

EEE3032 – Diagnostic Self-Test

Computer Vision and Pattern Recognition

Dr John Collomosse

Centre for Vision Speech and Signal Processing
Department of Electronic Engineering
University of Surrey

May 18, 2010

1 Introduction

Computer Vision and Pattern Recognition (CVPR) is a Level 3 undergraduate taught module, running in the Spring semester. The course covers software techniques that enable computers to recognise meaningful patterns in a visual signal (e.g. an image or video) and thereby to reason about the world. An example of a CVPR system is the London Congestion Charging system that can recognise the presence of cars in video, read their license plate, and process charging information.

Like many engineering topics, Computer Vision and Pattern Recognition is a form of applied mathematics. The purpose of this document is to help you assess your level of ability in the relevant maths topics, to help you make a decision as to whether you will be comfortable with material on this course.

1.1 Course Requirements

CVPR requires a basic knowledge of linear algebra. Linear algebra concerns vectors and matrices, and how matrix operations may be used to manipulate vector quantities. This is important because in CVPR we often deal with shapes (polygons) whose vertices are vector quantities (points in 2D space). Matrices can be used to transform these polygons in 2D space — for example matrices can be used to rotate, scale, or translate (shift) a shape. You need to be comfortable with these geometric interpretations of matrix transformations and the basic linear algebra from the out-set of this course. Modules covering these topics include Computer Graphics (*ee2.cvp*) and Computer Vision (*ee2.ivp*) modules. Level 2 Engineering Mathematics (EEE2005) covers some of this material although not from a geometric perspective. If you did not take those modules you can learn about these topics to the required level by undertaking some pre-reading before the module starts using the accompanying self-study guide.

Parts of CVPR require knowledge of some basic statistics and probability. You should be confident with the basic concepts of probability theory, to the level of a typical high-school maths course (for example the Statistics 1 module in AS Level Maths). At a couple of points in this module you will need an elementary working knowledge of complex numbers, i.e. be comfortable with using numbers that have a real and an imaginary component. All Electronic Engineering students will have covered complex numbers in their first year. If you have not encountered complex numbers, it is sufficient for this course to treat them as a 2D vector quantity. The question in this self-test gives an example of the kind of manipulation you should be able to perform.

Finally, this is a software course and you will need to be able to write small programs to do the lab exercises and the coursework. The lab exercises are to be programmed in Matlab, and the coursework in C or C++. If you have programmed before, you will find Matlab a straightforward language and in labs we will approach it from the basics — CVPR does not require prior experience in Matlab. However for the

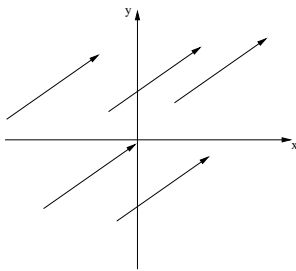
coursework assignment you will need an intermediate level of capability in C/C++ programming. If you have taken Electronic Engineering modules in C at Level 1 or 2, or the C++ module at Level 3, then this will be sufficient. Again you can use the self-test as a guide to what we expect.

2 Take the EEE3032 Self-Test

You should attempt this 4 part self-test on your own. A pass mark is given for each section of the test (based on 1 mark per question). If you “pass” all sections then you should feel confident that you are in a position to undertake the module. Solutions are given at the end of this document in section 3.

2.1 Linear Algebra (pass at 60% = 5 correct)

If you have difficulties with any questions in this test section (2.1), reading the self-study guide (pre-reading) for the module.



1. How many different lines, and how many different vectors are there in the above diagram?
2. Compute $|a|$, the magnitude (length) of 2D vector $a = (3, 4)^T$. Note in this test (and module) vectors are written in column form, i.e. $a = (3, 4)^T$ is shorthand for $a = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$.
3. Given a pair of vectors a and b , write down an expression for the angle θ between the two vectors.
4. Multiply these two matrices together: $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} p & q \\ r & s \end{pmatrix}$
5. The following matrix M transforms 2D point p in some way, to new location p' . Describe (in words) the transformation.

$$p' = Mp$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

If $\theta = \pi/2$ and $p = (5, 0)^T$, then what are the coordinates of p' ? (try to answer this last part by inspection, i.e. without numerically working the example)

