Lecture plan

Gröbner bases in C++
- Using the Boost library
- The const keyword
- Operator overloading
- Using the standard template library
- Regular expressions
- Predicates
One cool thing that Mathematica can do is solve polynomial equations in many variables with a fair degree of success.

Underpinning many of these calculations is the theory of Gröbner bases.

We will use Gröbner bases to perform the “effective Nullstellensatz” (“zero locus theorem”).

Can we determine when a set of polynomials admits a simultaneous solution?

Example 1: Use Gaussian elimination on linear equations such as:

\[ x + y + z + 3 = 0 \]
\[ 2x + 2y + 2z + 5 = 0 \]

Example 2: Use GCD algorithm on equations in one variable:

\[ (x - 2)(x - 3)(x - 5)^2 = 0 \]
\[ x - 3 = 0 \]
A monomial is a term such as $x_0^2 x_1^3 x_3^5$ in $n$ variables.

**Definition**

A *Monomial Ordering*, $<$ is a total order on the set of monomials in $n$ variables satisfying:

- $M < N$ implies $MP < NP$
- $M < MP$ for all monomials $M$, $N$, $P$. 
Lexicographic order

Example

Lexicographic order is given by:

\[ x_0^{a_0} x_1^{a_1} \ldots x_{n-1}^{a_{n-1}} < x_0^{b_0} x_1^{b_1} \ldots x_{n-1}^{b_{n-1}} \]

if either:

- \( a_0 < b_0 \)
- \( a_0 = b_0 \) and \( a_1 < b_1 \)
- \( a_0 = b_0, \ a_1 = b_1 \) and \( a_2 < b_2 \)
- \( \ldots \)
Reduction

**Definition**

The term with highest order in a polynomial $p$ is denoted $\text{lm}(p)$ and is called the lead monomial. We similarly define the lead coefficient $\text{lc}(p)$ and the lead term $\text{lt}(p) = \text{lc}(p)\text{lm}(p)$.

**Definition**

If $f$ and $g$ are polynomials and some term of $f$ is divisible by $\text{lm}(g)$. Write $cm$ for this term of $f$ and define the one step reduction of $f$ by $g$ by:

$$\text{red}_1(f, g) = f - \frac{c}{\text{lc}(g) \text{lm}(g)} g$$

Otherwise $f$ is said to be irreducible by $g$.

- The one step reduction doesn’t contain the monomial $m$
- All terms greater than $m$ in our ordering remain unchanged
Given a finite set of polynomials $G$ write $\text{red}(f, G)$ for the total reduction. This is the result of applying one step reduction repeatedly using the elements of $G$ until we end up with something irreducible. (We need to be slightly more precise on the “repeatedly” to guarantee that this is unique)

If we call this total reduction $h$ we may write:

$$f = h + \sum_{g \in G} q_g g$$

so reduction is in some sense a generalization of polynomial division and $h$ can be considered the remainder and the process can be called multivariate division.
Gröbner basis

Definition

A Gröbner basis $G$ of an ideal $I$ in a polynomial ring is a set of polynomials generating the ideal such that multivariate division of any polynomial in the ideal $I$ by $G$ gives remainder 0.

- For simplicity we assume that our polynomials are over $\mathbb{C}$ which is algebraically closed.
- Recall Hilbert’s weak Nullstellensatz which tells us that the ideal generated by a set of polynomials contains 1 if and only if the polynomials have no common zero.
- Therefore a Gröbner basis gives an answer to the effective Nullstellensatz. The polynomials have a common solution if and only if the remainder of 1 on division by $G$ is 0.
S polynomial

**Definition**

Given polynomials $f$ and $g$ write $M$ for the monomial

$$M = \text{lcm}(\text{lm}(f), \text{lm}(g))$$

By construction this is divisible by $\text{lm}(f)$ and $\text{lm}(g)$.

Define

$$S(f, g) = \frac{M}{\text{lm}(f)} f - \frac{M}{\text{lm}(g)} g$$

In other words, $S$ is designed to represent cancelling the leading terms of $f$ and $g$.

