



Image processing for CAL

F21DG 2016-2017: Design and Code Project

Ross Brunton - H00128171

Teymoor Rasheed - H00128887

Mark Young - H00157656

Calum Andrew John Turner - H00125390

Supervisors

Prof Greg John Michaelson

Dr Robert Stewart

Dr Paulo Garcia

Section 1: Background	3
Our Project	3
Dataflow Model	3
ORCC	3
CAL (CAL Actor Language)	4
Section 2: Low Level Image Processing Algorithms	6
Addition	6
Addition Actor	6
Subtraction	7
Subtraction Actor	7
Multiplication	8
Multiplication Actor	8
Logical Operators	9
Logical Actors	9
Division	10
Division Actor	10
Contrast Stretching	11
Actor Structure	11
Cache Actor	11
Contrast Stretch Actor	12
MaxMin Actor	12
Intensity Histogram	13
Intensity_histogram Actor	13
Reflect	14
Reflect Actor	14
Rotate	15
Rotate Actor	15
Translate	16
Translate Actor	16
Connected Components Labelling	17
Actor Diagram (connected_components)	18
Above Actor	18
Double Cache Actor	18
Label Actor	19
Replace Actor	19
Reduce Actor	20
Actor Performance Overview	20
Section 3: Histogram of Oriented Gradient	22
Section 4: HOG implementation	24
8x8 Cell Tiler	24

Tiler Actor	24
Gradient	25
Histogram Binning	26
Block Generation	27
Block Normalization	28
Section 5: Future Steps	29
Sliding Window for the entire image	29
Gradient before cells	29
Integrating the Algorithm into an SVM	29
Implement remaining image processing algorithms	29
References	31
Appendix A: Code Sample Listing	32

Section 1: Background

Our Project

The project was undertaken by four 5th year MEng Software Engineering students at Heriot-Watt university. As a group project for the final year, The objective was to use ORCC and CAL to implement several image processing algorithms.

For the first half of the project, the group independently worked on several smaller algorithms in order to familiarise themselves with the tools they were using. After this, they then worked collaboratively on implementing a Hog Descriptor generation system.

Dataflow Model

A traditional program consists of a single "control flow" usually located at a single point in memory which advances forward one instruction at a time. This is the model that languages such as Java or C use, where a single line of code is executed at a time before moving onto the next.

A dataflow computation, on the other hand, breaks the algorithm down into multiple "actors", each of which has it's own "execution state" (Sousa, 2012). That is, each actor acts completely independently from other actors and multiple actors can run code at the same time.

Actors have input and output streams, which are joined to other actors allowing them to communicate. This results in a graph, where the streams are the edges and the actors are the nodes. The main computation in an actor is done by reading some number of values from inputs, possibly updating an internal state, and then writing some number of values to outputs. By joining many actors like this, more and more complicated algorithms can be created.

The primary advantage of this model is that actors can be distributed across multiple computation nodes easily, allowing a high degree of parallelism. Each actor can do its own thing, without the need for locks. Different actors can even be located on different physical machines, provided they can communicate over a network.

The dataflow model is incredibly useful when the developer wants to implement an algorithm on an FPGA (Pell and Mencer, 2011). FPGA designs generally sacrifice the functionality of high performance CPUs in favour of smaller, purpose built microprocessors. Since an individual actor contains very simple logic and only needs to communicate using well defined connections, it is incredibly well suited for FPGAs.

ORCC

Orcc(Orcc, 2014) is an open source development tool for Eclipse whose main purpose is to aid developers with dataflow infrastructure. Orcc makes it easy for developers to compile dataflow descriptors to hardware instructions or other programming languages such as C. Orcc is an extension of the Eclipse Xtext SDK for CAL features and with the Graphiti SDK library provides a high level diagram editor. Orcc therefore is dependent these two Eclipse libraries. The Orcc compiler is a source-to-source compiler that will compile the data flow descriptors into other languages source code rather than generating machine code.

Orcc provides graph editing tools that make it quick and easy to create dataflow networks. This interface makes it easy to understand how actors are connected as they can often have multiple inputs and outputs which would be difficult to capture as text.

Other useful features of Orcc are the debugger and text assistance. It provides useful debugging for dataflow programs and a simulator so you can test your programs within Eclipse. The text assistance is based on the Xtext library and aids developers with syntax and error highlighting.

The main backend and compiler output for Orcc is C. As C is the main backend it has had the most time spent on it and the most features. Orcc also supports Distributed Application Layer (DAL) and Transport-Trigger Architecture (TTA) backends.

CAL (CAL Actor Language)

CAL is an actor language created in 2001 as a part of the Ptolemy II (Davis et al., 2001) project at UC Berkeley. The concept of actors was first introduced by Carl Hewitt (1977) as a way to model distributed knowledge-based algorithms. In CAL an actor is a computational entity with input ports, output ports, states and parameters.

CAL is a programming language designed for writing (dataflow) actors, which are stateful operators that transform input streams of data objects (tokens) into output streams. CAL has been compiled to a variety of target platforms, including single and multicore processors and programmable hardware.

CAL has been used in several application areas such as Video encoding Frameworks (Bhattacharyya et al., 2011), cryptography (Ahmad et al., 2012) and in this case image processing.

Actors perform their computation in a sequence of steps called 'firings'. In each of those steps, the actor may:

- Consume tokens from its input ports.
- Modify its internal state.
- Produce tokens at its output ports.

Consequently describing an actor involves:

- Describing its interface to the outside.
- The ports and their input types.
- The structure of its internal state, as well as the steps it can perform.
- What these steps do (in terms of token production and consumption, the update of the actor state), and how to pick the step that the actor will perform next.

Example:

```
actor Example () In ==> Out :  
  action In: [x] ==> Out: [x] end  
end
```

The first line declares the actor name, followed by a list of parameters (which is empty, in this case), and the declaration of the input and output ports. In front of the ==> sign there are input ports (only one port named In, in the example) and the output port is after it (in this example there is only one port named Out).

The second line defines an action. Actions describe what process happens during the step that an actor takes, and therefore steps consist of executing an action. Actors can have any number of actions but in this example `Example` only has one (`action In: [a] ==> Out: [a] end`).

When an actor takes a step, it may consume input tokens and produce output tokens. The action in `Example` demonstrates how to specify token consumption and production. The parts in front of the `==>`, which are called input patterns, specify how many tokens to consume and from which ports, and what to call those tokens in the rest of the action. There is one input pattern in this action, `In:[x]`. In the example one token is to be read (consumed) from input port `In`, and the token is to be called `x` in the rest of the action. Such an input pattern also defines a condition that must be met for this action to fire. If the required token is not present, this action will not be executed.

On the output side of an action following the `==>` sign, the output expressions define the number and values of the output tokens that will be produced on each output port by each firing of the action.

Detailed information about CAL can be found here:

<https://embedded.eecs.berkeley.edu/caltrop/docs/LanguageReport/CLR-1.0-r1.pdf>

Section 2: Low Level Image Processing Algorithms

During the first part of the project, a number of algorithms were implemented by individual group members in order to gain experience with the toolchain.

The following sections describe the algorithms that have been worked on, as well as the actors that make the algorithms up.

For each actor, a list is given of the actions that can fire, in priority order, as an itemized list.

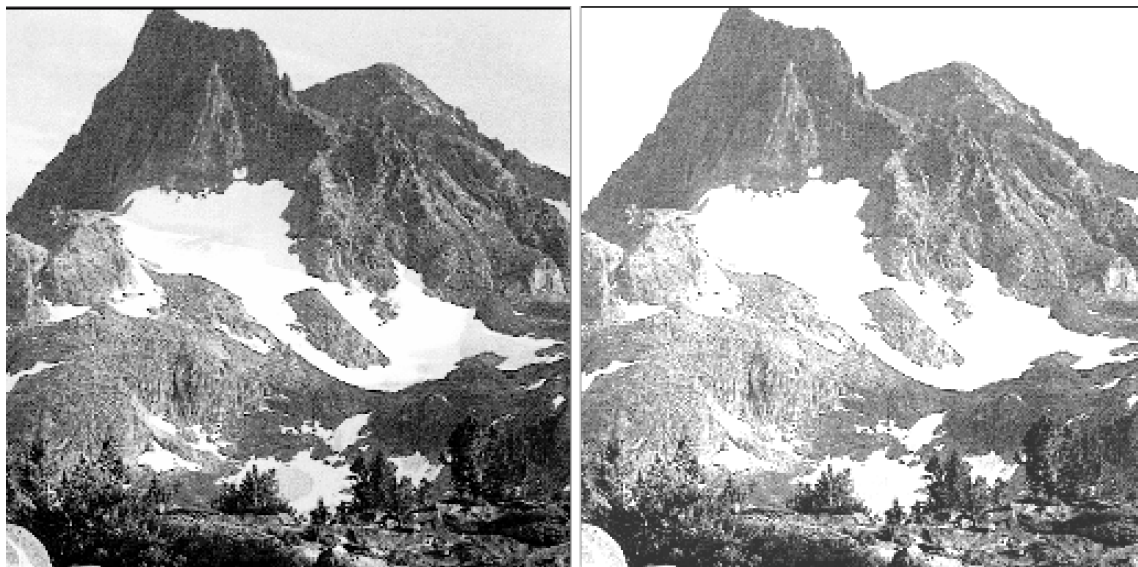
Each actor can exist in one of a number of states, given in the actor description. Each state has its own set of actions. Note that some actors do not use states, and thus no state is listed for them.

