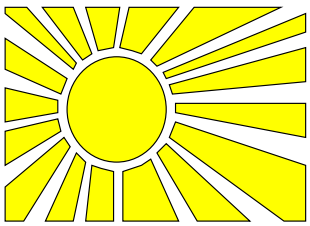
EPCC-SS2001-08

Modelling of fracture growth in rocks

Helen May Gibson

Abstract

The evolution of fractures in rocks is an area of active research in the department of Geology and Geophysics at the University of Edinburgh. The patterns of crack propagation are complex and of great interest, especially as cracks provide pathways for fluid flow, and so predicting where cracks are is useful in areas such as determining water quality, assessing the long-term safety of nuclear waste disposal, and optimising oil production from fractured reservoirs. This project seeks to advance the understanding of crack propagation by simulating cracks in rocks computationally using a parallel code developed by Dr Javier Sabadell [1] that calculates the displacements of a rock under an applied force, and extending the code by adding in cracks as failed elements, calculating the stresses acting on them, and eventually, showing how the cracks propagate.



Contents

1 Introduction

This project seeks to simulate the stresses on, and growth of, cracks in elastic rocks. The network of cracks can allow water or contaminants to migrate through an otherwise impervious rock. Thus being able to predict the growth and arrangement of cracks can be very useful in real world applications.

If there is a crack in a rock and a force is applied, the crack will grow if the stress at its tip exceeds the strength of the rock at that point. If there are several cracks, their positioning relative to each other affects the extent to which they feel the stresses. Cracks parallel to each other are shielded from the stresses and grow relatively slowly. On the other hand, if two cracks are along the same line, with their tips facing each other, they will feel the stresses more, and grow faster. As cracks grow towards each other, they may eventually join.

If you have a large space with a heterogeneous strength, and lots of failed elements, the calculations become very complicated and it is necessary to use numerical methods.

A serial code to model the stresses round a system of cracks was written by Dr Javier Sabadell. His code applies boundary conditions, external and internal forces, and generates cracks and displays stresses.

However, for the code to be able to model bigger, more complex systems, it should work faster. Also, if the code was to simulate cracks growing, each time the crack was extended, the system would have to be re-calculated. This, again, would require fast computation. Thus a parallel approach was necessary.

Dr Sabadell started work on a parallel code called `elasem`. It was parallelised using a message passing interface (MPI) [2]. It used a partitioning algorithm, *METIS* [4], to divide up the work between the processors. It also used the *Aztec* parallel library [3], an iterative solver package that simplifies the parallelisation process when dealing with linear systems of equations. This allowed `elasem` to model the elasticity of a rock by calculating the displacements of the nodes under an applied force.

This project started from Dr Sabadell's parallel code. The project extended the code to allow fractures to be generated in the rock being modelled, and to calculate the stresses at the tips. Future work will provide a mechanism for the cracks to propagate.

2 Overview of `elasem`

Dr Sabadell took a finite element approach to the problem. The rock is modelled by a grid which is divided up into elements. Each element consists of four grid-points to make up a square. Then calculations involving displacements and gradients can be made using the points round the edge of the element.

Forces can be made to act on nodes in the grid. The resulting displacements are calculated using the linear algebra parallel library, *Aztec*. These displacements can be displayed graphically. Examples of this can be seen in figures 2 to 4 in the results section.

In order to introduce cracks, the displacements were set to zero at the nodes where the crack would be. This introduces a barrier in the grid so the stresses are transferred to the end of the barrier, similar to the way the stresses are transferred to the tip of a physical crack.

The stress matrix and its invariants can be calculated for a given grid-point. The eventual idea is to use the stress invariants calculated at the tips of the crack and the strength of the material at that point to determine whether the node at the tip of the crack would break. If so, the stress invariants at the nodes surrounding the broken node should be calculated. The node with the largest value obtained should be tested next for breaking. If its stress invariant exceeds the strength of the material at that point, its neighbours should be examined and so on, until the crack comes across a material strength greater than the stress at its tip. The crack should be extended to include all the elements that were calculated to fail. The program could then continue to run, increasing the force acting on the crack linearly with time, until cracks join up.

Observing the evolution of the cracks, and the final network of cracks formed, will give an insight into the patterns of crack growth. Further runs with differing initial cracks will allow more clues to be drawn, bringing closer the goal of being able to predict crack growth and arrangements.

