

## Computational Projects

### Lecture 1: Introduction & simple algorithms

Dr Rob Jack, DAMTP

<http://www.maths.cam.ac.uk/undergrad/catam/part-ia-lectures>

Moodle: [CATAM IA: Lectures and MATLAB Sessions](#)

## Overview

- CATAM : Computer-Aided Teaching of All Mathematics
- There are no end-of-year examination questions
- You will work on projects during the academic year (Parts IB & II)
- You earn marks by submitting project reports (along with associated computer code)

## plan for Part IB

- MATLAB manual (exercises) & introductory project over summer
- Supervision at start of Michaelmas term (organised by colleges)
- 2 core projects due start of Lent term
- 2 additional projects (selection of 4) due start of Easter term
- Maximum credit: 160 tripos marks
- For the average student, CATAM contributes 20-25% of marks

## plan for Part II

- Many projects on offer (approx 30)
- Different weights due to differing levels of effort
- Each person submits 3-5 projects (depending on weights)
- Maximum credit: 150 Tripos marks

## plan for Assessment

- you must communicate the results of each project in a written report
- must be typeset electronically (eg Microsoft Word, LibreOffice, LaTeX, ...)
- marks:
  - 40% for computational work,
  - 50% for mathematics,
  - 10% for quality of write-up & remarks

Plagiarism will not be tolerated, people who cheat risk getting zero marks for the whole CATAM course

## Resources

- CATAM webpage (with links to other content)  
[www.maths.cam.ac.uk/undergrad/catam](http://www.maths.cam.ac.uk/undergrad/catam)
- MATLAB booklet
- CATAM moodle page
- Part IB and Part II manuals (new one published each summer, late July-early August)
- These lectures — PDF files and MATLAB code on webpage
- CATAM News (on website) — updates/corrections/FAQ's
- CATAM helpline: [catam@maths.cam.ac.uk](mailto:catam@maths.cam.ac.uk)

## CATAM aims

- you will learn to use a computer to solve mathematical problems
- the main focus is on mathematics, but you will also learn and apply some new skills:
  1. **algorithms**: converting maths to "recipes" that a computer can follow, and characterising their efficiency
  2. **programming**: implementing algorithms using a computer language; testing and debugging
  3. **communication**: writing clear and precise reports

## Computers in mathematics

- applied
  - integration of ODEs & modelling dynamical systems
  - simulations of weather, climate change, fluid flow, materials, quarks, galaxies...
- applicable
  - statistical analyses of "big data"
  - modelling financial markets
- pure
  - investigating cases to develop hypotheses and conjectures
  - large primes, graph colouring, probability, ...

## Algorithms

**algorithm:** a set of rules for carrying out a mathematical (or other) operation ...

... set out in a *sufficiently precise* way that a computer can follow them

Like a mathematical proof, an algorithm needs to be *unambiguous*: the computer will execute the algorithm that you give, not the one that you "obviously" meant to write(!)

## Matrix multiplication

Consider 2  $n \times n$  matrices  $A$  and  $B$

The matrix elements of their product,  $C = AB$  are given by

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$$

For example, if

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad B = \begin{pmatrix} p & q \\ r & s \end{pmatrix}$$

Then

$$C = \begin{pmatrix} ap + br & aq + bs \\ cp + dr & cq + ds \end{pmatrix}$$

This is all true as mathematics but it is not an *algorithm*.

