A mean-shift tracker: implementations in C++ and Hume

Iain Wallace[1] Aug/Sep 2005

A Nuffield Foundation undergraduate research bursary working at
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Whilst an undergraduate at the University of York

[1] iaw@macs.hw.ac.uk

Table of Contents

TABLE OF CONTENTS 1

INTRODUCTION 3

HOW DOES IT WORK? 4

What does it do? 4

The Algorithm 5
- The colour-model 5
- Limitations of the algorithm 5

The C++ 6
- The `colourModel` class 6
- The main code 6

The Hume 7
- With lists 7
- With vectors 8
- With boxes 9

METHOD 11

C++ to Hume with lists 11

Lists to vectors 12

Vectors to boxes 13

PERFORMANCE AND METRICS 13

Speed of execution 14
- Test setup 14
- Results 14

Code size 15

SUMMARY 16

BIBLIOGRAPHY 17

APPENDIX A. RUNNING THE CODE 18

C++ 18
- Files 18
- Compiling 18
- Input image sequences 18
- Usage 18
- Editing the Histogram Function 19
- The Utility Programs 19

The Hume 20
- Performance note 20

Compatibility note 20
Files 20
Usage 20

APPENDIX B. CODE 22

meanshift.cpp 22

colourModel.h 28

colourModel.cpp 29

meanshift-lists.hume 32

meanshift-vectors.hume 39

meanshift-boxes.hume 47

Introduction

This technical report covers the work I carried out whilst working on a Nuffield Foundation Undergraduate research bursary at Heriot-Watt University in the Dependable Systems Group. It was carried out in August/September 2005 before my final year (masters) at the University of York.

The project was to implement a prototype motion tracking system in C++ and then compare this version with a Hume [4] implementation.

This report describes the implementation of a mean-shift tracking algorithm (as described by D. Comaniciu, V. Ramesh and P. Meer [1]) in C++, three Hume implementations and comparisons between the four.

The three Hume implementations that were developed are; a list-recursion based version, vector based without recursion and an entirely box-based approach. The methodologies to convert the algorithm between these approaches are also detailed, as is benchmark data illustrating the performance differences between the approaches.

A mean-shift tracker was chosen, as it is a simple tracker to understand, yet shows good results on a wide variety of test data. It is also computationally inexpensive compared to other trackers, and does not require any time-consuming tuning of parameters, as there is no motion or sensor model.

How does it Work?

What does it do?

All of the four versions of the code implement a mean-shift tracker. The input is an image sequence, and an initial position of a target to track, and the output is a track of the target over the image sequence. Below are sample output frames of the C++ version tracking a person walking across a lobby (data from "CAVIAR test case scenarios"[2]):

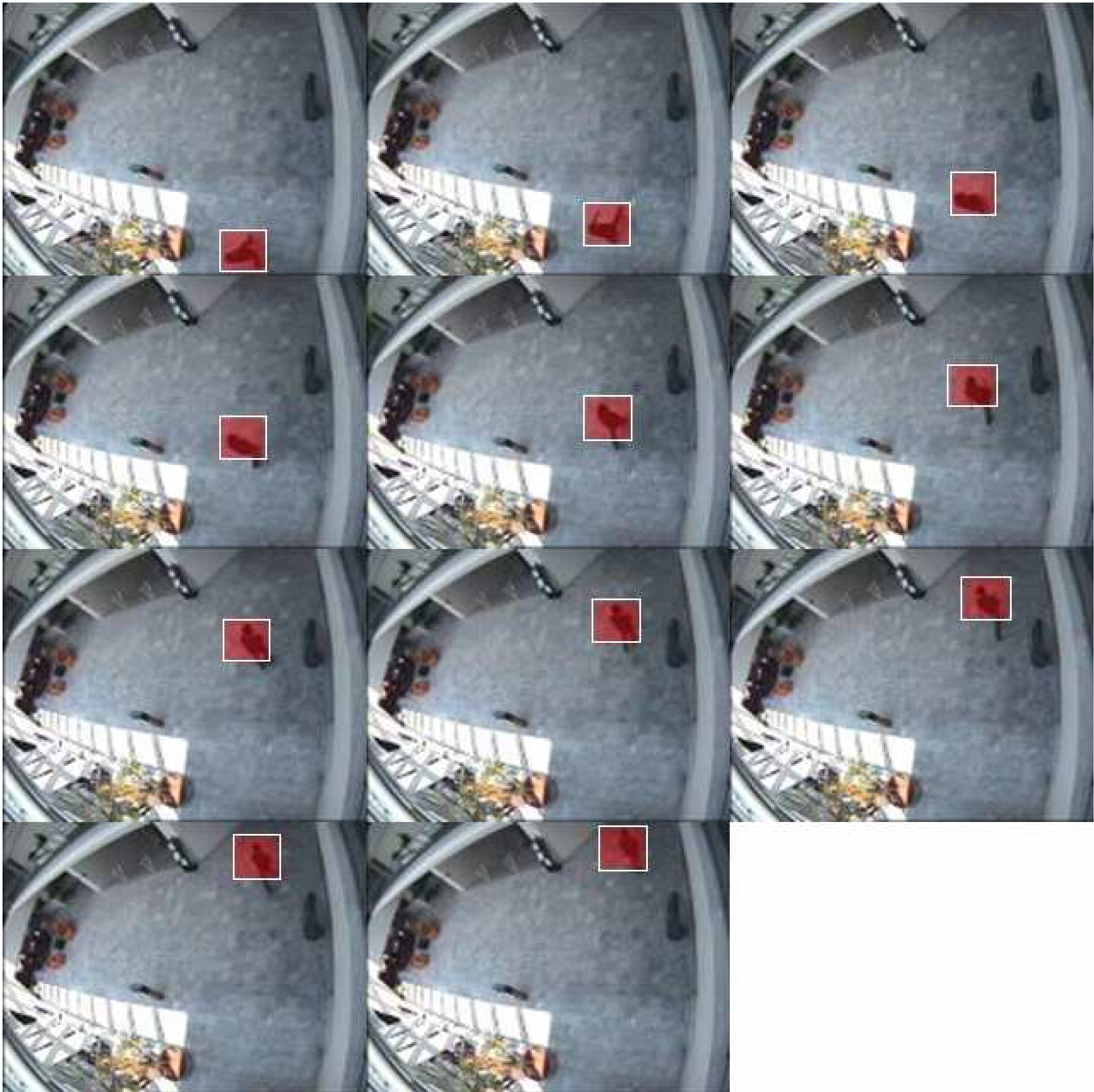

Figure 1: Every tenth frame from tracker output

[2] http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/

The Algorithm

The algorithm used to perform the tracking is the mean-shift algorithm described by D. Comaniciu, V. Ramesh and P. Meer [1].

The system is initialised with two coordinates describing a box containing the target, this is the "target window" below.

The basic steps are as follows:

1. Initialise the target colour-model, $\hat{q}_u$ using equations (2) and (4) from the above paper. These equations reduce to be simply a kernel-weighted histogram over the target area.
2. In the same way, compute $\hat{p}_u$, the colour-model for the target, on the current centre position of the target.
3. Update the weights for each pixel in the target window, using equation (10).
4. Compute the target displacement, using equation (11), and add it to the current centre position.
5. Repeat steps 2-4 over the current frame until convergence, that is the displacement is zero.
6. load the next frame, and repeat from step 2.

In practice, the algorithm is also limited to a certain number of iterations over any given frame to speed it up. Also, the iterations stop in the case of "oscillation" where on one loop the displacement (dx) moves it back to the previous position, and so on. That is, when $dx_t + dx_{t+1} = 0$.

The colour-model

An important consideration and the only real "tuning" that can be done to the tracker is the choice of histogram function used. If a bin is used for all possible colours in a 24bit image, then there are 256*256*256= 16 million bins! As the target window is likely to be thousands of pixels at the most, the model will contain 0 for most of these bins – clearly a waste. Also there are several summing operations performed over the model to normalise it, which would be very costly for 16 million bins. As a result, the feature space must be quantised, and possibly use less than three colour channels. The trade-off to be made is speed versus accuracy of tracking – if the feature space is too sparse then the target will be lost. Similarly if less colour channels are used, they should be chosen so that they best differentiate the object from the background.

Limitations of the algorithm

The tracking algorithm is unable to cope with certain conditions, mostly due to the fact that the search for the target in frame $n+1$ is started at the location for frame n. This means that if the target is fast moving relative to the frame-rate, then the target will not be present in the search area, so the tracker will fail. The other main issue is that the search-window is of constant size, so if the target "zooms" by moving towards or away from the camera, the reference target model will be wrong.

Fast moving targets could be accounted for by incorporating a motion model, e.g. by feeding the output into a Kalman filter [2]. Zooming can be handled by using the CAMSHIFT (continually adjusted mean shift, [3]) method which adjusts the window size.

The C++

The C++ code is simple in its structure, using only the main procedure, a handful of functions and one class for representing the colour models. The C++ code was developed first to have a reference implementation of the algorithm for comparison, and the C++ is also easier to understand.

The `colourModel` class

This class is used for the $\hat{q}_u$ and $\hat{p}_u$ models, and contains methods to update the model, index the model and get the bin number for a particular pixel in the window (used for the delta functions).

The most important method is `updateModel()`, which is called to calculate the colour models (in steps 1 and 2 above). The function it calculates can be described as "each bin, u, in the model is equal to the normalised sum of all kernel values for the pixels falling in that bin" (see eqn. (2) in [1]).

This is implemented by a 2D loop over the window (lines 90-102) which finds the bin for each pixel (using `findBin()`), and uses that as an index into the model array, which then has the kernel value for that pixel added to it. A 2D array the size of the window representing the kernel is passed into the method. As the bin for each pixel is calculated, it is stored in the bins array – this is to avoid re-calculating it when the weights are updated.

The next two loops (lines 104-112) simply then sum all the values, and then divide each value to normalise the model. The model is initialised to zero, so that any bins not found in the window will be correct.

The `findBin()` function is the histogram function used to assign a particular RGB value to a bin in the feature space. This is done by using the 3 values as an index into a 3-dimensional space represented by a 1-dimensional array.

The main code

The kernel, and the derivative kernel, are pre-calculated as the program initialises. This is because they are constant and depend only on the size of the target window. Memory is also allocated for the weights array, as it is passed around using pointers to prevent unnecessary copying.

The first frame is loaded, and the model $\hat{q}_u$ calculated for the initial centre position before the main loop is entered. The main loop loads each frame, and has an inner loop to iterate over each frame calculating the displacement. This inner loop, representing steps 2-4 above, terminates when the displacement equals zero, the loop count reaches 20 or the current displacement plus the previous equals zero (to catch oscillation).

Informally, the weights array is a 2-dimensional array the size of the window, with each value equal to the square root of $\hat{q}_u / \hat{p}_u$ for u equal to the bin of the corresponding image pixel (see eqn. (10) in [1]). As has been mentioned above, many pixels may share the same bin, and the model will contain many bins which are zero in value. Because of this, to calculate the weights efficiently and without

divide-by-zero exceptions, the `updateWeights()` function first loops through the entire model, setting a temporary array, `R`, to 0 where $\hat{p}_u$ equals 0. For non-zero values of $\hat{p}_u$ the division and square-root are calculated. This avoids un-necessary computation, as if the value is non-zero then at least one pixel must exist in the window for that bin, so the calculation will be needed. The next step is to loop over the window (lines 117 to 123) and set each weight value to the value of `R` indexed by the bin of the current pixel. The bin is found using the values pre-calculated at the model update stage.

Informally the displacement is calculated by "the sum of: each pixel's relative position * its weight * the kernel derivative all over the sum of: the weight * the kernel derivative".

This is calculated efficiently (lines 132-143) by looping over the target window and for each pixel calculating the weight*derivative once, then keeping a running total for the sum of this value, and the sum of this value times the relative position. The division is then carried out after the loop.

The Hume

There are three versions of the Hume ([4]) code, each version was developed using the previous one as a base.

With lists

In the first version of the code all calculation is carried out using recursive functions, with lists for the data structures. Boxes are only used for the input and output of frames. The simple box configuration is as follows:

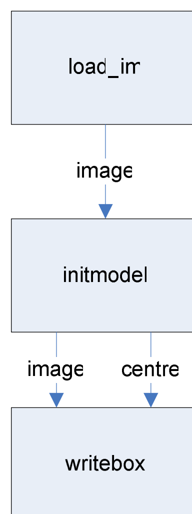

Figure 2: Box configuration for vector and list based implementations

The main box which loops round loading frames is the `initmodel` box. It has two rules, one to match the initial condition based on frame number and a second to process the frames. The first rule calls `updateModel` to create the $\hat{q}_u$ model.

The main function that `updateModel` calls that does the bulk of the work is `doUpdate`, which is the equivalent to `updateModel()` in the C++, though operates differently due to the lists. The three data structures of interest to the function are the image, the model and the kernel. As the image is a vector so random-access is cheap, the functions behaviour to read it is not important. As mentioned above, the model will likely contain more empty entries than filled ones, so even although it's a list, random access is not that expensive, as it will be required for relatively few

values. However, each value from the 2d kernel list must be accessed, so the `doUpdate` and `doYUpdate` functions operate on the kernel recursively to avoid repeated random access into the serial data-structure.

