
Jonathon Hare
Sina Samangooei
David Dupplaw

# The OpenIMAJ Tutorial

# CONTENTS

# PREFACE

**OpenIMAJ** is a set of libraries and tools for multimedia analysis. OpenIMAJ is very broad and contains everything from state-of-the-art computer vision (e.g. SIFT descriptors, salient region detection, face detection, etc.) and advanced data clustering, through to software that performs analysis on the layout and structure of webpages.

OpenIMAJ is primarily written in pure Java and, as such, is completely platform independent. The video capture and hardware libraries contain some native code but Linux (x86 and x86_64 are supported currently; ARM support is coming soon), OSX and Windows are supported out of the box (under both 32 and 64 bit JVMs). It is possible to write programs that use the libraries in any JVM language that supports Java interoperability, such as Groovy, Jython, JRuby or Scala. OpenIMAJ can even be run on Android phones and tablets.

The OpenIMAJ software is structured into a number of modules. The modules can be used independently, so if, for instance, you were developing data clustering software using OpenIMAJ you wouldn't need to acquire the modules related to images. The list on the following page illustrates the modules and summarises the functionality in each component.

This tutorial aims to instruct the reader on how to get up and running writing code using OpenIMAJ. Currently the tutorial covers the following areas:

1. Getting started with OpenIMAJ using Maven
2. Processing your first image
3. Introduction to clustering, segmentation and connected components
4. Processing video
5. Finding faces
6. Global image features
7. SIFT and feature matching

In the future we hope to add more content to the tutorial covering the following:

- Image and video indexing using ImageTerrier
- Compiling OpenIMAJ from source
- Tracking features in video
- Audio processing
- Hardware interfaces

- Advanced local features
- Scalable processing with OpenIMAJ/Hadoop
- Machine learning

## The OpenIMAJ Modules

| | |
|---|---|
| openimaj | |
| core ..... | Submodule for modules containing functionality used across the library. |
| core ..... | Core library functionality concerned with general programming problems rather than multimedia specific functionality. Includes I/O utilities, randomisation, hashing and type conversion. |
| feature ..... | Core notion of features, usually denoted as arrays of data. Definitions of features for all primitive types, features with location and lists of features (both in memory and on disk). |
| audio ..... | Core definitions of audio streams and samples/chunks. Also contains interfaces for processors for these basic types. |
| image ..... | Core definitions of images, pixels and connected components. Also contains interfaces for processors for these basic types. |
| video ..... | Core definitions of a video type and functionality for displaying and processing videos. |
| video-capture .... | Cross-platform video capture interface using a lightweight native interface. Supports 32 and 64 bit JVMs under Linux, OSX and Windows. |
| math ..... | Mathematical implementations including geometric, matrix and statistical operators. |
| audio ..... | Submodule for audio related functionality. |
| processing ..... | Implementations of various audio processors (e.g. multichannel conversion, volume change, ...). |
| ... | |

```
...
├─ image ..... Submodule for image related functionality.
│  ├─ processing ..... Implementations of various image, pixel and connected component processors (resizing, convolution, edge detection, ...).
│  ├─ feature-extraction .....
│  Methods for the extraction of low-level image features,
│  including global image features and pixel/patch classification models.
│  ├─ local-features ... Methods for the extraction of local features. Local features are descriptions of regions of images (SIFT, ...) selected by detectors (Difference of Gaussian, Harris, ...).
│  └─ faces ..... Implementation of a flexible face-recognition pipeline, including pluggable detectors, aligners, feature extractors and recognisers.
├─ machine-learning .. Algorithms which aid the classification and search of data.
│  ├─ nearest-neighbour Brute force and KD-Tree implementations of exact and approximate KNN.
│  └─ clustering ..... Various clustering algorithm implementations for all primitive types including random, random forest, K-Means (Exact, Hierarchical and Approximate), ...
├─ hadoop ..... Extensions to enable interaction with the Apache Hadoop Map-Reduce implementation.
│  ├─ core-hadoop ..... Reusable wrappers to access and create sequence-files and map-reduce jobs.
│  └─ tools ..... Tools that provide Map-Reduce jobs that can be run on a Hadoop cluster.
│     ├─ HadoopClusterQuantiserTool .....
│     Distributed feature quantisation tool.
│     ├─ HadoopFastKMeans .....
│     Distributed feature clustering tool.
│     ├─ HadoopGlobalFeaturesTool .....
│     Distributed global image feature extraction tool.
│     ├─ HadoopImageDownload .....
│     Distributed image download tool.
│     ├─ HadoopLocalFeaturesTool .....
│     Distributed local image feature extraction tool.
│     ├─ SequenceFileIndexer .....
│     Tool for building an index of the keys in a Hadoop SequenceFile.
│     └─ SequenceFileTool .....
│     Tool for building, inspecting and extracting Hadoop SequenceFiles.
├─ hardware ..... Various interfaces to hardware devices that we've used in projects built using OpenIMAJ.
│  ├─ serial ..... Interface to hardware devices that connect to serial or USB-serial ports.
│  ├─ gps ..... Interface to GPS devices that support the NMEA protocol.
│  └─ compass ..... Interface to an OceanServer OS5000 digital compass.
└─ ...
```

```
...
├─ web ..... Support for analysing and processing web-pages.
│  ├─ core-web ..... Implementation of a programatic offscreen web
│  │                 browser and utility functions.
│  └─ analysis ..... Utilities for analysing the content and visual layout of a
│                    web-page.
├─ video ..... Support for analysing and processing video.
│  ├─ video-processing . Various video processing algorithms, such as shot-
│  │                     boundary detection.
│  └─ xuggle-video ..... Plugin to use Xuggler as a video source. Allows most
│                        video formats to be read into OpenIMAJ.
├─ thirdparty ..... Thirdparty code that has been integrated into
│  │                OpenIMAJ.
│  └─ klt-tracker ..... Implementation of the Kanade-Lucus-Tomasi feature
│                       tracker.
└─ tools ..... Commandline tools exposing OpenIMAJ functionality
   ├─ CityLandscapeClassifier .....
   │  Tool for classifying images as being cityscapes/land-
   │  scapes (or natural/unnatural).
   ├─ FaceTools ..... Tools for face detection and recognition.
   ├─ FeatureVisualisation .....
   │  Tools for visualising certain types of image feature.
   ├─ FlickrCrawler .... Tool for downloading image datasets from Flickr.
   ├─ GlobalFeaturesTool .....
   │  Tool for extracting global features from images.
   ├─ ImageCollectionTool .....
   │  Tool for creating collections of images from various
   │  sources.
   ├─ LocalFeaturesTool  Tool for extracting local features from images.
   ├─ OCRTools ..... Tool for applying OCR to images.
   └─ WebTools ..... Tools for extracting and analysing the layout and visual
                     characteristics of webpages.
```