3 Operation of `elasem`

`elasem` relies on several input files in addition to the libraries mentioned in the introduction. They are described in detail in the appendices, but a brief overview of these is given below:

`options.d`:

selects the solver used by the *Aztec* library, as well as scaling, preconditioning, number of iterations, and various other parameters that have to be set for the library to work.

`parameters.d`:

specifies the parameters for the *Aztec* library, such as the tolerance for the convergence test.

`else2_stat.d`:

states the name of the data file and the degree of the spectral element.

data file:

the data file named in `else2_stat.d` contains all the details of the grid:

- the co-ordinates of the grid-points,
- the grid-points in each element,
- the grid-points with Dirichlet boundary conditions (fixed displacements), and Neumann boundary conditions (fixed velocity). (There is also the possibility of including absorbing boundary conditions, but the code does not deal with these, and an error is given)
- the grid-points on which horizontal or vertical forces act,
- and the Young's Modulus, density and Poisson's ratio of the material in each element.

The degree of the spectral element is a number, stored in a variable called `ns`, that indicates how fine the grid should be. When `ns` is set to 1, the grid is as fine as indicated in the input file. If

ns is greater than 1, this indicates that the grid will be refined by the program at run time, and the number of nodes increased. The finer the grid, the more detail can be seen.

The first thing the parallel code does is read in all the grid data from the data file, and store the values in arrays. The Lamé Parameters, λ and μ are calculated from the Young's Modulus and Poisson's ratio of the material, and are used later in the calculations of the stresses. The grid is then divided up between the processors using *METIS*. Figure 1 shows how *METIS* divides up the square grid between processors. You may need colour to view this.

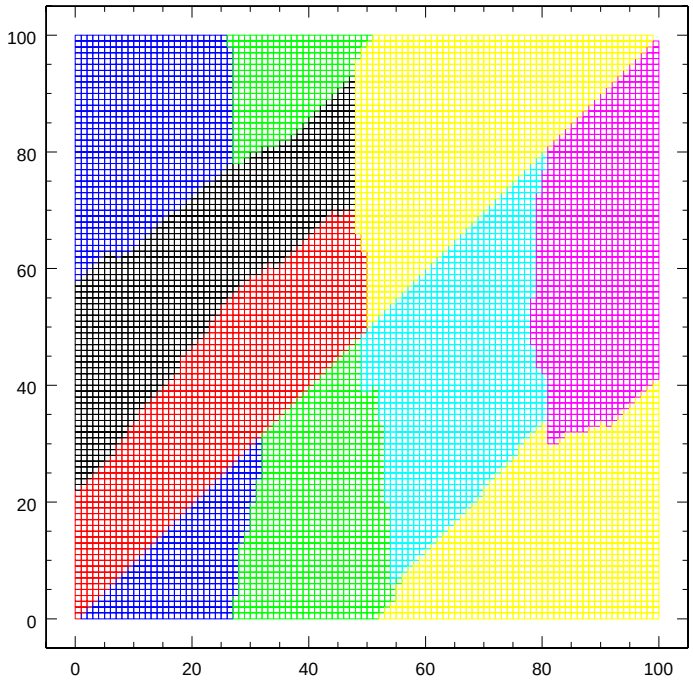


Figure 1: Square grid partitioned by *METIS*.

If ns, the degree of spectral element, is greater than one, the grid is divided up further. The stiffness matrix is then calculated. It contains directional information of the Young's Modulus at each point. Then the tensor form of Hooke's law – a system of equations of the form Ax = b where A is the stiffness matrix, b is the stress tensor and x is the strain tensor – is solved to get the displacements. During this, each processor is only responsible for the grid-points to which it was assigned. The displacements in the x and y direction for each grid-point are stored in an array, each processor only holding information for its grid-points. The array is then collated onto the first processor and printed out in a form that can be read by a commercial plotting package called *tecplot* [5]. This, however, can be converted to a format compatible with a public plotting package called *plotmtv* [6] using a program written by Dr Sabadell.

There were a variety of machines available to run the code at the Edinburgh Parallel Computing Centre (EPCC). However, there was a problem trying to run the code on the Sun systems. The *Aztec* library is written in C but has a Fortran interface. MPI is initialised outside *Aztec* and the MP2 communicator used is passed down to the library by one of the routines. Unfortunately the *Aztec* library assumes that the same representation is used for a C and a Fortran communicator.

