EPSRC Summer School on Computer Vision
Vision Algorithmics

Adrian F. Clark
VASE Laboratory, Dept. Electronic Systems Engineering
University of Essex, Colchester, CO4 3SQ
⟨alien@essex.ac.uk⟩

Contents

1 Introduction

The purpose of a PhD is to give training and experience in research. The principal output of a PhD is one or more contributions to the body of human knowledge. This might suggest that the main concern of a thesis is describing the contribution, the "Big Idea," but the reality is different: the thesis must present experimental evidence, analysed in a robust way, that *proves* the Big Idea is indeed valid. For theses in the computer vision area, this gathering and analysis of experimental evidence invariably involves programming algorithms. Indeed, informal surveys have found that PhD students working in vision typically spend over 50% of their time in programming, debugging and testing algorithms. Hence, becoming experienced in vision software development and evaluation is an important, if not essential, element of research training.

However, there are other things to consider in this context. Vision software has a reputation, entirely deserved, for being *fragile*: a technique that works perfectly well on one type of imagery — say imagery of natural scenes acquired from a mobile robot in a research laboratory — may fail dismally when applied to imagery acquired from a camera in a car driving along a road. If computer vision is ever going to be used significantly outside research laboratories, techniques *must* be made more robust. In the author's opinion at least, this is the main challenge that the upcoming generation of vision researchers needs to get to grips with.

Many researchers consider the programming of vision algorithms to be straightforward because it simply instantiates the underlying mathematics. This attitude is totally naïve, for it is well known that mathematically-tractable solutions do not guarantee viable computational algorithms. Indeed, the author considers a programming language to be a formal notation for expressing algorithms, one that can be checked by a machine, and a program to be the formal description of an algorithm, incorporating mathematical, numerical and procedural aspects.

Rather than present a dusty survey of the vision algorithms available in the 100+ image processing packages available commercially or for free, this essay tries to do something different: it attempts to illustrate, through the use of examples, the major things that you should bear in mind when coding up your own algorithms or looking at others' implementations. It concentrates largely on *what can go wrong*. The main idea the author is trying to instill is a suspicion of all results produced by software! Example code is written in C but the principles are equally valid in C++, Java, Matlab, Python or any other programming language you may use.

Having made you thoroughly paranoid about software issues, the document will then briefly consider what packages are available free of charge both to help you write vision software and analyse the results that come from it. That will be followed by a brief discussion that leads into the exploration of performance characterisation in a later lecture in the Summer School.

2 Failure of Programming Model

Let us consider the first area in which computer vision programs often go wrong. The problem, which is a fundamental one in programming terms, is that the conceptual model one is using as the basis of the software has deficiencies.

Perhaps the best way to illustrate this is by means of an example. The following C code performs image differencing, a simple technique for isolating moving areas in image sequences.

```
typedef unsigned char byte;

void sub_ims (byte **i1, byte **i2, int ny, int nx)
{
  int y, x;

  for (y = 0; y < ny; y++)
    for (x = 0; x < nx; x++)
      i1[y][x] = i1[y][x] - i2[y][x];
}
```

The procedure consists of two nested loops, the outer one scanning down the lines of the image and the inner one accessing each pixel of the line; it was written with exposition in mind, not speed. Let us consider what is good and bad about the code.

Firstly, the code accesses the pixels in the correct order. Two-dimensional arrays in C are represented as 'arrays of arrays,' which means that the last subscript should cycle fastest to access adjacent memory locations. On PC- and workstation-class machines, all of which have virtual memory subsystems, failure to do this could lead to large numbers of page faults, substantially slowing down the code. Incidentally, it is commonly reported that this double-subscript approach is dreadfully inefficient as it involves multiplications to subscript into the array — but that is complete bunkum!

There are several bad aspects to the code. Arrays `i1` and `i2` are declared as being of type `unsigned char`, which means that pixel values must lie in the range 0–255; so the code is constrained to work with 8-bit imagery. This is a reasonable assumption for the current generation of video digitizers, but the software cannot be used on data recorded from a 10-bit flat-bed scanner, or from a 14-bit remote sensing satellite, without throwing some potentially important resolution away.