The function to return the bin number of a particular pixel operates as in the C++, however note that the indexing is from 1 in the Hume, as this avoids confusion with the vectors which index from 1 in the current version of Hume.

The other rule in the main box calls `updateCentre` on each frame. This function then calls `findNewCentre` with the displacement value for the current frame, this then loops recursively until the convergence conditions are met.

The `computeDisplacement` function works quite differently to the equivalent C++ function. Most of the work is done using `twodmixmap`. This takes in two equal-sized 2D lists, and a function that takes two arguments, the output is one 2D list that's the corresponding values of the original lists applied to the function. This allows for efficient calculation of the sums over the search window required by equation (11) (Comaniciu et al, [1]). This is used to calculate the products for the weight-lists and the kernel derivative, and also the relative pixel positions – which are pre-calculated in a constant list. The `sums`, `sum` and `csums` (co-ordinate version of `sums`) are simple functions to do the sums required before the division.

The `updateWeights` function, and its helpers, are actually misleading names – nothing is updated, rather a new 2D weight lists is calculated. This is done recursively, using extra parameters for X and Y in the helper functions to correctly terminate. `updateWY` is where the actual calculation takes place, and the actual calculation is the same as the C++. If $\hat{p}_u$ for a particular pixel is zero, then the weight is zero, else it's equal to the square-root of $\hat{q}_u / \hat{p}_u$. Unfortunately this requires random access into the model structures, but there is no way round this, except to iterate through the models, which would be far slower.

With vectors

The vector implementation is very similar to the list-based version, only the recursion has been removed in favour of `vecdef` and there are no lists. As such functions such as `twodmixmap` are re-written to perform the same function on 2D vectors, in the form of `vec2dmixmap`.

As vectors may be accessed randomly with no performance penalty, the `updateModel` function (and its helpers) become much simpler, as they can loop over the window and index the kernel and model as they wish.

The main changes happen in the `updateWeights` function (and the kernel/derivative functions, which follow basically the same structure). As before, there is no update, rather a new weight array is created using `vecdef` calls. Lets are used so that partial application can be used for the function passed to `vecdef`, which should only have one argument. This allows the function which generates the value in the weight vector for a particular X,Y coordinate to have the X,Y coordinate passed to it. This is best illustrated by the simple code to generate the derivative kernel (lines 193-207):

```
evalDeriv = vecdef x_size evalDcolumns;

evalDcolumns X = let evalD Y = computeDKernel X Y
             in
             vecdef y_size evalD;

computeDKernel X Y = kernelDeriv (X-half_x) (Y-half_y);

kernelDeriv x y = if ((ned x y) > 1.0)
              then 0.0
              else 1.0;
```

In this example `x_size`, `y_size`, `half_x` and `half_y` are constant.

With boxes

In the box-based implementation, there are still no lists – only vectors are used, but only the constants are calculated by functions (the same ones used in the vector version), the rest of the calculation is performed by boxes. The wiring of these boxes is best illustrated by the overleaf diagram (loop-back wires in following table):

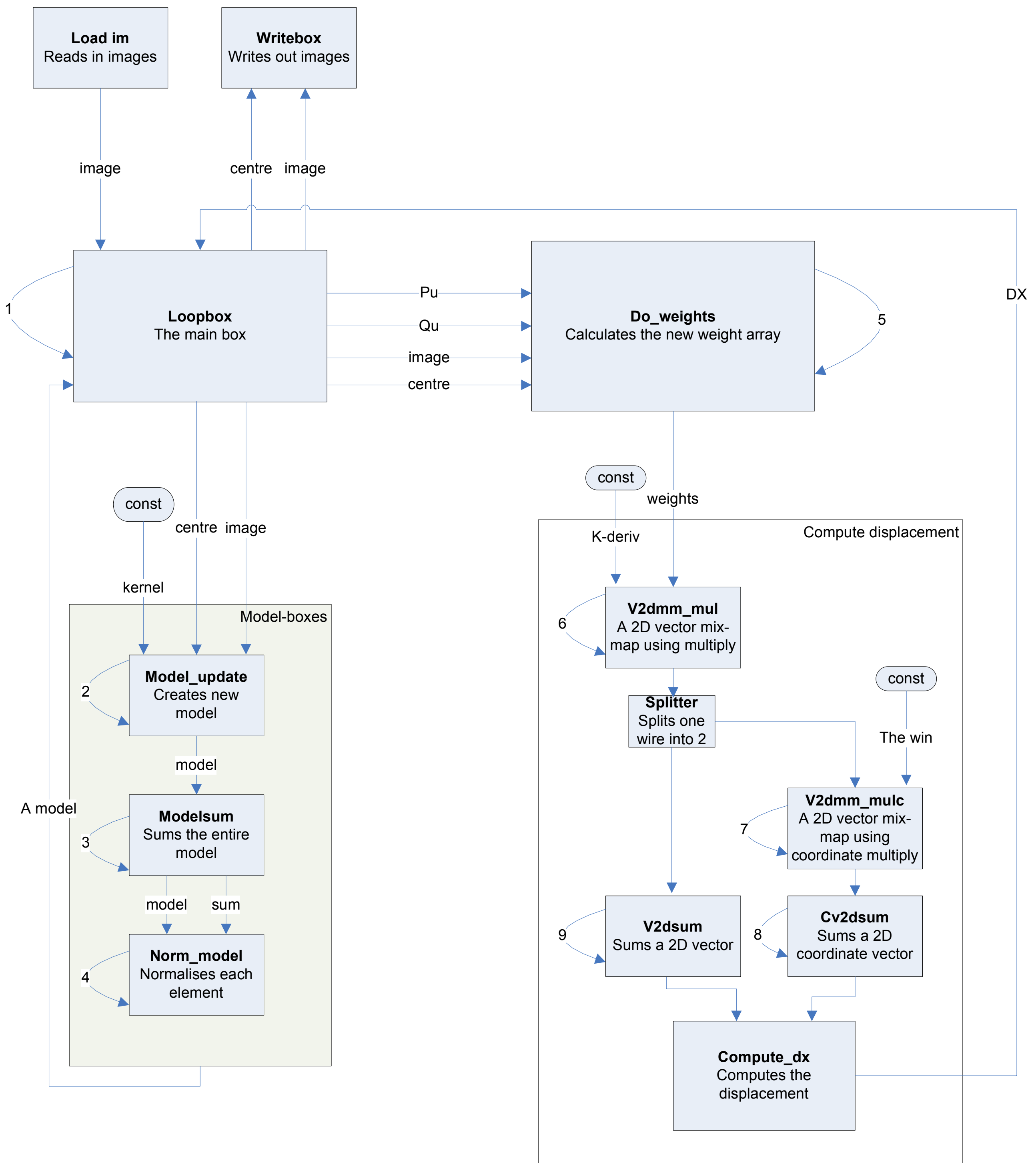

Figure 3: Box wiring of the mean-shift tracker

Loop Number	Data on loop-back wires
1	State, centre, frames_left, Qu, dx, image, loopcount
2	Kernel, image, centre, x, y, model
3	Accumulator, model, size
4	Sum, size, model
5	Weights, Pu, Qu, image, centre, x, y
6	Vector1, vector2, x, y, result
7	Vector1, vector2, x, y, result
8	Vector, accumulator, x, y
9	Vector, accumulator, x, y

The white grouping boxes identify functions that are performed by more than one box. For example, to calculate the target-model there are three boxes. The first creates the model, the second sums it and the third uses the model and the sum to normalise each element.

The section that computes the displacement does so in the same way as the list or vector based implementations, only here the "mixmap" functions are replaced by boxes that perform the same task, likewise the summing boxes.

The `loopbox` is the "main" box that binds all the functions together, and iterates over each frame. The operation of this box is shown below:

```
1     output the first frame to the model boxes to calculate Qu and
      wait
2     Qu input from model boxes, load it into a loop-back wire
3     Output the frame and current centre to model boxes (get Pu),wait
4     Output Pu,Qu,frame and current centre to do_weights and wait for
      the displacement to return
5     If a convergence condition is not met, update the current centre
      and goto 3.
6     Else output the updated centre position to writebox, load a new
      frame and goto 3.
```

The `do_weights` box operates in the same fashion as the vector and list based versions, the only change here is that the looping is done by the box, using loop-back wires to keep state.

Method

This section covers the techniques I used to develop the "final" box-based Hume implementation. Some detail of how the versions differ has already been presented above, this section aims to cover the methods used and the reasons for them.

C++ to Hume with lists

The initial version was in C++. The main reason for this was that I have used C++ extensively before and am familiar with it, and also it is a natural choice of language to implement a tracking system in. It is a natural choice, as it is a fast, mature language, and there are many libraries – e.g. the image library – available to speed development. It was also of interest to see how easy it would be to implement a C++ program in Hume.

The biggest difference to overcome was the change from passing around pointers to data structures which are then used to assign values, to passing around references and being unable to do assignment. With lists this problem is somewhat solved, as they allow insertion – however it is not constant time as it is with arrays in C++.

The other main change is using recursive functions to iterate over lists rather than loops-within-loops. This is a fairly simple transformation to make however, consider pseudo-code to loop over a 100x100 2D data structure setting it to 0:

```
For x = 0 to x = 100
      For y = 0 y = 100
            Array[x][y] = 0
      End for
End for
```

With recursion and lists, this becomes 3 functions:

```
do_loop = loop_x 100

loop_x x = if (x = 0)
            then loop_y x
            else ((loop_y x 100):(loop_x (x-1)))

loop_y x y = if (y = 0)
              then 0
              else (0:(loop_y x (y-1)))
```

The x parameter is passed down to the loop_y function to show how the list could be updated based on both the X and the Y coordinate, as this is what's most commonly required. The "(H:T)" notation is used to indicate the head and the tail of the list, as it is in Hume.

The biggest conceptual change between the C++ and Hume implementations is in the use of "mixmap" functions, to apply a function over a pair of 2D data-structures. This was in part necessary to ensure that lists were accessed sequentially – i.e. as efficiently as possible, and it also makes for clearer code. This shows one of the advantages of a language such as Hume – the ability to pass functions as parameters.

Lists to vectors

The conversion from list based recursion to vectors, with no recursion, is a simple one. However, it relies on having a built in vecdef function, which defines a vector of a given (constant) size using a supplied function. The function supplied should take one argument, which is the position in the vector. This creates a problem with 2D structures, where both the X and the Y coordinate are needed by the inner-loop, however this can be overcome by using let for partial application of functions as shown below:

```
do_loop = vecdef 100 loop_x

loop_x X = let loop_y Y = inside_loop X Y
            in
            vecdef 100 loop_y

inside_loop x y = 0
```

This code does the same as the previous code in the list section, but using vectors. Note that although it just sets each element to 0, the X and Y coordinate are still passed to the function defining each component's value.

There is one fatal flaw in this approach – although it works in the current version of the Hume interpreter, let should not be used to define functions according to the specification. In addition, partial application of any kind is also forbidden.

The use of `vecdef` also shows up a weakness is the type declaration syntax for the current version of Hume. Vector types must be defined with a range, and this range must be a constant but only constant digits are permitted by the syntax – not constant expressions. The implication is that use of `vecdef`, to define vectors whose length is a constant expression, prevents the program being typed. This implementation gets away with it however, as there is currently no type checking.

Other utility functions such as mixmap are easily re-written in this new style to accommodate vectors. The list insertion routines can be replaced with the built-in `update`.

Vectors to boxes

The first step in creating a box based implementation was to write replacement "utility" boxes that performed the functions such as 2D mixmap and summing. A box which loops over a 2D structure and applies a function to the elements is relatively simple, to continue with the example used above, we have:

```
box do_loop
in (input_vector,x,y,vector)
out (x',y',vector',ouput_vector)

match
(w, _*,_*,*) -> (100,  100, w,                  *)  |
(*, -1,-1,*) -> (-1,   -1   *                   *)  |  --accepting state
(*, 0, _*,v) -> (-1,   -1,  *,                  v)  |
(*, x ,0 ,v) -> ((x-1),100, v,                  *)  |
(*, x, y, v) -> (x    ,y-1,(update2d v x y 0), *);
```

The x,y and vector wires are wired up as loop-back wires, and the input comes in on `input_vector`, the output on `output_vector`. The `update2d` utility function is simply the normal vector `update` applied to a 2D vector.

This example has slightly more functionality than the previous examples in that it accepts an input vector and "returns" a vector. This is to illustrate a useful technique for utility boxes. As the box is expected not to run just once, but to be "called" repeatedly, it is important that it returns to a runnable state once processing is complete. This is achieved by the use of "-1" on both the counters to allow it to indicate an accepting state. As the first rule matching on the input is before the accepting state, it will take precedence in the event of further input requiring processing.

It is easy to see how this simple box could be extended to perform any of the previous implementation's functions that require looping – many more examples can be found in the code (see the appendix).

Performance and metrics

The most important measure of performance is whether or not the different implementations give the same output track for given input data. Having tested the four implementations on several test data sets, they all give the same output. The only difference is in the boxes version of the Hume. It differs in that there is no rule to match the "oscillation" convergence condition, as it would add more, unnecessary complication in the form of additional wires. The difference this makes to the output, is that in the case where with each iteration the centre jumps between the same two pixels, it may come to rest on the pixel that the other versions do not land on.