TUTORIAL 1

# GETTING STARTED WITH OPENIMAJ USING MAVEN

Apache Maven is a project management tool. Maven performs tasks such as automatic dependency management, project packaging and more. We **strongly** encourage anyone using OpenIMAJ to use Maven to get their own project started. We've even provided a Maven **archetype** for OpenIMAJ (basically a project template) that lets you get started programming with OpenIMAJ quickly.

You can find out more about Apache Maven at `http://maven.apache.org`.

OpenIMAJ requires Maven 3. You can check if you have Maven installed already by opening a terminal (or DOS command prompt) and typing:

```
mvn -version
```

If Maven is found the, version will be printed. If the version is less than 3.0, or Maven was not found, go to `http://maven.apache.org` to download and install it. Once you've installed Maven try the above command to test that it is working.

To create a new OpenIMAJ project, run the following command:

```
mvn -DarchetypeCatalog=http://octopussy.ecs.soton.ac.uk
    /m2/releases/archetype-catalog.xml
        archetype:generate
```

Maven will then prompt you for some input. Firstly, when prompted, choose the `openimaj-quickstart-archetype` and choose the 1.0.4 version. For the `groupId`, enter something that identifies you or a group that you belong to (for example, I might choose `uk.ac.soton.ecs.jsh2` for personal projects, or `org.openimaj` for OpenIMAJ sub-projects). For the `artifactId` enter a name for your project (for example, `OpenIMAJ-Tutorial01`). The version can be left as `1.0-SNAPSHOT`, and the default package is also OK. The `openimajVersion` must be set as `1.0.4`. Finally enter `Y` and press return to confirm the settings. Maven will then generate a new project in a directory with the same name as the `artifactId` you provided.

At the time of writing, the latest release version of OpenIMAJ is `1.0.4`. Future versions of the archetype should automatically select the `openimajVersion` variable based on the chosen archetype version.

The project directory contains a file called `pom.xml` and a directory called `src`. The `pom.xml` file describes all of the dependencies of the project and also contains instructions for packaging the project into a fat jar that contains all your project code

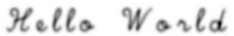
and resources together with the dependencies. If you find that you need to add another library to your project, you should do so by editing the `pom.xml` file and adding a new dependency. The `src` directory contains the code for your project. In particular, `src/main/java` contains your java application code and `src/test/java` contains unit tests.

The default project created by the archetype contains a small "hello world" application. To compile and assemble the "hello world" application you `cd` into the project directory from the command line and run the command:

```
mvn assembly:assembly
```

This will create a new directory called target that contains the assembled application jar (the assembled jar is the one whose name ends with -jar-with-dependencies.jar). To run the application, enter:

```
java -jar target/OpenIMAJ-Tutorial01-1.0-SNAPSHOT-jar-
    with-dependencies.jar
```




The application will then run, and a window should open displaying a picture with the text "hello world". Closing the window, or ctrl-c on the command line, will quit the application.

## Integration with your favourite IDE

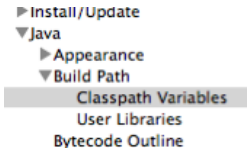
We could now go ahead and start playing with the code in a text editor, however this really isn't recommended! Using a good Integrated Development Environment (IDE) with auto-completion will make your experience much better.

Maven integrates with all the popular IDEs. The OpenIMAJ developers all use Eclipse (`http://www.eclipse.org`) so that is what we're most familiar with, however we should be able to help getting it set up in a different IDE if you wish.

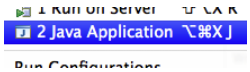
Integration with Eclipse is quite simple. From the command line, inside the project directory, issue the command:

```
mvn eclipse:eclipse
```

This will generate Eclipse project files in the same directory. In Eclipse you can then import the project into the Eclipse workspace (File>import..., choose "Existing projects into workspace", select the project directory, make sure "Copy projects into workspace" is **unchecked**, then click "Finish"). The project should then appear in the workspace and you'll be able to look at the App.java file that was generated by the archetype.