What happens if, for some particular values of the indices `y` and `x`, the value in `i2` is greater than that in `i1`? The subtraction will yield a negative value, one that cannot be written back into `i1`. Some run-time systems will generate an exception; others will silently reduce the result modulo 256 and continue, so that the user erroneously believes that the operation succeeded. In any case, one will not get what one expected. The simplest way around both this and the previous problem is to represent each pixel as a 32-bit signed integer rather than as an unsigned byte. Although images will then occupy four times as much memory (and require longer to read from disk, *etc.*), it is always better to get the right answer slowly than an incorrect one quickly. Alternatively, one could even represent pixels as 32-bit floating-point numbers as they provide the larger dynamic range (at the cost of reduced accuracy) needed for things like Fourier transforms. On modern processors, the penalty in computation speed is negligible for either 32-bit integers or floating-point numbers. Indeed, because of the number of integer units on a single processor and the way its pipelines are filled, floating-point code can sometimes run faster than integer code!

You should note that this problem with overflowing the representation is not restricted to subtraction: addition and multiplication have at least as much potential to wreak havoc with the data. Division requires even more care for, with a purely integer representation, one must either recognise that the fractional part of any division will be discarded or remember to include code that rounds the result, as in

```
i1[y][x] = (i1[y][x] + 255) / i2[y][x];
```

for 8-bit unsigned imagery; the number added changes to 16383 for 16-bit unsigned imagery, and so on.

There are ways of avoiding these types of assumptions. Although it is not really appropriate to go into these in detail in this document, an indication of the general approaches is relevant. Firstly, by making a class (or structured data type in non-object-oriented languages such as C) called '`image`', one can arrange to record the data type of the pixels in each image; a minimal approach in C is:

```
struct {
  int type;
  void *data;
} image;
...
image *i1, *i2;
```

Inside the processing routine, one can then select separate loops based on the value of `i1->type`.

The code also has serious assumptions implicitly built in. It assumes that images comprise only one channel of information, so that colour or multi-spectral data require multiple invocations. More seriously, it assumes that an image can be held entirely in computer memory. Again, this is reasonable for imagery grabbed from a video camera but unreasonable for 6000×6000 pixel satellite data or for images recorded from 12 Mpixel digital cameras.

The traditional way of avoiding having to store the entire image in memory is to employ line-by-line access:

```
for (y = 0; y < ny; y++) {
  buf = getline (y);
  for (x = 0; x < nx; x++)
    ...operate on buf[x]...
  putline (y, buf);
}
```

This doesn't actually have to involve line-by-line access to a disk file as `getline` can return a pointer to the line of an image held in memory: it is *logical* line-by-line access. This approach has been used by the author and colleagues to produce code capable of being used *unchanged* on both serial and parallel hardware as well as on compute clusters ("grid computing").

3 Programming Versus Theory

Sticking with this notion of using familiar image processing operations to illustrate problems, let us consider convolution performed in image space. This involves multiplying the pixels in successive regions of the image with a set of coefficients, putting the result of each set of computations back into the middle of the region. For example, the well-known Laplacean operator uses the coefficients

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

while the coefficients

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

yields a 3×3 blur. When programming up any convolution operator that uses such a set of coefficients, the most important design question is *what happens at the edges?* There are several approaches in regular use:

- don't process the edge region, which is a one-pixel border for a 3×3 'mask' of coefficients, for example;
- reduce the size of the mask as one approaches the edge;
- imagine the image is reflected along its first row and column and program the edge code accordingly;
- imagine the image wraps around cyclically.

Which of these is used is often thought to be down to the whim of the programmer. *But it is not* as only one of these choices matches the underlying mathematics.

The correct solution follows from the description of the operator as a *convolution* performed in image space. Mathematically, convolution is usually performed in Fourier space: the image is Fourier transformed to yield a spectrum; that spectrum is multiplied by a filter, and the resulting product inverse-transformed. The reason that convolutions are not programmed in this way in practice is that, for small-sized masks, the Fourier approach requires more computation, even when using the FFT discussed below. The use of Fourier transformation, or more precisely the *discrete Fourier transform*, involves implicit assumptions; the one that is relevant here is that the data being transformed are cyclic in nature, as though the image is the unit cell of an infinitely-repeating pattern. So the only correct implementation, at least as far as the theory is concerned, is the last option listed above.