There is also some difference in the intermediate data – for example the kernel data structures. This seems to occur due to the different precision in the Hume implementations compared to the C++ - Hume in theory allows the specification of precision, but this is not yet implemented.

Speed of execution

Another measure of performance is the speed of execution. As I was working with somewhat incomplete versions of the Hume interpreter, the speed of execution relative to the C++ is not really relevant, but included for interest. Or far more relevance is the difference between the Hume versions.

Test setup

All the tests were carried out on "jove", which is a dual 933Mhz Pentium 3 machine, with 1Gb of RAM, running linux kernel version 2.6.8-1.521smp and GCC 3.3.3. The C++ was compiled with the options "-O2 -Wall -ansi -pedantic -ffast-math". Times were measured using the unix "time" command to get user-execution time.

The data used was again from the CAVIAR project, and was footage of some shoppers in a shopping mall[3]. So that the test could be run in a reasonable time-frame (due to the poor performance of the Hume) the images were reduced to 320x240, and the tests run over the first 150 frames. The PPM image format was used for the C++, and the Hume used the concatenated and stripped PPMs produced by the C++ utilities.

Results

Implementation	Total time	Time less I/O
C++	0m21.44s	~0.3s
Hume – Vectors	22m15.972s	~3m53s
Hume - Lists	66m18.510s	~47m56s
Hume - Boxes	75m31.86s	~57m9s

The "Time less I/O" column is the time taken just for processing – without counting file I/O. This number was obtained by using test programs to measure the time taken to load and save one frame, and then scaling this up to gain an I/O time value for the entire sequence.

The results are as expected. The C++ is fastest by a long way, the vector version is the fastest Hume version, followed by the lists then the boxes.

As the lists offer no advantage- they are simply the easiest to code, using simple recursion and they do not allow for random-access it is easy to see why they are slower than the vectors. As vectors allow for constant time update and reading, this provides great advantages – mostly in access to the large model arrays.

[3] http://groups.inf.ed.ac.uk/vision/CAVIAR/CAVIARDATA1/ EnterExitCrossingPaths1cor.mpg

The box version, despite its use of vectors is very slow. This is most likely attributed to the fact that for each "cycle" every box must have its rules evaluated, even although most of the time, most of the boxes do not do anything. This extra evaluation of rules adds considerably to the cost.

Code size

Another interesting comparison is the amount of code needed by the various approaches. The follow table shows a few metrics comparing the relative sizes:

Version	No. Boxes	No. Functions	(utility)	No. Characters
C++	N/A	12	N/A	11067
Hume - Lists	3	49	20	7411
Hume - Vectors	3	43	16	6457
Hume - Boxes	13	25	9	11572

The "(utility)" column lists the number of those functions that are utility ones e.g. `map` etc. The no. of characters measure may not be very accurate – the results were found using "wc" and "grep" to strip the lines of comments, though the formatting-spaces remained. Also it ignores the helper programs used to prepare the input for the Hume versions and the image library used by the C++.

It is no surprise that the boxes version is largest – the syntax for wiring is quite verbose, and what were originally simple functions are now complex rules to be matched.

The size for the C++ is also fairly irrelevant – it could have been written to be smaller, instead of written to be fast and readable.

Summary

In summary, I have implemented a box-based motion tracker in Hume, which does not use any recursive functions to carry out its task. I also implemented a C++ version for comparison, and two other Hume versions were created in the course of developing the boxes version.

Comparing the versions, it is no surprise that the C++ is many times faster – however, that difference is only a factor of 60 for the fastest Hume version (22 minutes for the vectors, compared to 21 seconds for the C++). And this is using an interpreted implementation of Hume – I would hope that future Hume compilers would see even greater speed gains.

A Hume implementation also does not require vastly more code than a C++ one – though it is not as functional, and requires input pre-processing. However, I feel that the Hume code is not as readable as the C++ - the boxes version in particular is tricky to read and understand, it is difficult to get an immediate intuition of what a program does from lines of rules to be matched.

Bibliography

[1] *"kernel-based object tracking"* D. Comaniciu, V. Ramesh, P. Meer, IEEE Trans. Pattern Analysis and Machine Intelligence, vol25 (5),pp564-577, August 2003

[2] *"Kalman Filtering"* Mohinder S. Grewal, Angus P. Andrews, Wiley-Interscience 2001, ISBN: 0-471-39254-5

[3] *"Computer Vision Face Tracking For Use in a Perceptual User Interface"*, Gary R. Bradski, Intel Technology Journal Q2 '98, 1998

[4] *"The Hume Report, Version 0.3*", Kevin Hammond, Greg Michaelson, Technical report, School of Maths and Computer Sciences, Heriot-Watt University/School of Computer Science, University of St Andrews, 2005. http://www-fp.dcs.st-and.ac.uk/hume/report/hume-report.ps

Appendix A. Running the code

The Hume CVS repository contains a copy of this report, all the code and some example output from the tracker run on various test sequences.

C++

Files

You should have these files from the archive

CImg-docs	-- The readme and license doc for the CImg library.
CImg.h	--the CImg library (it's self-contained)
colourModel.cpp/.h	--the colour model object, used by meanshift.cpp
initTracker.cpp	--code for the program to get initialisation co-ords
Makefile	--the linux makefile for the code.
meanshift.cpp	--code for the meanshift tracker.
seqcons.cpp	--for constructing image sequence files for the hume.
seqdes.cpp	--for extracting images from the hume image sequences

Compiling

"make" will make all the targets. These are as follows:

initTracker
meanshift
seqcons
seqdes

Targets can be made individually. There are no dependencies, though the "imagemagik" toolbox will mean that more image formats can be handled (to see if you have it, check if you have "convert").

Input image sequences

The input images can be in any format supported by "convert" if you have it installed, otherwise they must be simple formats such as ppm (see CImg docs for details). The input should be a series of individual frames. If you have a video file, and wish to split it into frames, I used the excellent Virtualdub (www.virtualdub.org).

Frame name should not be padded with 0s.

E.g. "frame0" to "frame150" is ok, whereas "frame000" to "frame150" is not.

Usage

initTracker

All of the meanshift programs require initialisation parameters of the top left, and bottom right, coordinates of a box containing the target in the first frame. To make this easier, the initTracker program lets you select these by clicking on the image.

```
initTracker first_frame
```

Will run the program. First click the top left corner, then the bottom right - the target area will then be highlighted and the terminal will ask if this is ok. If you press 'n' then the process repeats, if you press 'y' the program exits with the coordinates of the target.

meanshift

This runs the tracker.

Typing "meanshift" will show the usage, it's best explained by example though:

```
./meanshift ./redcup/redcup .ppm 0 99 77 63 148 151
```

In this case, the images are all in ./redcup, they are all .ppm and they are named redcup0.ppm to redcup99.ppm (so the name_stem is ./redcup/redcup.

The top left corner is 77,63 and the bottom right is 148,151

Output will appear in the same place as the input images, and the same filenames but with "_out" appended to the name.

Editing the Histogram Function

This is the only way you can really "tune" the tracker, see the report or the paper for details on this. It is set up initially to use a 3 colour 16 bins per colour model, that is a feature space of 4096. If this is changed then "make clean" MUST be run before a recompile as meanshift.cpp uses the header too!

The bins-per-colour (BPC) can be changed in colourModel.h, but be sure to set BINSIZE to 256/BPS, and if less colour channels are to be used, then change NUMBINS too (e.g. "NUMBINS BPC" for only one channel) and edit the "findBin" method to return 0 for the unused channels (see comments for details).

The Utility Programs

seqcons

This constructs image sequences for the hume. "./seqcons" will display usage. Note the output is on std_out. Again an example is easiest:

```
./seqcons ./redcup/redcup .ppm 0 99 > redcup.seq
```

This converts the redcup frames (in the meanshift example) to a single redcup.seq file.

The .seq file will be ascii ppm image data, concatenated and with no headers.

seqdes

This takes in a image sequence file (such as output from the hume code) and outputs an image sequence. Note that you need to know the original image dimensions. An example:

```
./seqdes ./out./output .jpg 0 99 redcup.seq 320 240
```

This would output the redcup.seq sequence to ./out/output0.jpg to ./out/output99.jpg and says that it's a sequence of 320x240 images.

The Hume

Performance note

These all operate the same way, except the boxes version, which does not catch "oscillation" as a termination condition (see the report). They run MUCH slower than the C++ code - again see the report.

Compatibility note

Currently (19/9/05) the code only runs on the latest version of the Hume interpreter (contact Robert Pointon for details). It requires support for the "vecdef" and "vecmap" operations, and also "update".

Perhaps more controversially, the vectors version uses "let" to define functions (for partial application of functions) - this shouldn't be allowed in Hume.

Note the boxes version doesn't do this (except for the constants, which could be generated by boxes too).

Also, it uses the formatting input of the interpreter to load image data into 3d vectors.

Files

meanshift-boxes.hume --implemented using boxes for everything but calculating the constants

meanshift-lists.hume -- Implemented in functions, using lists and recursion.

meanshift-vectors.hume -- implemented using vectors, primarily vedefs.

Usage

All of the versions are run the same way. For example:

```
hume meanshift-boxes.hume 2> output.seq
```

The output is an image sequence on std_err - hence the pipe command above. This is because the Hume file output doesn't flush until the EOF character is written, so for large sequences memory usage is huge. Note that “2>” is the syntax for piping to std_err only for the “bash” shell, depending on your shell you may need to use a different symbol.

The std_err will display progress in terms of traces on the number of frames remaining.

To set the input sequence file, the constant "im_seq" stream must be set in the "program parameters section.

The size of the image frames is defined in the "image" type declarations, and the number of frames is unfortunately hard-coded in the main looping box itself - see the comments in the files for help - it's currently set for 150 frames.

target details are specified by the TL_X, TL_Y, BR_X,BR_Y constants, for the Top Left and Bottom Right X,y coordinates.

To adjust the histogram model used, edit the BPC constant (bins-per-colour) and the NUM_CHANNELS constant. Note that if a different number of colour channels are used then the "findbin" function must also be changed - it's currently set to use 128 bins in the red channel only, and the example 3 channel "findbin" function is currently commented out.