**Theorem**

$G$ is a Gröbner basis if and only if the remainder on dividing $S(f, g)$ by $G$ is zero for all elements $f, g \in G$. 
Buchberger’s algorithm

- Input: A set of polynomials $F$ that generates $I$
- Output: A Gröbner basis $G$ for $I$

1. $G := F$
2. For every $g_i, g_j$ in $G$
3. Reduce $S(g_i, g_j)$, with the multivariate division algorithm relative to the set $G$ until the result is not further reducible.
4. If the result is non-zero, add it to $G$.
5. Repeat steps 2-4 until all possible pairs are considered, including those involving the new polynomials added in step 4.
6. Output $G$
Consider the ideal generated by the leading terms of $G$. These ideals give us an ascending chain of ideals. So by the Hilbert Basis theorem this chain eventually becomes constant.

Warning: there is no guarantee (from this argument) how long it will take. It can be very slow.

Performance depends heavily upon the choice of monomial ordering. Lexicographic ordering is normally a poor choice.

There are better Gröbner basis algorithms than Buchberger’s.

There are better algorithms for solving the weak Nullstellensatz than using Gröbner bases.
Credit

- I’ve borrowed heavily from Wikipedia to create this summary.
- Gröbner bases were introduced by Buchberger, but he was kind enough to name them after his supervisor.
- The book “Ideals, Varieties and Algorithms” is a great and very easy reference.
We can solve the effective Nullstellensatz by using Buchberger’s algorithm to find a Gröbner basis and then seeing if it contains 1.

Most of the work will be writing the classes to store Monomials and Polynomials. We can just copy the algorithms above blindly.

We’ll also write a class to capture the idea of a monomial order.
Monomial - sketch of design

A general monomial is $x_0^{i_0}x_1^{i_1} \ldots x_{n-1}^{i_{n-1}}$. Here’s the key behaviour we expect:

- **We will store a vector called powers** containing the powers $i_0, i_1, \ldots i_{n-1}$.
- **The number $n$ is called the dimension of the monomial.**
- **The number $i_0 + i_1 + \ldots + i_{n-1}$ is the degree of the monomial.**
- **You can multiply two monomials to obtain a new monomial.**
- **You can test to see if one monomial divides another and, if so, compute the quotient (needed for the S-polynomial)**
- **You can compute the least common multiple of two monomial (needed for the S-polynomial)**
- **You can test if two monomials are equal**
- **You can compare monomials in lexicographic order**
New C++ features we will use

- We would like to overload *, < and << so we can multiply monomials, compare them and print them out just as easily as we can multiply, compare and print numbers.
- We will use the `const` keyword to make it clear what methods modify a Monomial
class Monomial {
public:
    std::vector<int> powers;

    /** Get the degree of the monomial */
    int getDegree() const;
    /** Get the dimension (i.e. number of variables) */
    int getDimension() const;
    /** Does this divide other? */
    bool divides(const Monomial& other) const;
    /** Returns this/other */
    Monomial quotient(const Monomial& other) const;

    Monomial(std::vector<int> powers)
    : powers(powers) {};
};

/** Write a monomial to a stream */
std::ostream& operator<<(std::ostream& out, const Monomial& monomial);

/** Multiply two monomials */
Monomial operator*(const Monomial& x, const Monomial& y);

/** Compare two monomials using < */
bool operator<(const Monomial& x, const Monomial& y);

/** Comparison */
bool operator==(const Monomial& x, const Monomial& y);

/** Compute the lowest common multiple of two monomials */
Monomial lcm(const Monomial& x, const Monomial& y);
The const keyword

- You can use `const` to indicate that a parameter to a method is not changed by the method.

```cpp
double computeArea( const Circle& circle ) {
    return circle.radius*circle.radius*pi;
}
```

```cpp
double computeArea( const Circle& circle ) {
    return circle.radius = 1000; // compiler error
}
```

- It is often advised that by default you should pass objects by `const` reference.
The const keyword on member functions

- Sometimes member functions don’t change an object, you can use the `const` keyword to show this:

```cpp
class Circle {
public:
    void resize(double newRadius);
    void computeArea() const;

private:
    double radius;
};
```

- You might use multiple `const` declarations — for example some on the parameters and an additional "member" declaration.
**The const keyword on references and pointers**

- References and pointers can also be constant to indicate that you can’t change what they point to.
- It’s quite normal to return a `const` reference to a member variable. This combines the speed of reference passing with the safety of knowing the internals of your object can’t be changed.
- (Remember you must never return a reference to a local variable)
class Chart {
public:
    void resize( double newRadius );

    void addPoint( double x, double y ) {
        xValues.push_back(x);
        yValues.push_back(y);
    }

    const vector<double>& getXValues() {
        return xValues;
    }

    const vector<double>& getYValues() {
        return yValues;
    }

private:
    vector<double> xValues;
    vector<double> yValues;
};
Note that it is impossible for `xValues` and `yValues` to contain different numbers of elements.