4 Numerical Issues

One thing that seems to be forgotten far too often these days is that floating-point arithmetic is not particularly accurate. The actual representation of a floating-point number is $m \times 2^e$, analogous to scientific notation, where m, the *mantissa*, can be thought of as having the binary point immediately to its left and e, the *exponent*, is the power of two that 'scales' the mantissa. Hence, (IEEE-format) 32-bit floating-point corresponds to 7–8 decimal digits and 64-bit to roughly twice that. Because of this and other issues, *floating-point arithmetic is significantly less accurate than a pocket calculator!* The most common problem is that, in order to add or subtract two numbers, the representation of the smaller number must be changed so that it has the same exponent as the larger, and this involves right-shifting binary digits in the mantissa to the right. This means that, if the numbers differ by about 10^7, all the digits of the mantissa are shifted out and the lower number effectively becomes zero. The 'obvious' solution is to use 64-bit floating-point (`double` in C) but this simply postpones the problem to bigger numbers, it does not solve it.

You might think that these sorts of problems will not occur in computer vision; after all, images generally involve numbers only in the range 0–255. But that is simply not true; let us consider two examples of where this kind of problem can bite the unwary programmer.

The first example is well-known in numerical analysis. The task of solving a quadratic equation crops up surprisingly frequently, perhaps in fitting to a maximum when trying to locate things accurately in images, or when intersecting vectors with spheres when working on 3D problems. The solution to

$$ax^2 + bx + c = 0$$

is something almost everyone learns at school:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

and indeed this works well for many quadratic problems. But there is a numerical problem hidden in there.

When the discriminant, $b^2 - 4ac$, involves values that make $b^2 \gg 4ac$, the nature of floating-point subtraction, alluded to above, can make $4ac \to 0$ relative to b^2 so that the discriminant becomes $\pm b$... and this means that the lower of the two solutions to the equation is $-b + b = 0$.

Fortunately, numerical analysts realized this long ago and have devised mathematically-equivalent formulations that do not suffer from the same numerical instability. If we first calculate

$$q = -\frac{1}{2}\left(b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}\right)$$

then the two solutions to the quadratic are given by

$$x_1 = c/q$$

and

$$x_2 = q/a.$$

Even code that looks straightforward can lead to difficult-to-find numerical problems. Let us consider yet another simple image processing operation, finding the standard deviation

(SD) of the pixels in an image. The definition of the SD is straightforward enough

$$\frac{1}{N}\sum_{i=1}^{N}(x_i-\bar{x})^2$$

where $\bar{x}$ is the mean of the image and N the number of pixels in it. (We shall ignore the distinction between population and sample SDs, which is negligible for images.) If we program this equation to calculate the SD, the code has to make two passes through the image: the first calculates the mean, $\bar{x}$, while the second finds deviations from the mean and hence calculates the SD.

Most of us probably learnt at school how to re-formulate this: substitute the definition of the mean in this equation and simplify the result. We end up needing to calculate

$$\sum x^2 - \frac{(\sum x)^2}{N}$$

which can be programmed up using a single pass through the image as follows.

```
float sd (image *im, int ny, int nx)
{
  float v, var, sum = sum2 = 0.0;
  int y, x;

  for (y = 0; y < ny; y++) {
    for (x = 0; x < nx; x++) {
      v = im[y][x];
      sum = sum + v;
      sum2 = sum2 + v * v;
    }
  }
  v = nx * ny;
  var = (sum2 - sum * sum/v) / v;
  if (var <= 0.0) return 0.0;
  return sqrt(var);
}
```

I wrote such a routine myself while doing my PhD and it was used by about ten people on a daily basis for about two years before someone came to see me with a problem: on his image, the routine was returning a value of zero for the SD, even though there definitely was variation in the image.

After a few minutes' examination of the image, we noticed that the image in question comprised large values but had only a small variation. This identified the source of the problem: the single-pass algorithm involves a subtraction and the quantities involved, which represented the sums of squares or similar, ended up differing by less than one part in $\sim 10^7$ and hence yielded inaccurate results from my code. I introduced two fixes: the first was to change `sum` and `sum2` to be `double` rather than `float`; and to look for cases where the subtraction in question yielded a result that was was small or negative and, in those cases, calculate only the mean and then make a second pass through the image. It's a fairly *ad hoc* solution but proved adequate, and I and my colleagues haven't been bitten again.

