Exercise 1 (easy)

Toby Wiseman (Imperial College)

Mathematica Summer School, Porto 2014

AIM : The aim of this exercise is to solve a 2 - dimensional p.d.e. boundary value problem.

[This exercise is intended as a simple introduction to solving elliptic p.d.e.s. It will form the basis for the harder exercise 2 which will implement solving a gravitational problem.]

The p.d.e. is defined in terms of a function f on a 2 – dimensional rectanglular domain with coordinates x, y with $x \in (0, 1)$ and $y \in (0, 2\pi)$

We choose Dirichlet boundary conditions for f at x = 0. Defining a boundary function bfn[y], then f[x = 0, y] = bfn[y].

We choose the y coordinate to be **periodic**.

We will require f to have a **reflection symmetry** at the boundary x = 1, so that we can extend f to the domain $x \in (0, 2)$, $y \in (0, 2\pi)$ with f[x, y] = f[2 – x, y].
[Equivalently we may say that we wish to find solutions to our problem in the space of functions that are even at x = 1;
Note also that this implies a Neumann condition at x = 1, but in fact is stronger as it implies all odd normal derivatives must vanish.]

We will discretize the p.d.e. on a **uniform** grid in x, y, and use the *Mathematica* function 'Interpolation' to compute the stencils.

Please run the following to show the problems in exercise 1;

The number of grid points in the x, y directions will be NX, NY; we will take a small number of points for simplicity of demonstrating the results you should obtain;

```
NX = 3;

NY = 3;
```

We will choose the points in the x - direction to be located at the following;

```
tmp1 = Table[(ii - 1.)/(NX - 0.5), {ii, 1, NX}]
```

We impose the x reflection boundary condition at x = 1 by taking two copies of our grid in x as follows; note that we have picked our x spacing above carefully so that after have done this we still obtain a uniformly spaced grid.

```
Join[tmp1, Reverse[2 - tmp1]]
```

And the points in y will be located at;

```
tmp2 = Table[2 π (ii - 0.5)/NY, {ii, 1, NY}]
```

We impose the y periodicity by taking copies of our grid and attaching them at y=0 and y=2π as follows; Again note that we have chosen our y points carefully so that when we extend the grid as below again it is uniformly spaced.

```
Join[-2 π + tmp2, tmp2, 2 π + tmp2]
```

Problem 1 : create the basic domain

The first task is to create an array ' grid' whose elements contain the (x, y) coordinates of each point in the domain, together with a variable name. These are b1, b2, ... for the boundary points which will hold the Dirichlet data for the problem. The remainder are called f1, f2, ... and are the variables we will wish to solve for, given the p.d.e. You should write some code to generate an array 'grid' that looks like the following.

```
grid = {{{0.`, 1.0471975511965976`}  b1},
        {{0.`, 3.141592653589793`}  b2},
        {{0.`, 5.235987755982988`}  b3},
        {{0.4`, 1.0471975511965976`}  f1},
        {{0.4`, 3.141592653589793`}  f2},
        {{0.4`, 5.235987755982988`}  f3},
        {{0.8`, 1.0471975511965976`}  f4},
        {{0.8`, 3.141592653589793`}  f5},
        {{0.8`, 5.235987755982988`}  f6}};
```

Problem 2 : create an extended domain to implement reflection and periodic boundary conditions