Appendix B. Code

meanshift.cpp

```
//meanshift.cpp - a meanshift tracker
//Iain Wallace, 08/2005

//Based on the equations in "kernel-based object tracking"
//D. Comaniciu, V. Ramesh, P. Meer, IEEE Trans. Pattern
//Analysis and Machine Intelligence, vol25 (5),pp564-577,
//August 2003

//References to equations, and numbers, refer to this paper.

//Also used matlab code by Zsolt L. Husz from HWU for reference

//Some notes on terminology.
//"window" refers to the user-defined region covering the target.

//TODO put references in to the eqns in the paper

#include "CImg.h"
#include <iostream>
#include <sstream>
#include <string>
#include <time.h>

#include "colourModel.h"

using namespace cimg_library;
using namespace std;

double kernel(int x,int y,int half_x,int half_y)
{
	//This comes from a simplified version of eqn(12).
	//Note that this makes use of the fact that a lot of the kernel terms cancel out,
	//as it is primarily used in eqn(2) and eqn(3).
	//the distance to the point, normalised to unit radius from the centre
	double euclideanDistance = sqrt( pow( ( (double)(x)/(double)(half_x) ) ,2.0) +pow( ( (double)(y)/(double)(half_y) ) ,2.0) );
	if (euclideanDistance > 1)
		return( 0.0);
	else
		return(1.0-pow(euclideanDistance,2));
}

void evalKernel (double*** kArray,int half_x,int half_y)
{
	//This function calculates the Epanechnikov kernel over
	//the size of the window.
	//x and y vary accoding to local co-ords with 0,0 at the centre
```

```
	for (int x = -half_x;x < half_x;++x)
	{
		for (int y = -half_y;y < half_y;++y)
		{
			(*kArray)[x+half_x][y+half_y] = kernel(x,y,half_x,half_y);
		}
	}
}

void evalKernelDeriv (int*** kArray,int half_x,int half_y)
{
	//This function calculates the derivative of the Epanechnikov kernel over
	//the size of the window. Appears as "g" in the paper.
	//x and y vary accoding to local co-ords with 0,0 at the centre
	double euclideanDistance;
	for (int x = -half_x;x < half_x;++x)
	{
		for (int y = -half_y;y < half_y;++y)
		{
			euclideanDistance = sqrt( pow( ( (double)(x)/(double)(half_x) ) ,2.0) +pow( ( (double)(y)/(double)(half_y) ) ,2.0) );
			if (euclideanDistance > 1)
				(*kArray)[x+half_x][y+half_y] = 0;
			else
				(*kArray)[x+half_x][y+half_y] = 1;

		}
	}
}
void writeFrame(CImg<unsigned char>* frame,int centre_x,int centre_y,int half_size_x,int half_size_y,string name)
{
	const unsigned char colour[3]={255,0,0};
	int x1 = centre_x - half_size_x;
	int y1 = centre_y - half_size_y;
	int x2 = centre_x + half_size_x;
	int y2 = centre_y + half_size_y;

	//Highlight the target in a fetching transparent red
	(*frame).draw_rectangle(x1,y1,x2,y2,colour,0.4);

	//This code writes the frame (with box) to a file.
	//Filename will be the original, with "_out" appended before the extension.
	string fileName(name);
	fileName.insert((name.size()-4),"_out");
	(*frame).save(fileName.c_str());

}


void updateWeights(double*** weights,colourModel *Pu,colourModel *Qu,unsigned int x_size,unsigned int y_size)
{
	//This calculates the pixel weights for the window, based on the colour model.
	//Described in eqn(10)
```

```
	double R[NUMBINS];

	for (int i = 0;i< NUMBINS;++i)
	{
		//Note if there're no pixels in a bin we'll never need to use the value,
		//so set corresponding R to 0
		//Necessary, as otherwise there will be divide-by-zero errors.
		if ((*Pu)[i]==0)
		{
			R[i] = 0.0;
		}
		else
		{
			R[i] = sqrt( ((*Qu)[i]/(*Pu)[i]) );
		}
	}

	for (unsigned int x = 0;x<x_size;++x)
	{
		for (unsigned int y = 0;y<y_size;++y)
		{
			(*weights)[x][y] = R[(*Pu).theBin(x,y)];
		}
	}
}

void computeDisplacement(double*** weights,int*** kArray,unsigned int centre_x,unsigned int centre_y,int half_x,int half_y,int *dx,int *dy)
{
	double weight_sum = 0;
	double x_sum =0, y_sum=0;
	double curPixelWeight;

	for (int x = -half_x;x < half_x;++x)
	{
		for (int y = -half_y;y < half_y;++y)
		{
			//the bottom half of eqn(11) (sum the weights under the kernel)
			curPixelWeight = (*weights)[x+half_x+1][y+half_y+1]*(*kArray)[x+half_x+1][y+half_y+1];
			weight_sum += curPixelWeight;
			//The top half of Eqn(11)
			x_sum += x*curPixelWeight;
			y_sum += y*curPixelWeight;
		}
	}
	//do the division
	*dx = (int)floor(x_sum/weight_sum);
	*dy = (int)floor(y_sum/weight_sum);
}

int main(int argc,char **argv)
{
    if (argc != 9)
    {
```

```
    //TODO make the usage more helpful once I know what it does!
     cout << "Usage: meanshift name_stem extension start_frame_no end_frame_no TL-X TL-Y BR-X BR-Y" << endl;
     cout << "name-Stem = the invariant part of the image's filenames" << endl;
     cout << "extension = the file extension of the images" << endl;
     cout << "start and end frame no - fairly self-explanatory" << endl;
     cout << "TL-X = top-left corner X co-ord" << endl;
     cout << "TL-Y = top-left corner Y co-ord" << endl;
     cout << "BR-X = bottom-right corner X co-ord" << endl;
     cout << "BR-Y = bottom-right corner Y co-ord" << endl;
     exit(1);
}

    //First off, save the arguments
    int startNo = atoi(argv[3]);
    int endNo = atoi(argv[4]);
    string nameStem(argv[1]);
    string extension(argv[2]);
    //Use a stringstream for creatign image filenames, as it's easier to manipulate
    stringstream curFrameName(ios_base::in | ios_base::out);
    int x0 = atoi(argv[5]);
    int y0 = atoi(argv[6]);
    int x1 = atoi(argv[7]);
    int y1 = atoi(argv[8]);

    int curFrameNo = startNo;

    //calculate the centre of the window, and the half-size (size from centre to edge)
    //Note this implicitly makes the window an odd size, which is required so there is
    //a centre pixel (not halfway)
    int centre_x = (int)floor((x1-x0)/2.0)+x0;
    int centre_y = (int)floor((y1-y0)/2.0)+y0;
    const int half_size_x = centre_x - x0;
    const int half_size_y = centre_y - y0;
    const int x_size = 2*half_size_x+1;
    const int y_size = 2*half_size_y+1;

    cout << "Initialising the weight array" << endl;
    //declare a 2d array the size of the window
    //This is used to store the pixel weights, as described in eqn(10)
    double **weight_array;
    weight_array = new double*[x_size];
    for (int i=0;i<x_size;++i)
          weight_array[i]  = new double[y_size];

    cout << "Initialising the kernel" << endl;
    //declare a 2d array the size of the window, and fill it with the kernel function

    double **kernel_array;
    kernel_array = new double*[x_size];
    for (int i=0;i<x_size;++i)
          kernel_array[i]  = new double[y_size];
    //the value of the kernal is static, as the only variable is the position of the pixels
    //relative to the centre, and the window size is constant.
```

```
evalKernel(&kernel_array,half_size_x,half_size_y);

cout << "Initialising the kernel derivative" << endl;

//now the derivative of the kernel
//AS with the kernel, this is needed, but also constant (as the kernel is constant)
int **kernelDeriv_array;
kernelDeriv_array = new int*[x_size];
for (int i=0;i<x_size;++i)
      kernelDeriv_array[i]  = new int[y_size];

evalKernelDeriv(&kernelDeriv_array,half_size_x,half_size_y);

cout << "Initialising the model (Qu)" << endl;

//this is odd - why does it clear the buffer rather than set it??
//Hence the code "seems" to put nameStem in the stringstream twice, but
//actually only does once.
curFrameName.str(nameStem);
curFrameName << nameStem << curFrameNo << extension;

cout << "Loading: " << curFrameName.str().c_str() << endl;
CImg<unsigned char> frame(curFrameName.str().c_str());

//we need the model of the window
colourModel Qu;

Qu.updateModel(&frame,centre_x,centre_y,half_size_x,half_size_y,&kernel_array);


cout << "Writing the first frame" << endl;
//NOTE: currently the writeFrame routine just draws the rectangle, output is
//to display only.
writeFrame(&frame,centre_x,centre_y,half_size_x,half_size_y,curFrameName.str());
//CImgDisplay main_disp(frame,"output",0,0);


colourModel Pu;
int dx = 0, dy=0, pdx = 0, pdy = 0;

//for some stats
int avIters = 0;
int maxIters = 0;
int minIters = 99999;

time_t start, end;
time(&start);
cout << "****Beginning tracking****" << endl;
for (curFrameNo = startNo+1;curFrameNo <= endNo;++curFrameNo)
{
      //Load the frame
      curFrameName.str(nameStem);
      curFrameName << nameStem << curFrameNo << extension;
```

```
		cout << "Loading: " << curFrameName.str().c_str() << endl;

		frame = frame.load(curFrameName.str().c_str());

		//We iterate over the current frame until it converges.
		int loopCount = 0;
		bool exit = false;
		while (exit == false)
		{
			pdx = dx;
			pdy = dy;

			loopCount++;
			Pu.updateModel(&frame,centre_x,centre_y,half_size_x,half_size_y,&kernel_array);
			updateWeights(&weight_array,&Pu,&Qu,x_size,y_size);
			//now compute the displacement
			computeDisplacement(&weight_array,&kernelDeriv_array,centre_x,centre_y,half_size_x,half_size_y,&dx,&dy);
			//cout << "dx = " << dx << " dy = " << dy << endl;
			centre_x += dx;
			centre_y += dy;
			//Check if we've converged
			//Also, strictly a better convergence rule could be used
			// - see steps 4-6 on p567 in the paper.
			//There is a check for "oscillation" due to rounding errors.
			if (((dx == 0) && (dy == 0)) || (loopCount > 20) || ( (pdx + dx == 0) && (pdy+dy==0) ))
				exit = true;
		}
		avIters += loopCount;
		if (loopCount > maxIters)
			maxIters = loopCount;
		if (loopCount < minIters)
			minIters = loopCount;

		//now write out the frame
		//NOTE: currently the writeFrame routine just draws the rectangle, output is
		//to display only.
		writeFrame(&frame,centre_x,centre_y,half_size_x,half_size_y,curFrameName.str());
		//main_disp.display(frame);
	}
	time(&end);
	cout << "****Done tracking****" << endl;
	cout << "Average iterations per frame = " << (avIters/curFrameNo) << endl;
	cout << "Max iters = " << maxIters << endl;
	cout << "Min iters = " << minIters << endl;
	cout << "Ave. Fps: " << (curFrameNo/difftime(end,start)) << "fps" << endl;
	cout << "Ave. ips: " << (avIters/difftime(end,start)) << "ips" << endl;


	//be a good little program and free the memory :-)
	for (int i=0;i<x_size;++i)
	{
		delete[] weight_array[i];
		delete[] kernel_array[i];
```

```
			delete[] kernelDeriv_array[i];
		}
		delete[] weight_array;
		delete[] kernel_array;
		delete[] kernelDeriv_array;
		exit(0);
}
```

colourModel.h

```
//colourModel.h - defines the colourModel object
//This is used to create the weighted histogram used by the mean-shift tracker.

// Iain Wallace 16/08/05

#ifndef COLOURMODEL_H
#define COLOURMODEL_H
//Uses the CImg library, found at http://cimg.sourceforge.net/
#include "CImg.h"

//IMPORTANT: Changes to these values WILL cause crashes unless you do "make clean"
//then rebuild meanshift.o aswell, as it uses them!
#define BPC 16                //bins per colour
#define BINSIZE 16            //size of each bin MUST BE BPC/256
#define NUMBINS BPC*BPC*BPC   //total number of bins
//if only two colours are used then
//#define NUMBINS BPC*BPC
//if only one colour
//#define NUMBINS BPC

class colourModel
{
public:
		colourModel();          //initially the histogram will be empty.
		~colourModel();


		//The [] operator indexes into m_model.
		//Note index is 0 to numBins-1
		double operator[](unsigned int bin);

		//returns the bin-no. of a RGB value
		unsigned int findBin(unsigned char R,unsigned char G,unsigned char B);

		//empties the model for updating.
		void clearModel();

		//takes in a kernel covering the ROI too, and instead of a regular histogram,
		//creates the model for the ROI based on the kernel too.
		//Used for Qu and Pu
```

```
      void updateModel(cimg_library::CImg<unsigned char>* image,  //The image in question
                               unsigned int centreX,    //X co-ord of centre
                               unsigned int centreY,    //Y co-ord of centre
                                int half_x,     //half the x-size of the window
                                int half_y,     //half the y-size of the window
                               double*** kArray);       //the kernel



       //returns the bin-number of the pixel at x,y in the binallocation table
       //unsafe! well kinda, returns 0 and prints to std-err for errors
       //IMPORTANT! This takes an index 0-x_size, not the -half_x to half_x indexing
       //used elsewhere!
       unsigned int theBin(unsigned int x, unsigned int y);


private:
        double m_model[NUMBINS];
        unsigned int** bins;
        bool binsInit;
        unsigned int mX_dim,mY_dim;
};
#endif //COLOURMODEL_H
```

colourModel.cpp

```
//colourModel.cpp - the colourModel object
//Iain Wallace 16/08/05

#include "colourModel.h"
#include <math.h>
#include <iostream>
using namespace cimg_library;
using namespace std;

colourModel::colourModel()
{
      //model must be initialised to 0
      memset( m_model, '\0', sizeof(m_model) );
      //Se we can tell if the memory's been allocated
      binsInit = false;
}

colourModel::~colourModel()
{
      //Free the bins memory, if it's assigned.
      if (binsInit)
      {
          for (unsigned int i=0;i<mX_dim;++i)
                delete[] bins[i];
```

```
			delete[] bins;
		}
}

double colourModel::operator[](unsigned int bin)
{
	//Use GTE as index is from 0
	if (bin >= NUMBINS)
	{
		cerr << "ERROR! Tried to access a model bin that doesn't exit!" << endl;
		return 0;
	}
	return m_model[bin];
}

unsigned int colourModel::findBin(unsigned char R,unsigned char G,unsigned char B)
{
	//return the bin number of a pixel according to its RGB value.
	//Note that the constants defined in the header file, and this function
	//control the only real tuning of the tracker.

	//scale the colours
	unsigned int r,g,b;


	r = (unsigned int)floor( (float)(R/BINSIZE) );
	g = (unsigned int)floor( (float)(G/BINSIZE) );
	b = (unsigned int)floor( (float)(B/BINSIZE) );
	//If, for example, the blue channel is not to be used, then change for:
	//b = 0;


	return (r + BPC*g + BPC*BPC*b);
}

void colourModel::clearModel()
{
	memset( m_model, '\0', sizeof(m_model) );
}


//This performs the function described by eqn(2) and eqn(4) (effectively the same)
//It saves the bin that each pixel is allocated too, as this prevents it being
//re-calculated when the weights are updated.
void colourModel::updateModel(cimg_library::CImg<unsigned char>* image,  //The image in question
                              unsigned int centreX,    //X co-ord of centre
                              unsigned int centreY,    //Y co-ord of centre
                              int half_x,              //half the x-size of the window
                              int half_y,              //half the y-size of the window
                              double*** kArray)        //the kernel
{
	 clearModel();
	 //first time this is called, create bins with the size of the window
```

```
        //NOTE: more cunning memory management would be required for reszizing windows!
        if (!binsInit)
        {
             cout << "Allocating a bin allocation table" << endl;
             //allocate the memory
             mX_dim = 2*half_x+1;
             mY_dim = 2*half_y+1;
             bins = new unsigned int*[mX_dim];
             for (unsigned int i=0;i<mX_dim;++i)
                   bins[i]  = new unsigned int[mY_dim];
             binsInit = true;
        }


       for (int x = -half_x;x <= half_x;++x)
       {
             for (int y = -half_y;y <= half_y;++y)
             {
                   //update the bin allocation table at the same time
                   int iX = centreX + x;
                   int iY = centreY + y;
                   bins[x+half_x][y+half_y] = findBin((*image)(iX,iY,0),(*image)(iX,iY,1),(*image)(iX,iY,2));
                   //adding on the kernel function, instead of summing like a regular histogram,
                   //accounts for the delta function in the eqns.
                   m_model[bins[x+half_x][y+half_y]]+= (*kArray)[x+half_x][y+half_y];
             }
       }
       //The model values must by normalised.
       double total = 0;
       for (int i = 0;i< NUMBINS;++i)
       {
             total += m_model[i];
       }
       for (int i = 0;i< NUMBINS;++i)
       {
             m_model[i] /= total;
       }
}

unsigned int colourModel::theBin(unsigned int x, unsigned int y)
{
       if ((x > mX_dim) || (y > mY_dim) || (!binsInit))
       {
             cerr << "ERROR! Attempted to access a pixel out of the window!" << endl;
             return( 0);
       }
       else
       {
             return(bins[x][y]);
       }
}
```