5 The Right Algorithm in The Right Place

The fast Fourier transform (FFT) is a classic example of the distinction between mathematics and 'algorithmics.' The underlying discrete Fourier transform is normally represented as a matrix multiplication; hence, for N data points, $O(N^2)$ complex multiplications are required to evaluate it. The (radix-2) FFT algorithm takes the matrix multiplication and, by exploiting symmetry properties of the transform kernel, reduces the number of multiplications required to $O(N \log_2 N)$; for a 256×256 pixel image, is a saving of roughly

$$\left(\frac{256^2}{256 \times 8}\right)^2 = \left(\frac{2^{16}}{2^{11}}\right)^2 = 1024 \text{ times!}$$

These days, an FFT is entirely capable of running in real time on even a moderately fast processor; a DFT is not.

On the other hand, one must not use cute algorithms blindly. When median filtering, for example, one obviously needs to determine the median of a set of numbers, and the obvious way to do that is to sort the numbers into order and choose the middle value. Reaching for our textbook, we find that the 'quicksort' algorithm (see, for example, [1]) has the best performance for random data, again with computation increasing as $O(N \log_2 N)$, so we choose that. But $O(N \log_2 N)$ is its *best-case* performance; its *worst-case* performance is $O(N^2)$, so if we have an unfortunately-order set of data, quicksort is a poor choice! A better compromise is probably Shell's sort algorithm: its best-case performance isn't as good as that of the quicksort but its worst-case performance is nowhere near as bad — and the algorithm is simpler to program and debug.

Even selecting the 'best' algorithm is not the whole story. Median filtering involves working on many small sets of numbers rather than one large set, so the way in which the sort algorithm's performance scales is not the overriding factor. Quicksort is most easily implemented recursively, and the time taken to save registers and generate a new call frame will probably swamp the computation, even on a RISC processor; so we are pushed towards something that can be implemented iteratively with minimal overhead — which might even mean a bubble sort! But wait: why bother sorting at all? There are median-finding algorithms that do not involve re-arranging data, either by using multiple passes over the data or by histogramming. One of these is almost certainly faster.

Where do you find out about numerical algorithms? The standard reference these days is [2], though I should warn you that many numerical analysts do not consider its algorithms to be state-of-the-art, or even necessarily good. (A quick web-search will turn up many discussions regarding the book and its contents.) The best numerical algorithms that the author is aware of are those licensed from NAg, the Numerical Algorithms Group. All UK higher education establishments should have NAg licenses.

Considerations of possible numerical issues such as the ones highlighted here are important if you need to obtain accurate answers or if your code forms part of an active vision system.

6 Vision Packages

If you've been reading through this document carefully, you're probably wondering whether it is ever possible to produce vision software that works reliably. Well, it *is* possible — but you cannot tell whether or not a piece of software has been written carefully and uses sensible numerical techniques without looking at its source code. For that reason, the author, like many other vision researchers, has a strong preference for open-source software. This wish to see 'inside the box' mitigates against Matlab, which has become quite popular in the vision community in recent years, and that is a shame because it provides fairly comprehensive graphical facilities and an adequate command language, though there will be more to say concerning the latter below.

What is out there in there in the open source world? There are many vision packages, though all of them are the results of fairly small groupings of researchers and developers. Particular ones that the author has looked at recently include:

OpenCV: `http://opencv.sourceforge.net/`
Growing from a project sponsored by Intel, OpenCV has become increasingly popular since it was used in DARPA's Grand Challenge concerning autonomous navigation around a course in the Mojave desert. Intended for active vision, and therefore fairly fast, it is becoming a popular repository for algorithms, ranging from Tsang's stereo calibration algorithm to the face recognition technique of Viola and Jones. There are also compatible implementations of other mainstream algorithms, *e.g.* SIFT.

VXL: `http://vxl.sourceforge.net/`
A heavily-templated C++ class library being developed jointly by people in a few research groups, including (the author believes) Oxford and Manchester. VXL grew out of Target Jr, which was a prototype of the Image Understanding Environment, a well-funded initiative to develop a common programming platform for the vision community. (Unfortunately, the IUE's aspirations were beyond what could be achieved with the hardware and software of ten years ago, and it failed. The community has not yet come back together to work towards a coherent end, sadly.)

Tina: `http://www.tina-vision.net/`
Tina is probably the only vision library currently around that makes a conscious effort to provide facilities that are statistically robust. While this is a great advantage and a lot of work has been put into making it more portable and easy to work with, it is fair to say that a significant amount of effort needs to be put into learning to use it.