We now need to implement the periodic behaviour in y, and the reflection boundary condition at x=1. We achieve this in a simple way by taking copies of 'grid' and after appropriate manipulation glue them together into an array 'gridfull', to double the x domain as described above, and add a copy of the y domain at both y=0 and y=2π as above. You should write code to generate from 'grid' the following extended array' gridfull'.

```
{{{0.`, -5.235987755982989`}  b1},
 {{0.`, -3.141592653589793`}  b2},
 {{0.`, -1.0471975511965983`}  b3},
 {{0.4`, -5.235987755982989`}  f1},
 {{0.4`, -3.141592653589793`}  f2},
 {{0.4`, -1.0471975511965983`}  f3},
 {{0.8`, -5.235987755982989`}  f4},
 {{0.8`, -3.141592653589793`}  f5},
 {{0.8`, -1.0471975511965983`}  f6},
 {{2.`, -5.235987755982989`}  b1},
 {{2.`, -3.141592653589793`}  b2},
 {{2.`, -1.0471975511965983`}  b3},
 {{1.6`, -5.235987755982989`}  f1},
 {{1.6`, -3.141592653589793`}  f2},
 {{1.6`, -1.0471975511965983`}  f3}
```

```
gridfull = {{1.2`, -5.235987755982989`}  f4
            {1.2`, -3.141592653589793`}  f5
            {1.2`, -1.0471975511965983`} f6
            {0.`, 1.0471975511965976`}   b1
            {0.`, 3.141592653589793`}    b2
            {0.`, 5.235987755982988`}    b3
            {0.4`, 1.0471975511965976`}  f1
            {0.4`, 3.141592653589793`}   f2
            {0.4`, 5.235987755982988`}   f3
            {0.8`, 1.0471975511965976`}  f4
            {0.8`, 3.141592653589793`}   f5
            {0.8`, 5.235987755982988`}   f6
            {2.`, 1.0471975511965976`}   b1
            {2.`, 3.141592653589793`}    b2
            {2.`, 5.235987755982988`}    b3
            {1.6`, 1.0471975511965976`}  f1
            {1.6`, 3.141592653589793`}   f2
            {1.6`, 5.235987755982988`}   f3
            {1.2`, 1.0471975511965976`}  f4
            {1.2`, 3.141592653589793`}   f5
            {1.2`, 5.235987755982988`}   f6
            {0.`, 7.330382858376184`}    b1
            {0.`, 9.42477796076938`}     b2
            {0.`, 11.519173063162574`}   b3
            {0.4`, 7.330382858376184`}   f1
            {0.4`, 9.42477796076938`}    f2
            {0.4`, 11.519173063162574`}  f3
            {0.8`, 7.330382858376184`}   f4
            {0.8`, 9.42477796076938`}    f5
            {0.8`, 11.519173063162574`}  f6
            {2.`, 7.330382858376184`}    b1
            {2.`, 9.42477796076938`}     b2
            {2.`, 11.519173063162574`}   b3
            {1.6`, 7.330382858376184`}   f1
            {1.6`, 9.42477796076938`}    f2
            {1.6`, 11.519173063162574`}  f3
            {1.2`, 7.330382858376184`}   f4
            {1.2`, 9.42477796076938`}    f5
            {1.2`, 11.519173063162574`}  f6};
```

Problem 3 : use 'Interpolation' to create finite difference stencil

We may now directly use the *Mathematica* 'Interpolation' function using the argument 'gridfull' to generate a finite difference approximation to a function. This will automatically build in our boundary conditions.

The below uses 2nd order interpolation, but higher order is simply achieved by changing the option InterpolationOrder→2. [Note you shouldn't use higher order than the minimum of NX and NY. So don't try higher orders until you have increased these!]

```
fninterp = Interpolation[gridfull, InterpolationOrder → 2];
```

Likewise we may construct 1st and 2nd derivatives of this interpolation function.

```
dxfninterp[x_, y_] = D[fninterp[x, y], x];

dyfninterp[x_, y_] = D[fninterp[x, y], y];

dxxfninterp[x_, y_] = D[fninterp[x, y], {x, 2}];

dyyfninterp[x_, y_] = D[fninterp[x, y], {y, 2}];

dxyfninterp[x_, y_] = D[fninterp[x, y], x, y];
```

Write code to use grid and gridfull to create a (flat) list in the domain $x\epsilon(0,1)$ $y\epsilon(0,2\pi)$ of;

1) the x coordinates of each point (called xval)
2) the y coordinates of each point (called yval)
3) the function value at each point (called fval)
4) the x derivative of the function at each point (called dxfval)
5) likewise the other derivaties (called dyfval, dxxfval, dyyfval, dxyfval)