meanshift-lists.hume

```
--meanshift.hume - a meanshift tracker
--Iain Wallace, 08/2005

--Based on the equations in "kernel-based object tracking"
--D. Comaniciu, V. Ramesh, P. Meer, IEEE Trans. Pattern
--Analysis and Machine Intelligence, vol25 (5),pp564-577,
--August 2003

--References to equations, and numbers, refer to this paper.
--Also see my C++ implementation of this algorithm,
--as I feel it's easier to understand.


--The output images will be to std-err, so it should be
--piped into a file, which seqdes can then be run on.
--The std out will display the traces, that is the number
--of frames remaining and the current centre location.

program

-----------------------------------
--Types----------------------------
-----------------------------------
--some basic types, used all over the place
type Int = int 32;
type Float = float 32;
type model = [Float];
-----------------------------------
--Program parameters---------------
-----------------------------------
--this is a sequence of images, ascii ppm with headers stripped, concatenated into one file.
--output is the same, use the C++ progs to construct/split the files
stream im_seq from "person-sm.seq";
stream ims_out to "person-track-out.seq";

stream outputscreen to "std_out";
stream outerror to "std_err";



--CHANGEME this section, specific to image size
type pixel = vector 1..3 of Int;
type im_row = vector 1 .. 320 of pixel;
type image = vector 1..240 of im_row;

type coord = vector 1..2 of nat 64;
```

```
--CHANGEME these are target details
constant TL_X = 109;
constant TL_Y = 59;

constant BR_X = 134;
constant BR_Y = 127;

constant half_x = ((BR_X - TL_X) div 2);
constant half_y = ((BR_Y - TL_Y) div 2);

constant x_size = (2*half_x);
constant y_size = (2*half_y);

constant init_centre = << (TL_X + half_x ),(TL_Y + half_y)>>;

--CHANGEME These are the settings used to define
--the histogram. The things to change are BPC (bins per colour)
--and the Number of colour channels. (NUM_CHANNELS)
--Note that changing the number of channels requires selecting
--and appropriate "findbin" function below.
constant BPC = 128;
constant BINSIZE = (256 div BPC);
constant NUM_CHANNELS = 1;
constant  NUMBINS = (BPC ** NUM_CHANNELS)+1;
expression "numbins";
expression NUMBINS;
-----------------------------------
--Utility functions----------------
-----------------------------------

--helper functions to update a 2d vector (image)
update2d tdvector x y item = update tdvector x (update (tdvector@x) y item);
--note update can be defined in terms of vecdef like so
--myupdate vec pos val =  let
--                          ff i = if i == pos
--                                 then val
--                                 else vec@i
--                       in vecdef (length vec) ff;

sqr :: Float -> Float;
sqr x = x*x;

--Define the map function
map f [] = [] ;
map f (h:t) = f h:map f t;

--function to multiply two numbers
mul x y = x*y;

--this maps a function over 2 EQUAL SIZE lists, resulting in one list
--that's the combination of the two.
--Kinda like a zip  and a fold all rolled into one.
mixmap f [] [] = [];
```

```
mixmap f (h1:t1) (h2:t2) = ((f h1 h2):(mixmap f t1 t2));

--mixmaps 2 equal size 2d lists together, returns a 2d list
--again like a map over 2dlists, and a fold.
twodmixmap f [] [] = [];
twodmixmap f (h1:t1) (h2:t2) = (( mixmap f h1 h2 ):(twodmixmap f t1 t2));

--define foldr
foldr f z [] = z;
foldr f z (x:xs) = f x (foldr f z xs);

--and an "add" function
add x y = x + y;

--sum a list
sum l = foldr add 0.0 l;

--sum the sublists in  a list of lists
sums l = map sum l;

--returns a pixel in the image, indexed using the centre and a range 0-x_size
getP :: image -> Int -> Int -> coord -> pixel;
getP im x y cen = (im@((cen@2)+y-half_y))@((cen@1)+x-half_x);

divCoord :: coord -> Float -> coord;
divCoord dx sum = << (((dx@1) as Float)/sum) as Int,(((dx@2) as Float)/sum) as Int >>;

mulCoord :: coord -> Float -> coord;
mulCoord dx sum =  << (((dx@1) as Float)*sum) as Int,(((dx@2) as Float)*sum) as Int >>;

addCoord x y = << x@1 + y@1, x@2 + y@2>>;

csum l = foldr addCoord <<0,0>> l;

csums l = csum (map csum l);
-----------------------------------
--a function returning a list-of-lists, representing the Epanechnikov kernel
--over the search window

--This comes from a simplified version of eqn(12).
--Note that this makes use of the fact that a lot
-- of the kernel terms cancel out, as it is
--primarily used in eqn(2) and eqn(3).


normalisedEuclideanDistance :: Int -> Int -> Float;

normalisedEuclideanDistance x y = sqrt( sqr((x as Float) / (half_x as Float)) + sqr((y as Float) / (half_y as Float))  );

--just to rename it to make the code a bit more managble.
ned x y = normalisedEuclideanDistance x y;

kernel :: Int -> Int -> Float;
```

```
kernel x y = if ((ned x y) > 1.0)
               then 0.0
                      else (1.0 - sqr(ned x y));

evalKcols :: Int -> Int -> [Float];
evalKcols columnNo ypos  = if (ypos ==  half_y)
                                then [kernel columnNo ypos]
                                else ((kernel columnNo ypos):(evalKcols columnNo (ypos+1)));

evalKrows :: Int -> [[Float]];

evalKrows xpos = if (xpos == half_x)
                    then [evalKcols xpos (0-half_y)]
                    else ((evalKcols xpos (0-half_y)):(evalKrows (xpos+1)));

--the list returned is indexed first by X then by Y
--that is, it's a list of columns
--a list of lists of floats.
evalKernel :: [[Float]];
evalKernel = evalKrows (0-half_x);

------------------------------------------------------
--Functions to calculate the derivative kernel, as a list
--of lists, like the kernel. This is what appears as the
--"G" term in the equations in the paper.

kernelDeriv :: Int -> Int -> Float;
kernelDeriv x y = if ((ned x y) > 1.0)
                     then 0.0
                     else 1.0;

evalDcols :: Int -> Int -> [Float];
evalDcols columnNo ypos  = if (ypos ==  half_y)
                                then [kernelDeriv columnNo ypos]
                                else ((kernelDeriv columnNo ypos):(evalDcols columnNo (ypos+1)));

evalDrows :: Int -> [[Float]];

evalDrows xpos = if (xpos == half_x)
                    then [evalDcols xpos (0-half_y)]
                    else ((evalDcols xpos (0-half_y)):(evalDrows (xpos+1)));

evalDeriv :: [[Float]];
evalDeriv = evalDrows (0-half_x);

------MODEL STUFF-----------------------------
--this is where things start to get a bit tricky...
--Note that results are not the same as the C++!
--..could be rounding differences etc. though.... :-S

--scale a colour into the histogram's range
scale :: Int -> Int;
scale c = (c div BINSIZE);
```

```
--Find the bin number of a pixel (R,G,B vector)
findBin :: pixel -> Int;

--this is the 3-channel findbin
--findBin P = scale(P@1) + BPC*scale(P@2) + BPC*BPC*scale(P@3) + 1;--cos we don't index from 0!!

--this is the red-channel only findbin.
findBin P = scale(P@1) +1;

--adds a value onto the value in a given model position
--Note the first position in the list  is position 0!!
addToModel :: Int -> Float -> model -> model;
addToModel pos val (h:t) = if (pos == 0)
                              then ((h+val):t)
                              else (h:(addToModel (pos -1) val t));

--this initialises the model to be entirely 0, of a given size.
makeEmptyModel :: Int->  model;
makeEmptyModel size  = if (size ==0)
                            then []
                            else (0.0:(makeEmptyModel (size -1 )));

--this assumes certain constants are already defined
-- returns a model, NUMBINS in size
--WARNING this is where it starts to get tricky,
-- just start crying now ;-)

--This performs the function described by eqn(2) and eqn(4) (effectively the same)
--it's the same as the "updatemodel" function in colourmodel in the C++.

doYUpdate :: image -> coord -> Int -> Int -> model -> [Float] -> model;
doYUpdate im cen Xno y tmodel (h:t) = if (y == half_y)
                                          then (addToModel (findBin((im@((cen@2)+y))@((cen@1)+Xno))) h tmodel)
                                         -- then 5/0
                                          else doYUpdate im cen Xno (y+1) (addToModel (findBin((im@((cen@2)+y))@((cen@1)+Xno))) h tmodel)  t;

doUpdate :: image -> coord -> Int -> Int -> model -> [[Float]] -> model;

doUpdate im cen x y tmodel (h:t) = if (x == half_x)
                                       then (doYUpdate im cen x y tmodel h)
                                      -- then 5/0
                                        else doUpdate im cen (x+1) y (doYUpdate im cen x y tmodel h)  t;


--we need to be able to normalise the model histogram

norm :: model -> Float -> model;
norm [] theSum = [];
norm (h:t) theSum = (h/theSum):(norm t theSum);

normaliseModel :: model -> model;
normaliseModel tmodel = norm tmodel (sum(tmodel));
```

```
updateModel :: image -> coord  -> [[Float]] -> model;
updateModel frame centre kernel = normaliseModel(doUpdate frame centre (0-half_x) (0-half_y) (makeEmptyModel NUMBINS) kernel);

--------------------------------
--Functions used to update the weights, note these could probably be more efficent
--they have more in common with the equation, rather than the C++ function.

--This calculates the pixel weights for the window, based on the colour model.
--Described in eqn(10)


updateWeights :: model -> model -> image -> coord -> [[Float]];
updateWeights Pu Qu frame cen = updateWX 0 Pu Qu frame cen;

updateWX :: Int -> model -> model -> image -> coord -> model;
updateWX x Pu Qu frame cen = if (x == x_size)
                              then [updateWY x 0 Pu Qu frame cen]
                              else ((updateWY x 0 Pu Qu frame cen):(updateWX (x+1) Pu Qu frame cen));


--FIXME this uses list indexing ARRRG! (and into models at that!)
updateWY :: Int -> Int -> model -> model -> image -> coord -> [Float];
updateWY x y Pu Qu frame cen = if (y == y_size)
                                    then (if ((Pu@(findBin(getP frame x y cen))) == 0.0)
                                           then [0.0]
                                           else [sqrt((Qu@(findBin(getP frame x y cen)))/(Pu@(findBin(getP frame x y cen))))])
                                    else (if ((Pu@(findBin(getP frame x y cen))) == 0.0)
                                            then(0.0:(updateWY x (y+1) Pu Qu frame cen))
                                            else ((sqrt((Qu@(findBin(getP frame x y cen)))/(Pu@(findBin(getP frame x y cen))))):(updateWY x (y+1) Pu Qu
frame cen)));

--------------------------------
--Below are the functions used to calculate the displacement
--this is eqn(11) in the paper.

--FIXME arg, i do the mixmap several times! no need!
--Also, incredibly cryptic code!


computeDisplacement :: [[Float]] -> [[Float]] -> coord -> coord;
computeDisplacement weights kderiv = divCoord (csums (twodmixmap mulCoord theWin (twodmixmap mul weights kderiv)))  (sum (sums(twodmixmap mul weights
kderiv)));

--------------------------------
--functions to loop the whole op over the frame untill convergence
--this is ugly!!!!
--Some slight error, if you run it to convergence on 1 frame, it shifts
--the centre 1 pixel to the right.

findDx :: coord -> coord -> coord -> Int -> image ->model-> coord;
findNewCentre centre dx old_dx loopcount frame Qu  = if ( (dx == <<0,0>>) || (loopcount > 4) || ((addCoord dx old_dx) == <<0,0>>) )
                                                           then centre
```

```
                                            else findNewCentre (addCoord centre dx) ( computeDisplacement (updateWeights (updateModel
frame (addCoord centre dx) theKern) Qu frame (addCoord centre dx)) theDeriv) dx (loopcount + 1) frame Qu;

updateCentre :: coord -> frame -> model  -> coord;
updateCentre centre frame Qu = findNewCentre centre ( computeDisplacement (updateWeights (updateModel frame centre theKern) Qu frame centre) theDeriv)
<<0,0>> 0 frame Qu;



--------------------------------
--create a window-sized list-of-lists with the centre-relative
--coords in.

evalWcols :: Int -> Int -> [coord];
evalWcols columnNo ypos  = if (ypos ==  half_y)
                               then [<< columnNo, ypos >>]
                               else ((<<columnNo, ypos>>):(evalWcols columnNo (ypos+1)));

evalWrows :: Int -> [[coord]];

evalWrows xpos = if (xpos == half_x)
                    then [evalWcols xpos (0-half_y)]
                    else ((evalWcols xpos (0-half_y)):(evalWrows (xpos+1)));

evalWin :: [[coord]];
evalWin = evalWrows (0-half_x);

--------------------------------
--evaluate the constants, speeds things up later.

constant theWin = evalWin;
constant theKern = evalKernel;
constant theDeriv = evalDeriv;

---------------------------------
--a box that loads in an image. Makes use of the cunning I/O
--in the interpreter.

box load_im

in (im::image)
out (im'::image)
match
 i -> i;

wire load_im (im_seq) (initmodel.im);

---------------------------------

--------------------------------
--a box to loop over each frame

box initmodel
```

```
in (im::image,cen::coord,num_ims::Int,Qu::model)
out (im'::image,cen_out::coord,cen'::coord,num_left::Int,Qu'::model,frames_left::Int)

match
(i,c,150,_) -> (i,c,c,                      (149),(updateModel i c theKern),(149)) |
(i,c,n,  q) -> (i,c,( updateCentre c i q ),(n-1),q,                         (n-1));

wire initmodel
     (load_im.im',initmodel.cen' initially init_centre,initmodel.num_left initially 150,initmodel.Qu' initially [])--CHANGEME number of frames
     (writebox.im,writebox.pos,initmodel.cen trace, initmodel.num_ims,initmodel.Qu,writebox.num trace);

--the output box, draws a dot on the centre

box writebox
in (im::image,pos::coord,num::Int)
out (im'::image)

match
(_,_,0) -> (5/0                                ) | --the /0 makes it terminate
(i,p,_) -> ((update2d i (p@2) (p@1) <<0,255,0>>));

wire writebox
     (initmodel.im',initmodel.cen_out,initmodel.frames_left)
     (outerror);
```

