Mathematica implementation

```
clebschFunction[{x1_, x2_, x3_, x4_, x5_}] :=
 x1^3 + x2^3 + x3^3 + x4^3 + x5^3
cubicInCP3[{x_, y_, z_, w_}] :=
 clebschFunction[{x, y, z, w, -x - y - z - w}]
cubicInC3[{x_, y_, z_}] := cubicInCP3[{x, y, z, 1}]
point[u1_, v1_, u2_, v2_, lambda_] :=
 lambda {5, u1, v1} + (1 - lambda) {-5, u2, v2}
lines = Solve[{cubicInC3[point[u1, v1, u2, v2, 0]] == 0,
    cubicInC3[point[u1, v1, u2, v2, 1]] == 0,
    cubicInC3[point[u1, v1, u2, v2, 2]] == 0,
    cubicInC3[point[u1, v1, u2, v2, 3]] == 0},
    {u1, v1, u2, v2}];
```

Remarks

- This gives us 18 lines on the cubic surface. There are 9 more to find, but we can worry about that later.
- Look carefully at the definition of `clebschFunction` and `cubicInC3`. These are functions that apply to lists. For example `cubicInC3` is a function which takes a single parameter consisting of a list of three points.

```
sumOfSquares[x_,y_]:= x^2 + y^2
normSq[{ x_,y_ }]:= x^2 + y^2
sumOfSquares[3,4] == 25
normSq[{3,4}] == 25
```

Goal - plot these lines

- The first ingredient we need is to know how to plot "graphics primitives"
- To create a cylinder starting at point1, ending at point2 and with radius 0.1 we would use the following code:

```
point1 = {0,0,0};
point2 = {0,0,1};
cylinder = Cylinder[ {point1,point2}, 0.1];
Graphics3D[ cylinder ];
```

- There are lots of "graphics primitives" such as spheres, cones etc. that you can combine to create 3D pictures
- Use Graphics3D to display a list of primitives.

The ReplaceAll function

- The ReplaceAll function allows you to substitute values in an expression. Try the following

```
ReplaceAll[ x^2 + 3 x + 2, x->y]
ReplaceAll[ x^2 + 3 x + 2, 3->5]
ReplaceAll[ x^2 + 3 x + 2, {x->y, 3->z}]
```

- ReplaceAll is so useful there is a short hand for it /.

```
x^2 + 3 x + 2 /. x->y
```

The Map function

- Very often you want to apply the same function to every element of a list you can do this with the Map function.

```
squareIt[x_]:=x^2;
Map[ squareIt, {0,1,2,3,4}]
Map[ Factorial, {0,1,2,3,4}]
```

- Map is so useful it too has its own shorthand

```
squareIt[x_]:=x^2;
squareIt /@ {0,1,2,3,4}
Factorial /@ {0,1,2,3,4}
```

- Map is called "map" because if a function is a mapping, map applies this mapping to a list. Thus whenever you want to apply a function repeatedly, you'll probably want to use Map.

Anonymous functions

You might find it tedious to write a whole new function just because you want to perform an operation on every element of a list. You can define "anonymous functions" to minimize typing:

```
squareIt[x_]:=x^2;
squareIt /@ {0,1,2,3,4}
```

```
#^2 & /@ {0,1,2,3,4}
```

The # is a placeholder for the element of a list and the & means I've finished defining my function. Here's how to compute $\sin(x^x)$ for every element x of a list.

```
Sin(#^#) & /@ {0,1,2,3,4}
```

Putting it together

- We can plot the lines on the cubic surface by combining the `Cylinder`, `Map` and `ReplaceAll` functions
- We use `ReplaceAll` to find the coordinates where each line intersects the planes $x = 5$ and $x = -5$
- We use `Cylinder` to draw a cylinder of small radius connecting these points
- We use `Map` to apply the function to every line

Solution

```
findPoint1[ line_ ] := ReplaceAll[ {5, u1, v1}, line]
findPoint2[ line_ ] := ReplaceAll[ {-5, u2, v2}, line]
createCylinder[line_] :=
 Cylinder[ {findPoint1[line], findPoint2[line]}, 0.1]
cylinders = Map[ createCylinder, lines];
Graphics3D[ cylinders ]
```

- A "line" in this code is one of the items in the list returned by `Solve`
- Type `lines[[1]]` to see the first line, for example:

```
{u1 -> -5, v1 -> -1, u2 -> 5, v2 -> -1}
```

- So `findPoint1[ lines[[1]] ]` means the same as

```
ReplaceAll[ {5, u1, v1}, {u1 -> -5, v1 -> -1, u2 -> 5, v2 -> -1}]
```

  So `findPoint1` computes the point where the given line intersects the plane $x = 5$
- `findPoint2` finds the point where the line intersects the plane $x = -5$.
- It may seem strange to use the `ReplaceAll` function, but we don't have much choice: the output of `Solve` is a replacement expression.

