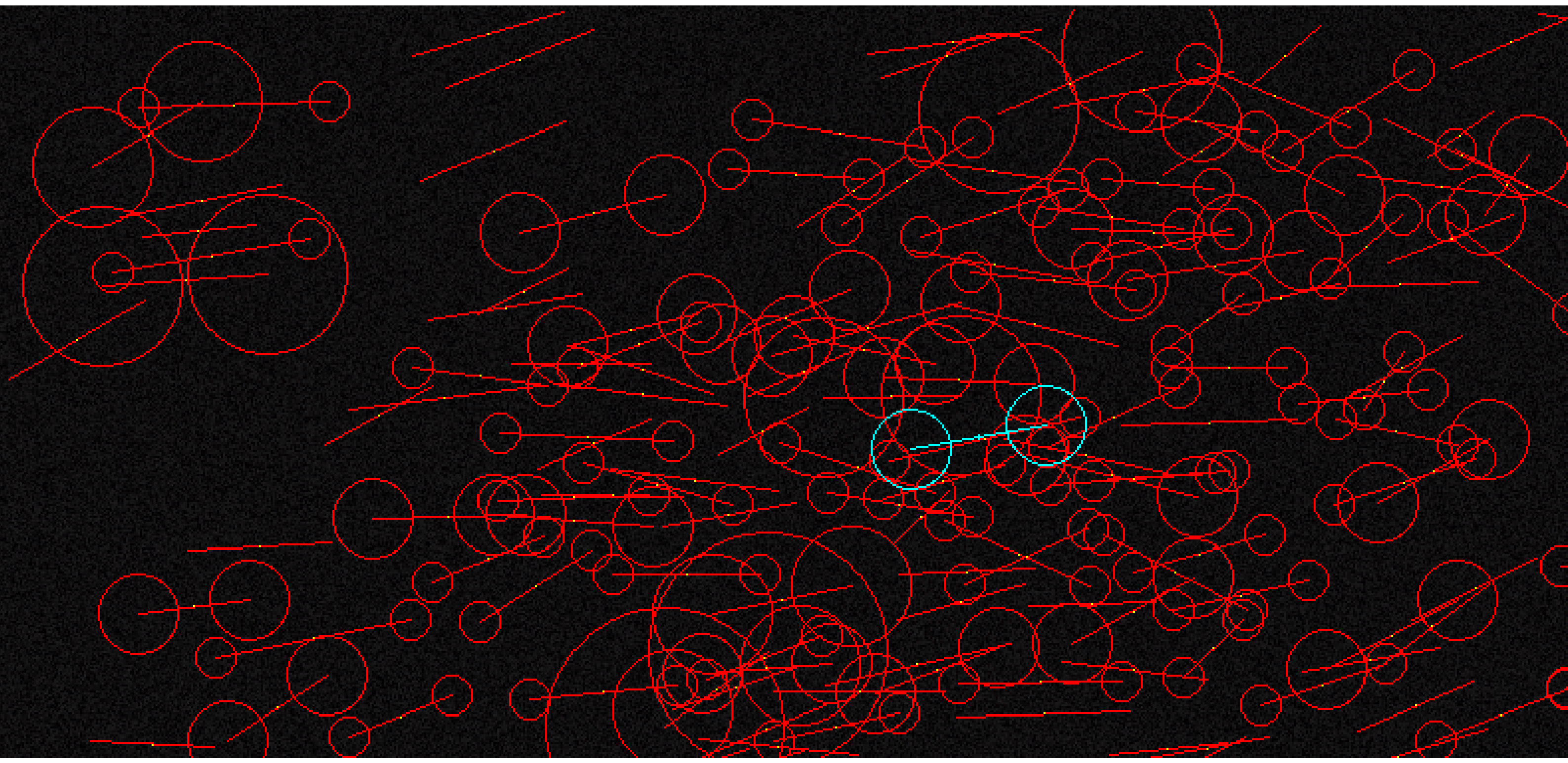

C# / .Net Implementation of Particle Filter for Eye Tracking

A 4-week project

Lars Pellarin
pellarin@itu.dk
280979-xxxx

Vedrana Andersen
vedrana@itu.dk
130274-xxxx

Project supervisor: Dan Witzner Hansen
IT University of Copenhagen, May 2006

Contents

Preface

This report describes a four-week project, which we conducted under supervision of Dan Witzner Hansen at the IT University in May 2006. In those short four weeks, we wrote our first 'Hello, world!' program in C#, and we implemented a generic particle filter framework for object tracking, plus a few simple filters to exemplify the program. Our filters can, after a user-guided initialization, successfully track eyes in a video sequence. The C# implementation of our filters is using `DWImageLibrary.dll`, provided by our supervisor, which enables us to import video AVI files and manipulate images efficiently.

1 Introduction

Our main goal for this project was to become familiar with the C# programming language and .NET framework. We chose to implement a particle filter as an extension of the Computer Vision project cluster, where we developed interest for tracking algorithms, and where we made our first tracking steps in Matlab. The secondary goal we had was to implement a rounded-up application, which (we can pretend) will reach the end user or someone that will extend it.

We joined the C#/.Net project cluster and during the first project week attended the lectures on C# given by Peter Sestoft. We worked on handed-out exercises, but also studied the tracking literature, and discussed the possible implementations of a particle filter. In the second week we became familiar with the provided C# image library, made the initial graphical user interface (GUI) and implemented the basic tracking algorithms. Improvements on GUI and implementation of more complex tracking models took place it the third and fourth week; testing our program and writing the report took a chunk of the last week.

Our program implements the basic structure of a particle filter widely used in Computer Vision for motion tracking. Our aim was to implement a particle filter framework which can be easily extended with new models and methods and here we benefited from C# .NET's generic capabilities and delegates (similar to anonymous inner classes or function pointers). We expected (and succeeded) to provide at least a filter based on background subtracted images, and one color-histogram based eye tracking model. All the filters should be able to work on recorded test videos, and output the tracking result by plotting the tracking path on screen and saving it in a file.

Having to implement a relatively large program in a programming language we were not familiar with meant that we did not have a clear picture of our implementation at the point of starting to code. Also, many of our decisions were greatly influenced with our curiosity and desire to learn C#. When we had to make a decision concerning the implementation, we often chose not to take the 'safe' way, but we tried to test as much of features provided by C#. Following the decision path of "Let's see how does the struct type, operator overloading, variable-arity…work?" we ended up with the implementation that is maybe not optimal, but has served as an excellent learning field.

Another set of decisions we were often faced with was wether to put the focus on algorithmic solutions or user-friendliness of our implementation. We agreed that it is particle filtering we want to work on, and that we will not spend lot of time on cosmetics, but we could not help being annoyed when our program broke only if we clicked the buttons in the wrong order. We chose a compromise solution to this problem, fool-proofing the program when we saw very strong need for it, and implementing those GUI features that we thought were needed or fun.

2 Background

As the first step in developing a our tracking program, we had to study the relevant theory. We had some knowledge of particle filtering from the Computer Vision project cluster, and have implemented a simple particle filter in one of the exercises. Still, to built a general particle filter, we had to try to imagine as many different ways of using it as possible. As for our eye-tracker, we had to find the appropriate method for locating the eye in the frame. We have therefore started our project by studying the relevant Computer Vision literature.

2.1 Particle Filtering

Tracking a moving object in a video sequence is an important aspect of computer vision. All tracking algorithms incorporate a dynamic model (describing how the object moves), and the analysis of a video frame for estimation (measurement) of an object's location. Both the motion and measurement are uncertain, noisy processes, giving rise to tracking algorithms based on theory of probability.