**IMPORTANT** By default Eclipse doesn't know about Maven and its repositories of jars. When you first import an OpenIMAJ project into Eclipse it will have errors. You can fix this by adding a new Java classpath variable (`Eclipse>Preferences>Java>Build Path>Classpath Variables`) called `M2_REPO`. The value of this variable is the location of your `.m2/repository` directory. For Unix systems this is usually found in your home directory, for Windows systems it is found in `C:\Documents and Settings\<user>\`.




Once you've opened the `App.java` file in Eclipse, you can right-click on it and select `Run as>Java Application` to run it from within Eclipse.

## Exercises

### Exercise 1: Playing with the sample application

Take a look at the App.java from within your IDE. Can you modify the code to render something other than “hello world” in a different font and colour?

TUTORIAL 2

# PROCESSING YOUR FIRST IMAGE

In this section we'll start with the "hello world" app and show you how you can load an image, perform some basic processing on the image, draw some stuff on your image and then display your results.

Loading images into Java is usually a horrible experience. Using Java `ImageIO`, one can use the `read()` method to create a `BufferedImage` object. Unfortunately the `BufferedImage` object hides the fact that it is (and in fact all digital raster images are) simply arrays of pixel values. A defining philosophy of OpenIMAJ is to *keep things simple* which in turn means in OpenIMAJ images are as close as one can get to being **just arrays of pixel values**.

To read and write images in OpenIMAJ we use the `ImageUtilities` class. In the `App.java` class file remove the sample code within the main method and add the following line:

```
MBFImage image = ImageUtilities.readMBF(
        new File("file.jpg"));
```

For this tutorial, read the image from the following URL:

```
MBFImage image = ImageUtilities.readMBF(
    new URL("http://dl.dropbox.com/u/8705593/sinaface.
        jpg"));
```

The `ImageUtilities` class provides the ability to read `MBFImages` and `FImages`. An `FImage` is a greyscale image which represents each pixel as a value between `0` and `1`. An `MBFImage` is a multi-band version of the `FImage`; under the hood it actually contains a number `FImage` objects held in a list each representing a band of the image. What these bands represent is given by the image's public `colourSpace` field, which we can print as follows:

```
System.out.println(image.colourSpace);
```

If we run the code, we'll see that the image is an RGB image with three `FImages` representing the red, blue and green components of the image in that order.

You can display any OpenIMAJ image object using the `DisplayUtilities` class. In this example we display the image we have loaded then we display the red channel of the image alone:

```
DisplayUtilities.display(image);
DisplayUtilities.display(
    image.getBand(0), "RedChannel");
```

In an image-processing library, images are no good unless you can do something to them. The most basic thing you can do to an image is fiddle with its pixels. In OpenIMAJ, as an image is just an array of floats, we make this is quite easy. Let's go through our colour image and set all its blue and green pixels to black:

```
MBFImage clone = image.clone();
for (int y=0; y<image.getHeight(); y++) {
    for(int x=0; x<image.getWidth(); x++) {
        clone.getBand(1).pixels[y][x] = 0;
        clone.getBand(2).pixels[y][x] = 0;
    }
}
DisplayUtilities.display(clone);
```

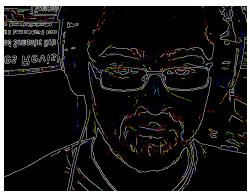
Note that the first thing we do here is to `clone` the image to preserve the original image for the remainder of the tutorial. The pixels in an `FImage` are held in a 2D float array. The rows of the image are held in the first array that, in turn, holds each of the column values for that row: `[y][x]`. By displaying this image we should see an image where two channels are black and one channel is not. This results in an image that looks rather red.

Though it is helpful to sometimes get access to individual image pixels, OpenIMAJ provides a lot of methods to make things easier. For example, we could have done the above like this instead:

```
clone.getBand(1).fill(0);
clone.getBand(2).fill(0);
```

More complex image operations are wrapped up by OpenIMAJ processor interfaces: `ImageProcessors`, `KernelProcessors`, `PixelProcessors` and `GridProcessors`. The distinction between these is how their algorithm works internally. The overarching similarity is that an image goes in and a processed image (or data) comes out. For example, a basic operation in image processing is **edge detection**. A popular edge detection algorithm is the *Canny edge detector*. We can call the Canny edge detector like so:

```
image.processInline(new CannyEdgeDetector2());
```

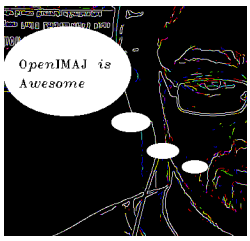


When applied to a colour image, each pixel of each band is replaced with the edge response at that point (for simplicity you can think of this as the difference between that pixel and its neighbouring pixels). If a particular edge is only strong in one band or another then that colour will be strong, resulting in the psychedelic colours you should see if you display the image:

```
DisplayUtilities.display(image);
```

Finally, we can also draw on our image in OpenIMAJ. On every `Image` object there is a set of drawing functions that can be called to draw points, lines, shapes and text on images. Let's draw some speech bubbles on our image:

```
image.drawShapeFilled(new Ellipse(700f, 450f, 20f, 10f,
    0f), RGBColour.WHITE);
image.drawShapeFilled(new Ellipse(650f, 425f, 25f, 12f,
    0f), RGBColour.WHITE);
image.drawShapeFilled(new Ellipse(600f, 380f, 30f, 15f,
    0f), RGBColour.WHITE);
image.drawShapeFilled(new Ellipse(500f, 300f, 100f, 70f
   , 0f), RGBColour.WHITE);
image.drawText("OpenIMAJ␣is", 425, 300, HersheyFont.
   ASTROLOGY, 20, RGBColour.BLACK);
image.drawText("Awesome", 425, 330, HersheyFont.
   ASTROLOGY, 20, RGBColour.BLACK);
DisplayUtilities.display(image);
```




Here we construct a series of ellipses (defined by their centre [x, y], axes [major, minor] and rotation) and draw them as white filled shapes. Finally, we draw some text on the image and display it.

