Revision

- What does `/@` mean?
- What does `/.` mean?
- What does

```
#^2 & /@ {1,2,3,4,5}
```

compute?

Mathematical discussion

- Dimension counting suggests that counting lines on cubics is a "sensible" problem. We reduced it to 4 equations in 4 unknowns.
- "sensible" means that we can reasonably expect the number of solutions to remain constant even if we perturb the cubic. To be precise our dimension counting is strong evidence that we can apply the implicit function theorem to show that the number of solutions remains constant for small perturbations.
- It's easy (exercise?) to check the details in this special case and show that the relevant differential in the implicit function theorem is indeed onto and so cubics near the Clebsch cubic also have 27 lines.
- So our result isn't such a special case as it seems.

Existential Varieties

- A variety over some field F is defined by the intersection and union of zero sets of polynomials.

$$V = \{(z_1, \ldots z_n) : (p_1(z) = 0 \,\&\, p_2(z) = 0 \ldots) \,||\, \ldots\}$$

- The expression after the colon could be called "an expression in the language of polynomial sets"
- What happens if we extend our language to allow uses of the symbols $\forall$ and $\exists$? For example:

$$V = \{(z_1, z_2) : \exists z_3 \; s.t. \; z_1^2 = z_3 \,\&\, z_2^2 = z_3\}$$

The expression on the right is "an expression in the language of fields". We could sets defined in this way "existential varieties". For example the space of cubic curves containing 27 lines would be an existential variety.

Quantifier Elimination

Theorem

(Tarksi) Over an algebraically closed field F, existential varieties and varieties are the same thing. Moreover there is a simple algorithm for converting an expression in the language of fields to one which doesn't contain the quantifiers $\forall$ or $\exists$.

Mathematica has implemented this algorithm:

```
Reduce[ Exists[z3, z1^2 == z3 && z2^2 == z3], {z2, z3}]
```

See Basu Pollack and Roy for a rigorous treatment (it's not a difficult proof)

Corollaries

Corollary

(Thom) Varieties defined as non-singular intersections of the zero sets of polynomials of fixed degree over $\mathbb{C}$ are diffeomorphic. For example all non-singular cubics curves are diffeomorphic. As are all non-singular cubic surfaces. As are all non-singular degree 8 surfaces . . .

Corollary

(Cayley–Salmon–Schläfli) The generic cubic surface contains 27 lines and intersect according to the Schläfli graph. In fact this result is true if you replace "generic" with non-singular but that isn't such an immediate corollary.

Exercises

★ Plot the intersection of the lines on the cubic as a graph. The Schläfli graph is usually defined as the graph obtained by taking 27 points and joining them if the corresponding lines on the cubic *do not* exist. I find the complement of the Schläfli graph more intuitive. Plot whichever you prefer. Place the 27 vertices evenly spaced on a circle. Search Wikipedia for Schläfli graph to see what your answer whould look like.

★ Pick two disjoint lines ℓ_1 and ℓ_2 on the Clebsch cubic surface. Choose a parameterization of each of these lines (i.e. a map $\phi_i : \mathbb{C} \to \ell_i$. Now define a map ψ mapping $\mathbb{C} \times \mathbb{C}$ to the Clebsch cubic surface by ψ takes (u, v) to the third intersection of the line through $\phi_1(u)$ and $\phi_2(v)$ with the cubic surface. Compute ψ explicitly and hence generate a parameteric plot of the cubic.

Discussion

- The last exercise demonstrates that non-singular cubic surfaces are rational. All we need is two skew lines on a cubic and we can construct a map from $\mathbb{C} \times \mathbb{C}$ which is "almost onto".
- If one understands a little about rational maps and the blow up construction, this map we have constructed can be seen as a biholomorphism of $\mathbb{CP}^1 \times \mathbb{CP}^1$ blown up at five points with the cubic surface.
- Since $\mathbb{CP}^1 \times \mathbb{CP}^1$ is just the blow up of $\mathbb{CP}^2$ at one point, we have pretty much shown that all non-singular cubic surfaces are given by blowing up $\mathbb{CP}^2$ at 6 points. This gives rise straightforwardly to a complete classification of cubic surfaces.
- We've shown how you can use Mathematica to actually compute the isomorphism explicitly once you're given the cubic surface.