For some actions, the code is provided under the respective action, unless the action is trivial or significantly complex. All the source files are located at the end of this report.

While implementing these algorithms, materials created by R. Fisher, S. Perkins, A. Walker and E. Wolfart (2003) were used as a reference.

The example image is from <https://homepages.cae.wisc.edu/~ece533/images/>.

Addition



This operator takes in two inputs and produces an output of the addition of the inputs. In the above example the first image is the original and the second image is the result of addition of 50 to every pixel value.

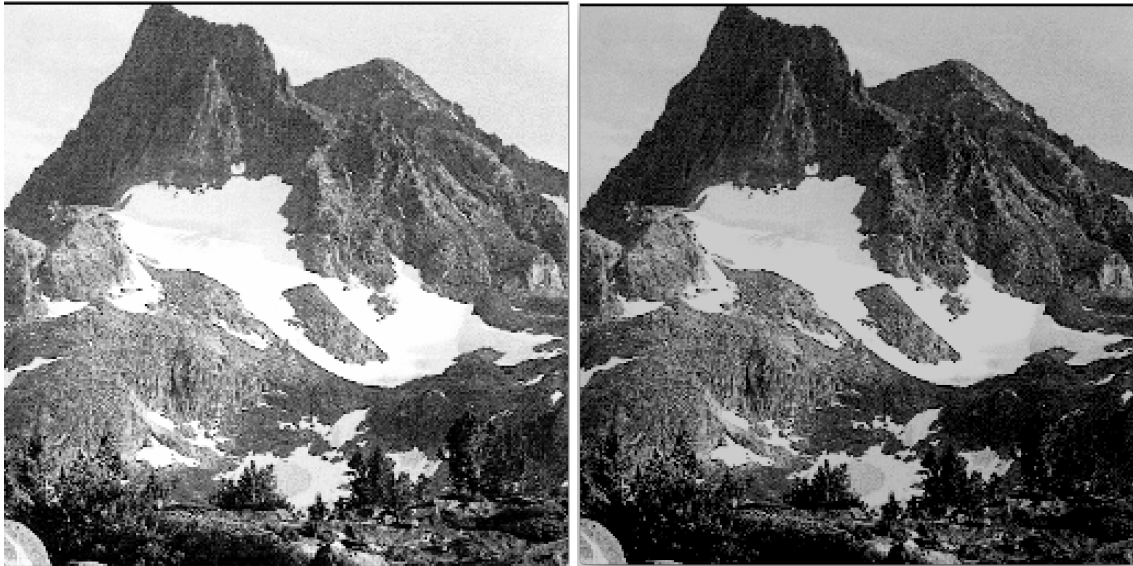
Addition Actor

```
actor addition()  
uint(size=8) Gin1, uint(size=8) Gin2  
==> uint(size=8) Gout
```

Actions (actor has no states):

- Read a value from both `Gin1` and `Gin2`, adds the values together, and outputs the result to `Gout`. If the value of the addition is above 255 then the output is 255.

Subtraction



This operator takes in two inputs and produces an output of the subtraction of the second input from the first. In the above example the first image is the original, the second image is the result of subtracting 50 from every pixel value.

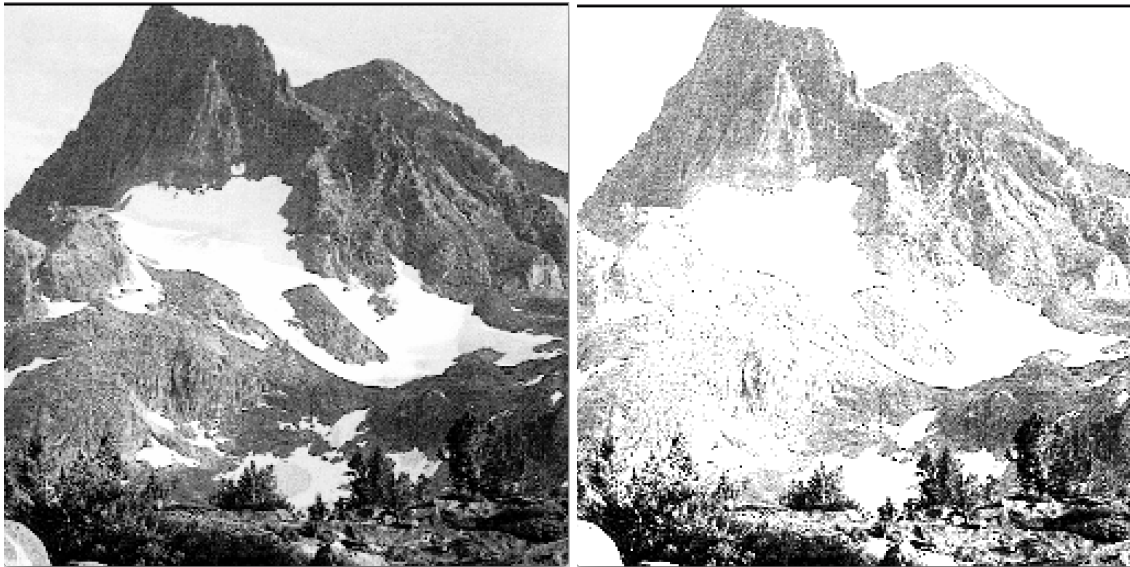
Subtraction Actor

```
actor subtraction()  
uint(size=8) Gin1, uint(size=8) Gin2  
==> uint(size=8) Gout
```

Actions (actor has no states):

- Read a value from both `Gin1` and `Gin2`, subtract the values, and outputs the result to `Gout`. If the subtraction would result in less than 0 then 0 is output.

Multiplication



This actor takes in two values; in the case above it was pixels of the source (left) image file, and the constant 2. Those two values are multiplied together and then output. This "brightens" the image as seen above.

Multiplication Actor


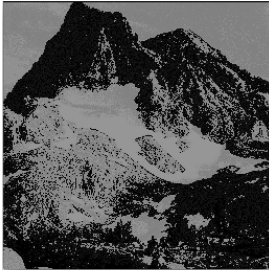
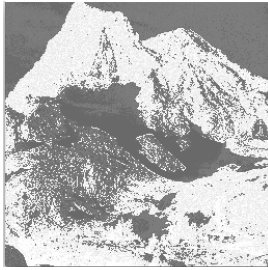
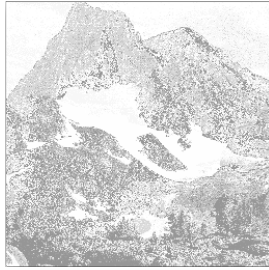
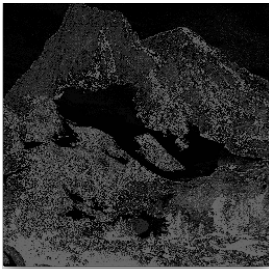
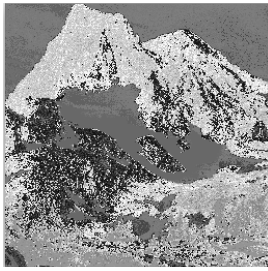
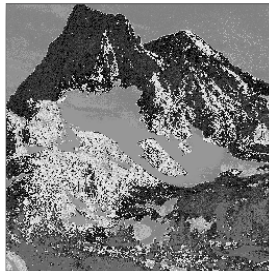
```
actor multiply()  
uint(size=8) Gin, uint(size=8) Factor  
==> uint(size=8) Gout
```

Actions (actor has no states):

- Read a value from both `Gin` and `Factor`, multiply them together, and output that value to `Gout`. If the calculated value would be greater than 255, output 255 instead.

```
action Gin:[x], Factor:[y] ==> Gout:[  
    if (y != 0) && x >= (255 / y)  
    then 255  
    else (x * y) end  
]
```

Logical Operators

			
	AND	NAND	OR
			
Original	NOR	XOR	XNOR

The logical operators take two images as input, and outputs a third image whose pixel values are just those of the first image applied with the specified bitwise operator to the corresponding pixels from the second. In the above example, each pixel was the first input and 150 was the second resulting in the output images.