This is not true for the Sun version of MPI hence this action resulted in a bus error and a core dump. Although it was possible to hack the *Aztec* library to work with the Sun version of MPI this was not done as it proved simpler to work on a linux Beowulf cluster using the MPICH implementation [7]. In fact the code had earlier been run on linux machines.

4 Method

This project worked on the parallel code described in §2 and §3. The first stage was to verify the results that Dr Sabadell obtained using his serial code. The next stage would involve writing code to allow the cracks to propagate and evolve.

Having deciphered the input data file, a program was written that would produce a data file containing the details of a grid similar to the ones that Dr Sabadell used for his serial code. His grid was square, with Dirichlet boundary conditions applied to the corners of the grid. `elasem` was slow to converge to a solution due to the strength of the test material, but applying Dirichlet Boundary conditions to all the edges was found to help.

In order to reproduce the tensile force that Dr Sabadell applied, it is necessary to have opposing forces acting along a pair of opposite edges, but the parallel code at present does not allow for this. Not enough time was available to extend the use of forces in the code. At the moment, the data file defines forces that act vertically on the top edge. The code to generate the data file can be seen in appendix C.

The next stage was to introduce cracks. A subroutine called `make_cracks` was written that could read in the start and end co-ordinates of a crack from an input file called `cracks.d`. The subroutine would then identify all the points in between these two co-ordinates. It relies on the grid being square and of known length (number of grid-points on one side, as input in the data file). Some pseudo code outlining the behaviour of this subroutine is shown in appendix B.

Once all the grid-points that define a crack are identified, they are stored in an array called `crackedNodes`. Dirichlet boundary conditions are applied to each of these nodes, thus setting the displacements in the nodes to zero. The Dirichlet boundary conditions for the cracks were applied in the same way as for those edges defined to have Dirichlet boundary conditions in the input file. Instead of setting the the node's component of the stiffness matrix, `effe`, to the value of the analytical solution for its co-ordinates, the component is set to zero, thus forcing the displacements to be zero. This causes the barrier in the grid necessary to simulate the crack.

`elasem` displays the displacements of the grid-points, whereas Dr Sabadell's original serial code displayed the stresses. The next stage in the project was thus to allow `elasem` to display the stresses.

In order to do this, the stress matrix had to be calculated for each grid-point from the stiffness matrix and values of λ and μ. As the problem is two dimensional, the matrices are two by two. The invariants of each stress matrix were calculated by finding the determinant and the diagonal sum of the matrix. As their name suggests, these quantities are invariant under rotation. The diagonal sum gives the volumetric component of the strain, and the determinant gives the stress that would be used in any failure criterion. Thus, it is the latter that is most useful to display.

Dr Sabadell wrote an additional subroutine, called `make_stress`, to calculate the stress matrices using the finite element method. However, there were complications. When these are re-

solved, the stress invariants will be passed to the subroutine `scrivi_tecplot_file`, rather than the displacements, so that the stresses can be plotted.

In the meantime, another subroutine called `grid_pt_stress` was written that could calculate the stress invariants at a given grid-point. This uses the finite difference method, and, like the `make_crack` subroutine, relies on the grid being square and of a known length. If the given grid-point is at the tip of a crack, the stress invariant can be compared with the strength of the material at that point to see if the crack should be extended.

However, whilst testing this subroutine, it was noticed that the stress invariants had different values depending on the number of processors used in the calculation. Further study showed that this also occurred in the original `elasem` with the original input file. Further work needs to be done to identify the source of this.

In spite of this, the displacement diagrams produced using 16 processors are what would be expected from the cracks generated, as can be seen in §5.

5 Results

Arrangements of cracks were generated that would reproduce some of the results that Dr Sabadell obtained. He plotted stresses, and they clearly showed that isolated cracks felt more stresses while adjacent parallel cracks shielded each other.

`elasem` is, at the moment, only capable of displaying the displacements, but these plots still give a good indication of the stresses felt.

The displacements were graphed using *Plotmtv*. The output is a picture of the grid, with each grid-point coloured according to the displacement it felt along one of the cartesian directions. There are two plots of each run, one coloured to show the x-components of the displacements, and the other to show the y-components. The labels on the axes, depth and lateral dimension, indicate the distance from the bottom left of the grid in the y and x directions respectively. The colour scale on the side of each graph indicates the value of the displacement associated with each colour. Negative displacements in the x-component plot indicate displacements to the left. Similarly, negative displacements in the y-component plot indicate displacements down the grid.