If the motion and measurement can be approximated by the linear model with Gaussian noise, an exact solution to a tracking problem can be found by Kalman filtering (Forsyth and Ponce, 2002, [2]; Trucco and Verri, 1998, [5]). However, many tracking problems can not be modeled by simple linear models.

Particle filtering is a probabilistic tracking method, where the relevant probability distributions are not represented explicitly, but are obtained by means of sampling. Using a large set of weighted, random samples (particles) it is possible to approximate the probability distributions needed for motion tracking. The method of employing sampling to approximate the quantitative problems is generally referred to as the *Monte Carlo* method.

Particles describe some object we want to locate in the frame. In a filter's most simple form, particles are just points spread out over a video image. More complex model is, for example, condensation algorithm used for tracking curves, where particles contain parametrization of the curve in terms of B-splines (Isard and Blake, 1998, [3]) . To track a feature in a video, a particle set is evaluated, resampled and propagated in every frame.

Evaluation

Evaluating particles means assigning weight to each particle according to some method. Weight is the measure of probability that the particle coincides with the object of interest, so evaluation step ensures that particles do represent the probability of finding the tracking object in the frame.

Evaluating methods vary depending on the nature of the tracking problem. One of the simplest evaluating schemes is to obtain the weight by comparing the pixel value at the particle position with the pixel value of the object. Depending on the model used, evaluations vary in complexity. Some of the possible evaluation methods include comparing intensity distributions, using correlation and using edge detection.

Resampling

Resampling is needed to ensure a good representation of the probability distribution. When resampling, we make a new set, where only the 'heavy' particles survive and possibly multiply (survival of the fittest). It is done to prevent the *degeneracy problem* which occurs when most particles have negligible weight. Instead of resampling in each iteration (frame) of filtering, a threshold on weight degeneracy can be used to determine if resampling is needed (Maskell, 2001, [1]).

Resampling can be done by randomly drawing a new set of particles, where each particles chance of 'survival' is determined by its weight, but alternative algorithms exist. For example, systematic resampling takes only $O(N)$ time, where N is the number of particles (Maskell, 2001, [1]).

Propagation

Propagation includes moving the particles according to some deterministic dynamics (drift) and then perturbing each particle individually (diffuse). It is often enough just to use diffuse propagation, which in practice means adding a certain amount of noise to the particle parameters. The resampling step has insured that the concentration of particles is high at positions where the probability of finding the object is big. Propagation step should then spread those highly concentrated particles over the neighboring region, so that in the next frame we again have a chance of locating the tracking object.

3 Program Description and Analysis

As mentioned previously, our implementation provides the general particle filtering framework, which is able to handle different particle models (we also call them *states*), different evaluation methods, different resampling methods and different propagation methods. A basic resampling method is also provided, as well as a couple of example states and example evaluation methods. Our intention was to implement the framework which can later on be extended with different models. In this section we bring the description of the general program design and analysis.

3.1 Generic Particle Filter

Active part

Envisioning a generic particle filter was not difficult; it should iteratively manipulate a set of states, evaluating, resampling and propagating it in every frame of the video. It is the video frames providing the basis for evaluation.

We decided to make an `IState` interface which defines minimal requirements for a particle filtering states, most importantly state's position and weight. Depending on the nature of the filtering problem, we would then define particle state classes that are suitable of handling all needed data.

The `ParticleFilter` class would have a field holding a set of particles in generic `List<IStates> particleSet` collection. It is manipulation of this collection that is the core of particle filtering. We discussed implementing resampling, evaluation, propagation and initialization method as delegates, and decision had to be made as to what should they take as an input, and what should the output be.

Since we stick only to those resampling schemes that do not change the number of particles, resampling delegate `ResampMethod` is simply taking a `List<IStates>` collection and is returning the resampled set of the same particles. Unlike resampling, propagation and evaluation are operations on a single `<IState>` particle, and an alternative to using delegates was to impose a `Propagate()` and `Evaluate()` method in the `IState` interface. Half-way trough the project we decided to use such `Propagate()` method instead of the propagation delegate, so a particle class should make sure that each particle knows how to propagate. For evaluation we persisted in using a delegate `EvalMethod`, which takes a `IState` particle and an image and returns a particle weight.

A tricky part of implementation was to decide how to initialize our particle filters. We chose to define an initialization delegate `InitMethod` that was supposed to create the collection of particles. We tried to make it as general as possible by letting it take an variable number of arguments using `params` modifier. Still, it often proved hard to use the initialization delegate, partly because many possible initialization schemes are imaginable, but also because different particles may require very different input for initialization. Quite a few times we went back to add input parameters to initialization delegate to make it deal with the specific eye-tracking methods we implemented. It would be desirable to deal with the initialization at the lower level, possibly in particle state class where the ‘true nature´ of particles is known.

Passive part

A particle filter that just manipulates a set of particles without showing some results would not be very useful. We have therefore provided a number of methods for visualizing the tracking result. To calculate the weighted mean for a set of particles we included `Add` and `Multiply` methods in our `IState` interface. We also imposed a number of Draw methods since different drawing styles may be preferable for different particles. We also define a `ToString` method in `IState`, so that the tracking data can be saved in a text file.

3.2 Three Tracking Models

Besides from implementing a generic particle filter with the underlaying classes, we have also implemented a number of tracking models and methods which can be combined in different ways. We can, however, divide our tracking schemes into three basic groups depending on the particle model and evaluation method. We will describe those three groups here.

All of our tracking models can share the resampling method, and we implemented two resampling methods. Those are described in the section 3.4.

Point tracking, difference evaluation

Tracking based on difference evaluation is the simplest tracking model, and we used it to test our implementation of the generic tracking filter. Video input for the difference evaluation is a sequence of background subtracted images (difference images). Those images have most of the pixel values close to zero, and bright patches at the positions of the moving object.

In this filtering method, the weights are assigned to particles directly from the pixel brightness, since we want to track the bright pixels. Propagation method is simply addition of Gaussian noise to the particle position. We have provided two initialization methods: random scatter over the whole video frame, and a Gaussian distribution around the user-defined point.

One eye tracking, histogram comparison evaluation

In this model we still track one point, but the evaluation method is based on histogram comparison, which is adapted for the purpose of eye-tracking. User input is needed to provide the inner and outer region of the eye in one of the frames from the sequence to obtain the histogram of the inner and the histogram of the outer region. In subsequent frames, corresponding histograms will be obtained for each of the particles and the weight is assigned according to histogram similarity (section 3.4).

We have implemented two different methods for obtaining the histograms from a user-defined region. In the first method the histograms are obtained by sampling every pixel in inner and outer region. In the other method we sample only a fixed number of random pixels in the two regions and we weight the pixels when making the histograms. Weighting is done according to the hat function (for inner region) and inverse hat function (for outer region), to de-emphasizes the influence of the pixels close to the border of inner and outer region. Hat function weights are linear in each direction and range from 0 at the image edges to 1 in the center.

In this tracking scheme particles are initialized around the position chosen by the user and propagation is still just adding Gaussan noise.

Two eyes tracking, histogram comparison evaluation

For this tracking model we implemented a state which tracks two eyes. The state is parameterized by the mean position between the eyes, the angle of the line connecting the eyes and the distance between the eyes. Evaluation is similar to the previous model, but this time it is four histograms that are extracted and compared with the histograms from the first frame. As in previous model, we can make either full histograms of the regions, or sampled and weighted histogram.

Propagation of two-eyes particles is based on adding Gaussian noise to the mean position, angle and distance between the eyes. Initialization is around the original position of the eyes chosen by the user.