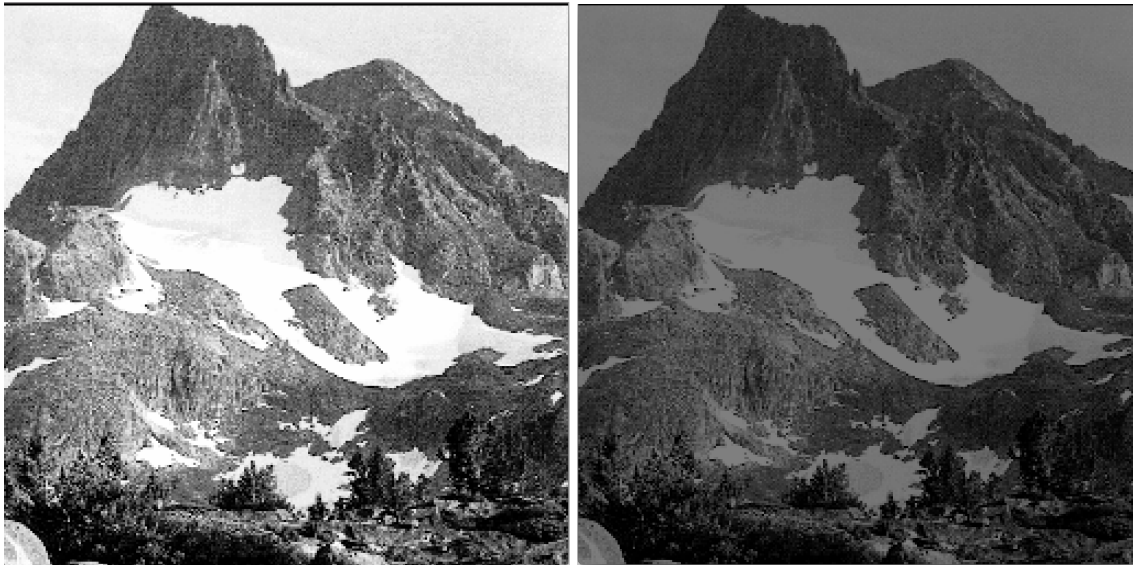
Logical Actors

```
actor OP()  
uint(size=8) Gin1, uint(size=8) Gin2  
==> uint(size=8) Gout
```

Actions (actor has no states):

- Read a value from both `Gin1` and `Gin2`, perform `OP` on them, and output the resulting value to `Gout`.

Division



The image division operator takes two identically sized images as input and produces a third image of the same size as the inputs whose pixel values are the pixel values of the first image divided by the corresponding pixel values of the second image.

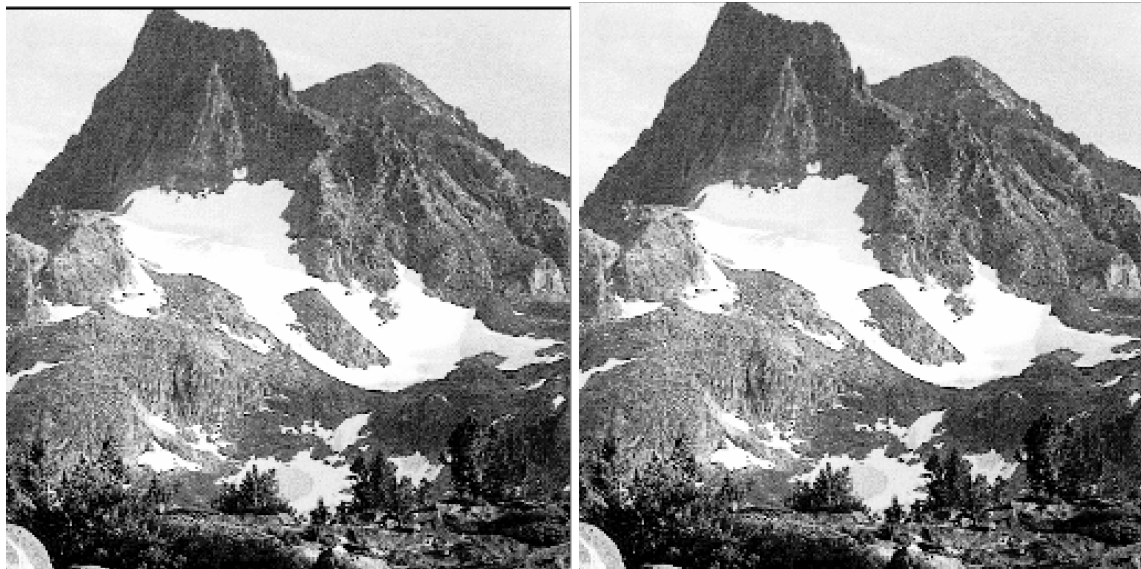
Division Actor

```
actor Division()  
uint(size=8) Gin1, uint(size=8) Gin2  
==> uint(size=8) Gout
```

Actions (actor has no states):

- Read a value from both `Gin1` and `Gin2`, divide them, and output that value to `Gout`.

Contrast Stretching



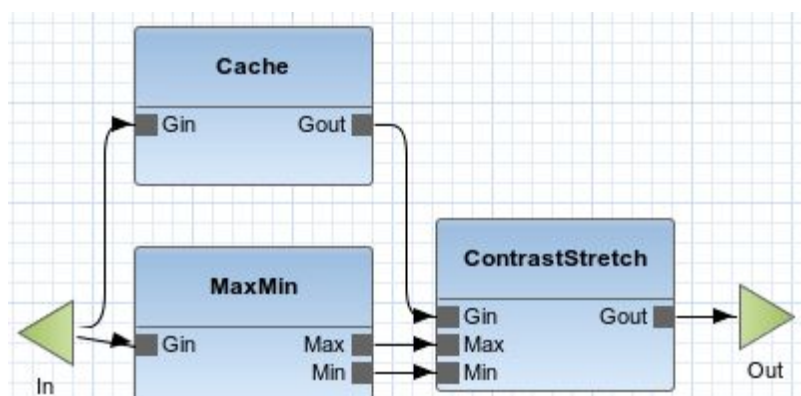
Possible values for the pixels are in the range from 0 to 255 (which determine brightness).

However, not all images use every value in this range, especially if they are very bright or very dark. What this actor does is take in the maximum and minimum pixel value from the image, and then image data. The pixels in the image have their values "stretched" such that instead of being in the range `[minimum:maximum]` they are in the range `[0:255]`.

The minimum and maximum values can be generated using a dedicated actor.

The end result of this is that the range of contrast is increased, making the images look like they have more depth. It is hard to see this in the image above, especially if it were printed out.

Actor Structure



Cache Actor

```
actor Cache()
```

```
uint(size=8) Gin
==> uint(size=8) Gout
```

This actor functions as a FIFO buffer large enough to store the image.

Actions (actor has no states):

- If there is space in the buffer, read from `Gin`, store it in the buffer and increment the write pointer.

```
action Gin:[x] ==>
guard start != last-1 && not (start = entries -1 && last = 0)
do
    store[start] := x;
    start := start + 1;
    start := start mod entries;
end
```

- If there is data in the buffer, read from the buffer and output that value to `Gout`, then increment the read pointer.

```
action ==> Gout:[out]
guard last != start
do
    out := store[last];
    last := last + 1;
    last := last mod entries;
end
```

Contrast Stretch Actor

```
actor ContrastStretch()
uint(size=8) Gin, uint(size=8) Max, uint(size=8) Min
==> uint(size=8) Gout
```

init (initial) state actions:

- Read a single value from `Max` and `Min` into `max` and `min`, set `p` to 0, then transition to the read state.

read state actions:

- When `p` is equal to or greater than the number of pixels in the image, transition to the init state.
- Read a single value from `Gin`, perform the contrast stretching algorithm using the `max` and `min` values, then output the output to `Gout`. Then increment `p`.

```
proc: action Gin:[x] ==> Gout:[
    (x - imageMin) * ((max - min) / (imageMax - imageMin)) + min]
do
    p := p + 1;
end
```

MaxMin Actor

```
actor MaxMin()
uint(size=8) Gin
==> uint(size=8) Max, uint(size=8) Min
```


Reads in an entire image and then outputs the maximum and minimum pixel values.