Ensuring object integrity in this way using private variables is called *encapsulation*.

Encapsulation is pretty much synonymous with data hiding, but emphasizes the idea of maintaining consistent state.
```cpp
class Monomial {
public:
    std::vector<int> powers;

    /** Get the degree of the monomial */
    int getDegree() const;
    /** Get the dimension (i.e., number of variables) */
    int getDimension() const;
    /** Does this divide other? */
    bool divides(const Monomial& other) const;
    /** Returns this/other */
    Monomial quotient(const Monomial& other) const;

    Monomial(std::vector<int> powers)
        : powers(powers) {};
};

/** Write a monomial to a stream */
std::ostream& operator<<(std::ostream& out, const Monomial& monomial);
/** Multiply two monomials */
Monomial operator*(const Monomial& x, const Monomial& y);
/** Compare two monomials using < */
bool operator<(const Monomial& x, const Monomial& y);
/** Comparison */
bool operator==(const Monomial& x, const Monomial& y);
/** Compute the lowest common multiple of two monomials */
Monomial lcm(const Monomial& x, const Monomial& y);
```
Pros and cons of const

Pros:

- **const** allows you to get the compiler to guarantee your data won’t be modified by other programs.
- **const** is commonly used by C++ libraries so you’re being an idiomatic C++ developer if you use it.

Cons:

- **const** is visual clutter that makes types harder to read.
- **const** statements must exactly match in declarations and definitions.
- **const** is a disease. If you want to use **const Circle** objects, then:
  - the author of Circle must have correctly added **const** to every member function that doesn’t modify the Circle
  - everyone who writes methods that take Circle objects must consider whether to add **const** or not.
Operator overloading

- It is often nice when defining mathematical classes to define the behaviour of mathematical operators like *, +, – and /. This is rarely useful for non mathematical classes.
- When writing data structures, you might define []
- Many objects, even non mathematical ones like strings, can support comparison operators like < and ==.
- Most objects should be printable using <<.
- It is standard to write so called “predicates” in C++. These are objects that overload “function calling”.

To redefine the behaviour of these operators we need to use the syntax of operator overloading.
Overloading * for monomials

Here’s the declaration:

```cpp
/** Multiply two monomials */
Monomial operator*(const Monomial& x, const Monomial& y);
```

Here’s the definition:

```cpp
Monomial operator*(const Monomial& m1, const Monomial& m2) {
    int d = m1.getDimension();
    ASSERT(m2.getDimension() == d);
    vector<int> powers(d);
    for (int i = 0; i < d; i++) {
        powers[i] = m1.powers[i] + m2.powers[i];
    }
    return Monomial(powers);
}
```
Overloading * summary

Either

- Write a function called `operator*` that takes two parameters and returns the desired value

Or

- Write a `member` function called `operator*` that takes a single parameter and returns the desired value.
Overloading «

Overload « to make it easy to print your objects
- The first parameter should be a reference to an ostream.
- The second parameter should be a (const) reference to your class.
- *It should return the ostream that was passed in.*

Declaration:

```cpp
/** Write a monomial to a stream */
std::ostream& operator<<(std::ostream& out, const Monomial& monomial);
```
The only interesting point is that we return `out`
Consider

```cpp
std::cout << x << y << z;
```

This is equivalent to

```cpp
(((std::cout << x) << y) << z);
```

So each use of `<<` returns a stream that is passed onto the next use of `<<`.

- All types of output stream (e.g. writing to file, the internet, a printer) are subclasses of `ostream` so we’re using polymorphism here. You can write a monomial to any kind of stream.
- We used `<<` in the first lecture and you finally understand it.
Overloading `<`

```cpp
/** Compare two monomials using < */
bool operator<(const Monomial& x, const Monomial& y);
```

- This method returns a `bool` indicating if `x` is less than `y`.
- You might also want to overload operators such as `<=`, `>=`, `==` etc.
- Many C++ libraries only require you to overload `<` and work out the rest from that.
- We’ve overloaded `==` as an example.
Summary

We’ve written a Monomial class that looks like a standard C++ class. We’ve included

- `const` support
- Operator overloading of *, < and ==
- Printing using `<<`.