In the author's opinion, the biggest usability problem with current computer vision packages is that they cater principally for the programmer: they rarely provide facilities for prototyping algorithms via a command language or for producing graphical user interfaces; and their visualisation capabilities are typically not great. These two deficiencies are ones that Matlab, for example, addresses very well.

Essentially all of the vision packages available today, free or commercial, are designed for a single processor; they cannot spread computation across a cluster, for example, and that is a problem for algorithms that may take minutes to run and consequently tens of hours to evaluate. There is a definite need for something that does this, provides a scripting language and good graphical capabilities, is based around solid algorithms, and is well tested. The author has recently started working on such a beast based around the capabilities of numerical Python ('numpy'); perhaps next year, some of the fruits of that will be worth including in this lecture. In the meantime, contact him if you're interested in joining in.

With the exception of OpenCV, which includes a (limited) machine learning library, vision packages work on images. Once information has been extracted from images, you usually have to find other facilities. This is another distinct shortcoming, as the author is certain that many task-specific algorithms are very weak in their processing of the data extracted from images. However, there are packages around that can help:

Image format conversion. It sometimes seems that there are as many different image formats as there are research groups! You are almost certain to need to convert the format of some image data files, and there are two good toolkits around that makes this very easy, both of which are available for Windows and Unix (including Linux and MacOS X). Firstly, there is NETPBM, an enhancement of Jef Poskanzer's PBMPLUS (`http://netpbm.sourceforge.net/`): this is a set of programs that convert most popular formats to and from its own internal format. If you use an image format within your research group that isn't already supported by NETPBM, the author's advice to you is to write compatible format converters as quickly as possible; they'll prove useful at some point. The second package worthy of mention is ImageMagick (`http://www.imagemagick.org/`), which includes both a good converter and a good display program. There are also interfaces between ImageMagick and the Perl, Python and Tcl interpreters, which can be useful.

Statistical software. The statistics research community has been good at getting its act together, much better than the vision community, and much of its research now provides facilities that interface to `R` (`http://www.r-project.org/`). `R` is a free implementation of S (and S+), commercial statistical packages that grew out of work at Bell Laboratories. `R` is available for both Unix and Windows (it's a five-minute job to install on Debian Linux, for example) and there are books around that provide introductions for all levels of users. Unusually for free software, it provides good facilities for graphical displays of data.

Neural networks. Many good things have been said about netlab (`http://www.ncrg.aston.ac.uk/netlab/`), a Matlab plug-in that provides facilities rarely seen elsewhere.

Genetic algorithms. Recent years have seen genetic algorithms gain in popularity as a robust optimisation mechanism. There are many implementations of GAs around the Net; but the author has written a simple, real-valued GA package with a root-polishing option which seems to be both fast and reliable. It's available in both C and Python; do contact him if you would like a copy.

Whatever you choose to work with, whether from this list or not, treat any results with the same suspicion as you treat those from your own code.

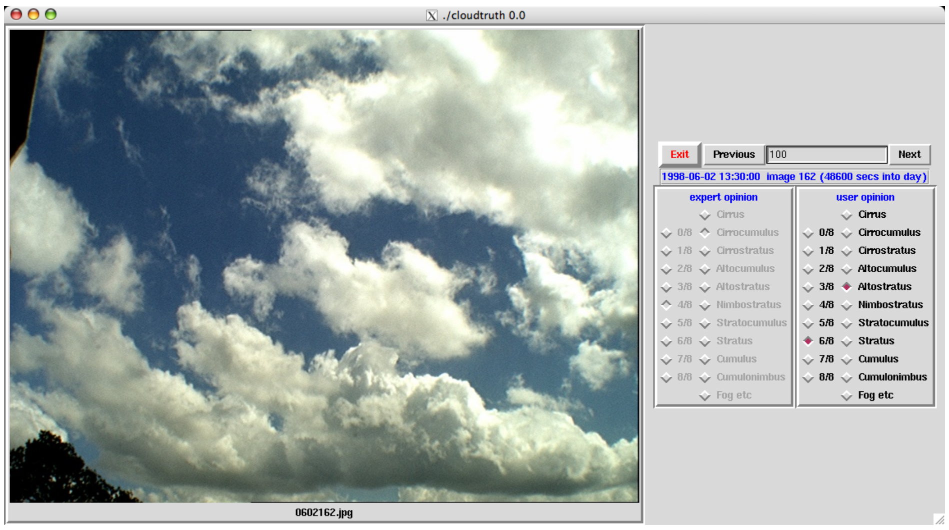
7 Massaging Program Outputs

One of the things you will inevitably have to do is take the output from a program and manipulate it into some other form. This usually occurs when "plugging together" two programs or when trying to gather success and failure rates. To make this easier, there are two specific things that make your life easier:

- learn a scripting language; the most suitable is currently Python, largely because of the excellent `numpy` and `scipy` extensions;
- don't build a graphical user interface (GUI) into your program.

Scripting languages are designed to be used as software 'glue' between programs; they all provide easy-to-use facilities for processing textual information, including regular expressions. Which of them you choose is largely up to personal preference; the author has used Lua, Perl, Python, Ruby and Tcl — after a long time working with Perl and Tcl has finally settled on Python. Indeed, most of the author's data analysis programming is done in a scripting language as it is really a much more time-efficient way to work.

Building a GUI into a piece of research code is a mistake because it makes it practically impossible to use in any kind of automated processing. A much better solution is to produce a comprehensive command-line interface, one that allows most internal parameters and thresholds to be set, and then use one of the scripting languages listed above to produce the GUI. For example, the author built the interface shown in Figure 1 using Tcl and its Tk graphical toolkit in about two hours. The interface allows the user to browse through a set of nearly 8,000 images and an expert's opinion of them, fine-tuning them when the inevitable mistakes crop up. This 250-line GUI script works on Unix (including Linux and MacOS X) and Windows entirely unchanged.



100

1998-06-02 13:30:00 image 162 (48600 secs into day)

user opinion: [✓ 6/8] [✓ Altostratus]

Figure 1: Custom-designed graphical interface using a scripting language

8 Concluding Remarks

The aim of this essay is to make you aware of the major classes of problem that can and do crop up in vision software. Beyond that, the major message to take home is *not to believe results*, either those from your own software or anyone else's, without checking. A close colleague will often spend a day getting a feel for the nature of her imagery, and she can spot a problem in a technique better than anyone else I've ever worked with. That is not, I suggest, entirely coincidence.

Another important thing is to see if changing tuning parameters has the effect that you expect. For example, one of the author's students recently wrote some stereo software. Its accuracy was roughly as anticipated but, following my suggestion that she explore the software using simulated imagery, found that accuracy decreased as the image size increased. She was uneasy about this and, after checking with me for a 'second opinion,' took a good look at her software. Sure enough, she found a bug and, re-running the tests after fixing it, found that accuracy then increased with image size — in keeping with both our expectations. Without both a feel for the problem and carrying out experiments to see that expected results do indeed occur, the bug may have lain dormant in the code for weeks or months, only to become apparent after more research had been built on it — effort that would have to be repeated.

This idea of exploring an algorithm by varying its parameters is actually an important one. Measuring the success and failure rates as each parameter is varied allows the construction of receiver operating characteristic (ROC) curves, currently one of the most popular ways of presenting performance data in vision papers.

However, it is possible to go somewhat further. Let us imagine we have written a program that processes images of the sky and attempts to estimate the proportion of the image that has cloud cover (a surprisingly difficult task); this is one of the problems that the data in Figure 1 are used for. A straightforward evaluation of the algorithm, the kind presented in most papers, would simply report how often the amount of cloud cover detected by the program matches the expert's opinion. However, as the expert also indicated the predominant type of cloud in the image, we can explore the algorithm in much more detail. We might expect that a likely cause of erroneous classification is mistaking thin, wispy cloud as blue sky. As both cirrus and altostratus are thin and wispy, we can run the program against the database of images and see how often the estimate of cloud cover is wrong when these types of cloud occur. This approach is much, much more valuable: it gathers evidence to see if there is a particular type of imagery on which our program will fail, and that tells us where to expend effort in improving the algorithm. This is performance *characterisation*, the first step towards producing truly robust vision algorithms — and that forms the basis of the material that Neil Thacker will present in a later Summer School lecture.

References

[1] Robert Sedgewick. *Algorithms in C.* Addison-Wesley, 1990.

[2] William H. Press, Saul A. Teukolsky, Willian T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992.