

# Hard Sums and APIs

SS12 : Research Programming in the Scientific Computing Group

Chris Goodyer

Scientific Computation Group

# Outline

- 1 Scientific Computing
- 2 Code Structure
- 3 Meshes
- 4 The MPI API
- 5 Visualisation

Development of fast, efficient algorithms to solve mathematical problems arising from ordinary and partial differential equations

Development of fast, efficient algorithms to solve mathematical problems arising from ordinary and partial differential equations

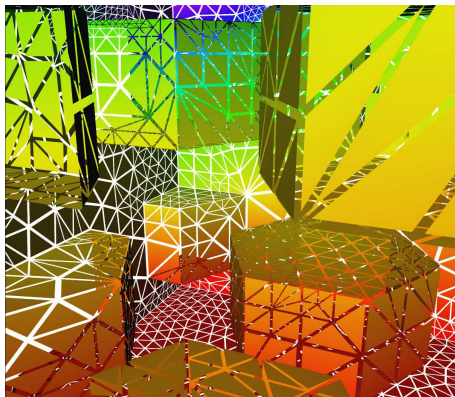
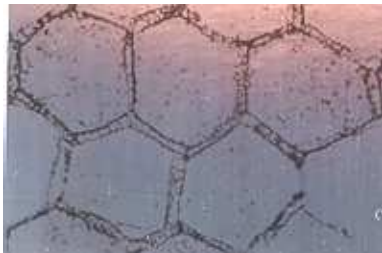
- Fundamental research
  - Numerical solution algorithms
  - Parallel Computing

Development of fast, efficient algorithms to solve mathematical problems arising from ordinary and partial differential equations

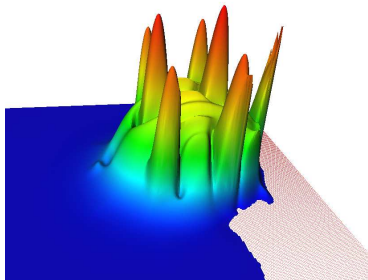
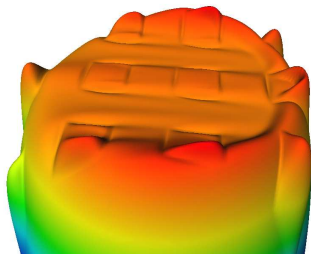
- Fundamental research
  - Numerical solution algorithms
  - Parallel Computing
- Application to real world problems
  - Solving demanding problems from other fields
  - Applying state-of-the-art algorithms to new problems

# Scientific Computing - Examples

- Chemical Diffusion through Skin



- Lubrication



- Biological Morphogenesis – A-life

Polyp

Medusa

Hydra

Two different scenarios – two different coding styles

Two different scenarios – two different coding styles

- Simple cases
  - Knock up simple codes to test hypotheses
  - Want something that is easy to write
  - Easy Visualisations?

Two different scenarios – two different coding styles

- Simple cases
  - Knock up simple codes to test hypotheses
  - Want something that is easy to write
  - Easy Visualisations?
- Specialised systems
  - Efficiency
  - Big problems

## Matlab

- Easy to write
- Lots of in-built functions

## Matlab

- Easy to write
- Lots of in-built functions

## C

- Fast
- Flexible

# Language choice

## Matlab

- Easy to write
- Lots of in-built functions

## C

- Fast
- Flexible

## Fortran

- Fast
- Lots of legacy code

Every real world problem is governed by three things:

- Mathematical equations
- What's it like at the start?
- What geometry are you interested in?