- Once we understand `findPoint1` and `findPoint2`, `createCylinder` is easy to understand
- We then use `Map` to apply `createCylinder` to every line
- This gives us a list of cylinders which we plot using Graphics3D

Using Show

- Now we have a plot of the lines
- We saw last week how to use `ContourPlot3D` to plot the surface itself
- To combine to sets of graphics, one can use the `Show` command

```
surface =
  ContourPlot3D[
   cubicInC3[{x, y, z}] == 0,
   {x, -5, 5}, {y, -5, 5}, {z, -5, 5},
   ContourStyle -> Opacity[0.5], Mesh -> None];
Show[ surface, Graphics3D[ cylinders]]
```

Summary

- We can define functions using patterns (more on this in a moment)
- We can use `Map` to automate tasks on lists
- We can use `ReplaceAll` to perfom substitutions
- `ReplaceAll` is the natural way to work with the output of Solve.
- Our code is infinitely more sophisticated than last week - we've started programming!

Making your code unreadable?

```
findPoint1[ line_ ] := {5, u1, v1} /. line
findPoint2[ line_ ] := {-5, u2, v2} /. line
createCylinder[line_] :=
 Cylinder[ {findPoint1[line], findPoint2[line]}, 0.1]
cylinders = createCylinder /@ lines;
Graphics3D[ cylinders ]
```

This is the same code as before except we've used the shorthand `/.` and `/@` for `ReplaceAll` and `Map`. An experienced Mathematica programmer would probably be more likely to write this less immediately readable version. Of course, once you are an experienced Mathematica programmer this should be reasonably readable.

Using Table for repetitive tasks

Another useful method of automating tasks in Mathematica is to use `Table`

```
Table[x^2, {x, 1, 10}]
Table[x*y, {x, 1, 10}, {y, 1, 10}]
Table[x*y, {x, 1, 10}, {y, x, 10}]
```

You can see that `Table` creates 2×2 tables when you supply 2×2 ranges. Use `Flatten` to get rid of unwanted nesting.

Exercises

★ Create a table of values of $n^2 + n + 41$ as n ranges from 0 to 39. Apply the function `PrimeQ` to every element of your table.

★ By using `Flatten` and `Table` create a list of all the composite numbers between 1 and 100 - don't worry if your list contains duplicates. You can treat lists as sets in Mathematica using the functions `Union`, `Complement`, `Intersection` and `DeleteDuplicates`. Use this to get a list of primes between 1 and 100.

★ Write a function `primes` that takes a parameter n and outputs a list of all the primes up to n.

★ Use the two dimensional graphics primitive `Circle` together with the command `Graphics` that displays two dimensional graphics to plot 12 small circles which are evenly spaced on a large circle (i.e. a clockface). Do this in two different ways: by using the `Table` command and by applying `Map` to the list `{1,2,3,4,5,6,7,8,9,10,11,12}`.

★ Use the function `Select` and the function `PrimeQ` to obtain a list of all prime numbers up to 100.

Summary

Here are some key functions for working with lists. Search for "List Manipulation" for even more.

- `Table`, `Array`, `{}`
- `First`, `Last`, `[[]]`, `Length`
- `Select`, `DeleteDuplicates`
- `Flatten`, `Join`, `Sort`, `Reverse`
- `Map`
- `Total`, `Count`

Schläfli graph

- We've found 18 lines on the cubic. We'd like to find all 27. In addition, we'd like to find which lines intersect. The graph of the intersections was found by Schläfli.
- As a first step, let's find a better representation of a line than a list of replacement expressions.
- Our "matrix representation" of a line consists of a list of two vectors which lie on the line. This is a valid representation if the resulting matrix has rank 2.
- How can we test if two such lines are equal? That they intersect?
- How can we transform the 18 replacement expressions we've found using Solve into a list of Matrices?

Testing for equality and intersection

```
equalsQ[ mRep1_, mRep2_] :=
   MatrixRank[ Join[ mRep1, mRep2 ]] == 2
intersectQ[ mRep1_, mRep2_] :=
   MatrixRank[ Join[ mRep1, mRep2 ]] == 3
```

Converting to matrix form

```
clebschFunction[{x1_, x2_, x3_, x4_, x5_}] :=
 x1^3 + x2^3 + x3^3 + x4^3 + x5^3
cubicInCP3[{x_, y_, z_, w_}] :=
 clebschFunction[{x, y, z, w, -x - y - z - w}]
cubicInC3[{x_, y_, z_}] := cubicInCP3[{x, y, z, 1}]
point[u1_, v1_, u2_, v2_, lambda_] :=
 lambda {1, u1, v1} + (1 - lambda) {-1, u2, v2}
lines = Solve[{cubicInC3[point[u1, v1, u2, v2, 0]] == 0,
    cubicInC3[point[u1, v1, u2, v2, 1]] == 0,
    cubicInC3[point[u1, v1, u2, v2, 2]] == 0,
    cubicInC3[point[u1, v1, u2, v2, 3]] == 0}, {u1, v1, u2, v2}];

p1[line_] := {1, u1, v1, 1} /. line
p2[line_] := {-1, u2, v2, 1} /. line
matrixRep[ replacementRep_ ] :=
 Simplify[{p1[replacementRep], p2[replacementRep]}]
lineMatrices = matrixRep /@ lines;
```