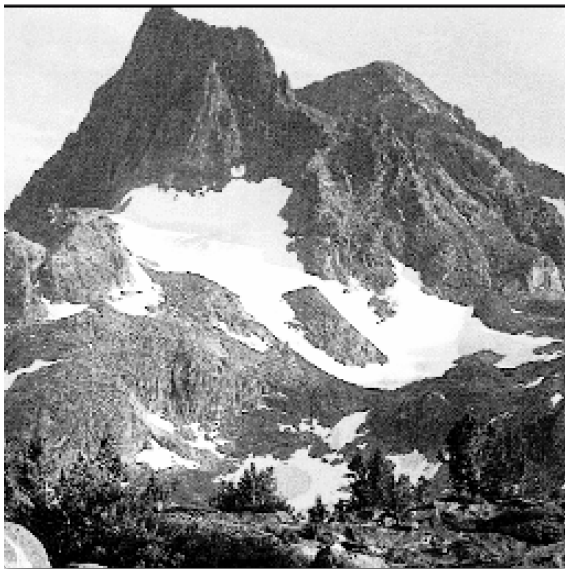
read (initial) state actions:

- If the entire image has been read, transition to the write state.
- Read an image pixel from `Gin`. If it is larger than `max` or smaller than `min`, set the appropriate value to the pixel value.

write state actions:

- Write `max` to `Max`, `min` to `Min`, reset all values and then transition to the read state.

Intensity Histogram



```
Bin 25 - 0
Bin 26 - 7
Bin 27 - 22
Bin 28 - 28
Bin 29 - 63
Bin 30 - 93
Bin 31 - 312
Bin 32 - 0
Bin 33 - 224
Bin 34 - 306
Bin 35 - 420
Bin 36 - 508
Bin 37 - 607
```

The Intensity histogram actor takes in all pixels of an image and creates a histogram of frequency of pixel values. There is a bin for each 256 pixel values. Once the entire image has been consumed the actor will then output the histogram, streaming each bins value from bin 0 to bin 255. The actor then resets itself before processing another image.

Intensity_histogram Actor

```
actor intensity_histogram ()
uint(size=8) Gin1
==> uint(size=16) Gout
```

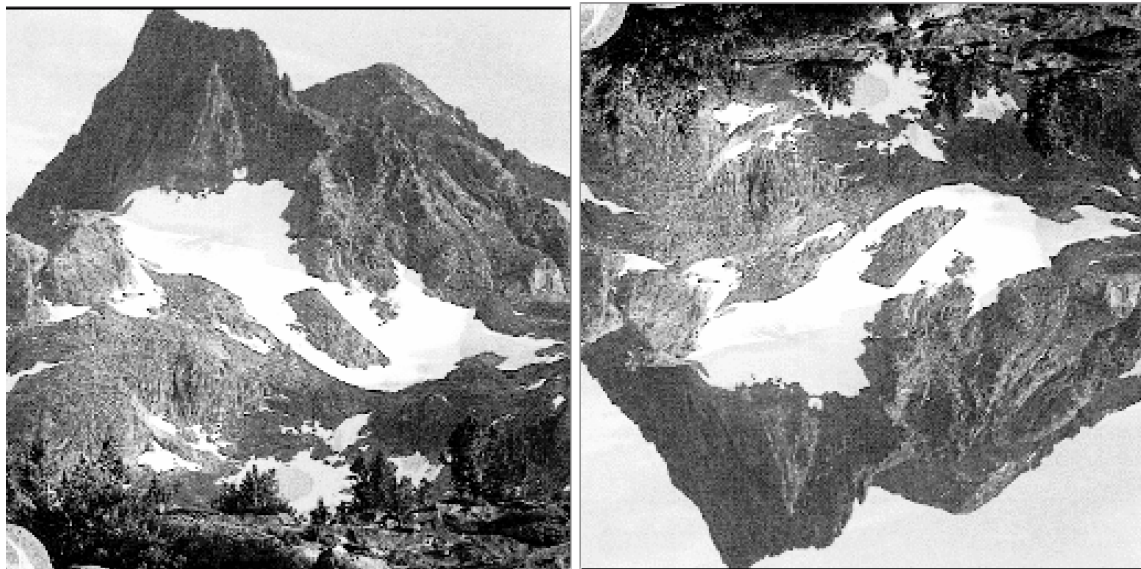
Initial state:

- Resets algorithm counters to 0, then move to the read state.

Read state:

- Sets all the 256 values within the histogram array to 0.
- For each pixel in the image, increment the pixel value bin by 1.
- Once the entire image has been processed, output each of the 256 histogram bin values.
- After all the bins have been output return to the init state.

Reflect



The reflection operator geometrically transforms an image such that image elements, i.e. pixel values, located at position $(x1,y1)$ in an original image are reflected entirely horizontally or vertically depending on the chosen orientation, which can be changed by modifying the constant “orientation”.

Reflect Actor

```
actor Reflect()  
uint(size=8) Gin ==> uint(size=8) Gout
```

Read state (initial) actions:

- Starting in the read state, it reads all the bytes from from the input image stream and put them into a buffer.

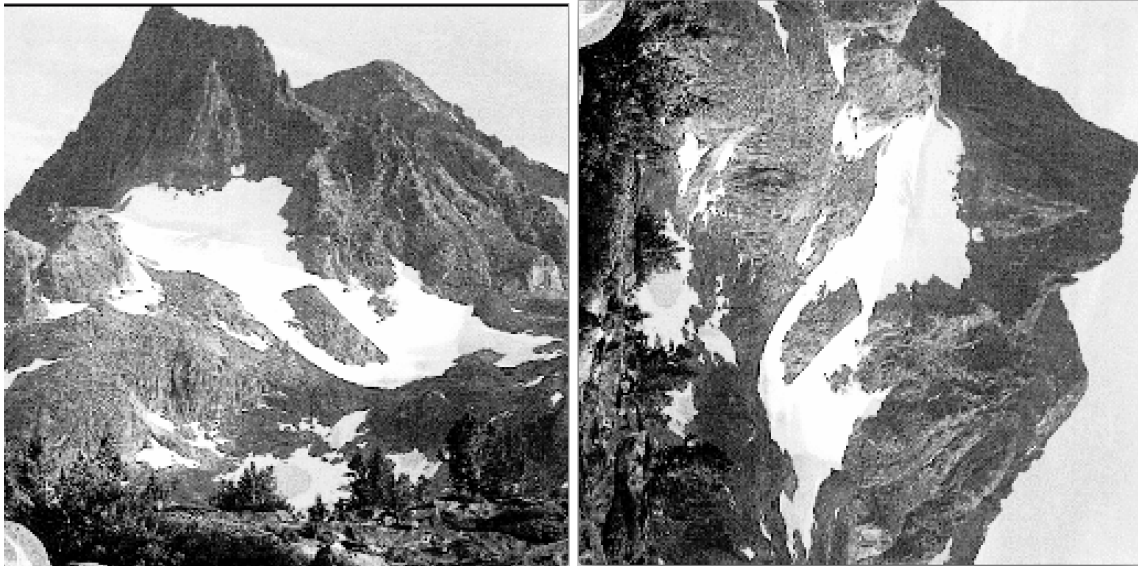
Transition state:

- Upon reading all bytes it transitions to the write state through the “transition” state which resets the `head` position in the buffer to 0.

Write state actions:

- Depending on the value of the chosen orientation, a specific section of the write state is used.
 - If horizontal was chosen then the output of the bytes is in a top to bottom, right to left order.
 - If vertical was chosen then the output of the bytes is in a bottom to top, left to right order.

Rotate



The rotation operator performs a geometric transform which maps the position (x_1, y_1) of a picture element in an input image onto a position (x_2, y_2) in an output image by rotating it through a user-specified (constant) angle θ about an origin O . In our implementation the origin is the center of the image and the angles can only be multiples of 90.

Rotate Actor

```
actor Rotate()  
uint(size=8) Gin ==> uint(size=8) Gout
```

Read state (initial) actions:

- Starting in the read state, the actor reads all the bytes from from the input image stream and put them into a buffer `bufferSize`.

