

EPSRC Vision Summer School

Vision Algorithmics

Adrian F. Clark

`<alien@essex.ac.uk>`

VASE Laboratory, Comp Sci & Elec Eng
University of Essex

Introduction

- Roughly 50% of your PhD time will be spent on practical work
- As vision software development and evaluation will consume so much time, it is in your interest to become proficient at it
- Vision software is **fragile** and needs to be made more robust
- My belief is that it's because too many researchers naïvely believe vision software development just involves “programming the maths”

Introduction

- Roughly 50% of your PhD time will be spent on practical work
- As vision software development and evaluation will consume so much time, it is in your interest to become proficient at it
- Vision software is **fragile** and needs to be made more robust
- My belief is that it's because too many researchers naïvely believe vision software development just involves “programming the maths”

Introduction

- Roughly 50% of your PhD time will be spent on practical work
- As vision software development and evaluation will consume so much time, it is in your interest to become proficient at it
- Vision software is **fragile** and needs to be made more robust
- My belief is that it's because too many researchers naïvely believe vision software development just involves “programming the maths”

Overview of the session

- 1 What can go wrong
 - Failure of programming model
 - Programming versus theory
 - Numerical issues
 - The right algorithm in the right place
- 2 Vision packages
- 3 Massaging program outputs
- 4 Concluding remarks

What can go wrong

I'll concentrate on four major causes of problems when programming vision software:

- Failure of programming model
- Programming versus theory
- Numerical issues
- The right algorithm in the right place

Each of these will be illustrated by simple examples.

What can go wrong

I'll concentrate on four major causes of problems when programming vision software:

- Failure of programming model
- Programming versus theory
- Numerical issues
- The right algorithm in the right place

Each of these will be illustrated by simple examples.

Programming models

Look at the following C code and try to spot what's good and bad about it.

```
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];
}
```


Good features

- The code is easy to read and (hopefully) understand.
- The code accesses the pixels in the correct order: in C, 2D arrays are ‘arrays of arrays,’ stored so that the last subscript addresses adjacent memory locations.
- Incidentally, it is commonly reported that this double-subscript approach is dreadfully inefficient as it involves multiplications to subscript into the array — complete bunkum!

Good features

- The code is easy to read and (hopefully) understand.
- The code accesses the pixels in the correct order: in C, 2D arrays are ‘arrays of arrays,’ stored so that the last subscript addresses adjacent memory locations.
- Incidentally, it is commonly reported that this double-subscript approach is dreadfully inefficient as it involves multiplications to subscript into the array — complete bunkum!

Good features

- The code is easy to read and (hopefully) understand.
- The code accesses the pixels in the correct order: in C, 2D arrays are ‘arrays of arrays,’ stored so that the last subscript addresses adjacent memory locations.
- Incidentally, it is commonly reported that this double-subscript approach is dreadfully inefficient as it involves multiplications to subscript into the array — complete bunkum!

Bad features

- The arrays are declared as `unsigned char`, so pixel values must lie in the range 0–255
- Hence, the code is constrained to work with 8-bit imagery; it cannot be used with 10-bit scanner images, 14-bit remotely-sensed data *etc.*
- The code fails when `i2[y][x] > i1[y][x]`

Bad features

- The arrays are declared as `unsigned char`, so pixel values must lie in the range 0–255
- Hence, the code is constrained to work with 8-bit imagery; it cannot be used with 10-bit scanner images, 14-bit remotely-sensed data *etc.*
- The code fails when $i2[y][x] > i1[y][x]$

Underflow and overflow

- Most run-time systems don't generate an exception for integer underflow or overflow, so you don't know when this kind of thing happens
- The problem due to subtraction is not unique: addition and multiplication are just as likely to cause problems
- Division is even worse as integer division discards the fractional part; so you have to do things like

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

for 8-bit data

Underflow and overflow

- Most run-time systems don't generate an exception for integer underflow or overflow, so you don't know when this kind of thing happens
- The problem due to subtraction is not unique: addition and multiplication are just as likely to cause problems
- Division is even worse as integer division discards the fractional part; so you have to do things like