## Exercises

### Exercise 1: DisplayUtilities

Opening lots of windows can waste time and space (for example if you wanted to view images on every iteration of a process within a loop). In OpenIMAJ we provide a facility to open a *named display* so that was can reuse the display referring to it by name. Try to do this with all the images we display in this tutorial. Only 1 window should open for the whole tutorial.

### Exercise 2: Drawing

Those speech bubbles look rather plain; why not give them a nice border?

TUTORIAL 3

# INTRODUCTION TO CLUSTERING, SEGMENTATION AND CONNECTED COMPONENTS

In this tutorial we'll create an application that demonstrates how an image can be broken into a number of regions. The process of separating an image into regions, or segments, is called **segmentation**. Segmentation is a widely studied area in computer vision. Researchers often try to optimise their segmentation algorithms to try and separate the **objects** in the image from the **background**.

To get started, create a new OpenIMAJ project using the Maven archetype, import it into your IDE, and delete the sample code from within the generated `main()` method of the `App` class. In the `main()` method, start by adding code to load an image (choose your own image):

```
MBFImage input = ImageUtilities.readMBF(
    new URL("http://..."));
```

To segment our image we are going to use a machine learning technique called **clustering**. Clustering algorithms automatically group similar things together. In our case, we'll use a popular clustering algorithm called **K-Means** clustering to group together all the similar colours in our image. Each group of similar colours is known as a **class**. The K-means clustering algorithm requires you set the number of classes you wish to find **a priori** (i.e. beforehand).

K-Means initialises cluster centroids with randomly selected data points and then iteratively assigns the data points to their closest cluster and updates the centroids to the mean of the respective clusters data points

Colours in our input image are represented in **RGB colour space**; that is each pixel is represented as three numbers corresponding to a red, green and blue value. In order to measure the similarity of a pair of colours the "distance" between the colours in the colour space can be measured. Typically, the distance measured is the **Euclidean** distance. Unfortunately, distances in RGB colour space do not reflect what humans perceive as similar/dissimilar colours. In order to work-around this problem it is common to transform an image into an alternative colour space. The **Lab colour space** (pronounced as separate letters, L A B) is specifically designed so that the Euclidean distance between colours closely matches the perceived similarity of a colour pair by a human observer.

The Euclidean distance is the straight-line distance between two points.

To start our implementation, we'll first apply a colour-space transform to the image:

```
input = ColourSpace.convert(input,ColourSpace.CIE_Lab);
```

We can then construct the K-Means algorithm:

```
FastFloatKMeansCluster cluster = new
    FastFloatKMeansCluster(3,2,true);
```

The first parameter is the dimensionality of the space (3 in this case corresponding to the **L**, **a**, and **b** dimensions of the colour vectors). The second argument is the number of clusters or classes we wish the algorithm to generate. The final boolean flag indicates whether the underlying algorithm should be the "exact" K-means algorithm (true) or an **approximate** algorithm (false). The approximate algorithm is much faster than the exact algorithm when there is very high-dimensional data; in this case, with only three dimensions, the approximate algorithm is not required. The OpenIMAJ K-Means implementation is multithreaded and automatically takes advantage of all the processing power it can obtain.

The FastFloatKMeansCluster algorithm takes its input as an array of floating point vectors (`float[][]`). We can flatten the pixels of an image into the required form using the `getPixelVectorNative()` method:

```
float[][] imageData = input.getPixelVectorNative(new
    float[input.getWidth() * input.getHeight()][3]);
```

The K-Means algorithm can then be run to group all the pixels into the requested number of classes:

```
cluster.train(imageData);
```

Each class or cluster produced by the K-Means algorithm has an index, starting from 0. Each class is represented by its centroid (the average location of all the points belonging to the class). We can print the coordinates of each centroid:

```
float[][] centroids = cluster.getClusters();
for (float[] fs : centroids) {
    System.out.println(Arrays.toString(fs));
}
```

Now is a good time to test the code. Running it should print the (`L, a, b`) coordinates of each of the classes.

We can now use the `FastFloatKMeansCluster` to assign each pixel in our image to its respective class. This is a process known as **classification**. The `FastFloatKMeansCluster` has a method called `push_one()` which takes a vector (the `L, a, b` value of a single pixel) and returns the index of the class that it belongs to. We'll start by creating an image that visualises the pixels and their respective classes by replacing each pixel in the input image with the centroid of its respective class:

```
for (int y=0; y<input.getHeight(); y++) {
    for (int x=0; x<input.getWidth(); x++) {
        float[] pixel = input.getPixelNative(x, y);
        int centroid = cluster.push_one(pixel);
        input.setPixelNative(x, y, centroids[centroid])
            ;
    }
}
```

We can then display the resultant image. Note that we need to convert the image back to RGB colour space for it to display properly:

```
input = ColourSpace.convert(input, ColourSpace.RGB);
DisplayUtilities.display(input);
```

Running the code will display an image that looks a little like the original image but with as many colours as there are classes.