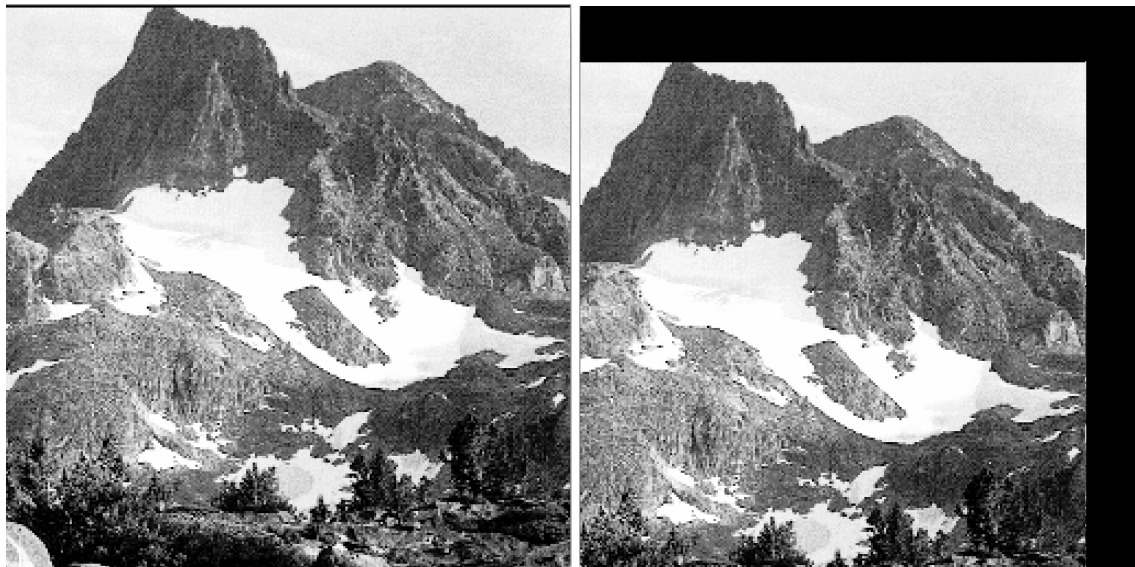
Transition State:

- Upon reading all bytes the actor transitions to the write state through the “transition” state which resets the head position in the buffer to 0.

Write state actions:

- Depending on the value of the chosen `angle` a specific section of the write state is used, each one is a 90 degree bracket.
 - If the angle is ≤ 90 the output image will be rotated 90 degrees clockwise.
 - Else if the angle is ≤ 180 the output image will be rotated 180 degrees clockwise.
 - Else the output image will be rotated 270 degrees clockwise.

Translate



The translate operator performs a geometric transformation which maps the position of each picture element in an input image into a new position in an output image. The image element located at (x_1, y_1) in the original is shifted to a new position (x_2, y_2) in the corresponding output image by displacing it through a user-specified translation (O_x, O_y) .

Translate Actor

```
actor Translate()  
uint(size=8) Gin ==> uint(size=8) Gout
```

Read state (initial) actions:

- Starting in the read state, the actor reads all the bytes from the input image stream and put them into a buffer `bufferSize`.

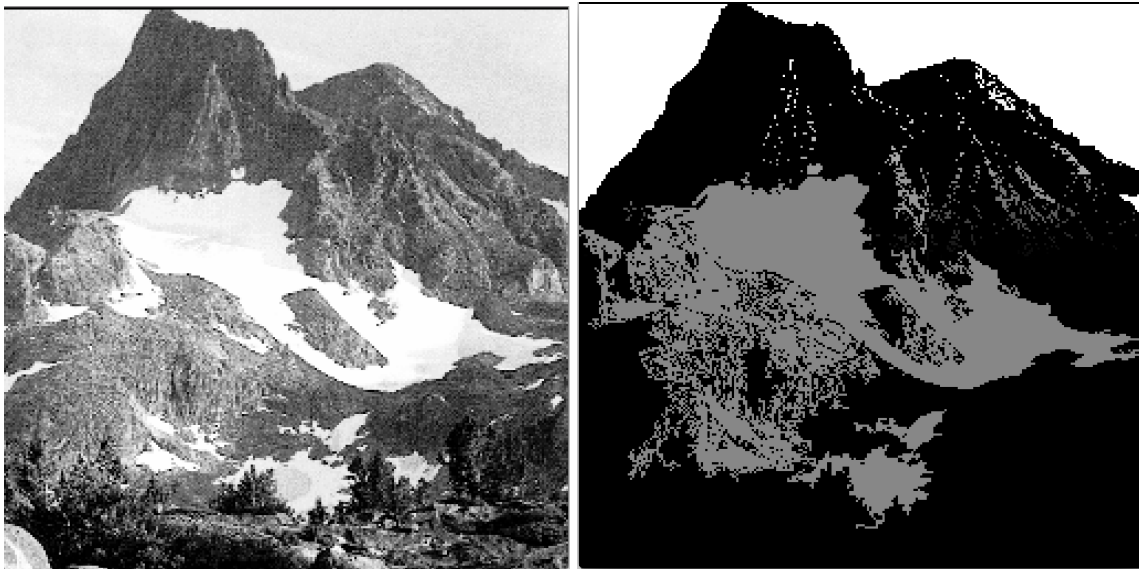
Transition State:

- Upon reading all bytes the actor transitions to the write state through the “transition” state which resets the `head` position in the buffer to 0.

Write state actions:

- In the write state the actor will iterate through the buffer sequentially which would represent top to bottom, left to right in the original image. The index value in the buffer is altered by the given `xOffset` and `yOffset` to find the translated position of the pixel to be output. If the index value is either outside the buffer boundaries, or from the next/previous “row” of pixels in the input image a value of 0 (black) will be output.

Connected Components Labelling



There are several actors handling this algorithm, acting like a pipeline.

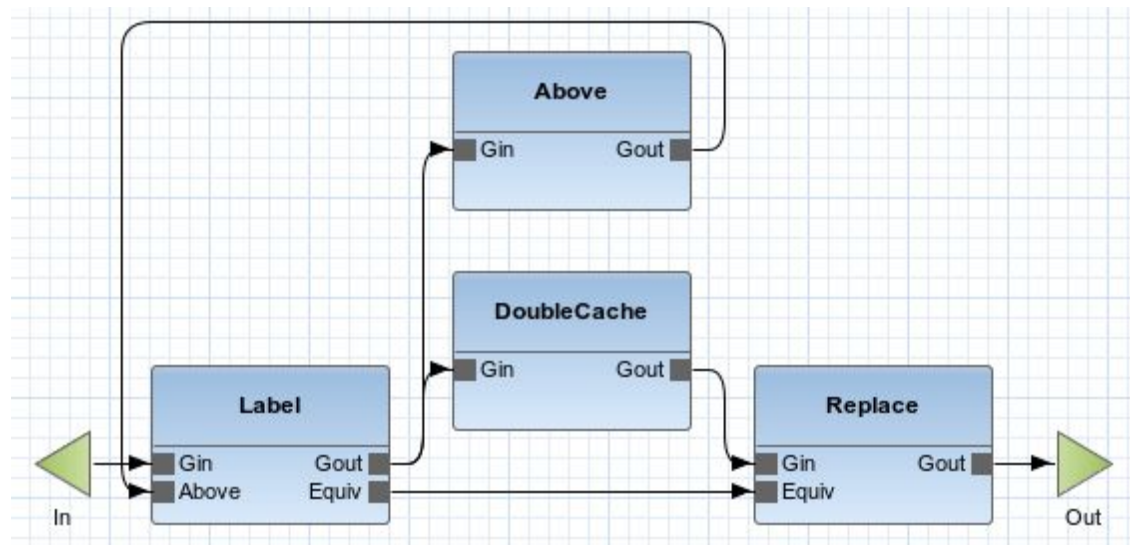
First is a binary threshold actor, which copies the image, changing any pixel values less than a given value to 0. This results in an image where bright areas are preserved, but darker areas are all set to 0. This is not included in the diagram below. This is fed into the main `connected_components` actor.

The job of the actor is to find which of these "light areas" are connected. It reads in the image, and outputs a 16 bit value for each pixel. This value has the following properties:

- If it was 0 in the input, it is 0 in the output.
- All pixels that are "connected" have the same output value.
- All pixels with the same output value are connected.

Since the output image has to use 8 bit values, another actor, `Reduce`, is provided which takes the 16 bit classes and maps each value to an 8 bit value. However, this typically isn't enough for most complex images due to the high number of small connected sections. Because of this, in the images above, the algorithm seems to have "given up" halfway through.

Actor Diagram (connected_components)



Above Actor

```
actor Above()  
uint(size=16) Gin  
==> uint(size=16) Gout
```

This actor alternates between reading and writing. It has a single "pointer" which tracks across the image. When reading, it populates an array of the current row. When writing, it outputs the three values above the "pointer". When the pointer goes off the right side of the image, the row being read is switched into the row being written as it goes onto the next line.

write state (initial) actions:

- If the row is equal to the height of the image or the actor has just started up, set `row` and `col` to 0, and empty the arrays `current` and `above`.
- Write the values at `col - 1`, `col` and `col + 1` to `Gout`, in that order, from the `above` array. Then transition into the `read` state.

read state actions:

- Read a value from `Gin`, and store it in `current[row][col]`. Increment `col`. If `col` is greater than the image width, set `col` to 0, increment `row` and copy `current` into `above`. In either case, transition to the `write` state.

Double Cache Actor

```
actor DoubleCache()  
uint(size=16) Gin  
==> uint(size=16) Gout
```

This actor functions as a FIFO pipe large enough to store a version of the image with 16 bit pixels.