meanshift-vectors.hume

```
--meanshift.hume - a meanshift tracker
--Iain Wallace, 08/2005

--Based on the equations in "kernel-based object tracking"
--D. Comaniciu, V. Ramesh, P. Meer, IEEE Trans. Pattern
--Analysis and Machine Intelligence, vol25 (5),pp564-577,
--August 2003

--References to equations, and numbers, refer to this paper.
--Also see my C++ implementation of this algorithm,
--as I feel it's easier to understand.


--The output images will be to std-err, so it should be
--piped into a file, which seqdes can then be run on.
--The std out will display the traces, that is the number
--of frames remaining and the current centre location.

--VECTOR VERSION!!!!
--This version uses vectors as opposed to the lists of the original.
--This poses probelms with types, as the vectors are created
--by "vecdef", and of constant size, but constant expressions
```

```
--can't be used to define types! This could be got around
--by manually entering the constant;s values everywhere instead
--of using expressions, and then typing would work.
--But it'd be a faff.

program

-----------------------------------
--Types----------------------------
-----------------------------------

--some basic types, used all over the place
type Int = int 32;
type Float = float 32;

--Arg i have to put this (possibly incorrect) type in to run it!
--it's the vecdef const-expression problem again!
type model = vector 1..4096 of Float;


-----------------------------------
--Program parameters---------------
-----------------------------------
--change these depending on the input data
--NOTE there are other coded-in values elsewhere in the code.
--Blame Hume!

--this is a sequence of images, ascii ppm with headers stripped, concatenated into one file.
--output is the same, use the C++ progs to construct/split the files
stream im_seq from "person-sm.seq";
stream ims_out to "person-track-out.seq";

stream outputscreen to "std_out";
stream outerror to "std_err";

--CHANGEME this section, specific to image size
type pixel = vector 1..3 of Int;
type im_row = vector 1 .. 320 of pixel;
type image = vector 1..240 of im_row;

type coord = vector 1..2 of nat 64;

--CHANGEME these are target details
constant TL_X = 109;
constant TL_Y = 59;

constant BR_X = 134;
constant BR_Y = 127;

--constant TL_X = 4;
--constant TL_Y = 4;
--constant BR_X = 7;
--constant BR_Y = 7;
```

```
constant half_x = ((BR_X - TL_X) div 2);
constant half_y = ((BR_Y - TL_Y) div 2);

constant x_size = (2*half_x);
constant y_size = (2*half_y);

constant init_centre = << (TL_X + half_x ),(TL_Y + half_y)>>;

--CHANGEME These are the settings used to define
--the histogram. The things to change are BPC (bins per colour)
--and the Number of colour channels. (NUM_CHANNELS)
--Note that changing the number of channels requires selecting
--and appropriate "findbin" function below.
constant BPC = 128;
constant BINSIZE = (256 div BPC);
constant NUM_CHANNELS = 1;
constant  NUMBINS = (BPC ** NUM_CHANNELS)+1;
expression "numbins";
expression NUMBINS;



-----------------------------------
--Utility functions----------------
-----------------------------------

--helper functions to update a 2d vector (image)
update2d tdvector x y item = update tdvector x (update (tdvector@x) y item);
--note update can be defined in terms of vecdef like so
--myupdate vec pos val =  let
--                          ff i = if i == pos
--                                 then val
--                                else vec@i
--                        in vecdef (length vec) ff;

sqr :: Float -> Float;
sqr x = x*x;

--function to multiply two numbers
mul x y = x*y;

vec2dmixmap f vec1 vec2 = let Xs x = vec2dmmY f vec1 vec2 x
                              in
                              vecdef (length vec1) Xs;

vec2dmmY f vec1 vec2 x = let Ys y = index f vec1 vec2 x y
                             in
                             vecdef (length (vec1@x)) Ys;

index f vec1 vec2 x y =  f ((vec1@x)@y) ((vec2@x)@y);

--and an "add" function
```

```
add x y = x + y;

--returns a pixel in the image, indexed using the centre and a range 0-x_size
getP :: image -> Int -> Int -> coord -> pixel;
getP im x y cen = (im@((cen@2)+y-half_y))@((cen@1)+x-half_x);

divCoord :: coord -> Float -> coord;
divCoord dx sum = << (((dx@1) as Float)/sum) as Int,(((dx@2) as Float)/sum) as Int >>;

mulCoord :: coord -> Float -> coord;
mulCoord dx sum =  << (((dx@1) as Float)*sum) as Int,(((dx@2) as Float)*sum) as Int >>;

addCoord x y = << x@1 + y@1, x@2 + y@2>>;

csumvec v size = if (size == 1)
                       then v@1
                       else addCoord (v@size) (csumvec v (size-1));

csum v = csumvec v (length v);

csums v = csum (vecmap v csum);

sum1d vec = sumvec vec (length vec);

sum2d vec = sum1d (vecmap vec sum1d);

---------------------------------
--a function returning a vector-of-vectors, representing the Epanechnikov kernel
--over the search window

--This comes from a simplified version of eqn(12).
--Note that this makes use of the fact that a lot
-- of the kernel terms cancel out, as it is
--primarily used in eqn(2) and eqn(3).

normalisedEuclideanDistance :: Int -> Int -> Float;

normalisedEuclideanDistance x y = sqrt( sqr((x as Float) / (half_x as Float)) + sqr((y as Float) / (half_y as Float))  );

--just to rename it to make the code a bit more managble.
ned x y = normalisedEuclideanDistance x y;

kernel :: Int -> Int -> Float;
kernel x y = if ((ned x y) > 1.0)
               then 0.0
                     else (1.0 - sqr(ned x y));

--functions to create a kernel in vectors
--TODO can't do the types because of the vecdef/constant vector thing
evalKernel = vecdef x_size evalKcolumns;

evalKcolumns X = let evalK Y = computeKernel X Y
                    in
```

```
                    vecdef y_size evalK;

--computeKernel must convert the range into -half_x to + half_x before calling the kernel.

computeKernel X Y = kernel (X-half_x) (Y-half_y);

------------------------------------------------------
--Functions to calculate the derivative kernel, as a vector
--of vectors, like the kernel. This is what appears as the
--"G" term in the equations in the paper.

kernelDeriv :: Int -> Int -> Float;
kernelDeriv x y = if ((ned x y) > 1.0)
                     then 0.0
                     else 1.0;

--TODO can't do the types because of the vecdef/constant vector thing
evalDeriv = vecdef x_size evalDcolumns;

evalDcolumns X = let evalD Y = computeDKernel X Y
                    in
                    vecdef y_size evalD;

--computeKernel must convert the range into -half_x to + half_x before calling the kernel.

computeDKernel X Y = kernelDeriv (X-half_x) (Y-half_y);



------MODEL STUFF------------------------------
--this is where things start to get a bit tricky...
--Note that results are not the same as the C++!
--..could be rounding differences etc. though.... :-S


--scale a colour into the histogram's range
scale :: Int -> Int;
scale c = (c div BINSIZE);

--Find the bin number of a pixel (R,G,B vector)
findBin :: pixel -> Int;

--this is the 3-channel findbin
--findBin P = scale(P@1) + BPC*scale(P@2) + BPC*BPC*scale(P@3) + 1;--cos we don't index from 0!!

--this is the red-channel only findbin.
findBin P = scale(P@1) +1;

--adds a value onto the value in a given model position
--Note the first position in the list  is position 1!!
--this uses "update" :-S
addToModel :: Int -> Float -> model -> model;
addToModel pos val tmodel = update tmodel pos (tmodel@pos + val);
```

```
--this initialises the model to be entirely 0, of a given size.
makemod :: Int -> Float;
makemod i = 0.0;

makeEmptyModel :: Int->  model;
makeEmptyModel size  = vecdef size makemod;


--this assumes certain constants are already defined
-- returns a model, NUMBINS in size
--WARNING this is where it starts to get tricky,
-- just start crying now ;-)

--This performs the function described by eqn(2) and eqn(4) (effectively the same)
--it's the same as the "updatemodel" function in colourmodel in the C++.

---vector based update

doUpdate im cen x tmodel kernel = if (x == x_size)
                                        then doYUpdate im cen x 1 tmodel kernel
                                        else doUpdate im cen (x+1) (doYUpdate im cen x 1 tmodel kernel) kernel;

doYUpdate im cen Xno y tmodel kernel = if (y == y_size)
                                             then addToModel (findBin( getP im Xno y cen)) ((kernel@Xno)@y) tmodel
                                             else doYUpdate im cen Xno (y+1) (addToModel (findBin (getP im Xno y cen)) ((kernel@Xno)@y) tmodel) kernel;

--we need to be able to normalise the model histogram

sumvec :: model -> Int -> Float;
sumvec tmodel size = if (size == 1)
                      then tmodel@1
                      else tmodel@size + (sumvec tmodel (size-1));

norm :: model -> Float -> model;
norm tmodel theSum = let
                     divide i = i/theSum
                    in
                     vecmap tmodel divide;

normaliseModel :: model -> model;
normaliseModel tmodel = norm tmodel (sumvec tmodel NUMBINS);

updateModel :: image -> coord  -> [[Float]] -> model;
updateModel frame centre kernel = normaliseModel(doUpdate frame centre 1 (makeEmptyModel NUMBINS) kernel);

--------------------------------
--Functions used to update the weights, note these could probably be more efficent
--they have more in common with the equation, rather than the C++ function.

--This calculates the pixel weights for the window, based on the colour model.
--Described in eqn(10)
```

```
--type info obviously now rubbish....
--updateWeights :: model -> model -> image -> coord -> [[Float]];

updateWeights Pu Qu frame cen = let updateX x = updateWX Pu Qu frame cen x
                                    in
                                    vecdef x_size updateX;

updateWX Pu Qu frame cen x = let updateY y = updateW Pu Qu frame cen x y
                                 in
                                 vecdef y_size updateY;

updateW Pu Qu frame cen x y = if ((Pu@(findBin(getP frame x y cen))) == 0.0)
                                 then 0.0
                                 else sqrt((Qu@(findBin(getP frame x y cen)))/(Pu@(findBin(getP frame x y cen))));

--------------------------------
--Below are the functions used to calculate the displacement
--this is eqn(11) in the paper.

--FIXME arg, i do the mixmap several times! no need!
--Also, incredibly cryptic code!


computeDisplacement :: [[Float]] -> [[Float]] -> coord -> coord;
computeDisplacement weights kderiv = divCoord (csums (vec2dmixmap mulCoord theWin (vec2dmixmap mul weights kderiv)))  (sum2d(vec2dmixmap mul weights
kderiv));

--------------------------------
--functions to loop the whole op over the frame untill convergence
--this is ugly!!!!
--Some slight error, if you run it to convergence on 1 frame, it shifts
--the centre 1 pixel to the right.

findDx :: coord -> coord -> coord -> Int -> image ->model-> coord;
findNewCentre centre dx old_dx loopcount frame Qu  = if ( (dx == <<0,0>>) || (loopcount > 4) || ((addCoord dx old_dx) == <<0,0>>) )
                                                         then centre
                                                         else findNewCentre (addCoord centre dx) ( computeDisplacement (updateWeights (updateModel
frame (addCoord centre dx) theKern) Qu frame (addCoord centre dx)) theDeriv) dx (loopcount + 1) frame Qu;

updateCentre :: coord -> frame -> model  -> coord;
updateCentre centre frame Qu = findNewCentre centre ( computeDisplacement (updateWeights (updateModel frame centre theKern) Qu frame centre) theDeriv)
<<0,0>> 0 frame Qu;



--------------------------------
--create a window-sized vector of vectors with the centre-relative
--coords in. TODO use to speed up some ops, well clarify at least :-)

evalWin = vecdef x_size evalWcols;
```

```
evalWcols X = let eval Y = getCoord X Y
                in
                vecdef y_size eval;

getCoord x y = <<x-half_x,y-half_y>>;


--------------------------------
--evaluate the constants, speeds things up later.

constant theWin = evalWin;
constant theKern = evalKernel;
constant theDeriv = evalDeriv;

---------------------------------
--a box that loads in an image. Makes use of the cunning I/O
--in the interpreter.

box load_im

in (im::image)
out (im'::image)
match
 i -> i;

wire load_im (im_seq) (initmodel.im);

---------------------------------
numframes = 150;
--------------------------------
--a box to loop over each frame

box initmodel
in (im::image,cen::coord,num_ims::Int,Qu::model)
out (im'::image,cen_out::coord,cen'::coord,num_left::Int,Qu'::model,frames_left::Int)

--note the initial conditons for the model Qu are spec'd as and empty list, but it's a vector!
match
(i,c,150,_) -> (*,*,c,                    (149),(updateModel i c theKern),*    ) |
(i,c,n,  q) -> (i,c,( updateCentre c i q ),(n-1),q,                        (n-1));

wire initmodel
     (load_im.im',initmodel.cen' initially init_centre,initmodel.num_left initially 150,initmodel.Qu' initially [])
     (writebox.im,writebox.pos,initmodel.cen trace, initmodel.num_ims,initmodel.Qu,writebox.num trace);

--the output box, draws a dot on the centre

box writebox
in (im::image,pos::coord,num::Int)
out (im'::image)

match
(_,_,0) -> (5/0) |
```

```
(i,p,_) -> ((update2d i (p@2) (p@1) <<0,255,0>>));

wire writebox
     (initmodel.im',initmodel.cen_out,initmodel.frames_left)
     (outerror);
```