The diagrams below were created from the following initial conditions: Dirichlet boundary conditions round all the edges with vertical forces along the top edge, pulling it up. The program was run on 16 processors. The cracks show clearly as the blue sections in the y-component image – blue representing zero displacement. In the x-component image you can see that the lower part of the grid is pulled in towards the crack, and the upper part is pulled away, much like how the material of a jumper moves when it is stretched.

Looking at figure 4 it can be seen that the displacements in the x direction corresponding to the lower crack are greater than the displacements in the same direction corresponding to the upper two cracks. This is because the lower crack is further from the upper pair and so the effects of the force are stronger on it than on the upper pair of cracks. This is due to the pair shielding each other from the force, while the independent crack is more exposed.

Although these plots show the displacements under an applied force, they still demonstrate the shielding effect that Dr Sabadell's stress plots showed. The results therefore correspond with Dr Sabadell's results.

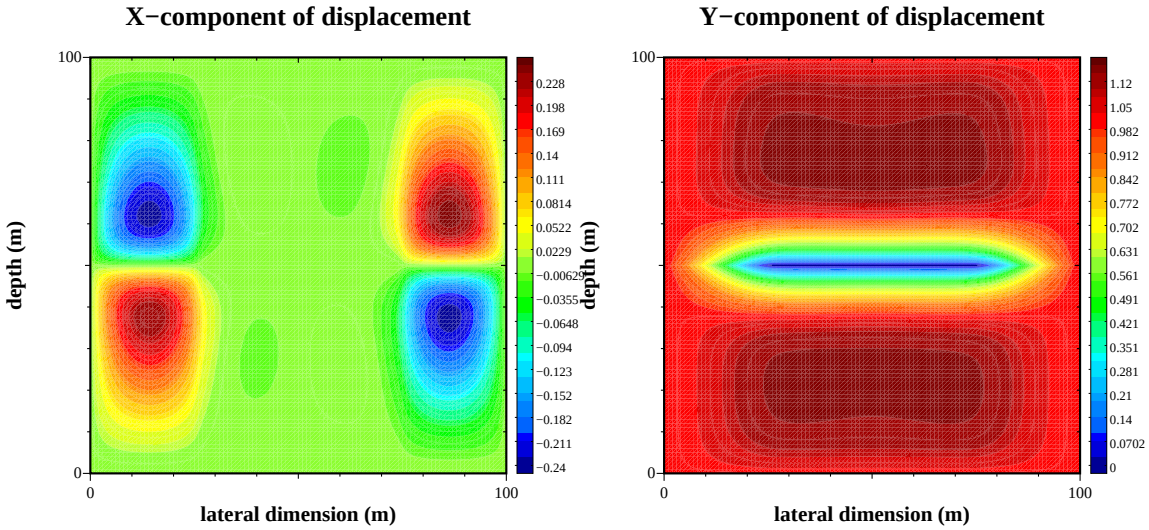


Figure 2: One crack, clearly visible in the y-component of displacement plot. In the x-component of displacement plot, four major regions of displacement are shown. The lower two indicate a pull towards the crack, and the upper two indicate a pull away from the crack.