Actions (actor has no states):

- If there is space in the buffer, read from `Gin`, store it in the buffer and increment the write pointer.
- If there is data in the buffer, read from the buffer and output that value to `Gout`, then increment the read pointer.

Label Actor

```
actor Label()
uint(size=8) Gin, uint(size=16) Above
==> uint(size=16) Gout, uint(size=16) Equiv
```

Takes in an image, and labels connected segments.

It takes entries from the input, and outputs a value (a "class") such that all points that connect have the same value. In a single pass it cannot fully calculate this, however. This is where the equiv output comes in. This functions as a list of pairs of classes which are touching, for use in the Replace actor.

Actions (actor has no states):

- If there is an entry in the `equiv` stack, pop from it as `x` and output `(last, x)` to `Equiv`
- If there is consumed the full image, reset everything (including the equiv stack and max label) and write `(0, 0)` to `Equiv`.
- Take a value `x` from `Gin`, a triplet `(l, m, r)` from `Above`. If `x` is 0, then set `last` to 0 and output 0 as the only thing this actor does.

Otherwise, between `l, m, r` and `last`, check how many non-zero values exist. Write one of these values to `Gout`, and push the rest onto the equiv stack.

In the case that there are no non-zero values among them, increment the max label value, and output the new value instead.

Set the variable `last` to the value output by this actor.

Replace Actor

```
actor Replace()
uint(size=16) Gin, uint(size=16) Equiv
==> uint(size=16) Gout
```

Takes in an image and a list of equivalent labels, and outputs such that all equivalent classes have the same value. Specifically, it takes a list of equivalence pairs, and groups all of them such that if it read a pair `(x, y)`, then `x` and `y` are in the same group.

Then it reads in "labels" for each pixel of an image, and outputs the index of the group that that label is in.

There is a large chunk of memory, which represents a big block of cons cells (linked lists). There is also an array of "heads" for the buckets. This makes a "bucket" a linked list of all labels which are the same.

replace (initial) state actions:

- If the entire image has been processed, or no action has happened yet for this actor, then zero the bucket head array (making all buckets empty).
- Read in a label value from `Gin`, and search the buckets for a matching label. Then output the ID of that bucket to `Gout`. If no bucket contains this label, then add it to a new bucket and output the id of that bucket to `Gout`.

learn state actions:

- If the transition flag is set, transition to the replace step.
- Read in two values (an equivalence pair) from `Equiv`. If the first value is 0, then set the transition flag with no other effect. Otherwise, find which buckets both values are in, and do the following:
 - If neither are in a bucket, create a new bucket and put them both in it.
 - If exactly one is in a bucket, add the other into that bucket.
 - If both are in the same bucket, do nothing.
 - If both are in different buckets, append one bucket to the other, and clear the old bucket's head.

Reduce Actor

```
actor Reduce()
uint(size=16) Gin
==> uint(size=8) Gout
```

This actor is not included in `connected_components`, since it is not required as part of the algorithm. It is only used to output 8 bit ints instead of 16 bit ones. This actor maps its inputs to its outputs such that the same value always produces the same output, and each output is only produced by a single input.

Actions (actor has no states):

- If enough pixels to have consumed a whole image have been processed, or nothing has happened yet for this actor, forget all the mappings.
- Read a value from `Gin` and look through the mappings. If it does not exist in the mappings, add it to the mappings with the next highest value. Output the mapped value to `Gout`.

Actor Performance Overview

The following executions were ran on an Intel core i5-2390T using Orcc's C backend. This processor has 2 physical cores, with 4 threads via hyperthreading. It runs at a base clock of 2.70GHz, turbo boosting to 3.5GHz.

Execution time was calculated using the UNIX `time` command on the command processing a 512x512 image, and gives the User time.

For memory usage, values are given in bytes and `w` and `h` represent the width and height of the source image. This figure only includes memory allocated by the algorithm itself, and not the framework, code, temporaries and so on. This was calculated by adding together the size of all the variables in the actor.

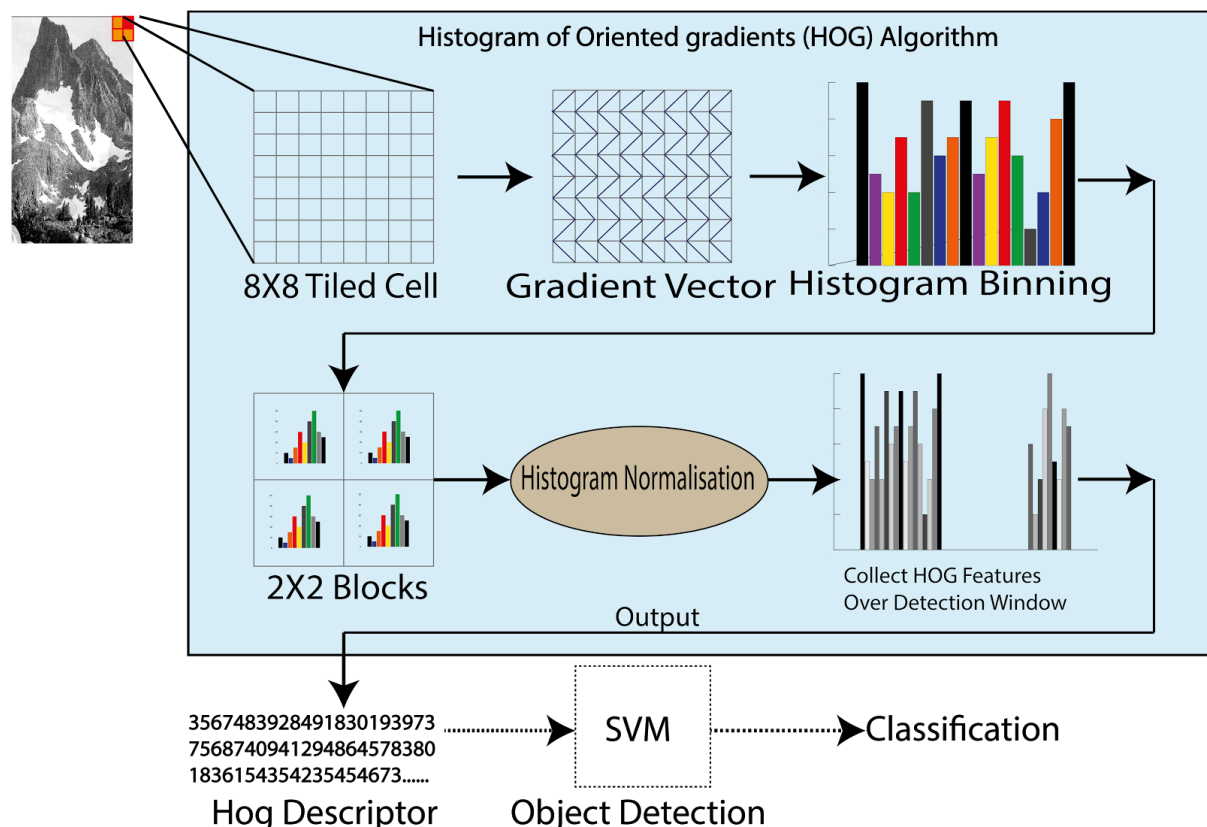
Line counts ignore the initial license documentation, and include the total lines across all actors.

Actor/Algorithm	Lines of code	Memory usage(bytes)	Time(s)
Addition	8	0	0.03s
Subtraction	8	0	0.04s
Multiplication	10	0	0.04s
AND	7	0	0.04s
NAND	7	0	0.03s
NOR	7	0	0.04s
OR	7	0	0.04s
XNOR	7	0	0.04s
XOR	7	0	0.04s
Division	7	0	0.04s
Contrast stretching	108	$50 + (w \cdot h)$	0.04s
Reflect	77	19	0.03s
Rotate	87	22	0.03s
Translate	77	26	0.04s
Connected components	336	$22,567 + (w \cdot 4) + (w \cdot h \cdot 4)$	0.79s
Intensity Histogram	60	524	0.02s

Section 3: Histogram of Oriented Gradient

A histogram of oriented gradients (HOG) is a feature descriptor function used within image processing that can be used to detect objects. It was developed by Navneet Dalal and Bill Triggs (Dalal and Triggs, 2005).

HOG is often used as a person detector and is very popular. The function generalises the image so that the object produces the same descriptor and therefore is easier to classify. HOG classifies an entire object as a global feature, instead of breaking the object down into individual features. This makes the function fairly simple and represents the object as a single feature.



The classification of the descriptor is done by a trained Support Vector Machine (SVM). An SVM is a machine learning algorithm that uses a supervised learning technique. Therefore to create a complete object detection, you first need to train a SVM for the subject domain (ie people). The SVM will classify the descriptor as either the object or not the object.