meanshift-boxes.hume

```
--meanshift.hume - a meanshift tracker
--Iain Wallace, 08/2005

--Based on the equations in "kernel-based object tracking"
--D. Comaniciu, V. Ramesh, P. Meer, IEEE Trans. Pattern
--Analysis and Machine Intelligence, vol25 (5),pp564-577,
--August 2003

--References to equations, and numbers, refer to this paper.
--Also see my C++ implementation of this algorithm,
--as I feel it's easier to understand.


--The output images will be to std-err, so it should be
--piped into a file, which seqdes can then be run on.
--The std out will display the traces, that is the number
--of frames remaining and the current centre location.
--run like "hume meanshift.hume 2> output.seq"

--VECTOR VERSION!!!!
--This version uses vectors as opposed to the lists of the original.
--This poses probelms with types, as the vectors are created
--by "vecdef", and of constant size, but constant expressions
--can't be used to define types! This could be got around
--by manually entering the constant;s values everywhere instead
--of using expressions, and then typing would work.
--But it'd be a faff.

--BOXES VERSION!!!!
--This version uses boxes to doo all of the main processing,
--rather than functions. However, the constants (kernel, derivative
--and relative window positions) are still calculated by functions.
--Note the equations calculated by the boxes aren't noted in comments
--this version's too complex for that - see the report.

program

-----------------------------------
--Types----------------------------
-----------------------------------
```

```
--some basic types, used all over the place
type Int = int 32;
type Float = float 32;

--Arg i have to put this (possibly incorrect) type in to run it!
--it's the vecdef const-expression problem again!
type model = vector 1..4096 of Float;

--a 2d vector of some sort, used cos of the type issues
type bigv = vector 1..999 of bigv2;
type bigv2 = vector 1..999 of Int;

--and one for floats
type bigvf = vector 1..999 of bigv2f;
type bigv2f = vector 1..999 of Float;

--and one for coords
type bigvc = vector 1..999 of bigv2c;
type bigv2c = vector 1..999 of coord;

-----------------------------------
--Program parameters---------------
-----------------------------------
--change these depending on the input data
--NOTE there are other coded-in values elsewhere in the code.
--Blame Hume!

--this is a sequence of images, ascii ppm with headers stripped, concatenated into one file.
--output is the same, use the C++ progs to construct/split the files
stream im_seq from "person-sm.seq";
stream ims_out to "person-track-out.seq";

stream outputscreen to "std_out";
--the standard error is used for output due to i/o flushing issues.
stream outerror to "std_err";

--CHANGEME this section, specific to image size
type pixel = vector 1..3 of Int;
type im_row = vector 1 .. 320 of pixel;
type image = vector 1..240 of im_row;

type coord = vector 1..2 of nat 64;

--CHANGEME these are target details
constant TL_X = 109;
constant TL_Y = 59;
constant BR_X = 134;
constant BR_Y = 127;

--a set of target details that are small, used for loop testing
--constant TL_X = 4;
--constant TL_Y = 4;
--constant BR_X = 7;
```

```
--constant BR_Y = 7;

--CHANGEME These are the settings used to define
--the histogram. The things to change are BPC (bins per colour)
--and the Number of colour channels. (NUM_CHANNELS)
--Note that changing the number of channels requires selecting
--and appropriate "findbin" function below.
constant BPC = 128;
constant BINSIZE = (256 div BPC);
constant NUM_CHANNELS = 1;
constant  NUMBINS = (BPC ** NUM_CHANNELS)+1;
expression "numbins";
expression NUMBINS;

constant half_x = ((BR_X - TL_X) div 2);
constant half_y = ((BR_Y - TL_Y) div 2);

constant x_size = (2*half_x);
constant y_size = (2*half_y);

constant init_cen = << (TL_X + half_x ),(TL_Y + half_y)>>;


-----------------------------------
--Utility functions----------------
-----------------------------------
sqr :: Float -> Float;
sqr x = x*x;

--helper function to update a 2d vector (image)
update2d tdvector x y item = update tdvector x (update (tdvector@x) y item);

--note update can be defined in terms of vecdef like so
--myupdate vec pos val =  let
--                          ff i = if i == pos
--                                 then val
--                                 else vec@i
--                       in vecdef (length vec) ff;

--Defines an empty 2d vector
zero i = 0.0;
vecdef2d x y = let Ys i = vecdef y zero
                in
                vecdef x Ys;

divCoord :: coord -> Float -> coord;
divCoord dx sum = << (((dx@1) as Float)/sum) as Int,(((dx@2) as Float)/sum) as Int >>;

mulCoord :: coord -> Float -> coord;
mulCoord dx sum =  << (((dx@1) as Float)*sum) as Int,(((dx@2) as Float)*sum) as Int >>;

addCoord x y = << x@1 + y@1, x@2 + y@2>>;
```

```
mul x y = x*y;

--return a pixel in the image.
--takes a centre, and an index 0 to x_size
getP :: image -> Int -> Int -> coord -> pixel;
getP im x y cen = (im@((cen@2)+y-half_y))@((cen@1)+x-half_x);

---------------------------------
--functions returning a 2d vector, representing the Epanechnikov kernel
--over the search window
---------------------------------

--This comes from a simplified version of eqn(12).
--Note that this makes use of the fact that a lot
-- of the kernel terms cancel out, as it is
--primarily used in eqn(2) and eqn(3).


normalisedEuclideanDistance :: Int -> Int -> Float;
normalisedEuclideanDistance x y = sqrt( sqr((x as Float) / (half_x as Float)) + sqr((y as Float) / (half_y as Float))  );

--just to rename it to make the code a bit more managble.
ned x y = normalisedEuclideanDistance x y;

kernel :: Int -> Int -> Float;
kernel x y = if ((ned x y) > 1.0)
               then 0.0
                      else (1.0 - sqr(ned x y));

--functions to create a kernel in vectors
--TODO can't do the types because of the vecdef/constant vector thing
evalKernel = vecdef x_size evalKcolumns;

evalKcolumns X = let evalK Y = computeKernel X Y
                     in
                     vecdef y_size evalK;

--computeKernel must convert the range into -half_x to + half_x before calling the kernel.

computeKernel X Y = kernel (X-half_x) (Y-half_y);

-----------------------------------------------------
--Functions to calculate the derivative kernel, as a vector
--of vectors, like the kernel. This is what appears as the
--"G" term in the equations in the paper.

kernelDeriv :: Int -> Int -> Float;
kernelDeriv x y = if ((ned x y) > 1.0)
                      then 0.0
                      else 1.0;

--TODO can't do the types because of the vecdef/constant vector thing
evalDeriv = vecdef x_size evalDcolumns;
```

```
evalDcolumns X = let evalD Y = computeDKernel X Y
                     in
                     vecdef y_size evalD;

--computeKernel must convert the range into -half_x to + half_x before calling the kernel.

computeDKernel X Y = kernelDeriv (X-half_x) (Y-half_y);



------MODEL STUFF-----------------------------

--scale a colour into the histogram's range
scale :: Int -> Int;
scale c = (c div BINSIZE);

--Find the bin number of a pixel (R,G,B vector)
findBin :: pixel -> Int;
--CHANGEME change this also depending on colour model
--this is the 3-channel findbin
--findBin P = scale(P@1) + BPC*scale(P@2) + BPC*BPC*scale(P@3) + 1;--cos we don't index from 0!!
--this is the red-channel only findbin.
findBin P = scale(P@1) +1;

--adds a value onto the value in a given model position
--Note the first position  is position 1!!
--this uses "update" :-S
addToModel :: Int -> Float -> model -> model;
addToModel pos val tmodel = update tmodel pos (tmodel@pos + val);


--this initialises the model to be entirely 0, of a given size.
makemod :: Int -> Float;
makemod i = 0.0;

makeEmptyModel :: Int->  model;
makeEmptyModel size  = vecdef size makemod;


--------------------------------
--create a window-sized vector of vectors with the centre-relative
--coords in. TODO use to speed up some ops, well clarify at least :-)

evalWin = vecdef x_size evalWcols;

evalWcols X = let eval Y = getCoord X Y
                in
                vecdef y_size eval;

getCoord x y = <<x-half_x,y-half_y>>;
```

```
--------------------------------
--evaluate the constants, speeds things up later.

constant theWin = evalWin;
constant theKern = evalKernel;
constant theDeriv = evalDeriv;
constant theEmpty = makeEmptyModel NUMBINS;

---------------------------------
--a box that loads in an image. Makes use of the cunning I/O
--in the interpreter.

box load_im

in (im::image)
out (im'::image)
match
 i -> i;
wire load_im (im_seq) (loopbox.im);

--------------------------------
--a box to loop over each frame

box loopbox
--in wires in this order:
--from load_im
--looping to self for state
--from the model boxes
--from computer_displacement

in (im::image,
    state::Int, cen::coord,num_ims::Int,Qu::model,old_dx::coord,loopcount::Int,cur_im::image,
    model_in::model,
    dx_in::coord)

--out wires listed in the order:
--to writebox
--looping to self - for state
--to do_weights
--to the model-boxes
out (im'::image, cen_out::coord, frames_left::Int,
     state'::Int,cen'::coord, num_left::Int, Qu'::model, dx::coord, cur_im'::image, loopcount'::Int,
     Pu_out::model, Qu_out::model, im_w::image, cen_w::coord,
     im_m::image, cen_m::coord)


--Note, depends on the unfair matching to catch dx=0 termination
--FIXME this won't terminate frame-looping cos of oscillation
--FIXME the loopcount should be a constant somewhere

--CHANGEME the 151,150 and 149s in here are for processing 150 frames.

--These rules could almost certainly be reduced to fewer somehow. But it hurts my brain just thinking about them :-(
```

```
--note the rules with 4 in - the start case is needed explicitly cos DX is reset to 0,0

--the states are as follows:
--The initialisation state, output to model boxes to create Qu
 --loads the now calc'd qu into the loop
--terminate as loopcount reached, load new image, reset LC, write out
--first iter, first dx done calc new model, next iter
--first iter, model done get dx
 --first iter, no dx, no model, calc model
--as above terminate  rule, but terminate cos dx is 0.
-- dx done calc new model, next iter
 --first frame, model done get dx
--first frame, no dx, no model, calc model

match
(i,*, c,151,_*,dx,     lc,*, *,*    ) -> (*, *,              *,    *,c,              150,  *, dx,     i, lc,    *,*, *, *,i, c              ) |
(*,*, c,150,_*,dx,     lc,i, m,*    ) -> (*, *,              *,    3,c,              149,  m, dx,     i, lc,    *,*, *, *,*, *              ) |
(i,_*,c,n,  Qu,dx,     0, im,*,*    ) -> (im,(addCoord c dx),(n-1),3,(addCoord c dx),(n-1),Qu,<<0,0>>,i, 4,     *,*, *, *,*, *              ) |
(*,1, c,n,  Qu,dx,     4, im,*,dx_in) -> (*, *,              *,    3,(addCoord c dx),n,    Qu,dx_in,  im,3,     *,*, *, *,im,(addCoord c dx)) |