For example, xval and yval should give;

```
xval = {0.`, 0.`, 0.`, 0.4`, 0.4`, 0.4`, 0.8`, 0.8`, 0.8`};

yval = {1.0471975511965976`, 3.141592653589793`, 5.235987755982988`,
   1.0471975511965976`, 3.141592653589793`, 5.235987755982988`,
   1.0471975511965976`, 3.141592653589793`, 5.235987755982988`};
```

For example, fval should look like;

```
fval = {b1, b2, b3, f1, f2, f3, f4, f5, f6};
```

And the derivatives, dxfval ... should look like;

```
dxfval = {-3.75` b1 + 5.` f1 - 1.25` f4, -3.75` b2 + 5.` f2 - 1.25` f5,
   -3.75` b3 + 5.` f3 - 1.25` f6, -1.25` b1 + 1.25` f4, -1.25` b2 + 1.25` f5,
   -1.25` b3 + 1.25` f6, -1.2499999999999998` f1 + 1.2499999999999998` f4,
   -1.2499999999999998` f2 + 1.2499999999999998` f5,
   -1.2499999999999998` f3 + 1.2499999999999998` f6};
```

```
dyfval = {0.238732414637843` b2 - 0.238732414637843` b3,
   -0.23873241463784295` b1 + 0.23873241463784317` b3,
   0.23873241463784295` b1 - 0.23873241463784317` b2,
   0.23873241463784298` f2 - 0.23873241463784298` f3,
   -0.23873241463784292` f1 + 0.23873241463784314` f3,
   0.23873241463784292` f1 - 0.23873241463784314` f2,
   0.23873241463784298` f5 - 0.23873241463784298` f6,
   -0.23873241463784292` f4 + 0.23873241463784314` f6,
   0.23873241463784292` f4 - 0.23873241463784314` f5};
```

```
dxxfval = {6.25` b1 - 12.5` f1 + 6.25` f4,
    6.25` b2 - 12.5` f2 + 6.25` f5, 6.25` b3 - 12.5` f3 + 6.25` f6,
    6.25` b1 - 12.5` f1 + 6.25` f4, 6.25` b2 - 12.5` f2 + 6.25` f5,
    6.25` b3 - 12.5` f3 + 6.25` f6, 6.250000000000001` f1 - 6.250000000000001` f4,
    6.250000000000001` f2 - 6.250000000000001` f5,
    6.250000000000001` f3 - 6.250000000000001` f6};
```

```
dyyfval =
  {-0.4559453263905199` b1 + 0.22797266319525994` b2 + 0.22797266319525994` b3,
   0.22797266319526` b1 - 0.45594532639052016` b2 + 0.2279726631952601` b3,
   0.22797266319526` b1 + 0.2279726631952601` b2 - 0.45594532639052016` b3,
   -0.4559453263905199` f1 + 0.22797266319525994` f2 + 0.22797266319525994` f3,
   0.22797266319526` f1 - 0.4559453263905201` f2 + 0.2279726631952601` f3,
   0.22797266319526` f1 + 0.2279726631952601` f2 - 0.4559453263905201` f3,
   -0.4559453263905199` f4 + 0.22797266319525994` f5 + 0.22797266319525994` f6,
   0.22797266319526` f4 - 0.4559453263905201` f5 + 0.2279726631952601` f6,
   0.22797266319526` f4 + 0.2279726631952601` f5 - 0.4559453263905201` f6};
```

```
dxyfval = {-0.8952465548919112` b2 + 0.8952465548919112` b3 + 1.193662073189215` f2 -
    1.193662073189215` f3 - 0.29841551829730373` f5 + 0.29841551829730373` f6,
   0.8952465548919111` b1 - 0.8952465548919118` b3 - 1.1936620731892147` f1 +
    1.1936620731892158` f3 + 0.2984155182973037` f4 - 0.29841551829730395` f6,
   -0.8952465548919111` b1 + 0.8952465548919118` b2 + 1.1936620731892147` f1 -
    1.1936620731892158` f2 - 0.2984155182973037` f4 + 0.29841551829730395` f5,
   -0.29841551829730373` b2 + 0.29841551829730373` b3 + 0.29841551829730373` f5 -
    0.29841551829730373` f6, 0.2984155182973037` b1 - 0.29841551829730395` b3 -
    0.2984155182973037` f4 + 0.29841551829730395` f6, -0.2984155182973037` b1 +
    0.29841551829730395` b2 + 0.2984155182973037` f4 - 0.29841551829730395` f5,
   -0.2984155182973037` f2 + 0.2984155182973037` f3 + 0.2984155182973037` f5 -
    0.2984155182973037` f6, 0.2984155182973036` f1 - 0.2984155182973039` f3 -
    0.2984155182973036` f4 + 0.2984155182973039` f6, -0.2984155182973036` f1 +
    0.2984155182973039` f2 + 0.2984155182973036` f4 - 0.2984155182973039` f5};
```