6. Using a 2×2 matrix transformation (such in the previous question) we can perform a wide range of useful geometric operations in 2D. Why is it impossible to perform translation (shifting) of a point using a 2×2 matrix? Describe *homogeneous coordinates* and how they can accommodate translation in a 3×3 matrix framework.
7. A 2D point p is acted upon by matrix transformation M such that $p' = Mp$. The transformation is:

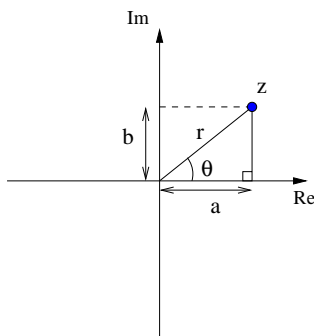
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Describe the action of M of p in the **active** sense, and in the **passive** sense. If you have not encountered this terminology before, refer to the self-study guide before answering this question.

2.2 Statistics and Probability (pass at 80% = 4 correct)

1. What are the mean, median and mode averages of this set of numbers $\{4, 3, 3, 4, 2, 6, 5, 4\}$?
2. Calculate the standard deviation and the variance of the above set of numbers.
3. The probability of throwing a 6 on a standard, fair six-sided die is $\frac{1}{6}$. You roll the die twice. What are the chance of rolling a double six?
4. Continuing from the previous question, when you roll the die twice what are the chances of throwing any double (e.g. the overall chance of throwing a double-1 or a double-2 or...)?
5. Bag A contains 3 oranges and 7 apples. Bag B contains 5 oranges and 5 apples. You pick a fruit from a bag. The chance that you picked Bag A is 20%. What is the probability you picked an apple?

2.3 Complex Numbers (pass at 100% = 1 correct)



1. The complex number z can be written in Cartesian form as $z = a + ib$, where $i = \sqrt{-1}$. z thus resembles a vector quantity and can be plotted on a 2D diagram (called an Argand diagram) as above.

Complex numbers are often used in signal processing to represent the magnitude and phase of a frequency component within a signal. These quantities are represented by r and θ respectively on the Argand diagram (and are also referred to as the polar form of a complex number $z = re^{i\theta}$).

Write down the expression for r and θ in terms of a and b , thus showing how to convert from the Cartesian form to the Polar form of z .

2.4 C/C++ Programming (pass at 50% = 3 correct)

1. How many times will XXXX be executed?

```
int i;
for (i=0; i<=10; i++) {
    XXXX
}
```

2. What will the following code output when run?

```
#include <stdio.h>

void foo(int a) { a=a+3; }

void bar(int* a) { (*a)=(*a)+3; }

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

    int a=3;

    foo(a);
    printf("%d",a);
    foo(a);
    printf("%d",a);
    bar(&a);
    printf("%d",a);
    foo(a);
    printf("%d",a);

}
```

3. The following code compiles but produces a Segmentation Fault (crashes) when run. What is the code supposed to do — and what is the likely reason for the crash?

```
#include <stdio.h>
#include <stdlib.h>

#define FILENAME "test.txt"

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

    FILE* fp=fopen(FILENAME,"r");

    int my_value;
    fscanf(fp,"%d",&my_value);
    printf("Value is %d",my_value);

    fclose(fp);

}
```

4. In the code below, *f* is an array of FEATURES (a user defined structure). Replace *XXXX* with a line of code that will print the value of field *a* from the second element of *f*.

```
struct FEATURE {  
  
    float a;  
    int b;  
  
};  
  
int main (int argc, char** argv) {  
  
    FEATURE f[3];  
  
    ... // some code to put values put into f  
  
    XXXX  
  
}
```

5. How many bytes apart are memory addresses *a* and *b*? Assume that an int takes 4 bytes of storage.

```
int main (int argc, char** argv) {  
  
    int a[100];  
  
    int* b=a+5;  
  
}
```

6. This question concerns dynamic memory allocation — you may or may not have covered this in your C/C++ course. Dynamic memory allocation may be useful but is not essential for your coursework.