Pattern matching

We've seen pattern matching before:

```
solveQuadratic[ a_, b_, c_ ]:= (-b + Sqrt[b^2 - 4 a c])/(2a)
normSq[ {x_,y_}] := x^2 + y^2
```

Differentiation

```
diff[ Sin[x_], x_] := Cos[x]
diff[ Cos[x_], x_] := -Sin[x]
diff[ f_ + g_, x_] := diff[f, x] + diff[g, x]
diff[ f_ g_ , x_] := diff[f, x] g  + f diff[g, x]
```

Here we haven't attempted to define diff all at once, we've simply given instructions on how to differentiate certain patterns.

Differentiation

```
diff[ Sin[x_], x_] := Cos[x]
diff[ Cos[x_], x_] := -Sin[x]
diff[ f_ + g_, x_] := diff[f, x] + diff[g, x]
diff[ f_ g_ , x_] := diff[f, x] g  + f diff[g, x]
diff[ c_ , x_] := 0  /; Element[c, Reals]
```

This last rule is harder to read - it says D[c,x] is zero if c is simply a real number

Exercises

★ Try out the differentiation example. Check it works for $3Sin[x] + 5Cos[x]$ plus any other tests you'd like to try.

★ Extend the definition so you can differentiate $f(x)^{g(x)}$ and just x. Check your answer with x^n, e^x and x^x.

★ Implement the chain rule. You'll need to use the ReplaceAll function (well that was how I did it anyway)

★ See how you get on trying to implement an integration function - don't devote your life to this!

Storage and indices

What happens if you omit an underscore?

```
f[x]:=x^2
f[y]:=y^3
```

This is pretty irritating a lot of the time. But it can be useful for lookup. For example:

```
colour[blackberry] = purple
colour[banana]=yellow
```

```
basisVector[1] = {1,0,0};
basisVector[2] = {0,1,0};
basisVector[3] = {0,0,1};
```

Subscripts

- What happens if a function is not defined? Mathematica just leaves it alone.
- You can use $x[1]$, $x[2]$, $x[3]$ to mean x_1, x_2, x_3. $x[1]$ is then just as good a symbol as any other

```
clebschCubic = Total[Table[x[i]^3, {i, 1, 5}]]
```

The Schläfli graph through pattern matching

- Create a list containing the following symbols $a[i]$ $(1 \leq i \leq 6)$, $b[i]$ $(1 \leq i \leq 6)$, $c[i,j]$ $(1 \leq i < j \leq 6)$
- Write a function `intersectQ` that uses pattern matching to decide if two of these symbols represent intersecting lines. The rules are:
  - the a_i are skew
  - the b_i are skew,
  - a_i intersects b_j if and only if $i \neq j$
  - a_i intersects c_{jk} if and only $i \in \{j,k\}$
  - b_i intersects c_{jk} if and only $i \in \{j,k\}$
  - c_{ij} intersects c_{kl} if and only if $\{i,j\} \cap \{k,l\} = \emptyset$.
- Assuming it is true that the lines on a cubic surface can be labelled so that there intersections follow these rules, plot the complement of the Schläfli graph. Use the function `Text` to label the vertices.

Discussion

The sets of lines a_i and b_i are called a "double six". There are many ways of labelling the lines on a cubic surface to match the above rules. Find at least one for the Clebsch cubic surface.