Of course, we still need to write conventional functions such as `lcm`, `quotient` and `divides`. You might consider using `/` instead of quotient but I felt monomial division was sufficiently different from standard division to make operator overloading more confusing than helpful.

Standard advice is only use operator overloading where it is obviously the best solution. It can make code hard to read if used “creatively”.
Exercises

- Implement the operators \(\leq, >\) and \(\geq\) and \(!=\) for Monomial. Make them all depend ultimately upon the definition of \(\lt\).
- Make sure you use \texttt{const} correctly in your definitions.
- Try removing some \texttt{const} keywords and seeing what compiler errors occur.
- Implement the operator \(\div\) for Monomials.
A polynomial class

There are some fairly obvious requirements of a polynomial class

- Overload the +, *, and - operators
- Overload == to test equality
- Overload « to print out nicely
- Implement reduce, lm, lc etc.

More interesting issues are:

- What data type should we use for the coefficients?
- What data structure should we use?
- Implement the += operator and -= operator
- Choosing a monomial ordering
- Implementing the = operator
The coefficients

- We will use the type

```cpp
boost::multiprecision::cpp_rational
```

for the coefficients.

- The Boost multiprecision library provides many classes for arithmetic with varying precision

- `cpp_rational` is an unlimited precision rational number

- Because it’s tedious to keep typing this long type name we create a local alias:

```cpp
typedef boost::multiprecision::cpp_rational BigRational;
```

- You can use `typedef` to create aliases for types. Don’t overuse it.
Using Boost

To use a library you must:

- Say where the header files are. These are the "include directories" under "VC++ directories".
- Say which compiled library files (lib) files to "link". Use the "Library directories" to say where the library files are to be found and Linker->Additional Dependencies to list the lib files.
- Say where dynamic link libraries are to be found and add them to the Executable Directories.
- Ensure dynamic link libraries are on the PATH when your program runs. For example they might be in the same directory as your executable or in a standard location on the PATH.

Each library will usually come with a set of instructions saying which of these steps you need to consider. For Boost one normally only needs the include directories as Boost is a “header only library”.
The data structure for Polynomial

- We will use a `map<Monomial,BigRational>`. This variable is called `coefficients`.
- In general one has a `map<KeyType,ValueType>`
- The `KeyType` must implement `<` (or else you must provide a comparison function separately)
- A `map` stores data in a tree structure so it can perform order \( \log(n) \) lookup from a key to a value
- By using a map we have a sparse data structure
- `map` also makes it easy to implement `lm` and `lc`
/**
 * Add a term to the polynomial
 */

void Polynomial::add(const BigRational& coefficient,
                     const Monomial& monomial) {
    if (coefficients.count(monomial) == 1) {
        BigRational value = coefficients[monomial];
        value += coefficient;
        coefficients[monomial] = value;
    } else {
        coefficients[monomial] = coefficient;
    }
    if (coefficients[monomial] == 0) {
        coefficients.erase(monomial);
    }
}
Map features used for add

- Use `map.count` to see how many values are stored against a given key.
- Use `map[key]` to find elements by key - you need to make sure the key really is in the map before doing this.
- Use `map[key] = ...` to insert elements.
- Use `map.erase` to remove elements with a given key.
Iterating through data

- We will also want to iterate through the data in a map
- Before we do this, let’s learn some new ways to iterate through a vector
Using `auto`

Use `auto` or `auto&` to indicate a variable or a reference variable whose type the compiler can pretty much deduce. For example you can use `auto` like this:

```cpp
void printRadius( Circle& circle ) {
    auto radius = circle.getRadius();
    std::cout << radius;
}
```

But not like this:

```cpp
void printRadius( auto& circle ) {
    auto radius = circle.getRadius();
    std::cout << radius;
}
```

Auto is new in C++11 and it makes container classes like map much easier to use.
Iterating through a vector - version I

Here’s the pre-C++11 way of “iterating” through a vector. One creates an object called an iterator which behaves like a pointer in that it overloads ++, --, *, and ==.

```cpp
void printVector( const vector<int>& vec ) {
    vector<int>::const_iterator i;
    i = vec.begin();
    while (i!=vec.end()) {
        std::cout << *i;
        i++;
    }
}
```