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

for 8-bit data

Underflow and overflow

- Most run-time systems don't generate an exception for integer underflow or overflow, so you don't know when this kind of thing happens
- The problem due to subtraction is not unique: addition and multiplication are just as likely to cause problems
- Division is even worse as integer division discards the fractional part; so you have to do things like

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

for 8-bit data

Extended representations

- The obvious solution to these problems is to use something with a longer representation than `unsigned char`, such as a 32-bit integer
- In fact, using a floating-point representation is attractive as it provides a greater dynamic range than integers (needed for Fourier-space processing, for example) and doesn't have the performance penalty it had a decade or so ago

Memory consumption

- The most fundamental decision built into the code is that the entire image can fit into memory; changing that would involve totally re-writing the code
- Is this important in these days where PCs have > 1 Gb RAM?
- It is if you want your software to be able to run on a handheld or a 'phone, or if it has to be used in an embedded system, or if it is astonishingly large

Memory consumption

- The most fundamental decision built into the code is that the entire image can fit into memory; changing that would involve totally re-writing the code
- Is this important in these days where PCs have > 1 Gb RAM?
- It is if you want your software to be able to run on a handheld or a 'phone, or if it has to be used in an embedded system, or if it is astonishingly large

Line-by-line access

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.

Programming versus theory

Think of programming a simple 3×3 blur. **What happens at the edges?**

- don't process the edge region
- 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

Only the last of these agrees with Fourier theory, which is the basis of convolution.

Programming versus theory

Think of programming a simple 3×3 blur. **What happens at the edges?**

- don't process the edge region
- 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

Only the last of these agrees with Fourier theory, which is the basis of convolution.

Programming versus theory

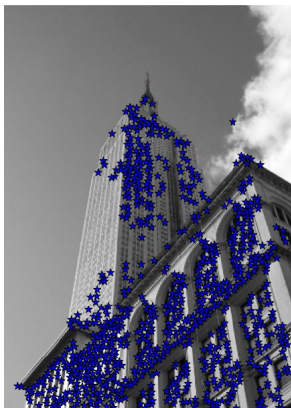
Think of programming a simple 3×3 blur. **What happens at the edges?**

- don't process the edge region
- 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

Only the last of these agrees with Fourier theory, which is the basis of convolution.

Systematic errors

Let's consider the corner detector due to Harris & Stephens.



The reason the corners are in the wrong place is that there is a *systematic error* in the algorithm. With a little care, you can overcome this.



What about the OpenCV implementation of Harris & Stephens?
Haven't you tested it?

Numerical issues

- A floating-point number is stored as $m \times 2^e$
- **Floating-point arithmetic is significantly less accurate than a pocket calculator!**
- 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 shifting binary digits in the mantissa
- If the numbers differ by about 10^7 , all the digits of the mantissa are shifted out and the lower number effectively becomes zero

Solution of a quadratic

The task of solving a quadratic equation crops up surprisingly frequently. The solution to

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

is something almost everyone learns at school:

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

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

Solution of a quadratic

The task of solving a quadratic equation crops up surprisingly frequently. The solution to

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

is something almost everyone learns at school:

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

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

Numerically-stable solution

If we first calculate

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

then the two solutions to the quadratic are given by

$$x_1 = c/q$$

and

$$x_2 = q/a$$

Calculating the standard deviation

- The definition of the s.d. is straightforward enough

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

- If we program this equation to calculate the s.d., the code has to make two passes through the image
- We can simplify the equation to produce

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

which requires only one pass through the image

```
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);
```

The right algorithm in the right place

- It is important to choose an efficient algorithm
- But don't go overboard: only do so where it makes sense

The fast Fourier transform

- The discrete Fourier transform (DFT) is naturally derived as a matrix multiplication; this takes $O(N^2)$ multiplications for an N -point transform.
- The FFT algorithm makes use of symmetry properties of the transform matrix to reduce the multiplication count to $O(N \log_2 N)$.
- For a 256×256 image, this gives a saving of about 1,000 times!