3.3 Graphical User Interface

We provided a graphical user interface (GUI), where user can interact with the filter. Many of the functions in the GUI are strongly influenced by the specific filters we implemented. It is in the GUI that the communication between the user, the video stream and the particle filter happens. This is most obvious during initialization, where video has to be loaded and the user has to select the filter settings and click on the video frame to initialize the filter.

The big challenge of our implementation was to find the balance between the general particle filter on one side, and the model-specific GUI on the other side. Keeping the GUI intuitive and simple, while still allowing the different tracking schemes proved challenging. An alternative solution would be to make a different GUI for each tracking model.

Among the other functions, our GUI allows:

- Loading video files, and basic video control (play and pause),
- Choosing from a set of available tracking models, evaluation methods and propagation parameters.
- Initializing the particle set by clicking and drawing in the video frame.
- Visualizing the tracking output as the path plot in the video frame.

A good initialization is crucial for the successful tracking. Since using a particle filter may require some knowledge of the relevant theory, we provide a few pre-initialized tracking demos, which are available from the GUI. The user should bear in mind that even though the demos are pre-initialized, particle tracking is a random process and a rather different results can be obtained in different runs of the same demo.

3.4 Basic Algorithms

When implementing different methods used by our particle filter, we relied on a number of algorithms, which we will briefly present here.

Resampling algorithms

If we have a set of N particles, with the weights w_i, where $i = 1, \dots, N$ we can resample them by drawing N particles randomly from the set, in such a way that particle's weight gives the probability that the particle will be drown. This can be implemented by normalizing particle weights so that they sum to 1, making a cumulative density function (CDF), and then N times drawing a random number $u \sim \mathrm{U}[0,1]$ from the uniform distribution on the interval $[0,1]$, checking where in the CDF the number u landed, and taking the corresponding particle to the new set. This mehtod is illustrated in the top of the figure 1.

Depending on the algorithm used for finding where in the (sorted) CDF the random number u lands, this resampling method takes $O(N^2)$ (as in our implementation) or $O(N \log N)$ time.

Alternative sampling algorithm that we implemented, systematic resampling, takes $O(N)$ time. It uses an algorithm based on order statistics and is easy to implement [1]. After building the CDF, a starting point $u \sim \mathrm{U}[0, 1/N]$ is drawn and localized in CDF. The remaining points are selected uniformly, by adding $1/N$ to u. Now we need to move only in one direction to localize the point in CDF. This method is illustrated in the bottom of figure 1.

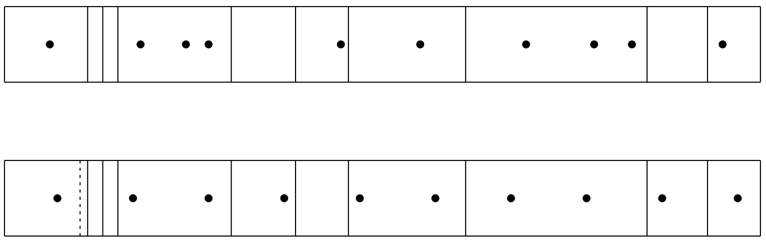

Figure 1: Illustrations of two sampling algorithms we implemented. The rectangular region represents cumulative density function (CDF) for the weights of ten particles Each slot corresponds to one particle. *Top:* Drawing 10 random numbers from $\mathbb{U}[0, 1]$, represented by the dots in a illustration. The chances are more dots will end in the slots corresponding to the particles with the higher weight, so the particles with large weights will survive, and those with small weights will not be selected. *Bottom:* Systematically resampling by drawing only the *first* sample from $\mathbb{U}[0, 1/10]$ (the area bordered with dotted line), and choosing other particles uniformly from the starting point. This approach ensures survival of the fittest (those having weights higher than 1/N) and maintains the dose of randomness.

Box-Muller transformation

All of our propagation methods are based exclusively on adding Gaussian noise to the parameters of the states, usually just to the position of the particle. The random number generator from the class `Random` provides uniform distribution of random numbers. We used the Box-Muller transformation to transform the uniform distribution to Gaussian (normal) distribution (Sestoft 2004, [4]).

Box-Muller transformation takes two random numbers $u_1 \sim \mathrm{U}[0, 1/N]$ and $u_2 \sim \mathrm{U}[0, 1/N]$, and transforms them to numbers

$$z_1 = \sqrt{-2\ln(x_1)}\cos(2\pi x_2)$$

$$z_2 = \sqrt{-2\ln(x_1)}\sin(2\pi x_2)$$

where ln is the natural logarithm. The numbers $z_1 \sim \mathrm{N}[0, 1]$ and $z_2 \sim \mathrm{N}[0, 1]$ are normally distributed with zero mean and variance 1.

Bhattacharyya coefficient

To compare two histograms we use the Bhattacharyya coefficient, which is a measure of the similarity between two discrete distributions. Battachharyya coefficient is used extensively in mean-shift object tracking, which employs mean shift iterations to maximize the similarity between two distributions (Zhu et al. 2002, [6]).

For two (normalized) histograms p and q, both with n bins, Bhattacharyya coefficient is calculated as

$$BC(p,q) = \sum_{i=1}^{n} \sqrt{p(n)q(n)}$$

which will obviously be one for two identical histograms, and close to zero for very two very different histograms.

We have been dealing with the color histograms which have a layer of bins for each color. We have simply concatenated the three histogram layers and calculated Bhattacharyya coefficient for the resulting histogram. Alternatively, we could have calculated Bhattacharyya coefficient for each color independently and then multiply the results.

When comparing inner region to another inner region, and one outer region with another, we assumed that the inner and outer regions are independent, so we multiplied the two obtained Bhattacharyya coefficients.

3.5 Possible Extensions

As it is often the case, when implementing our eye trackers we often saw the possibility to improve or extend the existing implementation, but we had could not fit it in the time frame of four weeks. We will anyway mention a few ideas here.

Angle and distance constraints

When tracking eyes it is reasonable to assume that the line connecting the two eyes is mostly horizontal, and that the distance between two eyes is not deviating a lot from the originally chosen value. It would therefore be desirable to put some constraint to those two parameters of the two-eyes state. As it is now, the only thing we can do is to make angle and eyes-distance noises very small, but then we have an inflexible filter that is not able to recover from possible loss of tracking object.

An easy and not very elegant solution to angle and distance constraint would be to change the propagation method, so that the angle is always sampled from a zero-mean Gaussian distribution, and distance noise should have mean at the initial eyes distance. Doing that, we would totaly ignore the previous angle and distance parameters of two-eyes state. A more elegant solution would therefore be to find a method that combines the previous parameters and the constraint.

Ellipse fitting

We have discussed the possibility of implementing an eye tracker based on fitting an ellipse. It could be done by having a state which carries a parametrization of ellipse for each eye. It would probably be enough to use the same ellipse (but shifted) for both eyes. An ellipse is parameterized by its centra point, the two radii and orientation. Orientation could initially be ignored. To initialize the filter user would need to fit the ellipse to the eye, so that it coincides (as good as possible) with the border between the iris and the sclera (white area) of the eye.

Evaluation would be based on sampling the pixel values on a number of normals to the ellipse, and finding the ‘mean’ normal. From the ellipse chosen by the user we would obtain the typical edge between the iris and sclera, which would capture the change between the dark and brighter pixel values. In subsequent frames the ellipse states would be evaluated according to the similarity of the the mean normal sampled across their ellipses, and the initial normal. This scheme could be further improved by weighting the pixels along the normal, similarly like we do in histogram sampling.

4 Implementation Description

In this section we will briefly describe the most important classes, which we use in our implementation.

DWImageLibrary

All image handling (importing and playing of AVI files, capturing and manipulating frames) is done via the provided `DWImageLibrary.dll`, which is based on OpenCV functions.