## Inputs

- Parameters
- Meshes

## Inputs

- Parameters
- Meshes

## What's coded

- Mathematical equations
- Solution methods

## Inputs

- Parameters
- Meshes

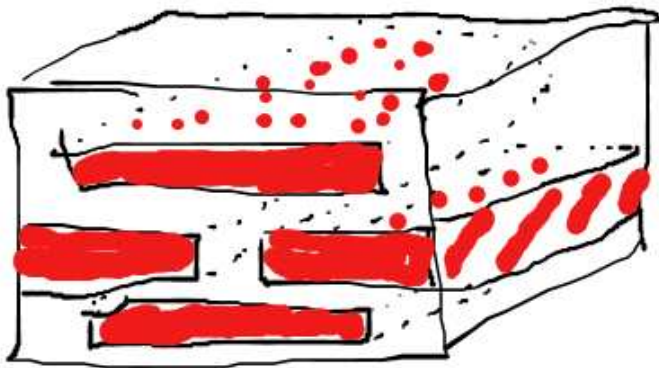
## What's coded

- Mathematical equations
- Solution methods

## Outputs

- Datasets - saved to formatted files
  - Meshes
  - Solution values

# Geometry to mesh



# Geometry to mesh

algebraic3d

```
solid bp1 = plane( 0, 0, 0 ; -1, 0, 0);  
solid bp2 = plane( 0, 0, 0 ; 0, -1, 0);  
solid bp3 = plane( 0, 0, 0 ; 0, 0, -1);  
solid bp4 = plane(1.05,0.70,0.45 ; 1, 0, 0);  
solid bp5 = plane(1.05,0.70,0.45 ; 0, 1, 0);  
solid bp6 = plane(1.05,0.70,0.45 ; 0, 0, 1);  
solid brick = bp1 and bp2 and bp3 and bp4 and bp5 and bp6;
```

```
solid corneo1a = orthobrick(2.50e-02, -5.00e-02, 2.50e-02; 1.025e+00, 12.50e-01, 1.25e-01);
```

```
solid corneo1b = orthobrick(1.075e+00, -5.00e-02, 2.50e-02; 2.075e+00, 12.50e-01, 1.25e-01);
```

```
solid corneo1 = corneo1a or corneo1b;
```

```
solid corneo2a = orthobrick(-5.00e-01, -5.00e-02, 1.75e-01; 5.00e-01, 12.50e-01, 2.75e-01);
```

```
solid corneo2b = orthobrick(5.50e-01, -5.00e-02, 1.75e-01; 1.55e+00, 12.50e-01, 2.75e-01);
```

```
solid corneo2 = corneo2a or corneo2b;
```

```
solid corneo3a = orthobrick(2.50e-02, -5.00e-02, 3.25e-01; 1.025e+00, 12.50e-01, 4.25e-01);
```

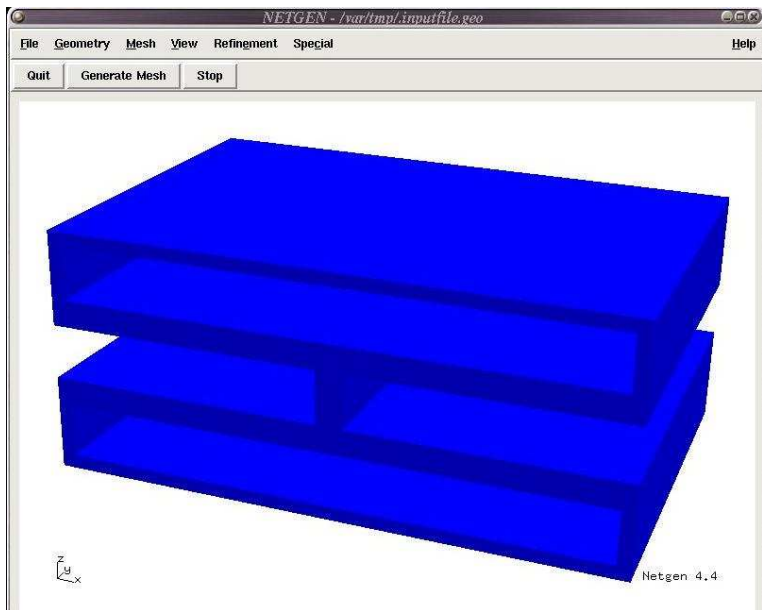
```
solid corneo3b = orthobrick(1.075e+00, -5.00e-02, 3.25e-01; 2.075e+00, 12.50e-01, 4.25e-01);
```

```
solid corneo3 = corneo3a or corneo3b;
```

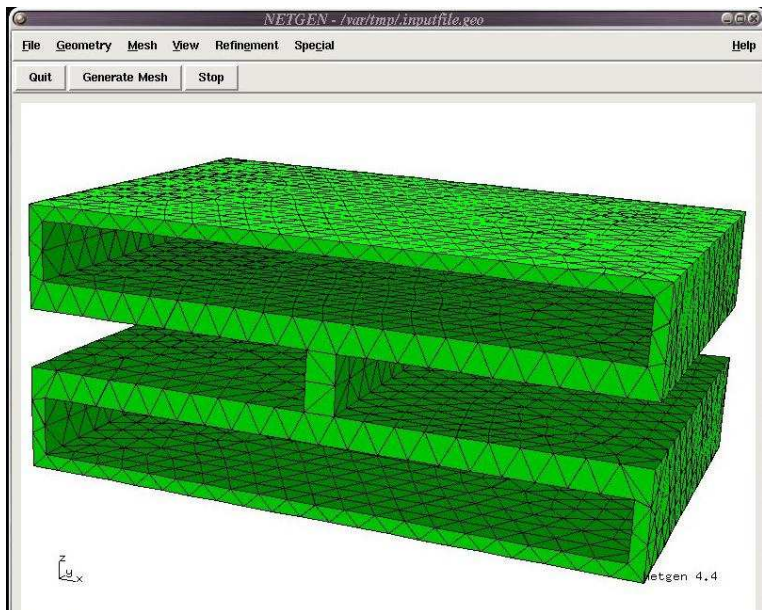
```
solid membrane = brick and not corneo1 and not corneo2 and not corneo3 -maxh=0.1;
```

```
tlo membrane;
```