Sorting for a median filter

However, you mustn't use fast algorithms blindly. For median filtering, for example, you need to determine the median of many sets of numbers.

- Textbooks show the 'quicksort' algorithm to be fastest; but its **worst-case** performance is poorer than many other sort algorithms.
- Quicksort is normally implemented recursively: for sorting 9 or 25 numbers, the procedure-call overhead probably dominates. In fact, Shell's sorting algorithm is probably better.
- However, there are median-finding algorithms that do not involve sorting; one of these is probably faster!

Sorting for a median filter

However, you mustn't use fast algorithms blindly. For median filtering, for example, you need to determine the median of many sets of numbers.

- Textbooks show the 'quicksort' algorithm to be fastest; but its **worst-case** performance is poorer than many other sort algorithms.
- Quicksort is normally implemented recursively: for sorting 9 or 25 numbers, the procedure-call overhead probably dominates. In fact, Shell's sorting algorithm is probably better.
- However, there are median-finding algorithms that do not involve sorting; one of these is probably faster!

Programming issues: a summary

- You're probably wondering whether it is ever possible to produce vision software that is efficient and works reliably.
- **Of course it is** — but you cannot tell without looking at the source code.
- Hence, many vision researchers have a strong preference for open-source software.

Programming issues: a summary

- You're probably wondering whether it is ever possible to produce vision software that is efficient and works reliably.
- **Of course it is** — but you cannot tell without looking at the source code.
- Hence, many vision researchers have a strong preference for open-source software.

Vision software

- Matlab:** good if you want to **use**, rather than **develop**, vision algorithms
- OpenCV:** looks as though it might become the normal way to disseminate vision algorithms — but it's intended for *speed* rather than *accuracy*
- Tina: a C library that makes a conscious effort to provide statistically-robust techniques, though not the easiest to use
- EVE: (Advertisement) A pure Python (`numpy`) set of some important image processing and vision techniques

Vision software

- Matlab:** good if you want to **use**, rather than **develop**, vision algorithms
- OpenCV:** looks as though it might become the normal way to disseminate vision algorithms — but it's intended for *speed* rather than *accuracy*
- Tina:** a C library that makes a conscious effort to provide statistically-robust techniques, though not the easiest to use
- EVE:** ⟨Advertisement⟩ A pure Python (`numpy`) set of some important image processing and vision techniques

Useful software

Numerical methods: NAG, BLAS, [numerical recipes]

Image format conversion: NETPBM and ImageMagick

Statistical software: R

Neural networks: netlab


Genetic algorithms: genesis, ECJ

Massaging program outputs

- At some point, you will have to take the output from a program and manipulate it into some other form
- learn a scripting language such as Perl, Python, Ruby or Tcl
- don't build a graphical user interface (GUI) into your program

Scripting languages are designed to be used as software 'glue' between programs; they provide facilities for processing text, including regular expressions.

./cloudtruth 0.0



0602162.jpg

Exit Previous 100 Next

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

expert opinion		user opinion	
0/8	◊ Cirrus	0/8	◊ Cirrus
1/8	◊ Cirrocumulus	0/8	◊ Cirrocumulus
2/8	◊ Cirrostratus	1/8	◊ Cirrostratus
3/8	◊ Altostratus	2/8	◊ Altostratus
4/8	◊ Altostratus	3/8	◊ Altostratus
4/8	◊ Nimbostratus	4/8	◊ Nimbostratus
5/8	◊ Stratocumulus	5/8	◊ Stratocumulus
6/8	◊ Stratus	6/8	◊ Stratus
7/8	◊ Cumulus	7/8	◊ Cumulus
8/8	◊ Cumulonimbus	8/8	◊ Cumulonimbus
	◊ Fog etc.		◊ Fog etc.

Concluding remarks

- **Do not believe results**, either those from your own software or anyone else's, without checking
- Don't be scared to spend a few hours getting a 'feel' for the nature of your data
- Check out your algorithm by varying the input data: ensure it does what you expect, or find out why your expectations are wrong
- See if you can find out how the performance of your algorithm depends on features of the input and use that as a way of improving it — not performance **evaluation** but performance **characterisation**