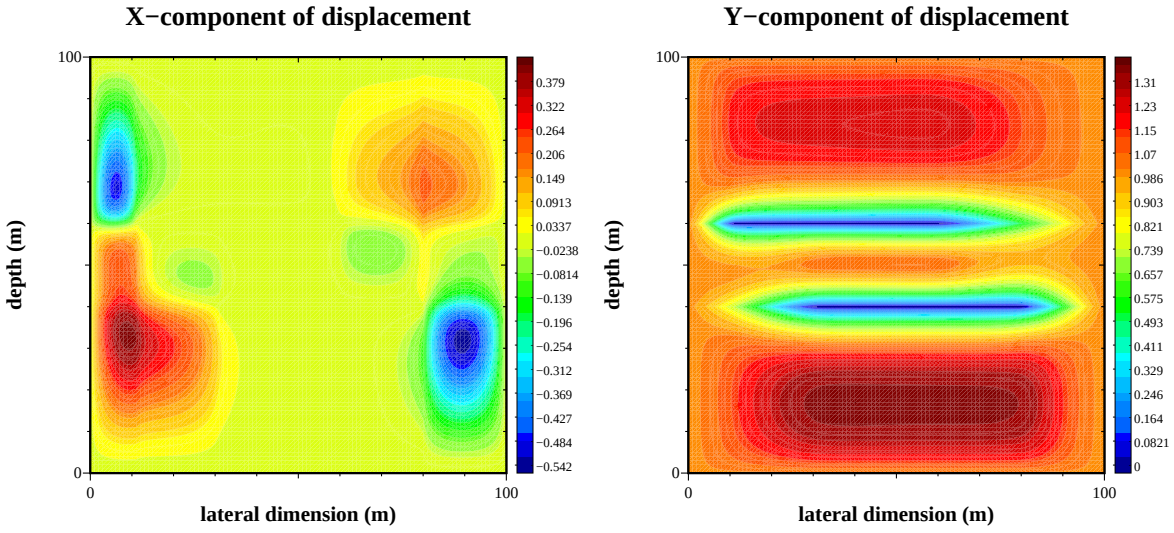


Figure 3: Two cracks, clearly visible in the y-component of displacement plot. In the x-component of displacement plot, the displacements are stronger at the more exposed tips, and very weak around the top right tip, as this tip is shielded from the stresses by the other crack.

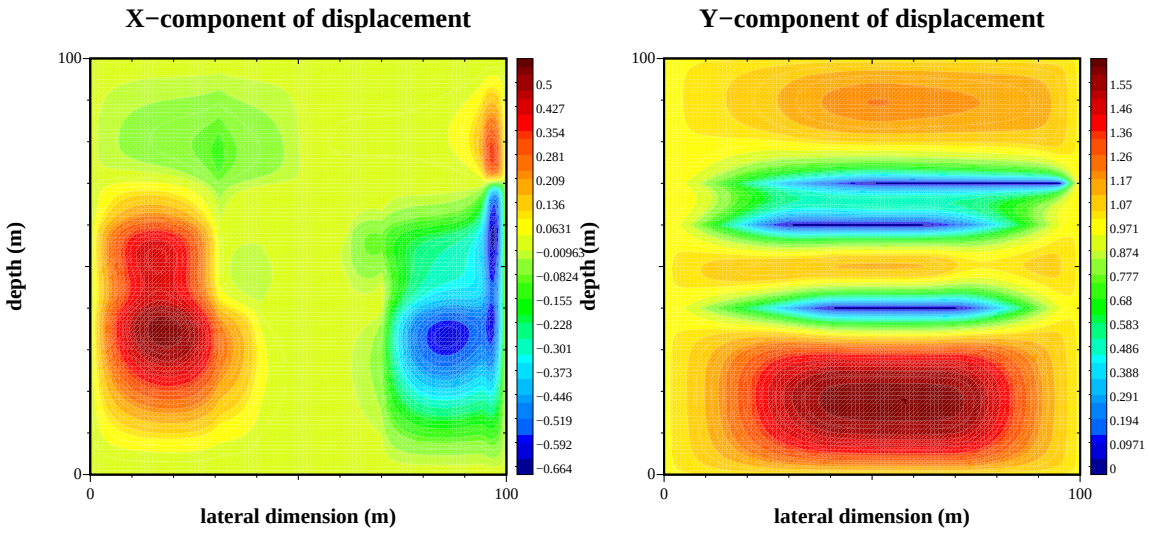


Figure 4: Three cracks, two close together, as you can see in the y-component of displacement plot, and a third further away. In the x-component of displacement plot, the displacements are very strong around the third, further away crack, and very weak around the top two, as they are shielding each other from the stresses.

6 Future Developments

There are several things that remain to be done in order to model cracks propagating in rocks more realistically. To begin with, the results on sixteen processors need to be verified, and the reason for the dependence of the results on the number of processors will need to be dealt with.

Once the bugs in the `make_stress` subroutine have been fixed, the code will be able to display the stress invariants.

The next stage would be to introduce a heterogeneous strength field to the grid, representing a polycrystalline ceramic material. The strength field must be heterogeneous, otherwise a crack, whose stress invariant at its tip exceeded the strength of the material, would continue to grow indefinitely until it reached the boundary. This would be appropriate for a homogeneous brittle material such as a perfect crystal, but is not appropriate for a polycrystalline ceramic material such as a rock. If the strength field was heterogeneous, the crack would only grow until it came across a node with a strength greater than the stress invariant at its tip.

A random strength field could be generated at run time using, for example, a Gaussian distribution to give each node a strength about a mean.