To actually produce a segmentation of the image we need to group together all pixels with the same class that are touching each other. Each set of pixels representing a segment is often referred to as a **connected component**. Connected components in OpenIMAJ are modelled by the `ConnectedComponent` class.

The `GreyscaleConnectedComponentLabeler` class can be used to find the connected components:

```
GreyscaleConnectedComponentLabeler labeler = new
    GreyscaleConnectedComponentLabeler();
List<ConnectedComponent> components = labeler.
    findComponents(input.flatten());
```

Note that the `GreyscaleConnectedComponentLabeler` only processes greyscale images (the `FImage` class) and not the colour image (`MBFImage` class) that we created. The `flatten()` method on `MBFImage` merges the colours into grey values by averaging their RGB values.

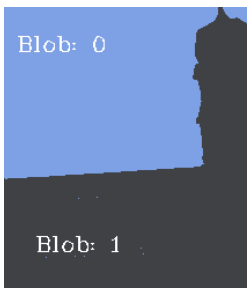
OpenIMAJ also contains a class called `ConnectedComponentLabeler` which can only be used on **binary** (i.e. pure black and white) `FImages`.

The `ConnectedComponent` class has many useful methods for extracting information about the shape of the region. Lets draw an image with the components numbered on it. We'll use the centre of mass of each region to position the number and only render numbers for regions that are over a certain size (50 pixels in this case):

```
int i = 0;
for (ConnectedComponent comp : components) {
    if (comp.calculateArea() < 50)
        continue;
    input.drawText("Point:" + (i++), comp.
        calculateCentroidPixel(), HersheyFont.
        TIMES_MEDIUM,20);
}
```

Finally, we can display the image with the labels:

```
DisplayUtilities.display(input);
```



## Exercises

### Exercise 1: The PixelProcessor

Rather than looping over the image pixels using two for loops, it is possible to use a `PixelProcessor` to accomplish the same task:

```
image.processInline(new PixelProcessor<Float[]>() {
    Float[] processPixel(Float[] pixel, Number[]...
        otherpixels) {
         ...
    }
});
```

Can you re-implement the loop that replaces each pixel with its class centroid using a `PixelProcessor`?

What are the advantages and disadvantages of using a `PixelProcessor`?

### Exercise 2: A real segmentation algorithm

The segmentation algorithm we just implemented can work reasonably well, but is rather naïve. OpenIMAJ contains an implementation of a popular segmentation algorithm called the `FelzenszwalbHuttenlocherSegmenter`.

Try using the `FelzenszwalbHuttenlocherSegmenter` for yourself and see how it compares to the basic segmentation algorithm we implemented. You can use the `SegmentationUtilities.renderSegments()` static method to draw the connected components produced by the segmenter.

TUTORIAL 4

# PROCESSING VIDEO

In this section we'll show you how to deal with videos using OpenIMAJ. We provide a set of tools for loading, displaying and processing various kinds of video.

All videos in OpenIMAJ are subtypes of the `Video` class. This class is typed on the type of underlying frame. In this case, let's create a video which holds coloured frames:

```
Video<MBFImage> video;
```

Exactly what kind of video is loaded depends on what you want to do. To load a video from a file we use the **Xuggle** library which internally uses `ffmpeg`. Let's load a video from a file (which you can download from here: `http://dl.dropbox.com/u/8705593/keyboardcat.flv`).

First you'll need to install Xuggle. Go to http://www.xuggle.com/xuggler/downloads/ and pick your platform then follow the instructions.

While this is downloading, let's write the code. If we want to load a video from a file we use a `XuggleVideo` object:

```
video = new XuggleVideo(new File("/path/to/keyboardcat.
    flv"));
```

The `XuggleVideo` class also has a constructor that lets you pass a URL to a video on the web without downloading it first.

If your computer has a webcam, OpenIMAJ also supports live video input. These are called capture devices and you can use one through the `VideoCapture` class:

```
video = new VideoCapture(320,240);
```

This will find the first video capture device attached to your system and render it as closely to $320 \times 240$ pixels as it can. To select a specific device you can use the alternative constructors and use the `VideoCapture.getVideoDevices()` static method to obtain the available devices.

Note that webcam capture does not require a Xuggle install and is completely built into OpenIMAJ.

To see if either of these kinds of video work, we can use `VideoDisplay` to display videos. This is achieved using the static function calls in `VideoDisplay` (which mirror those found in `DisplayUtilities` for images) like so:

```
VideoDisplay<MBFImage> display = VideoDisplay.
    createVideoDisplay(video);
```

Simply by creating a display, the video starts and plays. You can test this by running your app after Xuggle is installed.

**Note:** If you're using Linux or MacOSX, you might encounter an error trying to run this from Eclipse. This is due to library dependencies on Xuggle. To fix this, in Eclipse go to the run configuration (`Run>Run Configurations`). Once here click the `Environments` tab. Add a new environment variable called `DYLD_LIBRARY_PATH` on MacOSX or `LD_LIBRARY_PATH` on Linux and set this to the location of the `xuggler/lib` directory (this is likely to be `/usr/local/xuggler/lib` if you went with the default Xuggle install options).