We use the `DWAviReader` class for loading AVI files, and the `Start()` and `Stop()` methods for controls in combination with the C# `Timer` object set to an interval of the appropriate frame rate. For retrieving images from the video we use the `DWColorImage` class. Below is an example of loading an avi file, grabbing a frame and assigning it to a form element in the GUI:

```
PictureBox videoPanel;
Timer videoTimer;
DWColorImage image;

mov = new DWAviReader();
mov.Start(openFileDialog1.FileName);
videoTimer.Interval = (int)mov.FPS;
mov.GetImage(image);
videoPanel.Image = image.GetBitmap();
```

For the evaluation step we use `GetPixel()` method, and we visualize the particles with the drawing functions of the class: `SetPixel()` and `DrawEllipse()`.

4.1 Modeling particles

IState

We designed a state interface (`IState`), which describe the minimal state, and holds a method for propagating it, calculating the weighted average from a set of states and the methods for drawing the state on the screen.

Our `IState` interface defines the following methods and properties:

```
using DWImageLibrary;

public interface IState
{
    Position XYPosition { get; set; }
    double X { get; set; }
    double Y { get; set; }
```

```
        double Weight { get; set; }
        IState Copy();
        IState Add(IState s);
        IState Multiply(double d);
        void Propagate();
        void DrawState(DWColorImage image);
        void DrawWeight(DWColorImage image, double scale);
        void DrawMean(DWColorImage image);
        void ConnectTo(IState stat, DWColorImage image);
        string ToString();
    }
```

We have two classes implementing `IState`: the one-point state `SimpleState`, and the two-points state `EyesState`.

Position

We need a representation of our states' positions. The `System.Drawing` library has a `Point` class, but since we will work with Gaussian propagation noise, we need our coordinates to be of type `double`, so `Point`'s integer values will not do. We designed a `Position` struct type which meet this requirement, and furthermore is equipped with a method for computing the Euclidian distance to another `Position`, and a method for determining wether it is with in the bounds of an image and other such tools.

Sigma

Since the `IState` interface declares a `Propagate()` method, we designed a `Sigma` class, which is generating Gaussian noise, and we included `Sigma` objects in both of the states we implemented. The objects from the `Sigma` class are also used for noisy initialization of the particles.

As we need so many operations with adding Gaussian noise to the position of the states, we overloaded the + operator so when `Sigma` object is added to the `Position`, a new (noisily moved) position is returned.

4.2 Modeling Particle Filter

Particle Filter

We use a generic `List<T>` collection to hold our states in one field of the `ParticleFilter` class. The `ParticleFilter` class is the back bone structure for putting together a filtering system. Initialization, resampling and evaluation delegates are assigned to delegate fields, and initialization fills the `List<IState>` with values. Visualization methods are also appropriately placed in this class.

We write a summery class diagram here, as we think this my be important for further reading. For details, please see the code in the Appendix.

```
public delegate List<IState>  InitMethod  (int noStat, DWColorImage image,
                                            Sigma[] sigma, params Position[] initPos);
public delegate double        EvalMethod  (IState state, DWColorImage image);
public delegate List<IState>  ResampMethod(List<IState> particleSet);

public class ParticleFilter
{
    private List<IState> particleSet;
    private InitMethod im;
    private EvalMethod em;
    private ResampMethod rm;

    public ParticleFilter() { }
    public ParticleFilter(int number)
    public List<IState> PartSet { get; set; }
    public EvalMethod Em { get; set; }
    public ResampMethod Rm { get; set; }
    public InitMethod Im { get; set; }

    public void Evaluate(DWColorImage image)
    public void Resample()
    public void Propagate()
    public void Normalize()
    public void DrawStates(DWColorImage image)
    public void DrawWeights(DWColorImage image, double scale)
    public IState WeightedMean()
}
```

`Evaluate()`, `Resample()`, `Propagate()` invoke the respectable delegates and methods on `this.particleSet`.

`Normalize()` makes all weights of the particles in the set sum to 1. It is called once per frame from the evaluation method to ensure that our weights are converted into a probability measure, and is also useful for a consistent visualization of weights. Furthermore, the `Normalize()` ensures that the sum of the weights is never 0, in which case the population would 'die'.

`DrawStates(...)` invokes the visualization methods of each particle in the set as a kind of traversal of the commands to each state instance. `DrawWeights(...)` works like the `DrawStates(...)` but here we also pass the image and a scaling factor to make the normalized weights on average 1 pixel big when drawn on the image.

`WeightedMean()` returns the mean state/particle of the set.

Initialization delegate

We must design a delegate for the initialization which is flexible enough to adapt to different models. We cannot know the number of parameters describing the state model, so we use the parameter array for the initialization delegate.

```
List<IState> InitMethod(int n, DWColorImage im, Sigma[] s, params T[] initPos)
```

The parameters needed are

- `int n` the number of particles/states.
- `DWColorImage im` is an image from the DWImageLibrary. At least the size of the image could be handy for defining the 'field' for the particles, but it might also be needed to sample from it.
- `Sigma[] s` is an array of noise models. More than one are needed as there may be a difference between initialization and propagation noise. Two-eyes state needs even more noise for initialization.
- `params T[] initPos` are the position parameters needed for the model — if any.

For our filtering models we implemented 6 different initialization methods, two for each of the models.

Evaluation and Resampling delegate

As above, we use a delegate for defining the evaluation method:

```
delegate double EvalMethod(IState state, DWColorImage image)
delegate List<IState> ResampMethod(List<IState> particleSet)
```

We implemented two resampling methods, one based on true sampling and one that samples systematically. As for evaluation methods, we implemented five: one based on direct evaluation from the pixel value, and four histogram-based evaluations.

Please refer to section 3.2 and 3.4 for a more detailed description of the methods these delegates perform. For all the other implementation details please refer to the Appendix of the report.

5 User guide

File. Open a video AVI file from the file menu (uncompressed or Indeo 5 encoded avis are supported as a minimum). By opening a avi file, the ParticleFilter is reset to default values.

Demos. Select a ready made filter setup with suitable example videos.

Filter Setup. Choose initialization and evaluation methods from the drop down menus. Only certain methods go together — in general the complexity increase further down the menus, and some automatically pick appropriate companion methods from the other menu when selected. This has not been fully tested – use with care.

When choosing Histogram based methods, follow directions in the status bar at the bottom of the GUI. All selections on the image area must be made by dragging with the mouse in the top-left to bottom-right direction.