Another modification to the code would be to improve the force algorithms. As the code stands, it is only possible to specify a horizontal or vertical force of standard size. It would be an improvement to be able to vary the direction and strength. This would allow opposing forces and forces that increase with time.

Also a simple algorithm could be coded which could generate random cracks, and output their start and end co-ordinates into `cracks.d`. And then, as before, `cracks.d` would be read into `make_cracks`.

7 Conclusion

The parallel code is part way to being useful for geologists and geophysicists in their study of crack propagation. It can display the displacements round a user-defined crack in a user-defined grid, and it can calculate the stress invariants at a grid-point. Using 16 processors, the results seem reliable and illustrate the stress-shielding properties of a crack array.

This project has paved the way for future work. The documentation will allow others to begin at the point where we left off. Once the tasks listed in the above section have been completed, the code will be a valuable tool for studying crack propagation.

A Input and Output Files

A.1 Input Files

The original parallel program read in four files.

`options.d`:

The user can set the options of the *Aztec* library. The program works best using the second, default, solver. Scaling and preconditioning can be adjusted as required. Anything after the exclamation mark is not read, and states what each number represents. The contents of the square brackets is the default setting.

```
 2  ! [2] Solver: (CG->1,GMRES->2,CGS->3,TFQMR->4,BICGSTAB->5,LU->6)
 2  ! [1] Scaling: (None->1,Jacobi->2,BJacobi->3,row_sum->4,sym_diag->5,
                    sym_row_sum->6)
 5  ! [1] Precond.: (None->1,Jacobi->2,Neumann->3,ls->4,sym_GS->5,dom_dec->6)
 2  ! [2] Sub-domain solver: (lu->1,ilut->2,ilu->3,rilu->4,bilu->5,icc->6)
 1  ! [1] Convergence crit.: (r0->1,rhs->2,Anorm->3,noscaled->4,sol->5,
                              weighted->6)
 1  ! [1] Output: (all->-1,none->-2,warnings->-3,last->-4,k->k)
 0  ! [0] Use previous information from AZ_solve (no->0,recalc->1,reuse->2)
 0  ! [0] Level of graph fill-in
1000 ! [500] Maximum number of iterations
 3  ! [3] Polynomial order when using polynomial preconditioning
 0  ! [?] Submatrices factored with dom-dec algorithm
 1  ! [1] Overlap: (standard->1,symmetric->2)
30  ! [30] Krylov subspace size
 1  ! [1] Reordering (no->0,yes->1)
 1  ! [0] Keep matrix factorization (no->0,yes->1)
 1  ! [1] GMRES orthog. scheme (classical->1,modified Gram-Schmidt->2)
 1  ! [1] residual r~ (initial r->1,random->2)
```

`parameters.d`:

Similar to options.d, this file specifies the parameters for the *Aztec* library, such as the tolerance for the convergence test. At the moment all the parameters are set to their default value.

```
 1.d-06  ! [1.d-06] Tolerance for convergence test
 0.d0    ! [0.d0] Drop tolerance used with LU or ILUT preconditioners
 1.d0    ! [1.d0] Allowed increase of non-zero elements (LU factorization)
 1.d0    ! [1.d0] Damping or relaxation parameter used for RILU
  --     ! [ -- ] When options[AZ_conv] = AZ_weighted, the i'th local
                  component of the weight vector is stored in the
                  location params[AZ_weight+1]
```

`else2_input.d`:

The first line is the name of the data file. The second line is the value of ns. 1 indicates that the grid is divided up as the input file states, a number greater than 1 indicates that the grid will be made finer.

data file:

Named in else2_input.d, this file describes the sheet onto which cracks are placed. The first seven lines tell the code how many of each thing there are:

```
10201                    grid-points,
10000                    total elements,
    1                    materials,
  202                    nodal sources (body-forces),
  202                    elements' edges with Dirichlet b.c.,
    0                    elements' edges with Neumann b.c.,
    0                    elements' edges with absorbing b.c..
```

Then all the grid-points are numbered, and their x and y co-ordinates listed.

```
1  0.000000  0.000000  0.e0
2  0.000000  1.000000  0.e0
3  0.000000  2.000000  0.e0
4  0.000000  3.000000  0.e0
5  0.000000  4.000000  0.e0
6  0.000000  5.000000  0.e0
```