The code below will compile and run, but it has a fault that will become apparent when *N* is set very large. What is the fault, how would it manifest itself, and how would you fix it?

```
int main (int argc, char** argv) {  
  
    int N=1000;  
    while (N > 0) {  
        foo();  
        N--;  
    }  
  
}  
  
void foo(void) {  
  
    int i;  
    int* a=(int*)calloc(1000,sizeof(int)); // C++ coders, use int* a=new int[1000];  
    for (i=0; i<1000; i++) {  
        a[i]=i;  
    }  
  
}
```

3 Solutions — EEE3032 Self-Test

3.1 Solutions - Linear Algebra (pass at 60% = 5 correct)

1. There are 5 lines and 1 vector. A vector is simply a direction and magnitude. In 2D this can be expressed by a pair of numbers (x,y). A line is defined by both a vector (i.e. direction and length) *and a starting position in the 2D space.*
2. The magnitude of a 2D vector $a = (a_x, a_y)^T$ is given by $|a| = \sqrt{a_x^2 + a_y^2}$. So in this example the magnitude is $\sqrt{3^2 + 4^2} = 5$.
3. Standard matrix multiplication:
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} p & q \\ r & s \end{pmatrix} = \begin{pmatrix} ap + br & aq + bs \\ cp + dr & cq + ds \end{pmatrix}$$
4. The dot product defines the angular relationship between two vectors: $a \cdot b = |a||b| \cos \theta$. So, to compute the angle between a and b we first normalise the vectors (divide them by their magnitude), so that $|a| = |b| = 1$. We then compute the dot product and take the inverse cosine of the result.
5. Matrix M performs a rotation anti-clockwise, θ degrees, about the origin. For the numerical example, it should be apparent that $\theta = \pi/2$ will cause a 90 degree rotation anti-clockwise about the origin — so a point (5,0) lying on the x-axis moves to (0,5) i.e. lying on the y-axis.
6. Translation (shifting) of a point requires the addition of a constant to one or more of the coordinates. In a 2×2 framework this is impossible because anything we add to one coordinate (e.g. x) is a multiple of another coordinate (e.g. y) :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

Homogeneous coordinates introduce an additional dimension to points. Points specified as (x, y) become $(x, y, 1)$. A translation can be performed by:

$$\begin{pmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ 1 \end{pmatrix}$$

And indeed any of the linear (2×2) transformations can be performed using the identity 3×3 matrix with the upper-left 2×2 section populated with the desired matrix.

More generally, if the third row of the matrix departs from the identity then we will obtain an output point $(\alpha x, \alpha y, \alpha)^T$. To obtain the 2D location of the output point we need to divide through by the extra (homogeneous) coordinate α .

Homogeneous coordinates allow many common transformations (such as rotation, translation, and even perspective projection operations) to be performed in a single (3×3) matrix framework — which has mathematical and engineering advantages for example when combining together multiple transformations.

7. We can interpret the action of a matrix transform in either the active, or the passive sense. The mathematics does not change; they are two different ways of thinking about the operation. If this solution is unclear you are advised to consult the more lengthy description of these concepts in the self-study guide.

The active interpretation considers the point to be moving in the 2D space. Think of a computer game character moving around a screen on a stationary terrain map. The passive interpretation considers the 2D space to be moving around the point. Think of a computer game character fixed in the centre of a screen, with the terrain map scrolling behind it. The net result is the same, the game's character moves toward particular location on the map.

Returning to this example of matrix transform in the **active** sense, we describe p as being scaled by a factor of 2. So p is scaled away from the origin by M , to new location p' .

In the **passive** sense, we do not describe p as moving per se, but instead think of the space p is in being transformed...

Specifically we consider p as defined within the reference frame of M , which comprises basis vectors $i = (2, 0)^T$ and $j = (0, 2)^T$ (defined by the columns of M). This is different to our usual (or “root”) reference frame where $i = (1, 0)^T$ and $j = (0, 1)^T$. A point with coordinates $(5, 0)^T$ in reference frame M actually has coordinates $(10, 0)^T$ in the root reference frame. One step away from the origin in M takes us two steps away from the origin in the root frame.

In the passive sense we consider the matrix transformation M as defining a reference frame (here, $i = (2, 0)^T$ and $j = (0, 2)^T$), and that the coordinates of point p are defined initially within that reference frame. When we perform the matrix multiplication we interpret this as moving from the coordinate system of reference frame M to the coordinate system of the root reference frame... where the point will be numerically twice as far from the origin.