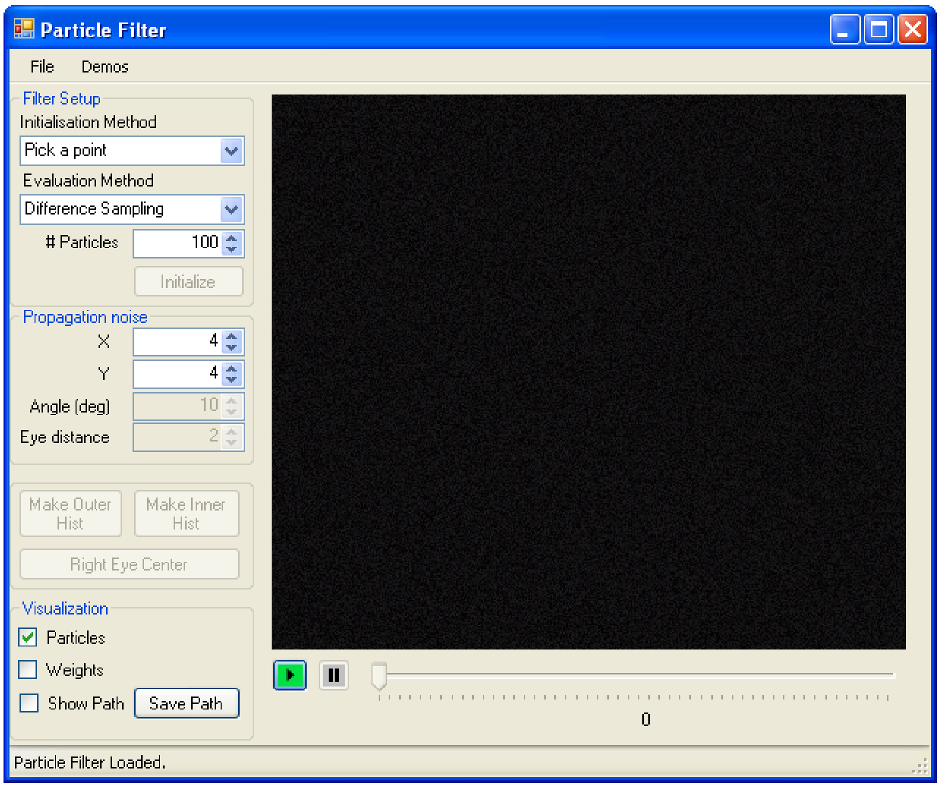

Figure 2: GUI for our particle filter immediately after loading a video and initialization of the default filter.

For the two-eye model, both eyes will be sampled according to the inner and outer region, so center the regions on the eye, and be careful with the size of the region.

Propagation Noise. Appropriate noise parameters become active upon changing the methods in Filter Setup. Changes are applied upon re-initialization, file loading or hitting play.

A filter can be re-initialized by choosing a new (or the same) initialization method, or - when using point based methods, by pausing/stopping the video and clicking on the image. This does now work for histogram based methods.

Visualization. Checking the boxes enables further visualization. As a minimum, the mean position is always drawn. At the end of every play session, the tracked path can be drawn to the screen, and saved to a file as coordinates in ASCII format.

6 Results and Testing

6.1 Results

We have, of course, used our particle filters on many AVI files, and many, many times. The particle filter *is* tracking, and we are generally satisfied with its performance. We will present some of the tracking results in this section.

The tracking success depends on a number of factors. Firstly, there are the properties of the video frame, how quick and how noisy is the object moving, how are the lightning conditions, were there any occlusions. . . ? For successful tracking AVI file has to be relatively 'nice'.

Propagation noise is the second important factor. It takes some time to get used to adjusting the noise parameters to get a good tracking. Usually we found that increasing the number of particles and increasing propagation noises generally improves the result, but the computation time can become an issue.

Lastly we stress again that particle filtering is a random process. We had some videos where we managed to track 4 out of 5 attempts, with identical initialization. And there were videos where we succeeded in 1 out of 5 attempts.

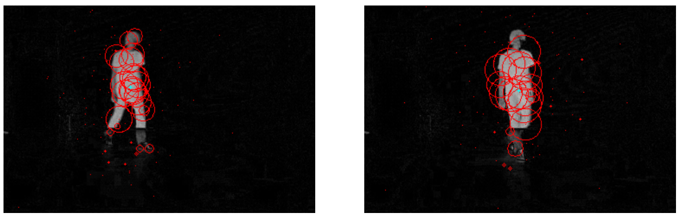

Figure 3: Tracking based on the direct evaluation of the background subtracted images. Particle weights are used for radius of the circles when plotting. The connection between the particle weight and the pixel value is obvious.

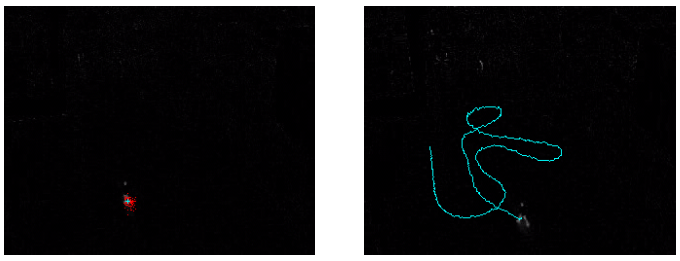

Figure 4: Direct evaluation used for tracking a person in a 632 frames long still-camera video. *Left:* A frame from the sequence with the particle cloud overlaid. *Right:* Tracking path.

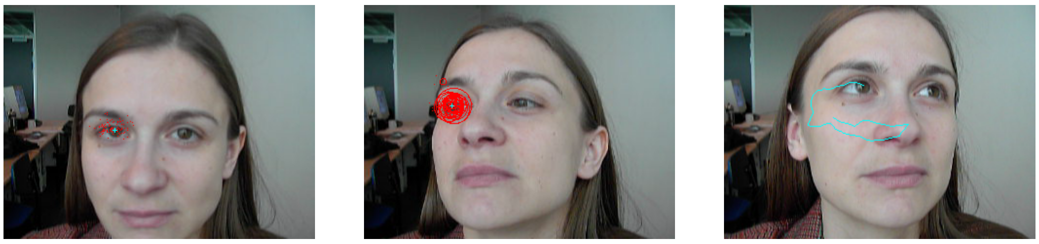

Figure 5: One-eye tracking based on histogram comparison. *Left:* A frame from the sequence with the particle cloud. *Middle:* A frame with the particles plotted as circles, where the radius corresponds to the weight. *Right:* Tracking path.

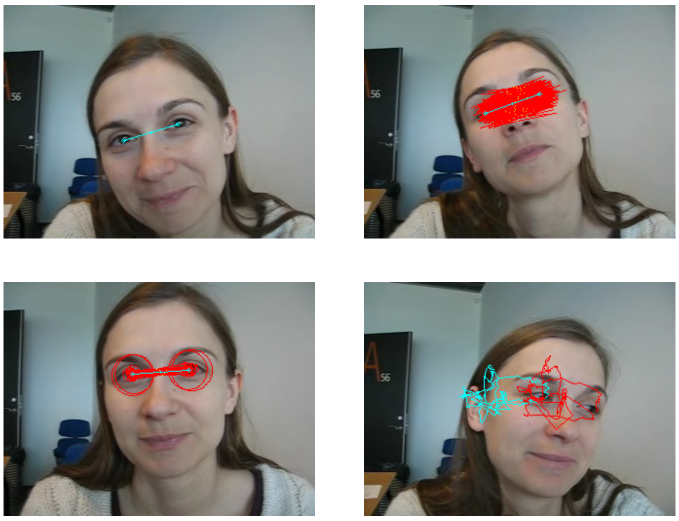

Figure 6: Two-eyes tracking. *Top left:* Only the mean shape of the particle set is plotted. *Top right:* Two-eyes states plotted as the lines connecting left and right eye. *Bottom left:* States plotted together with the weights. *Bottom right:* Tracking path.

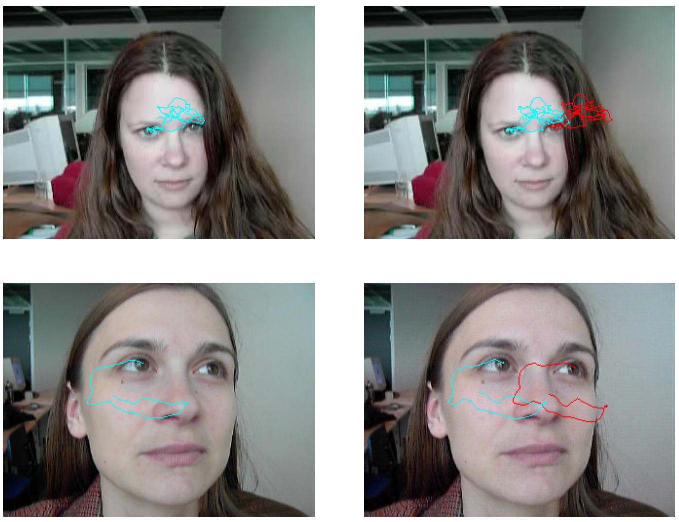
Figure 7: Comparing the tracking paths of one-eye and two-eyes tracking methods.