Recursion

The Fibonacci sequence gives a classic example of recursion

```
fibonacci[0] := 1
fibonacci[1] := 1
fibonacci[n_] := fibonacci[n - 1] + fibonacci[n - 2]
Table[ fibonacci[n], {n, 1, 10}]
```

A function is allowed to call itself. Of course, there's a danger that you'll get stuck in an infinite loop, but fortunately you'll normally quickly get a "stack overflow" telling you that something has gone wrong.

```
stackOverflow[n_] :=
  stackOverflow[n - 1] + stackOverflow[n - 2]
stackOverflow[1]
```

Efficiency

- The algorithm we've just implemented is very inefficient. Try computing the 30-th Fibonacci number.
- The problem is that `fibonacci` forgets its previous working.
- Suppose computing the n-th Fibonacci number takes u_n "computer operations"
- $u_0 = 1,\ u_1,\ u_n = u_{n-1} + u_{n-2} + 1$

The Mathematica idiom for storing values

```
fibonacci[n_] :=
  fibonacci[n] = fibonacci[ n - 1 ] + fibonacci[n - 2]
fibonacci[0] = 1;
fibonacci[1] = 1;
fibonacci[999]
```

Notice that Mathematica's pattern matching is intelligent. It prefers the most specific pattern. When it can't decide on this basis the order in the file becomes important with early definitions taking precedence.

How Mathematica Works

- Everything in Mathematica is stored as a tree of function calls
- Use `FullForm` to see Mathematica's internal representation:

```
FullForm[ (u + v) (x + y)]
```

```
Times[Plus[u,v],Plus[x,y]]
```

- Use `TreeForm` for a prettier view.

```
TreeForm[ (u + v) (x + y)]
```

- Everything in Mathematica is stored as a list of lists where the first element of the list is labelled.

Parts of an expression

You can access parts of an expression using `[[ ]]` or the function `Part`

- `Part[ Circle[ {x, y}, r], 1]` gives ...
- `Circle[ {x, y}, r][[1]]` gives ...
- `Circle[ {x, y}, r][[1]]` gives ...
- `Circle[ {x, y}, r][[1, 2]]` gives ...

There is nothing special about lists other than the shorthand `{}`.

- `FullForm[ {1, 2, 3}]` gives `List[1,2,3]`

Head

To access the tag name at the beginning use part 0 or Head

- `, yCircle[x, r][[0]]` gives ...
- `Head[ Circle[ x, y, r]]` gives ...

Many functions work equally well with any expression as they do with lists.

- Length[Circle[x, y, r]]

The function `AtomQ` is sometimes handy to see if an expression is a leaf.

Big Idea

- The big idea is that Mathematical expressions are simply data structures which have a hierarchical structure.
- Write some good code for pattern matching and working with lists and you'll have a powerful Maths library in no time.
- (Your users will probably want some pretty features like * instead of Times)

★ Write a function to recursively print out the contents of an expression. Use `Print` to print the Head and the atoms.

Solution

```
printRecursively[ x_ ] := Print[x] /; AtomQ[x];
printRecursively[ x_ ] :=
 Module[ {}, Print[ Head[x]]; printRecursively /@ x; ]
printRecursively[ (u + v) (x + y)]
```

Evaluation

- Mathematica automatically processes expressions by performing a sequence of rules
- The most obvious rule is to replace the value of a function definition with the value of the function
- What is `FullForm[ (3*5) + (6*7)]`?

Evaluation

- Mathematica automatically processes expressions by performing a sequence of rules
- The most obvious rule is to replace the value of a function definition with the value of the function
- What is `FullForm[ (3*5) + (6*7)]`?
- `FullForm[ Hold[ (3*5) + (6*7)]]` is closer to what we want.
- `Hold` prevents Mathematica performing its usual processing.