As with images, displaying them is nice but what we really want to do is process the frames of the video in some way. This can be achieved in various ways; firstly videos are `Iterable`, so you can do something like this to iterate through every frame and process it:

```
for (MBFImage mbfImage : video) {
    DisplayUtilities.displayName(mbfImage.process(new
        CannyEdgeDetector2()), "videoFrames");
}
```

Here we're applying a Canny edge detector to each frame and displaying the frame in a named window. Another approach, which ties processing to image display automatically, is to use an event driven technique:

```
VideoDisplay<MBFImage> display = VideoDisplay.
    createVideoDisplay(video);
display.addVideoListener(
  new VideoDisplayListener<MBFImage>() {
    @Override
    public void beforeUpdate(MBFImage frame) {
        frame.processInline(new CannyEdgeDetector2());
    }

    @Override
    public void afterUpdate(VideoDisplay<MBFImage>
        display) {
    }
  });
```

These `VideoDisplayListeners` are given video frames before they are rendered and they are handed the video display after the render has occurred. The benefit of this approach is that functionality such as looping, pausing and stopping the video is given to you for free by the `VideoDisplay` class.

## Exercises

### Exercise 1: Applying different types of image processing to the video

Try a different processing operation and see how it affects the frames of your video.

TUTORIAL 5

# FINDING FACES

OpenIMAJ contains a set of classes that contain implementations of some of the state-of-the-art face detection and recognition algorithms. These classes are provided as a sub-project of the OpenIMAJ code-base called `faces`. The OpenIMAJ maven archetype adds the face library as a dependency and so we can start building face detection applications straight away.

Create a new application using the quick-start archetype (see tutorial 1) and import it into your IDE. If you look at the `pom.xml` file you will see that the `faces` dependency from OpenIMAJ is already included. As you've already done the video-processing tutorial, we'll try to find faces within the video that your webcam produces. If you don't have a webcam, follow the video tutorial on how to use video from a file instead.

Start by removing the code from the main method of the `App.java` class file. Then create a video capture object and a display to show the video. Create a listener on the video display to which we can hook our face finder. The code is below, but check out the previous tutorial on video processing if you're not sure what it means.

```
VideoCapture vc = new VideoCapture( 320, 240 );
VideoDisplay<MBFImage> vd = VideoDisplay.
    createVideoDisplay( vc );
vd.addVideoListener(
  new VideoDisplayListener<MBFImage>() {
    @Override
    public void beforeUpdate( MBFImage frame ) {
    }

    @Override
    public void afterUpdate( VideoDisplay<MBFImage>
        display ) {
    }
  });
```

For finding faces in images (or in this case video frames) we use a face detector. The `FaceDetector` interface provides the API for face detectors and there are cur-

rently two implementations within OpenIMAJ - the `HaarCascadeDetector` and the `SandeepFaceDetector`. The `HaarCascadeDetector` is considerably more robust than the `SandeepFaceDetector`, so we'll use that.

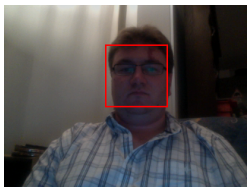
In the `beforeUpdate()` method, instantiate a new `HaarCascadeDetector`. The constructor takes the minimum size in pixels that a face can be detected at. For now, set this to 40 pixels:

```
FaceDetector<DetectedFace,FImage> fd = new
    HaarCascadeDetector(40);
```

Like all `FaceDetector` implementations, the `HaarCascadeDetector` has a method `detectFaces()` which takes an image. Because the `HaarCascadeDetector` uses single band images, we must convert our multi-band colour image into a single band image. To do this we can use the `Transforms` utility class that contains some static methods for converting images. The `calculateIntensity()` method will do just fine. Note that functionally the `calculateIntensity()` method does the same thing as the `flatten()` method we used earlier when used on RGB images.

```
List<DetectedFace> faces = fd.detectFaces(Transforms.
    calculateIntensity(frame));
```

The `detectFaces()` method returns a list of `DetectedFace` objects which contain information about the faces in the image. From these objects we can get the rectangular bounding boxes of each face and draw them back into our video frame. As we're doing all this in our `beforeUpdate()` method, the video display will end up showing the bounding boxes on the displayed video. If you run the code and you have a webcam attached, you should see yourself with a box drawn around your face. The complete code is shown below:



```
FaceDetector<DetectedFace,FImage> fd = new
    HaarCascadeDetector(40);
List<DetectedFace> faces =
            fd.detectFaces( Transforms.
                calculateIntensity(frame));

for( DetectedFace face : faces ) {
    frame.drawShape(face.getBounds(), RGBColour.RED);
}
```

OpenIMAJ has other face detectors which go a bit further than just finding the face. The `FKEFaceDetector` finds facial keypoints (the corners of the eyes, nose and mouth) and we can use this detector instead simply by instantiating that object instead of the `HaarCascadeDetector`. The `FKEFaceDetector` returns a slightly different object for each detected face, called a `KEDetectedFace`. The `KEDetectedFace` object contains the extra information about where the keypoints in the face are located. The lines of our code to instantiate the detector and detect faces can now be changed to the following:

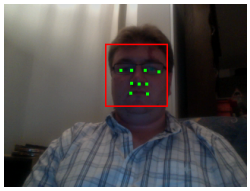
```
FaceDetector<KEDetectedFace,FImage> fd = new
    FKEFaceDetector();
List<KEDetectedFace> faces = fd.detectFaces( Transforms
    .calculateIntensity( frame ) );
```

If you run the demo now, you will see exactly the sameas before, as the `FKEFaceDetector` still detects bounding boxes. It may be running a bit slower though, as there is much

more processing going on - we're just not seeing the output of it! So, let's plot the facial keypoints.

To get the keypoints use `getKeypoints()` on the detected face. Each keypoint has a position (public field) which is relative to the face, so we'll need to translate the point to the position of the face within the video frame before we plot the points. To do that we can use the `translate()` method of the `Point2d` class and the `minX()` and `minY()` methods of the `Rectangle` class.