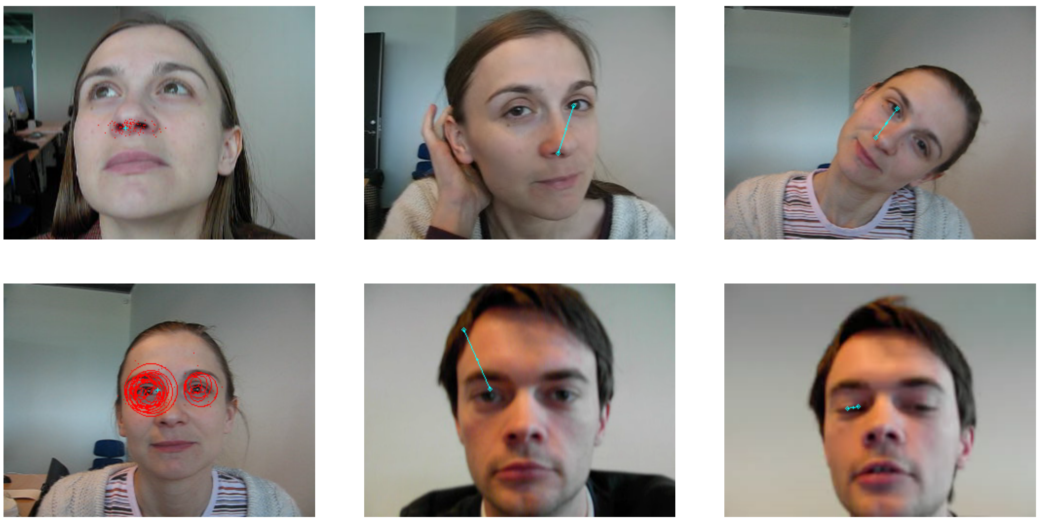
Figure 8: When things go wrong. *Top line:*A rather common situation: Nose tracker and eye-and-nose trackers. *Bottom left:* One-eye tracker tracking the right eye and particles jump to the left eye. *Bottom middle:* Eye-hair tracker. *Bottom right:* Two-eyes tracker collapses into one eye.

Chronologically, the first tracking method we developed was an evaluation based on pixel values. This can be used to track moving objects in a sequence of background subtracted images. On the figure 3 we bring a couple of images from such a filtering because they nicely illustrate the weights. We see particles spread over the image, but only those particles that coincide with the object get assigned higher weights, only those particles with higher weight contribute to weighted mean, and only those particles will be used again after resampling.

Figure 4 illustrates that even this simple tracking scheme can be useful — we have tracked a walking person in a video sequence taken by the still camera. We did not implement background subtraction, but have used a background subtracted sequence.

One-eye tracking is illustrated in figure 5. It proved very successful on all of the 'nice' videos we made, especially in the combination with the full histogram evaluation. In the 'difficult' videos it has sometimes happened that the particles get stranded on another feature having similar colors to the initial eye region – for example on the hair or the door. [1]

Figure 6 is a two-eyes tracking. We were again very pleased with filters performances and it was again a bit better in a combination with the full histogram evaluation. This filter was sometimes able to recover after the loss of track, but that occurred very sporadic. Some losses, like for example collapsing both points in one eye, proved to be non-recoverable.

In figure 7 we compare the paths obtained from the one-eye tracking and two-eye tracking. It is obvious that the two tracking methods return very similar paths.

Finally, figure 8 is a collection of some of the things that can go wrong. We saw quite a lot of images like those, and we felt that they also deserve a place in the report.

6.2 Testing

We have tested our eye-trackers on two videos. One was 267 frames long, easy-to-track movie of a person slowly moving the head. The other more difficult-to-track movie shows a person making abrupt head movements. The two methods we tested were one-eye tracker based on the full histogram, and the two-eyes tracker based on the sampled histogram. Analysis of the testing results was done using Matlab.

In both of the two movies we manually annotated the center of the eyes in each of the movie frames to obtain the annotated eyes path. We run a series of at least 5 trackings for each of the tests. We then calculated the mean tracked path and the

[1] We apologize for not providing a greater variation in the color of the eyes of the people in videos — there were not so many students willing to pose in the four-week period.