...etc

Then the elements are listed. The first column dictates the shape of the element (21 = rectangle), the second column indicates the material associated to that rectangle (1=first material). The following four columns indicate the four grid-points that make up the element, listed in clockwise order.

```
...
21   1       210       211       312       311
21   1       211       212       313       312
21   1       212       213       314       313
21   1       213       214       315       314
21   1       214       215       316       315
21   1       215       216       317       316
21   1       216       217       318       317
21   1       217       218       319       318
...
```

Then the boundary conditions are defined. Dirichlet ones fix the displacements. The first column, 9, specifies Dirichlet. The next two are the grid-points defining the edge with the boundary conditions

```
...
9      9797      9898
9      9898      9999
9      9999     10100
9     10100     10201
9         1       102
9       102       203
9       203       304
9       304       405
9...
```

The next lines describe the force. The first column is a label, the middle column is the grid-point where the force acts and the last column is the component of the force: 1 is horizontal, 2 is vertical.

```
...

  0      8586  1
  0      8687  1
  0      8788  1
  0      8889  1
  0      8990  1
  0      9091  1
  0      9192  1
...
```

The last line describes the material. The first number is the number of the material. The second number is the Young's Modulus of the material. The third is the Poisson's ratio and the fifth is the density.

```
matl 1 1.d0 1.d0 0.d0 1.d0
```

My additional subroutine requires another input file:

`cracks.d`:

The first line is the number of cracks. Then for each crack there is a line which contains four numbers, indicating the x and y co-ordinates of the start of the crack, and the x and y co-ordinates of the end of the crack, respectively.

A.2 Output Files

The main output files are `sol_an.d` and `else_stat.d`. They have to be passed through `./tec2mtv` before they can be viewed using *plotmtv*. `sol_an.d` only has meaningful data if the solution is analytical, i.e. if there are no forces acting on the grid.

Each processor outputs to a file called `fort.`num, where num is 10 + the number of the processor, i.e. processor zero outputs to `fort.10`, and processor 3 outputs to `fort.13`. The files contain the time taken to perform each part of the program.

Also, if there are more than 10 processors, another file is written: `dd_load.d`. This file states the percentage of the work that each processor had to do.

B Pseudo code of `make_cracks`

Before the call:

- read the number of cracks from `cracks.d`
- initialise array storing cracked nodes.
- call `make_cracks`

In `make_cracks`, for each crack:

- Read co-ordinates of start and end points
- Determine direction of crack in x and y direction; positive or negative
- Determine the ratio of steps in x direction to steps in y direction
  - If it is a horizontal crack, zero steps in y direction, length of crack steps in x direction.
  - else if it is a vertical crack, zero x steps, length of crack steps in y direction
  - else calculate gradient, use absolute value
    - if gradient less than 1, set x step to the integer of its inverse, y step set to 1
    - else set x step to 1, and the y step is the integer of gradient
- Find grid number of start co-ordinate using length of grid
- Store in next space in cracked nodes array
- Find number of steps needed
- maintain ratio of steps in x direction to steps in y direction
  - For steps in x direction add the length of the grid to (or subtract it from) the last grid-point
  - For steps in y direction add 1 to (or subtract 1 from) the last grid-point
- store each grid-point calculated in the cracked nodes array until the number of steps needed have been taken.