(*,2, c,n,  Qu,dx,     4, im,P,*    ) -> (*, *,              *,    1,c,              n,    Qu,dx,     im,4,     P,Qu,im,c,*, *              ) |
(*,3, c,n,  Qu,dx,     4, im,*,*    ) -> (*, *,              *,    2,c,              n,    Qu,dx,     im,4,     *,*, *, *,im,c              ) |
(i,3, c,n,  Qu,<<0,0>>,lc,im,*,*    ) -> (im,c,              (n-1),3,c,              (n-1),Qu,<<0,0>>,i, 4,     *,*, *, *,*, *              ) |
(*,1, c,n,  Qu,dx,     lc,im,*,dx_in) -> (*, *,              *,    3,(addCoord c dx),n,    Qu,dx_in,  im,(lc-1),*,*, *, *,im,(addCoord c dx)) |
(*,2, c,n,  Qu,dx,     lc,im,P,*    ) -> (*, *,              *,    1,c,              n,    Qu,dx,     im,lc,    P,Qu,im,c,*, *              ) |
(*,3, c,n,  Qu,dx,     lc,im,*,*    ) -> (*, *,              *,    2,c,              n,    Qu,dx,     im,lc,    *,*, *, *,im,c              );


wire loopbox
    (load_im.im',
     loopbox.state',loopbox.cen' initially init_cen ,loopbox.num_left initially 151,loopbox.Qu',loopbox.dx initially <<0,0>>,loopbox.loopcount'
initially 4,loopbox.cur_im',
     norm_model.norm_out,compute_dx.dx)

     (writebox.im,writebox.pos trace,writebox.num trace,
      loopbox.state trace,loopbox.cen , loopbox.num_ims,loopbox.Qu,loopbox.old_dx ,loopbox.cur_im,loopbox.loopcount,
      do_weights.pu_in,do_weights.qu_in,do_weights.im_in,do_weights.cen_in,
      model_update.im_in,model_update.cen_in);

--the output box, draws a dot on the centre
--note the state to write a '\0' for closing a file if output's to file.
box writebox
in (im::image,pos::coord,num::Int)
out (im'::image)

match
(_,_,0) -> ('\0') |
(i,p,_) -> ((update2d i (p@2) (p@1) <<0,255,0>>));

wire writebox
     (loopbox.im',loopbox.cen_out,loopbox.frames_left)
     (outerror);

-- this box creates the new model, passes it on to be normaliseed
```

```
box model_update
in (cen_in::coord, im_in::image,
    kernel::bigvf, frame::image, cen::coord, x::Int, y::Int, the_model::model)
out (kernel'::bigvf,frame'::image,cen'::coord,x'::Int,y'::Int,the_model'::model,
     model_out::model)
match
(c,im,_*,_*,_*,_*,  _*,  _*) -> (theKern,im,c,x_size,y_size,        theEmpty,                                                  *) |
(*,*, *, *, *, (-1),(-1),* ) -> (*,      *, *,(-1),  (-1),          *,                                                         *) | --the accepting state
(*,*,_*,_*,_*, 0,   _*,  m ) -> (*,      *, *,(-1),  (-1),          *,                                                         m) | --the output state
(*,*,k, f, c,  x,   0,   m ) -> (k,      f, c,(x-1), (length (k@x)),m,                                                         *) |
(*,*,k, f, c,  x,   y,   m ) -> (k,      f, c, x,    (y-1),         (addToModel (findBin( getP f x y c)) ((k@x)@y) m),*);


wire model_update
     (loopbox.cen_m , loopbox.im_m,
      model_update.kernel',model_update.frame',model_update.cen',model_update.x',model_update.y',model_update.the_model')

     (model_update.kernel,model_update.frame,model_update.cen,model_update.x,model_update.y,model_update.the_model,
      modelsum.m_in);


------------------------------------------------------------------------
--sums the model to pass it to the below normalising box
box modelsum
in (m_in::model,
    m::model,acc::Float,size::Int)
out (m'::model,acc'::Float, size'::Int,
     acc_out :: Int,m_out::model)
match
(m,_*,_*, _* ) -> (m,0.0,           NUMBINS,*, *) |
(*,*, 0.0,0  ) -> (*,0.0,           0,      *, *) | --accepting state
(*,m, ac, 0  ) -> (*,0.0,           0,      ac,m) | --final state
(*,v, ac, pos) -> (v,(ac + (v@pos)),(pos-1),*, *);

wire modelsum
     (model_update.model_out,
      modelsum.m',modelsum.acc' initially 0.0,modelsum.size' initially NUMBINS )

     (modelsum.m,modelsum.acc,modelsum.size,
      norm_model.sum_in trace,norm_model.m_in);
------------------------------------------------------------------------
--normalises the model

box norm_model
in (m_in::model,sum_in::Float,
    m::model,sum::Float,size::Int)
out (m'::model,sum'::Float, size'::Int,
     norm_out ::model)
match
(m,s,_*,_*, _* ) -> (m,                           s,   NUMBINS,*) |
(*,*,*, 0,  0  ) -> (*,                           0,   0,      *) | --accepting state
(*,*,v, _*, 0  ) -> (*,                           0,   0,      v) | --final state
(*,*,v, sum,pos) -> ((update v pos ((v@pos)/sum)),sum,(pos-1), *);
```

```
wire norm_model
     (modelsum.m_out,modelsum.acc_out,
      norm_model.m' ,norm_model.sum',norm_model.size' initially NUMBINS)

     (norm_model.m,norm_model.sum,norm_model.size ,
      loopbox.model_in);

-------------------------------------------------------------------------
--weights box

--helper function, ensures no /0 errors
w_update Pu Qu frame cen x y w = if ((Pu@(findBin(getP frame x y cen))) == 0.0)
                                  then update2d w x y 0.0
                                  else update2d w x y (sqrt((Qu@(findBin(getP frame x y cen)))/(Pu@(findBin(getP frame x y cen)))));

box do_weights
in (pu_in::model, qu_in::model, im_in::image, cen_in::coord,
    weights::bigvf, Pu::model, Qu::model, frame::image, cen::coord, x::Int, y::Int)

out (weights'::bigv,Pu'::model,Qu'::model,frame'::image,cen'::coord,x'::Int,y'::Int,
     weights_out::bigvf)
match
(p,q,i,c,*,*, *, *,    *,  _*,  _*  ) -> ((vecdef2d x_size y_size),        p, q, i,    c,  x_size,y_size,        *) |
(*,*,*,*,*,*, *, *,    *,  (-1),(-1)) -> (*,                               *, *, *,    *,  (-1),  (-1),          *) | --the terminating state
(*,*,*,*,w,_*,_*,_*,   _*, 0,   _*  ) -> (*,                               *, *, *,    *,  (-1),  (-1),          w) | --the output state
(*,*,*,*,w,Pu,Qu,frame,cen,x,   0   ) -> (w                              ,Pu,Qu,frame,cen,(x-1), (length (w@x)),*) | --finished a column
(*,*,*,*,w,Pu,Qu,frame,cen,x,   y   ) -> ((w_update Pu Qu frame cen x y w),Pu,Qu,frame,cen,x,     (y-1),         *); --the state that updates

wire do_weights
     (loopbox.Pu_out,loopbox.Qu_out,loopbox.im_w,loopbox.cen_w,
      do_weights.weights',do_weights.Pu',do_weights.Qu',do_weights.frame',do_weights.cen',do_weights.x',do_weights.y')

     (do_weights.weights,do_weights.Pu,do_weights.Qu,do_weights.frame,do_weights.cen,do_weights.x,do_weights.y ,
      v2dmm_mul.w_in);
-------------------------------------------------------------------------

--do the mix-map (mul)

box v2dmm_mul
in (w_in::bigvf,
    v1::bigvf,v2::bigvf,xdim::Int,ydim:: Int,result::bigvf)
out (v1'::bigvf,v2'::bigvf,xdim'::Int,ydim'::Int,result'::bigvf,
     result_out::bigvf)
match
(w,*, *, _*,  _*,  *     ) -> (w, theDeriv,x_size,y_size,         w,                                               *     ) |
(*,*, *, (-1),(-1),*     ) -> (*, *,       (-1),  (-1),           *,                                               *     ) | --final state
(*,_*,_*,0,   _*,  result) -> (*, *,       (-1),  (-1),           *,                                               result) | --detect when all
elements are processed
(*,v1,v2,x,   0,   result) -> (v1,v2,      (x-1), (length (v1@x)),result,                                          *     ) | --reached the bottom of a
column
(*,v1,v2,x,   y,   result) -> (v1,v2,      x,     (y-1),          (update2d result x y (mul ((v1@x)@y)((v2@x)@y)) ),*     );--hmm, uses update - urg.
```

```
wire v2dmm_mul
     (do_weights.weights_out,
      v2dmm_mul.v1',v2dmm_mul.v2',v2dmm_mul.xdim',v2dmm_mul.ydim' ,v2dmm_mul.result')

     (v2dmm_mul.v1,v2dmm_mul.v2,v2dmm_mul.xdim,v2dmm_mul.ydim ,v2dmm_mul.result,
      splitter.a);
-------------------------------------------------------------------------
--a box to split the output

box splitter
in (a::bigvf)
out (b::bigvf,c::bigvf)
match
(a) -> (a,a);

wire splitter
     (v2dmm_mul.result_out)

     (v2dsum.v_in,
      v2dmm_mulc.w_in);
-------------------------------------------------------------------------

--a box to do the sum
box v2dsum

in (v_in::bigvf,v::bigvf,acc::Float,xdim::Int, ydim::Int)
out (v'::bigvf,acc'::Float,xdim'::Int, ydim':: Int,acc_out::Float)
match
(v,*, _*, _*,  _*  ) -> (v, 0.0,             x_size,y_size,        *  ) |
(*,*, *,  (-1),(-1)) -> (*, *,               (-1),  (-1),          *  ) |
(*,_*,acc,0,   _*  ) -> (*, *,               (-1),  (-1),          acc) |
(*,v, acc,x,   0   ) -> (v, acc,             (x-1), (length (v@x)),*  ) |
(*,v, acc,x,   y   ) -> (v,(acc + ((v@x)@y)),x,     (y-1),         *  );

wire v2dsum
     (splitter.b,
      v2dsum.v',v2dsum.acc' initially 0.0,v2dsum.xdim',v2dsum.ydim')

     (v2dsum.v,v2dsum.acc,v2dsum.xdim,v2dsum.ydim,
      compute_dx.s);

-------------------------------------------------------------------------
--the mul-coord mixmap


box v2dmm_mulc
in (w_in::bigvf,
    v1::bigvf,v2::bigvc,xdim::Int,ydim:: Int,result::bigvf)

out (v1'::bigvf,v2'::bigvc,xdim'::Int,ydim'::Int,result'::bigvf,
     result_out::bigvc)
match
```

```
(w,*, *, _*,  _*,  *     ) -> (w, theWin,x_size,y_size,         w,                                                 *     ) |
(*,*, *, (-1),(-1),*     ) -> (*, *,     (-1),  (-1),           *,                                                 *     ) | --final state
(*,_*,_*,0,   _*,  result) -> (*, *,     (-1),  (-1),           *,                                                result) | --detect when all
elements are processed
(*,v1,v2,x,   0,   result) -> (v1,v2,    (x-1), (length (v1@x)),result,                                            *     ) | --reached the bottom
of a column
(*,v1,v2,x,   y,   result) -> (v1,v2,    x,     (y-1),          (update2d result x y (mulCoord ((v2@x)@y)((v1@x)@y)) ),*     );--hmm, uses update -
urg.


wire v2dmm_mulc
     (splitter.c,
      v2dmm_mulc.v1',v2dmm_mulc.v2',v2dmm_mulc.xdim',v2dmm_mulc.ydim' ,v2dmm_mulc.result')

     (v2dmm_mulc.v1,v2dmm_mulc.v2,v2dmm_mulc.xdim,v2dmm_mulc.ydim ,v2dmm_mulc.result,
      cv2dsum.v_in);

------------------------------------------------------------------------
--the box to sum the 2d vector of co-ords

box cv2dsum
in (v_in::bigvc,
    v::bigvc,acc::coord,xdim::Int, ydim:: Int)

out (v'::bigvc,acc'::coord,xdim'::Int, ydim':: Int,
     acc_out::coord)
match
(v,*, *,  _*,  _*  ) -> (v,<<0,0>>,                 x_size,y_size,        *  ) |
(*,*, *,  (-1),(-1)) -> (*,*,                       (-1),  (-1),          *  ) |
(*,_*,acc,0,   _*  ) -> (*,*,                       (-1),  (-1),          acc) |
(*,v, acc,x,   0   ) -> (v,acc,                     (x-1), (length (v@x)),*  ) |
(*,v, acc,x,   y   ) -> (v,(addCoord acc ((v@x)@y)),x,      (y-1),        *  );

wire cv2dsum
     (v2dmm_mulc.result_out,
      cv2dsum.v',cv2dsum.acc' ,cv2dsum.xdim',cv2dsum.ydim')

     (cv2dsum.v,cv2dsum.acc,cv2dsum.xdim,cv2dsum.ydim,
      compute_dx.c);
------------------------------------------------------------------------
--compute the dx value - nothing to do but a division

box compute_dx
in (c::coord,s::Float)
out (dx::coord)
match
(c,s) -> (divCoord c s);

wire compute_dx
     (cv2dsum.acc_out,
      v2dsum.acc_out)

     (loopbox.dx_in);
```