## Example: matrix multiplication

At its very simplest, an *algorithm* for computing  $C = AB$  might be

```
Loop over all the elements of the matrix C
(ie consider C(i,j) with i=1,2,..n and j=1,2,..n)

    Initially set C(i,j) to 0
    Loop over k, which takes values 1,2,..n
        Increment C(i,j) by A(i,k)*B(k,j);
    end loop over k
end loop over the elements of C
```

Note, we use indentation to show which instructions are "inside the loop" (i.e., which steps get repeated)

## Example: matrix multiplication

In Matlab we can compute  $C = AB$  by writing

```
for i = 1:n
    for j = 1:n
        C(i,j) = 0;
        for k = 1:n
            C(i,j) = C(i,j) + A(i,k)*B(k,j);
        end
    end
end
```

Let i "loop over" integer values 1 to n

"Initialize" C<sub>ij</sub> matrix element to 0

Summand

"=" is the assignment operator (this is not an equation!)  
i.e. the new C<sub>ij</sub> should take on the old value of C<sub>ij</sub>,  
plus this summand

Example: `simple_mult.m`

```

function C = mult(A,B)
%mult function : multiplies two matrices

% note: there is no output to the screen,
% we have ; at the end of all commands

[aRows,aCols] = size(A);
[bRows,bCols] = size(B);
C=zeros(aRows,bCols);

if ( aCols ~= bRows )
    error('matrix sizes not consistent with multiplication')
end

% we consider the i,j element of the new matrix C
% this requires two loops, one over i and the other over j
for i=1:aRows
    for j=1:bCols
        % for each element of C we have to do a sum with n terms
        for k=1:aCols
            C(i,j) = C(i,j) + A(i,k) * B(k,j);
        end
    end
end
end

```

end this box is similar to the algorithm on the last slide,  
but now it has been embedded into a MATLAB function

Example: `mult.m` (on website)

## MATLAB functions

In programming, functions are useful because...

They allow complex tasks to be accomplished with just one line of code

... makes programs more readable

... avoids rewriting similar algorithms several times (fewer chances to make mistakes)

They can be tested thoroughly at the time of writing and then reused many times


... helps with debugging

Example: `mult_test.m` (on website)

## Complexity

Suppose we have many matrix multiplications to do...  
how long does it take to multiply a pair of large matrices?

The *complexity of an algorithm* characterises the number of operations required:

- Initial assignment of each  $C(i,j)$  to zero  $n^2$
  - Number of multiplications  $n^3$
  - Number of additions  $n^3$
  - Number of further assignments  $+ n^3$
- $$3n^3 + n^2 = O(n^3)$$
- Complexity 

## Complexity

The *complexity* of the algorithm is  $O(n^3)$  in this case.

(the numerical prefactors are irrelevant for the complexity)

Simple matrix multiplication requires  $O(n^3)$  operations. So if we can multiply two  $3 \times 3$  matrices in  $\sim 10^{-9}$  seconds then multiplying two  $3000 \times 3000$  matrices will take  $\sim 1$  second.

... real-life computer programs (eg for solving partial differential equations) often need to multiply large matrices and to do this thousands of times...

... this can be slow, it's important to use efficient methods...

## Triangular matrices

If our matrices are upper triangular, we can save some operations...

$$\text{For } A = \begin{pmatrix} a & b \\ 0 & d \end{pmatrix} \quad B = \begin{pmatrix} p & q \\ 0 & s \end{pmatrix} \quad \text{and} \quad C = AB = \begin{pmatrix} ap & aq + bs \\ 0 & ds \end{pmatrix}$$

```
C = zeros(n,n)
for i = 1:n
    for j = i:n
        C(i,j) = 0;
        for k = i:j
            C(i,j) = C(i,j) + A(i,k)*B(k,j);
        end
    end
end
```

## Triangular matrices

$$\# \text{ of multiplications} = \sum_{i=1}^n \sum_{j=i}^n (j - i + 1) \approx \frac{n^3}{6}$$

```
for i = 1:n
    for j = i:n
        C(i,j) = 0;
        for k = i:j
            C(i,j) = C(i,j) + ...
                A(i,k)*B(k,j);
        end
    end
end
```