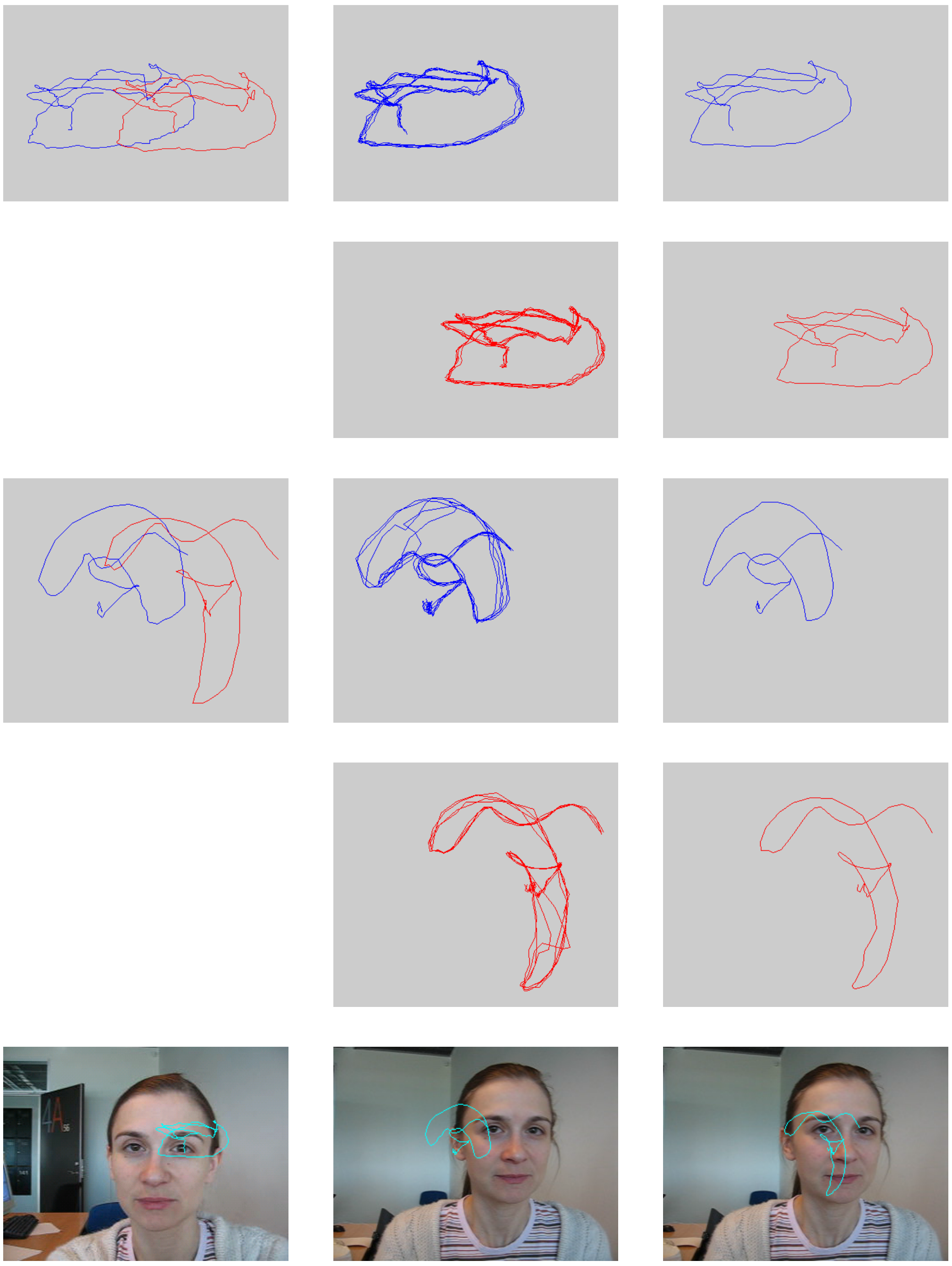

Figure 9: Testing the one-eye tracker, with the evaluation method based on full histogram comparison. *Left column:* Manually annotated paths for the left and right eye. *Middle column:* Tracked paths of all the testing trials for both the left and the right eye. *Right column:* The mean tracked path for each test. *Top rows:* The long and easy-to-track movie. *Middle rows:* Short and difficult-to-track movie. *Bottom row:* Tracking paths plotted by our implementation: Easy video (left eye), difficult video (left eye), and difficult video (right eye).

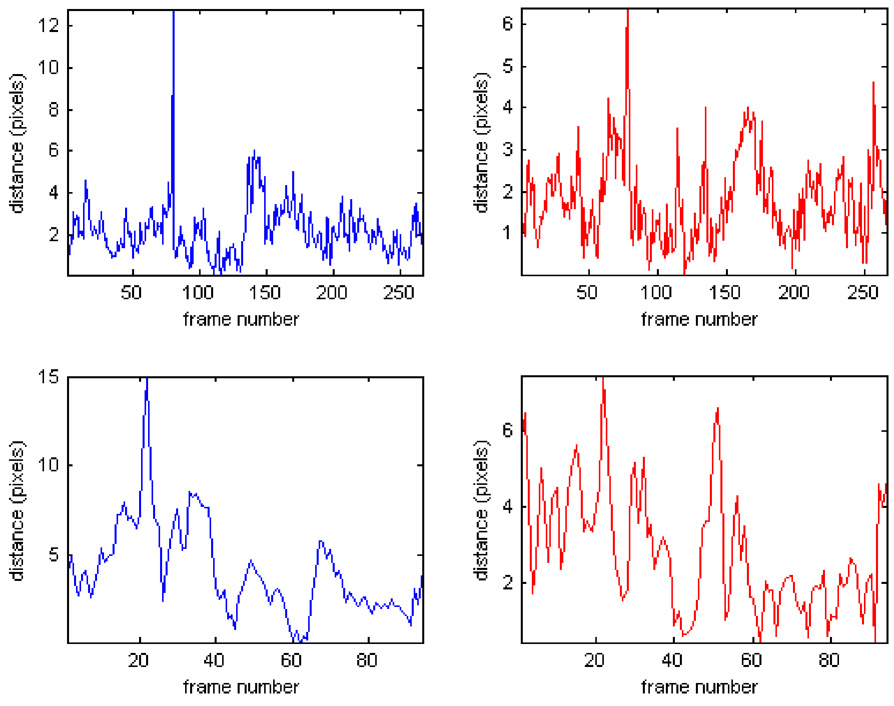

Figure 10: The tracking error across the frames for the one-eye tracking, expressed as the distance from the manually annotated path. *Top:* Easy movie, average tracking error 2 pixels. *Bottom:* Difficult movie, average tracking error 6 pixels (4 for left eye and 8 for right eye). *Left:* Left eye. *Right:* Right eye.

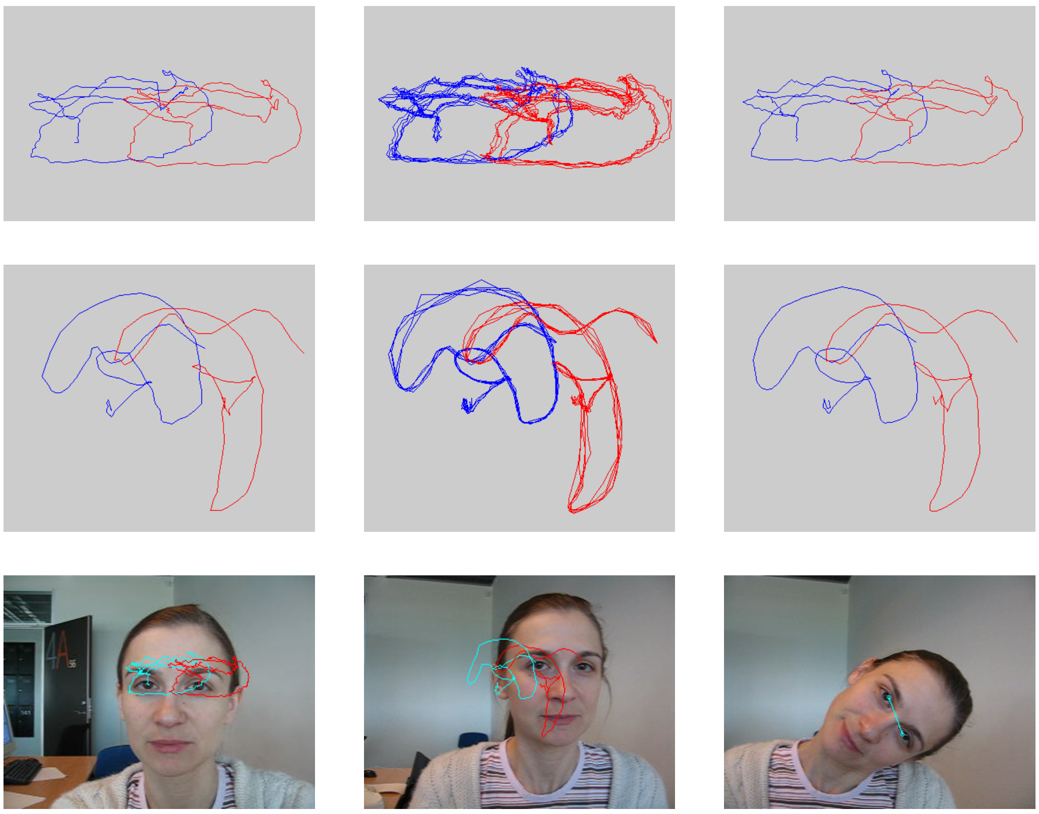

Figure 11: Testing the two-eyes tracker, with the evaluation method based on the full histograms. *Left column:* Manually annotated path. *Middle column:* The tracking paths of all test trials. *Right column:* The mean tracking path. *Top row:* Easy movie. *Middle row:* Difficult movie. *Bottom row:* Figures plotted by our implementation: Easy tracking path, difficult tracking path, and one frame from the difficult movie.

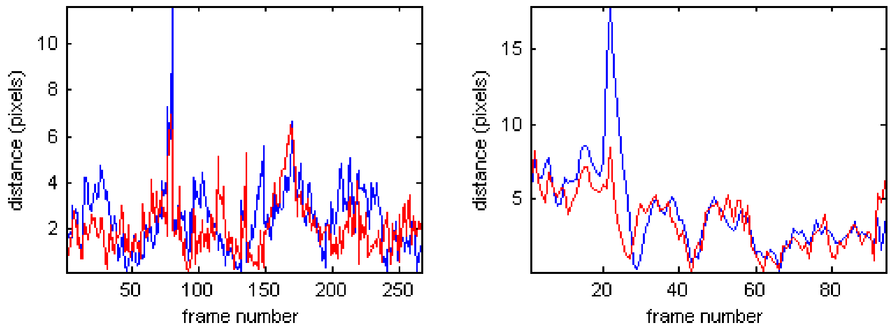

Figure 12: The tracking error across the frames for the two-eyes tracking based on the full histogram. *Left:* Easy movie, average tracking error 2 pixels. *Right:* Difficult movie, average tracking error 4 pixels.