Remarks

We've rerun our calculation of the lines using the planes $x = \pm 1$ instead of $x = \pm 5$. This makes the computation a bit easier for Mathematica to handle. The values of ± 5 were used before to give a prettier picture.
Clearly the cubic in $\mathbb{CP}^3$ is symmetric in the variables (x_1, x_2, x_3, x_4) but we've thrown away this symmetry when moving to homogeneous coordinates and choosing two planes.
If we apply the linear transformations corresponding to all the permutations of coordinates to the lines we've found, we expect to find every line on the cubic.

Here's how we can create a linear transformation corresponding to a permutation:

```
matrixForPerm[ perm_ ] :=
 Table[ If[perm[[i]] == j, 1, 0], {i, 1, 4}, {j, 1, 4}]
matrixForPerm[{1, 2, 4, 3}]
```

Note the `If` function.

```
If[test, valueIfTrue, valueIfFalse]
```

The `If` function occurs often in mathematical expressions, though this is the conventional notation:

$$m_{ij}^{\sigma} = \begin{cases} 1 & \sigma(i) = j \\ 0 & \text{otherwise} \end{cases}$$

Here m^{σ} is a matrix associated with a permutation σ of the set $\{1, 2, 3, 4\}$.

Computing all permutation matrices

Using the built in `Permutations` function we can generate all permutations of the list $\{1, 2, 3, 4\}$. Using `Map` we can then generate a list of all the desired linear transformations.

```
permutationMatrices =
  matrixForPerm /@ Permutations[{1, 2, 3, 4}];
```

Permuting the lines

Using `Map` again we can write a function `permuteLines` that takes a permutation and returns all the lines with the permutation applied:

```
permutationMatrices =
  matrixForPerm /@ Permutations[{1, 2, 3, 4}];
permuteLines[ perm_ ] := Module[{applyMatrix},
  applyMatrix[mRep_] := Transpose[ perm . Transpose[ mRep]];
  applyMatrix /@ lineMatrices
  ]
```

Permuting the lines

Using `Map` again we can apply `permuteLines` to every one of our permutations and `Flatten` the result to obtain a list of all the lines with all the symmetries applied.

```
permutationMatrices =
  matrixForPerm /@ Permutations[{1, 2, 3, 4}];
permuteLines[ perm_ ] := Module[{applyMatrix},
  applyMatrix[mRep_] := Transpose[ perm . Transpose[ mRep]];
  applyMatrix /@ lineMatrices
  ]
permutedLines =
  Flatten[ permuteLines /@ permutationMatrices, 1];
```

Note we pass the value 1 to `Flatten`. We only want it to remove one level of nesting.

Deleting duplicates

Clearly this will have generated a lot of duplicates. So lets remove them using `DeleteDuplicates`.

```
distinctLines = DeleteDuplicates[ permutedLines, equalsQ ];
Length[ distinctLines ]
```

Note that we're passing two parameters to `DeleteDuplicates` one is the list to process, the second is a function to use to test if two entries should be considered as duplicates.

Summary so far

```
(* Compute permutation matrices *)
matrixForPerm[ perm_ ] :=
 Table[ If[perm[[i]] == j, 1, 0], {i, 1, 4}, {j, 1, 4}]
permutationMatrices =
  matrixForPerm /@ Permutations[{1, 2, 3, 4}];
permutationMatrices =
  matrixForPerm /@ Permutations[{1, 2, 3, 4}];
(* Apply the permutation matrices to our lines *)
permuteLines[ perm_ ] := Module[{applyMatrix},
  applyMatrix[mRep_] := Transpose[ perm . Transpose[ mRep]];
  applyMatrix /@ lineMatrices
  ]
permutedLines =
  Flatten[ permuteLines /@ permutationMatrices, 1];
(* Remove duplicates and count the result *)
distinctLines = DeleteDuplicates[ permutedLines, equalsQ ];
Length[ distinctLines ]
```

Final touch

```
intersectionMatrix =
  Table[
  intersectQ[ distinctLines[[i]], distinctLines[[j]] ],
  {i, 1, 27},
  {j, 1, 27}];
MatrixForm[ intersectionMatrix ]
```

This completes the computation of the lines on the Clebsch cubic and their intersection matrix.