## Triangular matrices

$$\begin{aligned} \# \text{ of multiplications} &= \sum_{i=1}^n \sum_{j=i}^n (j - i + 1) = \sum_{i=1}^n \sum_{\ell=1}^{n-i+1} \ell \\ &= \sum_{i=1}^n \frac{1}{2}(n - i + 1)(n - i + 2) \\ &\approx \frac{1}{2} \sum_{i=1}^n (n - i)^2 = \frac{1}{2} \sum_{k=1}^n (k - 1)^2 \\ &\approx \frac{n^3}{6} \end{aligned}$$

```
for i = 1:n
    for j = i:n
        C(i,j) = 0;
        for k = i:j
            C(i,j) = C(i,j) + ...
                A(i,k)*B(k,j);
        end
    end
end
```

(last step using a Faulhaber's formula  
approximate equalities are valid at leading  
order in  $n$ )

## Complexity

Simple matrix multiplication algorithm:

	General $n \times n$	Triangular $n \times n$
• Initial assignment of each $C(i,j)$ to zero	$n^2$	$n^2$
• Number of multiplications	$n^3$	$\frac{1}{6}n^3$
• Number of additions	$n^3$	$\frac{1}{6}n^3$
• Number of further assignments	$+ n^3$	$+ \frac{1}{6}n^3$
	$3n^3 + n^2 = O(n^3)$	$\frac{1}{2}n^3 + n^2 = O(n^3)$

Fewer operations, but same complexity...  $O(n^3)$

The actual time taken to multiply matrices will depend on computational details (eg multiplying by zero should be a fast operation...)

... the *complexity* is a property of the algorithm, it can be *analysed*

## Strassen's algorithm

[ see eg wikipedia ]

You might think that all algorithms cost  $O(n^3)$ ...

For  $2 \times 2$  matrix multiplication  $C = AB$

$$\begin{array}{ll} \text{Let} & \begin{array}{l} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}), \\ M_2 = (A_{21} + A_{22})B_{11}, \\ M_3 = A_{11}(B_{12} - B_{22}), \\ M_4 = A_{22}(B_{21} - B_{11}), \\ M_5 = (A_{11} + A_{12})B_{22}, \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{21}), \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}). \end{array} \\ & \text{then} \quad \begin{array}{l} C_{11} = M_1 + M_4 - M_5 + M_7, \\ C_{12} = M_3 + M_5, \\ C_{21} = M_2 + M_4, \\ C_{22} = M_1 - M_2 + M_3 + M_6. \end{array} \end{array}$$

For  $2^n \times 2^n$  matrices, one can write an algorithm that involves operations on  $2 \times 2$  blocks, and use the method above for the blocks

For  $n \times n$  matrices, add zeros to make a  $2^n \times 2^n$  matrix

The complexity can be shown to be  $O(n^{\log_2 7}) \approx O(n^{2.8})$

## Scaling...

The Strassen algorithm has lower complexity (although it is certainly more *complicated*....)

Question: Is  $O(n^{2.8})$  really better than  $O(n^3)$ ?

Answer 1: As  $n \rightarrow \infty$ , yes it is!

Answer 2:

Suppose that the simple method costs  $6n^3$  and the Strassen method costs  $An^{2.8}$  for some  $A > 6$ ...

Then Strassen is faster for matrices of size  $n > (A/6)^5$ . If  $A$  is (eg) 60 then the simple method is still faster for matrices of sizes up to  $10^5 \times 10^5$ .

**General point...** small improvements in complexity are (often) useful only if  $n$  is very large

## Summary -- Complexity

- The complexity measures how the cost of an algorithm scales, when a parameter becomes very large (or small)
- Example parameters: size of a matrix, resolution of an image, accuracy required for the numerical solution to an equation
- Complexity only measures the scaling, the actual cost depends on details of the computation. For some fixed value of the parameter, the best scaling may not be the least cost...

## Next...

... next lecture will be on algorithms for solving equations (root finding)

... directly relevant for the introductory project

<http://www.maths.cam.ac.uk/undergrad/catam/part-ia-lectures>