# Geometry to mesh



# Geometry to mesh



# The generated mesh

## Nodes

```
12
0.00 0.00 0.00
0.00 0.00 1.00
2.00 0.00 0.00
2.00 0.00 1.00
0.00 1.00 0.00
0.00 1.00 1.00
2.00 1.00 0.00
2.00 1.00 1.00
1.06 0.00 1.00
1.06 1.00 1.00
1.00 1.00 0.00
1.00 0.00 0.00
```

## Tetrahedra

```
12
10 3 7 2
5 11 0 1
3 10 9 8
5 11 10 0
5 8 9 10
5 11 1 8
10 5 0 4
3 11 10 8
10 7 6 2
5 11 8 10
7 3 10 9
3 10 11 2
```

How to turn the equations into something the computer understands:

- See modules in your second and third year
- Continuous equations turned to matrices
- Solve matrix system at every timestep
- Improve mesh

# Reinventing the wheel?

- Lots of the ideas we want to use have been done before
- People release research codes to the community
- Huge development savings are made by not doing it yourself
- Learning curve

Typically these codes and libraries have an API – and hopefully good documentation

# MPI reminder

HelloWorld from multiple processors: [hello\\_mpi.c](#)

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

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

```
    int nprocs=1, pid=0;
```

```
    if (pid == 0)
```

```
        printf("Hello from proc %i of %i\n", pid, nprocs);
```

```
    else
```

```
        printf("Hello from proc %i\n", pid);
```

```
    return 0;
```

```
}
```

# MPI reminder

HelloWorld from multiple processors: [hello\\_mpi.c](#)

```
#include <stdio.h>
#include "mpi.h" ← the MPI library
int main(int argc, char** argv)
{
    int noprocs=1, pid=0;
    MPI_Init(&argc, &argv); ← start MPI
    MPI_Comm_size(MPI_COMM_WORLD, &noprocs);
        ↑ discover the number of processors
    MPI_Comm_rank(MPI_COMM_WORLD, &pid); ← get pid
    if (pid == 0)
        printf("Hello from proc %i of %i\n", pid, noprocs);
    else
        printf("Hello from proc %i\n", pid);
    MPI_Finalize(); ← end MPI
    return 0;
}
```

# MPI\_Send specification

## MPI\_Send

Performs a basic send

## Synopsis

```
#include "mpi.h"
```

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest,  
              int tag, MPI_Comm comm )
```

## Input Parameters

buf	initial address of send buffer (choice)
count	number of elements in send buffer (nonnegative integer)
datatype	datatype of each send buffer element (handle)
dest	rank of destination (integer)
tag	message tag (integer)
comm	communicator (handle)

# Passing Data Between Two Processors

p0 - sending

```
MPI_Send(&outgoing, 1, MPI_INT, pid+1, 301, MPI_COMM_WORLD);
```



p1 - receiving

```
MPI_Recv(&incoming, 1, MPI_INT, pid-1, 301, MPI_COMM_WORLD,  
        &status);
```

# Sharing Data Between All Processors

## Broadcasting to all

```
MPI_Bcast(solarray, length[p], MPI_DOUBLE, p,  
          MPI_COMM_WORLD);
```

# Sharing Data Between All Processors

## Broadcasting to all

```
MPI_Bcast(solarray, length[p], MPI_DOUBLE, p,  
          MPI_COMM_WORLD);
```

## Collate and combine results

```
MPI_Allreduce( in, out, count, MPI_INT, MPI_SUM,  
              MPI_COMM_WORLD );
```

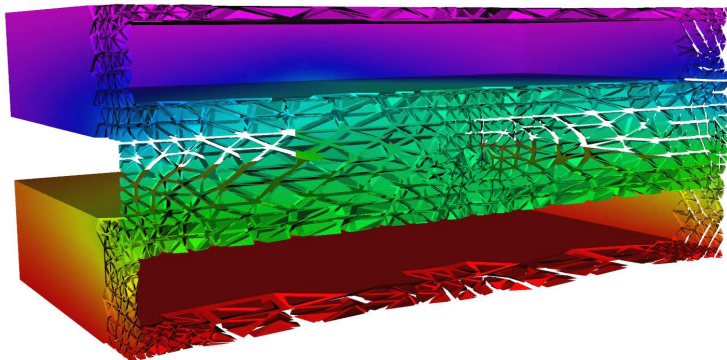






## Mesh and solution values

- Mesh stored similarly to inputs
- Solution values may be on nodes, edges, triangles or tetrahedra



# Summary

- Scientific Computing turns equations into pictures to give answers
- We want quick-to-write or quick-to-run codes
- We're not afraid to use existing libraries and tools
- We use both workstations and big parallel machines

# Summary

- Scientific Computing turns equations into pictures to give answers
- We want quick-to-write or quick-to-run codes
- We're not afraid to use existing libraries and tools
- We use both workstations and big parallel machines

## Morals

- Always plan what you're going to do first
- Always document your code
- Always test codes