HoldAll

- Some functions tell Mathematica NOT to apply rules to their arguments

```
i = 7;
Table[ i^2, {i, 1, 10}]
```

- If Mathematica was treating this like a normal function this would be equivalent to: `Table[ 7, {7, 1, 10}]`
- If we ask for `Attributes[Table]` we get
- `{HoldAll, Protected}`

Attributes

- Attributes of a function modify Mathematica's processing rules

```
Attributes[ Times ]
```

```
{Flat, Listable, NumericFunction, OneIdentity, -
Orderless, Protected}
```

- Flat means that when nested uses of Times occur, they should be flattened. It means much the same as associative. Compare the following:
- `FullForm[ Hold[x*(y*z)] ]`
- `FullForm[ x*(y*z) ]`

Other attributes

- `Orderless` means that the order of parameters is unimportant, so Mathematica should reorder them in a standard order - essentially alphabetic order.
- `Times[ y, x]` gives xy
- `Protected` means that normal users shouldn't be able to accidentally change the definition.
- `OneIdentity` means ...
- `Listable` means ...

SetAttribute

- Use `SetAttributes` to set an attribute.

★ Extend `diff` so it works on Matrices.

Back to HoldAll

Many Mathematica functions have the HoldAll attribute and it can be confusing

```
Clear[i];
expression = i^2;
range = {i, 1, 10};
Table[ expression, range]
```

produces an error. The fix is to use `Evaluate` which will perform the rules on an expression even if it is stored inside a `Hold`.

```
Table[ Evaluate[ expression], Evaluate[ range]]
Hold[ Evaluate[ 3*5 + 6*7]]
```

Sorting a list

- Sorting a list is the most basic algorithm taught to computer scientists
- It's the computer scientist version of the Euclidean algorithm.
- Try to come up with a good algorithm. To make it easy, I've printed some cards for you to sort as a team as quickly as possible.

A sort algorithm

```
sort[ {} ] := {}
sort[ {x_} ] := {x}
sort[  l_] := Module[{mid, lower, upper, midPoint},
  midPoint = l[[1]];
  lower = Select[l, # < l[[1]] &];
  mid = Select[l, # == l[[1]] &];
  upper = Select[l, # > l[[1]] &];
  Join[ sort[ lower], mid, sort[upper]]
  ]
```

We're making heavy use of anonymous functions here (# and &). We want to select only elements less than/equal to/greater than l[[1]]. This is a good example of how anonymous functions can improve your code.

Efficiency

```
exampleValues = Table[ Random[], {i, 1, 100000}];
Timing[ sort[ exampleValues]]
Timing[ Sort[ exampleValues]]
```

The first value output by `Timing` is the time taken. As you will see, Mathematica's `Sort` function is MUCH faster than our `sort` function. This is the sense in which Mathematica code is slow: it's built in functions are implemented in `C++` and are very fast, one's own code can be very slow.

Order of magnitude

Let's do a log-log plot of number of points in our list against the time taken to process it:

```
nValues = Table[10^i, {i, 3, 5, 0.25}];
sortRandom[ n_ ] := sort[ Table[ Random[], {j, 1, n}]];
timeSortRandom[ n_ ] := Timing[ sortRandom[n] ][[1]];
plotPoints =  {Log[#], Log[timeSortRandom[#]]} & /@ nValues;
ListPlot[ plotPoints, Joined -> True, Mesh -> All,
 AspectRatio -> Automatic]
```

The best you can do for a single threaded sorting algorithm is order $n \log n$, so I think we can be pretty happy with this.

Divide and conquer

- The sort algorithm is one instance of the general idea of using "divide and conquer" to speed up algorithms
- Recursion gives a convenient way of writing divide and conquer algorithms
- Undergraduate computer scientists spend a lot of time studying these kinds of algorithms and related data structures. They provide the building blocks of many sophisticated programs.

Functional programming

- We have now seen how to program in the "functional" programming style
- Functional programming is using functions, recursion, `Map`, lists and pattern matching to get results
- Some of you may have used procedural languages where `for` loops do most of the work. You can write `for` loops in Mathematica, but it's not good style.

Summary

- :=, =
- \@
- \.
- # &
- _, \;

Exercises

★ Rewrite the sort algorithm so that it does not use anonymous functions

★ Write a recursive program to draw a tree structure. Ideally make it look like a real tree.

★ Write a recursive program to create a 3D model of a Romanesco cauliflower.