3.2 Solutions - Statistics and Probability (pass at 80% = 4 correct)

1. Writing the numbers $\{x_1, x_2, \dots, x_N\}$, the mean is $\frac{1}{N} \sum_{i=1}^N x_i = 3.875$. The median of a set of numbers is the “middle” number in the set, having first sorted it (here, 4). The mode is the highest frequency number (here, 4).
2. Writing the set of numbers $\{x_1, x_2, \dots, x_N\}$, the mean $\mu = \frac{1}{N} \sum_{i=1}^N x_i$. The standard deviation of the set $\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$. The variance is the square of the standard deviation (hence often written σ^2). These quantities describes the distribution or “spread” of the numbers... i.e. how widely out they are around the mean value.
3. The events are independent, and so the combined (joint) probability of the two events occurring is simply the probability of one event multiplied by the probability of the other. The probability of throwing a six is $\frac{1}{6}$. The probability of throwing a six, following by another six is $\frac{1}{6} \times \frac{1}{6} = \frac{1}{36}$.
4. The probability of throwing a double of a particular number is $\frac{1}{36}$. The probability of throwing a double-anything is therefore $6 \times \frac{1}{36} = \frac{1}{6}$.
5. The probability of an apple given one has chosen bag A is $p(\text{apple}|A) = 0.7$. We call this a “conditional” probability. The conditional probability of an apple given bag B is chosen is $p(\text{apple}|B) = 0.5$. The probability of choosing bag A is $p(A) = 0.2$, and thus $p(B) = 0.8$. So, the probability of picking an apple is $p(\text{apple}) = P(\text{apple}|A)P(A) + P(\text{apple}|B)P(B) = (0.7 \times 0.2) + (0.5 \times 0.8) = 0.54$.

3.3 Solutions - Complex Numbers (pass at 100% = 1 correct)

1. The conversion is performed using $r = \sqrt{a^2 + b^2}$ and $\theta = \text{atan}(b/a)$. You can verify this with simple trigonometry by observing that a right-angled triangle is constructed against the *real* axis, with hypotenuse r and sides a and b . See the argand diagram in the question.

3.4 Solutions - C/C++ Programming (pass at 50% = 3 correct)

1. Executed 11 times, when $i=0$, $i=1, \dots$ and finally when $i=10$
2. The code will output `3366`. This is because the calls to `foo` pass value a by value, and the calls to `bar` pass a by reference. This means the a in `foo` is a local copy of the a in `main`, created when `foo` is called.. thus adding 3 to that local copy (which subsequently goes out of scope at the end of `foo` and is lost) has no effect. By contrast, deferencing the pointer to a in `bar` allows us to access the a local to function `main`.

3. The code is supposed to open the data file *test.txt*, read an integer from the file, and print it. A Segmentation Fault will arise if the file could not be found/opened. In that case variable *fp* will be NULL, and calling *fscanf* in that situation will cause the fault.
4. The code is as follows. The dot operator is used to reference fields in a struct, and array numbering in C starts at 0 (so the second element has index 1).

```
printf("%f",f[1].a);
```

Equivalently you could have used the arrow notation to access elements of the structure using a pointer to it... as *(f+1)* is a pointer to the second element:

```
printf("%f", (f+1)->a);
```

Note that *a* is a float, requiring the %f escape character in the printf.

5. The memory locations (pointers) *a* and *b* are 20 bytes apart. When we add 5 to *b* we added 5 lots of “int”-sized storage, and in this question you are told that an int takes 4 bytes (which is true for most common compilers/machines). If we had defined *a* and *b* as pointers to *char* then they would be 5 bytes apart (because a char takes 1 byte). Or if we defined them as pointers to *doubles* (which are 64bit = 8 bytes) the pointers would be 40 bytes apart. This is called *pointer arithmetic*.
6. The code has a “memory leak”. Memory is allocated in *foo* but not freed by the programmer when the variable *a* goes out of scope. This is different to Java, where memory gets freed automatically by the garbage collector when a variable goes out of scope. In C you must always remember to *free* anything you allocate (via *malloc* or *calloc*). If *N* were very large, you would run low on memory. You would notice this “by your code running slowly” as the operating system borrows storage from the hard disk to make up for the lack of free RAM (a process called paging/swapping).

Note, C++ users.. you would use “new” rather than “calloc” as indicated in the code comments within the question. To free up the memory you would use “delete” rather than “free”.