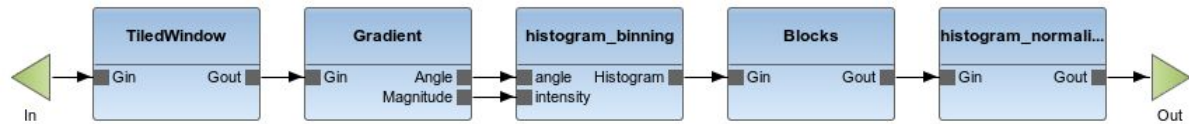
Note that the SVM part of the algorithm was not implemented within this project; it is purely a HOG description generator.

The above image shows the components of HOG and the process of how a HOG descriptor is created.

1. First the image is broken up into 8x8 pixel cells.
2. Gradient vectors for each pixel in the cell are calculated.
3. All the gradient vectors in a cell are placed in histogram bins depending on the gradient angle and magnitude.

4. These histograms, which now make up the image instead of the cells, are passed over by a 2x2 sliding window to create "Blocks".
5. Histograms are normalised by scaling all values in the block to values between 0 and 255.
6. Lastly all the blocks are concatenated together and the HOG descriptor is the resulting 3,780 values. This is made up of $7 * 15$ blocks * 4 histograms per blocks * 9 bins per histogram.

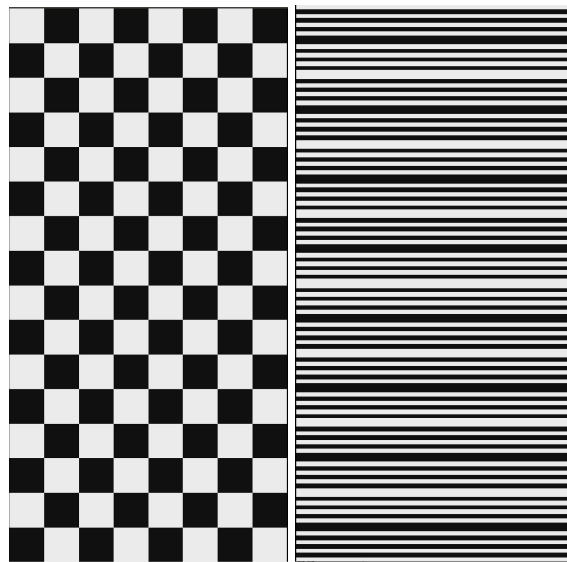
Section 4: HOG implementation



The image above shows the actors created for Hog as well as how they are connected together. The exact meaning and implementation of these individual actors are as follows.

McCormick's (2013) tutorial was followed while implementing the HOG algorithm.

8x8 Cell Tiler



The leftmost image is the input image which is taken in by the tiler actor and the rightmost image is the result from the actor.

This actor takes a 64 pixel wide by 128 pixel high image and iterates over it with an 8 pixel by 8 pixel tiler. The tiler starts in the top left corner and pixels are output in a left to right, top to bottom order. The first tile will output 64 pixels which are all from the first 8 by 8 square in the explicitly chosen checkbox input image. It will then move to the next square.

Tiler Actor

```
actor Tiler()
uint(size=8) Gin
==> uint(size=8) Gout
```

To describe and validate the implementation a specific input image was used which consists of 8 pixel by 8 pixel black and white squares.

The implementation uses a finite state machine with 3 states; read, transition and write.

Read state (initial) actions:

- the actor reads all the bytes from the input image stream and put them into a buffer. The buffer size is the image height * width, 64 x 128, which is fixed as these dimensions are what the state vector machine (SVM) was trained on.

Transition state actions:

- When the buffer has been filled the actor transitions to the write state via the “transition” stage, which resets the head to the beginning of the buffer to begin writing.

Write state actions:

- In the write state the actor begins with the x, y, xOffset and yOffset set to 0. Then incrementally move through the buffer and write to output 8 (window dimension) times, before resetting x to 0 and incrementing y to 1. This would translate into the first 8 pixels (left to right) from the first “square” in the input image shown above.
- Once again incrementally move through the buffer 8 times but by y * imagewidth, to locate the position in the buffer which would correspond to the second row of the first “square” in the input image.
- Once y has been incremented more times than the window dimension, the first square has been completed. x, y, and yOffset are reset to 0, and xOffset is incremented by 1 (which will be multiplied by window dimension during calculations). This will locate the position for the second column of “squares” in the input image.
- This process repeats until the xOffset exceeds the width of image. x, y, and xOffset are then reset to 0, and yOffset was incremented by 1 (also multiplied by window dimension during calculations to find the corresponding location in the buffer).

Reset state actions:

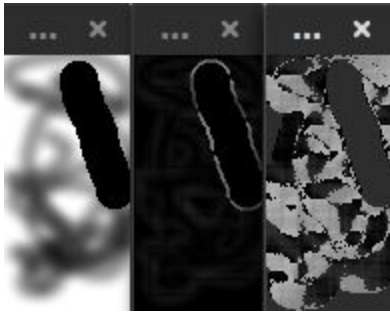
- Upon reaching the end of the buffer, and therefore the end of the image, the reset state will set all values back to 0 and transition back to the read state for further input (next image).

Using the custom test image as input the process results in an expected output of 8 (8 squares in a row)* 64 (8*8 dimension) pixel lines, alternating between black and white. The last square in a row is the same colour as the beginning of the next row, resulting in a double line, as shown in the output image.

Gradient

```
actor Gradient()  
uint(size=8) Gin  
==> uint(size=8) Angle, uint(size=8) Magnitude
```

The gradient actor reads in an 8 by 8 cell, then, for each pixel in the cell, output its angle and magnitude. For the precise mathematical definition of these values, consult the documentation for this actor, but briefly, it is the gradient between this pixel and its adjacent ones.



This screenshot shows an initial image (generated using Gimp), the magnitudes and the angles (in that order). The magnitudes outline the "edges" of the black blob, since that is where the image changes contrast the most, and also the fainter background paint.

The angle output looks like a mess because humans can't intuitively visualise them, but you can clearly see that there is no change on the black blob, and a rough outline of the paint trail in the background.

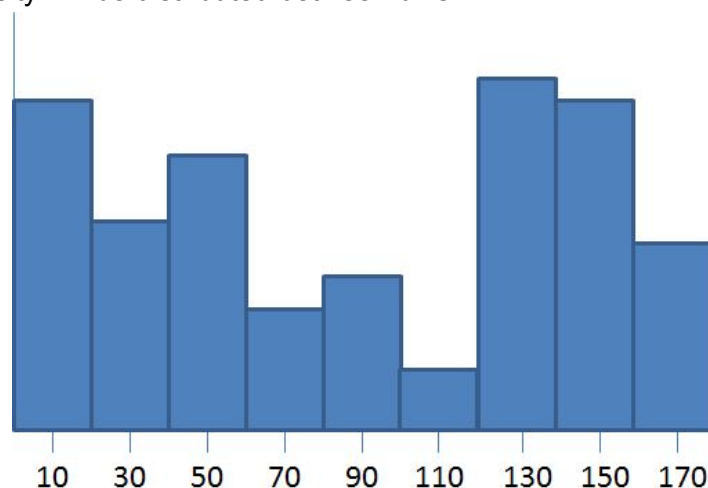
Actions (actor has no states):

- If the entire cell has been output, then reset both the read and write pointer to 0.
- If the whole cell has been output, then read the next pixel from `Gin` and store it in an internal buffer, then increment the amount of bytes that have been read.
- If the whole cell has been read, but all the values have not been output yet, get the current location in the cell. Calculate the angle and magnitude for this pixel using values in the buffer, output these values to `Angle` and `Magnitude`, and increment the bytes written.

Histogram Binning

```
actor histogram_binning()
uint(size=8) angle, uint(size=8) intensity
==> uint(size=16) Histogram
```

This algorithm takes in a gradient angle and intensity, and distributes the intensity to 9 histogram bins. This actor takes in 64 input pairs (which would be a single cell) and works out how the intensity will be distributed between bins.



The bins have the following values in degrees: 10, 30, 50, 70, 90, 110, 130, 150, 170.

The distribution logic calculates the distance an angle is from two bins. The closer bin will receive a larger split of the intensity. For example, if the angle was 25 degrees then $\frac{3}{4}$ of the intensity would go to bin 30 and $\frac{1}{4}$ would go to bin 10.

An example of how a pair of values are placed into bins would be:

```
If angle = 40 and intensity = 20 then
    bin30 += 10 and bin50 += 10
```

initArray state actions:

- Sets algorithm counters and histogram array to 0, then move to the read state

read state actions:

- Consumes 64 gradient angles and intensity values and bins them using the above mentioned logic.
- Then outputs the values of the 9 histogram bins
- Lastly returns to the initArray state.