Problem 4 : check our discretization

Let us check our discretization.

Create a replace list ‘varrepl’ to assign the values to the boundary points b1,b2,... and the interior points f1, f2, ... defined by the function;

```
fn[x_, y_] = Cos[y] (1 - 0.5 (1 - x)^2);
```

You should generate a list ‘varrepl’;

```
varrepl = {b1 → 0.25000000000000006`, b2 → -0.5`, b3 → 0.24999999999999964`,
    f1 → 0.41000000000000014`, f2 → -0.8200000000000001`, f3 → 0.4099999999999994`,
    f4 → 0.4900000000000001`, f5 → -0.98`, f6 → 0.48999999999999927`};
```

Use this list to compare your differenced derivatives to the actual derivatives. For example our 1st order x differences;

```
dxfval /. varrepl
```

whereas the exact values would give;

```
Table[D[fn[x, y], x] /. {x → xval[[ii]], y → yval[[ii]]}, {ii, 1, Length[xval]}]
```

These agree **exactly** as the function is quadratic in x, and we have used 2 nd order interpolation.

However, the same in y....

```
dyfval /. varrepl
```

```
Table[D[fn[x, y], y] /. {x → xval[[ii]], y → yval[[ii]]}, {ii, 1, Length[xval]}]
```

... shows only approximate agreement as we would expect. This agreement is **not very good** with very few points, but will quickly get better taking larger NX, NY (here we only have 3 points in each direction!). Experiment with this, for different InterpolationOrders (greater than 2), and check for improvement with increasing NX, NY.

Problem 5 : implement boundary conditions, initial guess and p.d.e. to be solved.

Now we will fix the boundary conditions at x = 0; let us choose these to be defined by bfn[y], so that f[0, y] = bfn[y]. Let' s pick the (periodic) function cosine;

```
bfn[y_] = Cos[y];
```

Then write code to create a replace list ' boundaryrepl' that sets the values of the boundary variables b1, b2, ... to the appropriate values defined by bfn. You should then have a list,

```
boundaryrepl = {b1 → 0.5000000000000001`, b2 → -1.`, b3 → 0.4999999999999993`};
```

Likewise in order to solve a pde, we will need an initial guess for a solution. For simplicity let us just use bfn[y] to define an initial guess fguess[x, y] = bfn[y]. Note that this correctly respects all the boundary conditions (including reflection symmetry at x=1).

Then write code to create a replace list for the interior variables f1, f2 ... that implements this initial guess. Call this ' interiorrepl' and it should look like;

```
interiorrepl = {f1 → 0.5000000000000001`, f2 → -1.`, f3 → 0.4999999999999993`,
   f4 → 0.5000000000000001`, f5 → -1.`, f6 → 0.4999999999999993`};
```

Now we are ready to solve a pde using this initial guess and boundary conditions. We will use the ' FindRoot' function. We must create a list of the equations at the interior points, and also a list of the variables (f1, f2 ....) to solve for together with an initial guess.

Suppose, for example, we wish to solve the Laplace equation; **D[f[x, y], {x, 2}] + 5 D[f[x, y], {y, 2}] == 0**

Then we may simply create a list of the equations using; **dxxfval + 5 dyyfval**

```
dxxfval + 5 dyyfval
```

This is a list of the LHS of our pde equation evaluated at all points on the grid domain. We must remove the boundary points from this list. Also we should use ' boundaryrepl' to put in the correct values for the boundary points b1, b2, ...