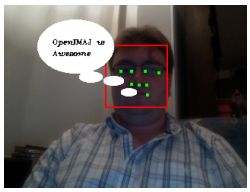
## Exercises



### Exercise 1: Drawing facial keypoints

Use the information above to plot the facial keypoints on the video.

### Exercise 2: Speech bubbles



Try and take the speech bubble from the previous image tutorial and make it come from the mouth in the video. **Hints:** use `getKeypoint(FacialKeypointType)` to get the keypoint of the left corner of the mouth and plot the ellipses depending on that point. You may need to use smaller ellipses and text if your video is running at 320x240.

TUTORIAL 6

# GLOBAL IMAGE FEATURES

The task in this tutorial is to understand how we can extract numerical representations from images and how these numerical representations can be used to provide similarity measures between images, so that we can, for example, find the most similar images from a set.

As you know, images are made up of pixels which are basically numbers that represent a colour. This is the most basic form of numerical representation of an image. However, we can do calculations on the pixel values to get other numerical representations that mean different things. In general, these numerical representations are known as **feature vectors** and they represent particular **features**.

Let's take a very common and easily understood type of feature. It's called a colour histogram and it basically tells you the proportion of different colours within an image (e.g. 90% red, 5% green, 3% orange, and 2% blue). As pixels are represented by different amounts of red, green and blue we can take these values and accumulate them in our histogram (e.g. when we see a red pixel we add 1 to our 'red pixel count' in the histogram).

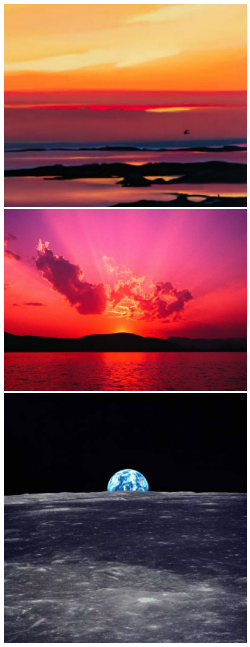
A histogram can accrue counts for any number of colours in any number of dimensions but the usual is to split the red, green and blue values of a pixel into a smallish number of 'bins' into which the colours are thrown. This gives us a three-dimensional cube, where each small cubic bin is accruing counts for that colour.

OpenIMAJ contains a multidimensional `Histogram` implementation that is constructed using the number of bins required in each dimension. For example:

```
Histogram histogram = new Histogram( 4, 4, 4 );
```

This code creates a histogram that has 64 ($4 \times 4 \times 4$) bins. However, this data structure does not do anything on its own. The `HistogramModel` class provides a means for creating a `Histogram` from an image. The `HistogramModel` class assumes the image has been normalised and returns a normalised histogram:

```
HistogramModel model = new HistogramModel( 4, 4, 4 );
model.estimateModel( image );
Histogram histogram = model.histogram;
```



You can print out the histogram to see what sort of numbers you get for different images. Let's load in 3 images then generate and store the histograms for them:

```
URL[] imageURLs = new URL[] {
    new URL( "http://users.ecs.soton.ac.uk/dpd/projects/
        openimaj/tutorial/hist1.jpg" ),
    new URL( "http://users.ecs.soton.ac.uk/dpd/projects/
        openimaj/tutorial/hist2.jpg" ),
    new  URL( "http://users.ecs.soton.ac.uk/dpd/projects
        /openimaj/tutorial/hist3.jpg" )
};
List<Histogram> histograms = new ArrayList<Histogram>()
    ;
for( URL u : imageURLs ) {
     HistogramModel model = new HistogramModel(4, 4, 4);
     model.estimateModel(ImageUtilities.readMBF(u));
     histograms.add( model.histogram );
}
```

We now have a list of histograms from our images. The `Histogram` class extends a class called the `MultidimensionalDoubleFV` which is a feature vector represented by multidimensional set of double precision numbers. This class provides us with a `compare()` method which allows comparison between two multidimensional sets of doubles. This method takes the other feature vector to compare against and a comparison method which is implemented in the `DoubleFVComparison` class.

So, we can compare two histograms using the Euclidean distance measure like so:

```
double distanceScore = histogram1.compare( histogram2,
    DoubleFVComparison.EUCLIDEAN );
```

This will give us a score of how similar (or dissimilar) the histograms are. It's useful to think of the output score as a **distance** apart in space. Two very similar histograms will be very close together so have a small distance score, whereas two dissimilar histograms will be far apart and so have a large distance score.

The Euclidean distance measure is symmetric (that is, if you compare `histogram1` to `histogram2` you will get the same score if you compare `histogram2` to `histogram1`) so we can compare all the histograms with each other in a simple, efficient, nested loop:

```
for( int i = 0; i < histograms.size(); i++ ) {
     for( int j = i; j < histograms.size(); j++ ) {
          double distance = histograms.get(i).compare(
              histograms.get(j), DoubleFVComparison.
              EUCLIDEAN );
     }
}
```

## Exercises

### Exercise 1: Finding and displaying similar images

Which images are most similar? Does that match with what you expect if you look at the images? Can you make the application display the two most similar images that aren't the same?

### Exercise 2: Exploring comparison measures

What happens when you use a different comparison measure (such as `DoubleFVComparison.INTERSECTION`)?