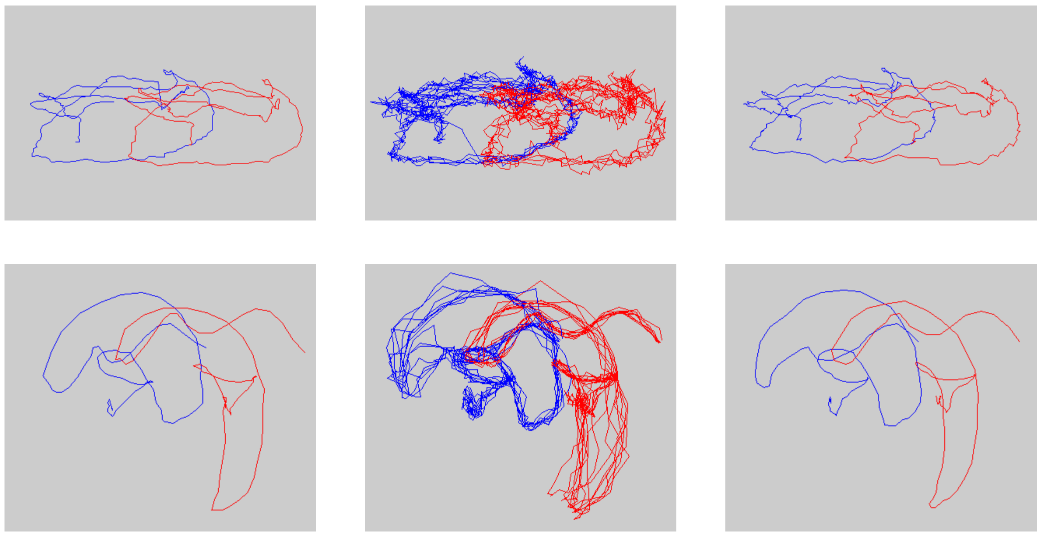

Figure 13: Testing the two-eyes tracker, with the evaluation method based on the sampled and weighted histograms. *Left column:* Manually annotated path. *Middle column:* The tracking paths of all test trials. *Right column:* The mean tracking path. *Top row:* Easy movie. *Bottom row:* Difficult movie.

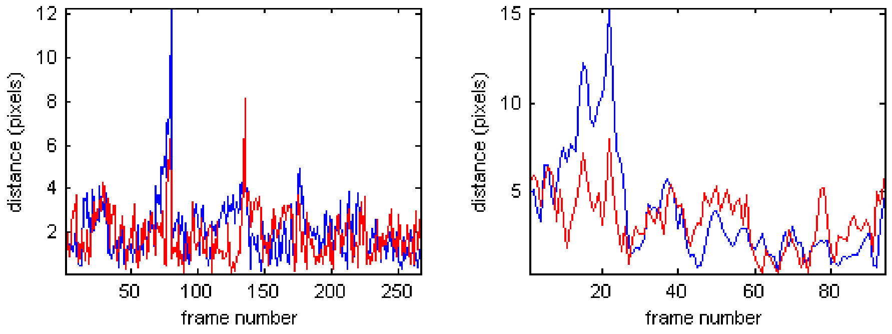

Figure 14: The tracking error across the frames for the two-eyes tracking based on sampled and weighted histogram. *Left:* Easy movie, average tracking error 2 pixels. *Right:* Difficult movie, average tracking error 4 pixels.

Euclidean distance between the annotated path and mean tracked path across the frames of the movie. It is the distance from the annotated path that we use as the measure of the error, but one has to take in consideration than annotation itself is not error-free.

For the easy movie we used 100 particles and the propagation noise with standard deviation of 4 pixels in both *x* and *y* direction (angle noise 10 degrees and distance between eyes 2 pixels) and for the difficult movie we used 500 particles with propagation noise set to 5 pixels standard deviation (15 degrees and 3 pixels).

In figure 9 we can compare the annotated path and the tracking paths obtained from our one-eye tracker. The general appearance of the tracking and annotated paths is very similar for the first movie. When tracking the eyes in the difficult movie we had a number of tracking losses, and had actually excluded one of the right-eye tracks from the results, because the loss occurred so early in the movie that the track was not useful.

Distances between the annotated and mean tracked paths are shown in figure 10. The tracker was rather successful in tracking the eyes. Average distance for the easy movie was only 2 pixels and for the difficult movie it was 6 pixels (4 for left and 8 for right eye).

The performance of the two-eyes tracker based on full histogram comparison is illustrated in next two figures. Figure 11 shows the annotated and tracking paths, and figure 12 shows distances to the annotated path.The average distance for easy movie was again 2 pixels, and the distance for the difficult movie was 4 pixels.

Figures 13 and 14 show corresponding results obtained from two-eyes tracker based on sampled and weighted histograms. We can see that the two-eyes tracker is more shaky, both in the easy and the difficult movie. Still, in the difficult movie it did a bit better then the one-eye tracker. Like with the one-eye tracker, we removed one track from the evaluation, since the loss was so early that the data did not make sense. Distances to the annotated path are in the figure14. The average distance for the easy movie is again 2 pixels, and for the difficult movie it is 4 pixel.

Our intention was also to provide speed tests for the two resampling method we implemented, but we never got around actually measuring the speed correctly.

7 Conclusion

We started our particle filtering project with the intention of learning how the C# programming language can be used for implementing a Computer Vision algorithm. We feel that we got what we came for. Our implementation is far from perfect, but if we were to start a similar project today, we would have a much better idea of what we could or should do.

All beginnings are slow — so was our beginning with C#. We spent the first week coding and changing and coding and changing and coding and changing again. Not even when we got familiar with C#, was the implementation process totally problem free. We have actually been despairing for a couple of days, not realizing that we obtain the histograms from the images we had already drawn our particles on. It took some time to discover unexpectedly pure colors in the bins of the histogram. We missed the console for outputting our data, and look forward to learning to visualize data when doing new implementations.

We have also learned that good communication is essential when programming in a group. The input of our full histogram method are corners of rectangular regions, first the outer, then the inner. Our sampling histogram method, on the other hand, takes the center of the rectangles and then the size, first inner then outer. This was sometimes a source of confusion, and that kind of things could have been avoided if we discussed and planned our implementation better.

It feels wrong handing in the implementation which brakes if the user clicks a special combination of buttons (we hope you will not find it), but it is a few hours to the deadline and we can not find the time to systematically analyze the methods of our GUI. Instead, we look at our implementation as a prototype, and it actually *works*. Given a nice video, and a correct combination of buttons, it *can* track the eyes across the frames very, very successfully. And we know now how to improve and extend it, so maybe...

References

[1] S. Arulampalam, S. R. Maskell, N. J. Gordon, and T. Clapp. A tutorial on particle filters for on-line nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 2001.

[2] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall, 2002.

[3] M. Isard and A. Blake. Condensation - conditional density propagation for visual tracking. *Int. J. Computer Vision*, 1988.

[4] P. Sestoft and H. I. Hansen. *C# Precisely*. Massachusetts Institute of Technology, 2004.

[5] E. Trucco and A. Verri. *Introductory Techniques for 3-D Computer Vision*. Englewood Cliffs, NJ: Prentice Hall, 1998.

[6] Z. Zhu, Q. Ji, and K. Fujimura. Combining Kalman filtering and mean shift for real time eye tracking under active IR illumination. 2002.