C Programs

The following is the code to generate the square grid that I specified. At the moment it specifies boundary conditions along all edges, but this can be modified to specify boundary conditions along the top and bottom edges only. It is written in C.

```
/* Author: Helen May Gibson
   Date  : August 2001

   The code outputs a data file of the correct format to be read by
   elasem

   It describes a grid 100 by 100 with Dirichlet boundary conditions
   along all edges, and vertical forces acting on the top edge.  The
   material is constant throughout, and is described in the last line
   of the file.

   The code for applying Dirichlet boundary conditions just to the
   four corners has been commented out.  Any changes made to number of
   edges with boundary conditions or forces must be reflected in the
```

```
   first seven lines of output.

*/

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

int main(void)
{
  double length_of_grid, distance;
  int  length, tot_num_gridpoints, tot_num_elements,
    count, i, j, a, b, c, d, row;

  FILE *out;

  out = fopen("gridIan100", "w");

  length_of_grid = 100;
  distance = 1.0; /* Distance between nodes */

  length = length_of_grid/distance + 1; /*number of gridpoints per side*/

  tot_num_gridpoints = length * length;

  tot_num_elements = (length-1)*(length-1);

  /* Outputs the first seven lines of the data file, stating the number of
     grid-points, elements, materials, grid-points with forces, and elements
     with each kind of boundary condition.

  */

  fprintf(out, " %d  grid-points,\n", tot_num_gridpoints);
  fprintf(out, " %d  total elements,\n", tot_num_elements);
  fprintf(out, "  1  materials,\n");
  fprintf(out, " %d  nodal sources (body-forces),\n", length);
  fprintf(out, " %d  elements' edges with Dirichlet b.c.,\n", 2*(length-1));
  fprintf(out, "  0  elements' edges with Neumann b.c.,\n");
  fprintf(out, "  0  elements' edges with absorbing b.c.,\n");

  count = 1;

  /* Lists grid-points and numbers them */

  for (i = 0; i < length; i++)
    for (j = 0; j < length; j++){
      fprintf(out, "%d  %lf  %lf  0.e0 \n", count, i*distance, j*distance);
```

```
    count++;
  }

if ((count-1) != tot_num_gridpoints)
  printf("\n count is not equal to the total number of grid-points!!!\n);

/* Elements described by clockwise rotation of grid-points

   21 indicates rectangle, 1 is the material number  */

count = 1;
i = 1;
row = 1;
while (i <= (tot_num_gridpoints - length - 1)){
  for (; i <= (row * length)-1; i++){
    a = i;
    b = i+1;
    c = i + length + 1;
    d = i + length;

    fprintf(out, "21   1   %d   %d   %d   %d\n", a, b, c, d);
    count++;
  }
  row++;
  i++;
}

if ((count-1) != tot_num_elements)
  printf("\n count is not equal to the total number of elements!!!\n");

/* Dirichlet b.c. */

for(i=1; i < length; i++)    /* Code for left edge */
  fprintf(out, "  9 %10d %10d\n", i, i+1);

for(i = 1; i < length; i++)  /* Code for top edge */
  fprintf(out, "  9 %10d %10d\n", i*length, (i+1)*length );

for(i = 0; i < length-1; i++) /* Code for bottom edge */
  fprintf(out, "  9 %10d %10d\n", (i*length)+1, (i+1)*length+1 );

for(i=1; i<length; i++)       /* Code for right edge */
     fprintf(out, "  9 %10d %10d\n",
             ((length-1)*length)+i, ((length-1)*length)+i+1 );



/*  Code to introduce 8 edges with Dirichlet b.c. which pin down the
```

```
      corners of the square; commented out

  fprintf(out, "   9         1         2\n");
  fprintf(out, "   9%10d%10d\n", length-1, length);
  fprintf(out, "   9%10d%10d\n", length, 2*length);

  fprintf(out, "   9%10d%10d\n", length*length-length, length*length);

  fprintf(out, "   9%10d%10d\n", length*length, length*length-1);

  fprintf(out, "   9%10d%10d\n",
          length*length-length+2, length*length-length+1);

  fprintf(out, "   9%10d%10d\n",
          length*length-length+1, length*length-2*length+1);

  fprintf(out, "   9%10d%10d\n", length+1, 1);

  */




  /* Forces along top */

   for(i=0; i < length; i++)
     fprintf(out, "  0%10d   2\n", i*length + length);

   /* Specifies the material.  Details in Documentation */

   fprintf(out, " matl  1  1.d0 1.d0 0.d0 1.d0\n");

  close(out);

  return 0;

}
```

References

[1] F. J. Sabadell. *1999 Tensile crack modelling: tests and preliminary results*, see http://www.glg.ed.ac.uk/ sabadell/report/.

[2] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker and Jack Dongarra. *MPI: The Complete reference.* The MIT Press. 1996.

[3] Sandia National Laboratories, see http://www.cs.sandia.gov/CRF/aztec1.html

[4] Family of Multilevel Partitioning Algorithms, see http://www-users.cs.umn.edu/ karypis/metis/.

[5] *Tecplot 9.0, Enjoy the View*, see http://www.amtec.com/Product_pages/tecplot9/index.html.

[6] Download from ftp://keel.mit.edu/pub/Plotmtv/.

[7] MPICH – A portable MPI Implementation, see http://www-unix.mcs.anl.gov/mpi/mpich/

Looking forward to my 4th year of an MPhys in Computational Physics, I am learning some useful parallel computing skills at the EPCC.

Supervisors: Dr Mario Antonioletti, Dr Ian Main, Mr Fahad Al-Kindy, Dr Javier Sabadell.