Block Generation

```
actor Blocks()
uint(size=16) Gin
==> uint(size=16) Gout
```

This functions as a 2x2 sliding window over the grid of histograms. A "frame" of 2x2 histograms starts at image position (0, 0) and those four histograms are output (left to right, top to bottom), which will be histograms at [(0, 0), (1, 0), (0, 1), (1, 1)]. This frame moves across to the right, making the next histogram [(1, 0), (2, 0), (1, 1), (2, 1)]. When this frame reaches the end of a row, it wraps to the row below it.

For this report, call the 2x2 histograms output "Blocks".

Internally there are two "rows" stored in full. One is for the row above the current location, and the other is for the row currently being read. When the end of the row is reached, they swap.

write (initial):

- If no actions have been run or the end of the source image has been reached, reset all values to 0.
- If on row or column 0, then transition to the read state.
- If all four histograms has been output (that is, $op = \text{histogram size} * 4$) for the block, transition to the read state.
- Use the op variable to determine which histogram to output ($op / \text{histogram size}$), and which value in that histogram to output ($op \bmod \text{histogram size}$). Output that value to Gout and increment op .

read:

- Read a full histogram from Gin, and store it the current row array at the current column location, then advance the current column. If the current column is now off the side of the image, reset column to zero, increment the row and swap the current row and the above row arrays. Then always transition to the write state.

Block Normalization

```
histogram_normalisation ()  
uint(size=16) Gin  
==> uint(size=8) Gout :
```

In order to make the histograms invariant to illumination and contrast changes, the histograms must be normalised. By taking the input from the block generation actor, take 4 histograms with 9 bins each, for a total of 36 values. Get the minimum and maximum values for all the bins in this block and use it to output the original 36 values normalised between 0 and 255.

Read (initial) state actions:

- This state will in read a single input and put it into a buffer and then increment the head position in the buffer. This will happen a total of 36 times at which the buffer will have 4 histograms, of 9 bins each, as values in the buffer.
- During this state it finds the maximum and minimum values in the buffer, which will then be later used to perform normalisation.

Transition state actions:

- Once the buffer has been filled the transition state will reset the head to the start of the buffer.

Write state actions:

- The write state iterates over the buffer and outputs each value as their min-max normalised value.

Reset state actions:

- Once the end of the buffer has been reached the reset state will reset the head and min max variables to their original values, before transitioning back to the read state for further input.

Section 5: Future Steps

Sliding Window for the entire image

In the complete HOG implementation images supplied can be any size, whereas the SVM and all subsequent actors are tuned to work specifically for a 64 wide, 128 high image. As such, a large sliding window of 64 x 128 iterates over the original image from top to bottom, left to right, in 1 pixel increments is used to output several "snapshots" of 64 x 128 images for the remainder of the implementation to process and the HOG descriptor to classify.

To recognize persons at different scales, the input image would be subsampled to multiple sizes. Each of these subsampled images would be used as input for the entire HOG implementation.

Gradient before cells

In the current implementation, the "Gradient" actor sits after the "Cell Tiler" actor, so it can't accurately detect gradient changes on the edges of these cells (since it doesn't "have" that data). While this is the order the tutorial that was followed, the order of these two actors can be swapped.

This would, however, require rewriting "Gradient" from scratch (since it currently reads in an entire 8x8 cell to use), and significant changes to "Cell Tiler" (since it would need to read and write 2 values per pixel instead of one).

This was not feasible during our project due to time restrictions, but would offer an improvement to robustness of the algorithm.

Integrating the Algorithm into an SVM

The current implementation is the HOG algorithm that creates HOG descriptors for an input image. This descriptor on its own will not tell you if the input image contains the object you are looking for. The descriptor needs to be input into a trained SVM algorithm to determine if the input image contains the desired object.

The easiest way to try this would be to take an existing SVM and, assuming the descriptors used to train it are created in the same method our algorithm does, input our HOG descriptors to retrieve a classification. Unfortunately, it is believed that even a slight difference in descriptor creation technique could alter the output hugely. Therefore it might be necessary to train a custom SVM using our HOG algorithm to obtain reliable results.

The first steps to creating a complete feature detection function would be to determine if our HOG implementation is different from the descriptor creation implementation of a trained SVM you wish to use. If this is not the case, then creating your own SVM using our HOG implementation.

Implement remaining image processing algorithms

During the first half a few "small" image processing algorithms were implemented. However, the following algorithms were not implemented; these would be a good place to start for anyone interested in following in the groups footsteps:

- Bitshift operations

- RGB to YUV transformation
- YUV to RGB transformation
- Affine Transform
- Classification
- Pixel value distribution
- Dilation
- Erosion
- Opening
- Closing
- Hit and Miss transform
- Thinning
- Thickening
- Skeletonization
- Robert's Cross Edge Detection
- Sobel Edge Detector
- Canny Edge Detection
- Compass Edge Detection
- Zero Crossing Detector
- Line Detector
- Distance Transform
- Fourier Transform
- Wavelet Transform
- Motion Estimation
- Optic Flow Computation
- Histogram Computation
- Statistical Distribution

References

- Ahmed, J, J., Li, S., Sadeghi, A. and Schneider, T. 2012. A Platform-Independent Crypto Tools Library Based on Dataflow Programming Paradigm. Berlin:Springer, pp 299-313.
- Bhattacharyya, S, S., Eker, J., Janneck, W, J., Lucarz, C., Mattavelli, M. and Raulet, M. 2009. *Journal of Signal Processing Systems*. Hingham:Kluwer, pp.251-263.
- McCormick, C, (2013). *HOG Person Detector Tutorial*. [ONLINE] Available at: <http://mccormickml.com/2013/05/09/hog-person-detector-tutorial/>. [Accessed 29 March 2017].
- Dalal N. and Trigg B. 2005. Computer Vision and Pattern Recognition. Montbonnot: INRIA.
- Davis, J., Hylands, C., Kienhuis, B., Lee, E., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Tsay, J., Vogel, B. and Xiong, Y. 2001. Heterogeneous concurrent modeling and design in java.
- Eker, J., and Janneck, W, J. 2001. An introduction to the Caltrop actor language. Berkeley.
- Hewitt, C. 1977. Viewing control structures as patterns of passing messages. Boston:Journal of Artificial Intelligence, pp.323–363.
- Pell, O. and Mencer, O., 2011. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News*, 39(4), pp.60-65.
- R. Fisher, S. Perkins, A. Walker and E. Wolfart. 2003. *Image Processing Operator Worksheets*. [ONLINE] Available at: <http://homepages.inf.ed.ac.uk/rbf/HIPR2/wksheets.htm>. [Accessed 29 March 2017].
- Sousa, T.B., 2012. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering* (Vol. 130).
- Orcc.sourceforge.net. (2014). *Orcc : Dataflow Programming Made Easy - Orcc*. [online] Available at: <http://orcc.sourceforge.net/> [Accessed 31 Mar. 2017].

Appendix A: Code Sample Listing

At the end of this report, several files of the implementation have been attached. This is a listing of all files attached.

These files can be found at:

<https://github.com/orcc/orc-apps> in the `ImageProcessing/src/image` directory when it has been merged. In that repository, each actor also has documentation in the same folder as the actor source.

- Image Arithmetic
 - `arithmetic/multiply.cal`
 - `arithmetic/addition.cal`
 - `arithmetic/subtraction.cal`
 - `arithmetic/division.cal`
- Logical Operators:
 - `arithmetic/LogicalAND.cal`
 - `arithmetic/LogicalINAND.cal`
 - `arithmetic/LogicalOR.cal`
 - `arithmetic/LogicalNOR.cal`
 - `arithmetic/LogicalXOR.cal`
 - `arithmetic/LogicalXNOR.cal`
- Contrast Stretching:
 - `utils/Cache.cal`
 - `point/contrast_stretch/ContrastStretch.cal`
 - `utils/MaxMin.cal`
- `image_processing/intensity_histogram.cal`
- Geometric operations:
 - `geometric/Reflect.cal`
 - `geometric/Rotate.cal`
 - `geometric/Translate`
- Connected Components Labeling:
 - `analysis/connected_components/Above.cal`
 - `analysis/connected_components/DoubleCach.cal`
 - `analysis/connected_components/Label.cal`
 - `analysis/connected_components/Reduce.cal`
 - `analysis/connected_components/Replace.cal`
- HOG Descriptor:
 - `hog_person_detector/constants.cal`
 - `hog_person_detector/tiler.cal`
 - `hog_person_detector/Gradient.cal`
 - `hog_person_detector/HistogramBinning.cal`
 - `hog_person_detector/Blocks.cal`
 - `hog_person_detector/histogram_normalization.cal`