Write code to strip off the boundary points, and replace the values for the boundary points, and store the resulting list in ' interiorequations'. This should look like;

```
interiorequations = {3.125000000000001` - 12.5` f1 +
    5 (-0.4559453263905199` f1 + 0.22797266319525994` f2 + 0.22797266319525994` f3) +
    6.25` f4, -6.25` - 12.5` f2 +
    5 (0.22797266319526` f1 - 0.4559453263905201` f2 + 0.2279726631952601` f3) +
    6.25` f5, 3.1249999999999956` +
    5 (0.22797266319526` f1 + 0.2279726631952601` f2 - 0.4559453263905201` f3) -
    12.5` f3 + 6.25` f6, 6.250000000000001` f1 - 6.250000000000001` f4 +
    5 (-0.4559453263905199` f4 + 0.22797266319525994` f5 + 0.22797266319525994` f6),
   6.250000000000001` f2 - 6.250000000000001` f5 +
    5 (0.22797266319526` f4 - 0.4559453263905201` f5 + 0.2279726631952601` f6),
   6.250000000000001` f3 + 5 (0.22797266319526` f4 + 0.2279726631952601` f5 -
       0.4559453263905201` f6) - 6.250000000000001` f6};
```

Problem 6 : Solve the Laplace equation

Then we must create a list of the interior variables we wish to solve for (ie. f1, f2....), together with an initial guess for each (using interiorrepl). This must take the form {{f1, f1guess}, {f2, f2guess}, ...}, with the appropriate values for the guesses. Write code to generate this list and store it in ' varandguess'. It should look like;

```
varandguess = {{f1, 0.5000000000000001`}, {f2, -1.`}, {f3, 0.4999999999999993`},
   {f4, 0.5000000000000001`}, {f5, -1.`}, {f6, 0.4999999999999993`}};
```

Now we can pass these arguements to FindRoot, and store the solution in the replacement list ‘solution’.

```
solution = FindRoot[interiorequations, varandguess];
```

In fact it is interesting to see how FindRoot does in solving. We may use the ' StepMonitor' option as;

```
solution = FindRoot[interiorequations, varandguess,
   StepMonitor :> Print["norm = ", Norm[interiorequations]]];
```

(Note; only one step is required as the pde is linear and each step is Newton-Rapheson which is exact for a linear system)

You should then find the following solution;

```
solution =
  {f1 → 0.26305016446675833`, f2 → -0.5261003289335167`, f3 → 0.2630501644667579`,
   f4 → 0.170024120647368`, f5 → -0.3400482412947359`, f6 → 0.17002412064736766`};
```

We may easily plot our solution using the code;

```
ListPlot3D[Table[
  {grid[[ii, 1, 1]], grid[[ii, 1, 2]], fval[[ii]] /. boundaryrepl /. solution}
  , {ii, 1, Length[grid]}], PlotRange → All]
```

Determine the exact analytic solution for this p.d.e. with its boundary conditions and compare it to our discretized solution.

Of course the agreement is not very good for such low numbers of points but increasing NX, NY to 10 each,

and taking InterpolationOrder → 6 should give a nice agreement.

Problem 7 : Solve a non-linear Laplace equation

Where these methods become much more powerful is when no exact solution is available. Now try to modify your equation to solve the ***non - linear*** elliptic pde;

D[f[x, y], {x, 2}] + 5 D[f[x, y], {y, 2}] - 2 f[x, y]^2 == 0

Just do the same manipulations (strip boundary points and replace boundary values) as above, but now on the list;

dxxfval + 5 dyyfval - 2 fval^2

You should then find a solution;

```
solution2 = {f1 → 0.20404183989284114`, f2 → -0.6915666859498503`,
   f3 → 0.20404183989284064`, f4 → 0.08474567161566736`,
   f5 → -0.5567675552494726`, f6 → 0.084745671615667`};
```

Check that the solution to this equation is consistent with converging to a (smooth) solution as you increase NX, NY. Also check that the reflection boundary condition at x=1 and periodicity of y are correctly implemented in the solution.