TUTORIAL 7

# SIFT AND FEATURE MATCHING

In this tutorial we'll look at how to compare images to each other. Specifically, we'll use a popular **local feature descriptor** called **SIFT** to extract some *interesting points* from images and describe them in a standard way. Once we have these local features and their descriptions, we can match local features to each other and therefore compare images to each other, or find a visual query image within a target image, as we will do in this tutorial.

Firstly, lets load up a couple of images. Here we have a magazine and a scene containing the magazine:

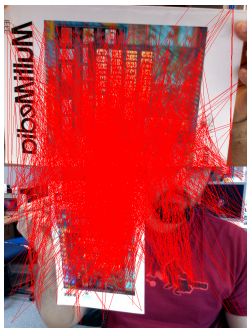
```
MBFImage query = ImageUtilities.readMBF(new
                                URL("http://dl.dropbox.com
                                    /u/8705593/query.jpg"))
                                    ;

MBFImage target = ImageUtilities.readMBF(new
                                URL("http://dl.dropbox.com
                                    /u/8705593/target.jpg")
                                    );
```

The first step is feature extraction. We'll use the **difference-of-Gaussian** feature detector which we describe with a **SIFT descriptor**. The features we find are described in a way which makes them invariant to size changes, rotation and position. These are quite powerful features and are used in a variety of tasks. The standard implementation of SIFT in OpenIMAJ can be found in the `DoGSIFTEngine` class:

```
DoGSIFTEngine engine = new DoGSIFTEngine();
LocalFeatureList<Keypoint> queryKeypoints = engine.
    findFeatures(query.flatten());
LocalFeatureList<Keypoint> targetKeypoints = engine.
    findFeatures(target.flatten());
```

Once the engine is constructed, we can use it to extract `Keypoint` objects from our images. The `Keypoint` class contain a public field called `ivec` which, in the case of a standard SIFT descriptor is a 128 dimensional description of a patch of pixels around a detected point. Various distance measures can be used to compare `Keypoints` to `Keypoints`.

The challenge in comparing `Keypoints` is trying to figure out which `Keypoints` match between `Keypoints` from some query image and those from some target. The most basic approach is to take a given `Keypoint` in the query and find the `Keypoint` that is closest in the target. A minor improvement on top of this is to disregard those points which match well with MANY other points in the target. Such point are considered non-descriptive. Matching can be achieved in OpenIMAJ using the `BasicMatcher`. Next we'll construct and setup such a matcher:



```
LocalFeatureMatcher<Keypoint> matcher = new
    BasicMatcher<Keypoint>(80);
matcher.setModelFeatures(queryKeypoints);
matcher.findMatches(targetKeypoints);
```

We can now draw the matches between these two images found with this basic matcher using the `MatchingUtilities` class:

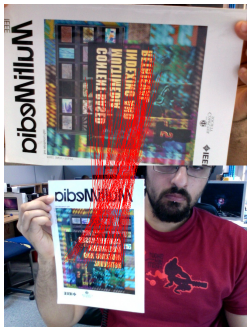
```
MBFImage basicMatches = MatchingUtilities.drawMatches(
    query, target, matcher.getMatches(), RGBColour.RED);
DisplayUtilities.display(basicMatches);
```

As you can see, the basic matcher finds many matches, many of which are clearly incorrect. A more advanced approach is to filter the matches based on a given geometric model. One way of achieving this in OpenIMAJ is to use a `ConsistentLocalFeatureMatcher` which given an internal matcher, a geometric model and a model fitter, finds which matches given by the internal matcher are consistent with respect to the model and are therefore likely to be correct.

An Affine transform models the transformation between two parallelograms.

To demonstrate this, we'll use an algorithm called Random Sample Consensus (RANSAC) to fit a geometric model called an **Affine transform** to the initial set of matches. This is achieved by iteratively selecting a random set of matches, learning a model from this random set and then testing the remaining matches against the learnt model.

We'll now set up our model, our RANSAC model fitter and our consistent matcher:



```
AffineTransformModel fittingModel = new
    AffineTransformModel(5);
RANSAC<Point2d, Point2d> ransac = new RANSAC<Point2d,
    Point2d>(fittingModel, 1500, new RANSAC.
    PercentageInliersStoppingCondition(0.5), true);

matcher = new ConsistentLocalFeatureMatcher2d<Keypoint
    >(
  new FastBasicKeypointMatcher<Keypoint>(8), ransac);

matcher.setModelFeatures(queryKeypoints);
matcher.findMatches(targetKeypoints);
MBFImage consistentMatches = MatchingUtilities.
    drawMatches(query, target, matcher.getMatches(),
    RGBColour.RED);
```

```
DisplayUtilities.display(consistentMatches);
```

The `AffineTransformModel` class models a two-dimensional Affine transform in OpenIMAJ. An interesting byproduct of this technique is that the `AffineTransformModel` contains the best transform matrix to go from the query to the target. We can take advantage of this by transforming the bounding box of our query with the transform estimated in the `AffineTransformModel`, therefore we can draw a polygon around the estimated location of the query within the target:

```
target.drawShape(query.getBounds().transform(
    fittingModel.getTransform().inverse()), 3,RGBColour.
    BLUE);
DisplayUtilities.display(target);
```

## Exercises

### Exercise 1: Different matchers

Experiment with different matchers; try the `BasicTwoWayMatcher` for example.

### Exercise 2: Different models

Experiment with different models (such as a `HomographyModel`) in the consistent matcher.

A `HomographyModel` models a **planar Homography** between two planes. Planar Homographies are more general than Affine transforms and map quadrilaterals to quadrilaterals.