All "container" classes have begin and end functions which allow you to iterate through them in this way. With vectors you have the alternative of using an index i but this option isn’t available if you want to iterate through a map.
Here’s one new way of “iterating” through a vector.

```cpp
void printVector( const vector<int>& vec ) {
    auto i = vec.begin();
    while ( i!=vec.end() ) {
        std::cout << *i;
        i++;
    }
}
```

The ugliest bit of the syntax is eradicated by using `auto`.
Iterating from the beginning to the end is so common, there’s a special shortcut

```cpp
void printVector(const vector<int>& vec) {
    for (auto i : vec) {
        std::cout << i;
        i++;
    }
}
```

This is almost readable!
void printMap(map<Monomial,BigRational>& m) {
    for (auto& pair : m) {
        std::cout << "Key = " << pair.first;
        std::cout << " Value = " << pair.second << "\n";
    }
}

It’s worth knowing that the pair object is of type

pair<Monomial,BigRational>

but thanks to auto we don’t have to type that.
Note that we’re using an auto& to prevent unnecessary copying of data.
Iterating through a map pre-C++11

Just for information here is what you used to have to do. C++ code will contain a lot of this sort of thing for many years to come:

```cpp
void printMap(const map<Monomial,BigRational>& m) {
    map<Monomial,BigRational>::const_iterator i;
    i = m.begin();
    while (i!=m.end()) {
        const pair<Monomial,BigRational> p = *i;
        std::cout << "Key = " << p.first;
        std::cout << " Value = " << p.second << "\n";
    }
}
```

Note the `const` modifiers. What has happened is that the type system has taken control and made us perform some very sophisticated reasoning to simply get our code to compile.
Iteration example

```cpp
/**
  * Compute the degree, this requires running through all the entries
  */
int Polynomial::getDegree() const {
    int degree = 0;
    for (auto& entry : coefficients) {
        if (entry.first.getDegree() > degree) {
            degree = entry.first.getDegree();
        }
    }
    return degree;
}
```
Partial iteration example

```cpp
/** Returns the lead monomial */
const Monomial& Polynomial::leadMonomial() const {
    auto it = getCoefficients().end();
    it--;
    return (*it).first;
}

/** Returns the coefficient of the lead monomial */
const BigRational& Polynomial::leadMonomialCoefficient() const {
    auto& it = getCoefficients().end();
    it--;
    return (*it).second;
}
```

- `.end()` returns an iterator that points one after the last element. This is standard C++ usage.
- `--` then points to the last element.
/**
 * Multiply two polynomials
 */

Polynomial operator*(const Polynomial& p1, const Polynomial& p2) {
    // note this implementation is sub-optimal
    ASSERT(p1.getDimension() == p2.getDimension());
    Polynomial ret( p1.getDimension());
    for (auto& entry1 : p1.getCoefficients()) {
        for (auto& entry2 : p2.getCoefficients()) {
            BigRational coefficient = entry1.second * entry2.second;
            Monomial monomial = entry1.first* entry2.first;
            ret.add(coefficient, monomial);
        }
    }
    return ret;
}
Converting monomials into polynomials

- Data of one class can be automatically converted into another class if you provide a constructor that takes one parameter.

```java
Polynomial(Monomial m) {
    dimension = m.getDimension();
    coefficients[m] = 1;
}
```

This allows you to write expressions such as

```java
Monomial m = ...  
Polynomial p = ...  
Polynomial q = m * p;
```

Without having to write a separate function to multiply polynomials

- (N.B. the actual code in the project is a little more complex)
The explicit keyword

- On the other hand the constructor.

```c
explicit Polynomial(int dimension) {
    this->dimension = dimension;
}
```

is marked explicit because we don’t want to automatically convert integers to polynomials in this way. It would just be confusing.

- As a general rule, add an `explicit` keyword if you write a 1-parameter constructor. (This rule is hard to remember!)

- (N.B. the actual code in the project is a little more complex)
Overloading operator $\texttt{+=}$

```cpp
/**
 *   Add a whole polynomial
 */
void Polynomial::operator+=(const Polynomial& p) {
    for (auto& entry : p.getCoefficients()) {
        add(entry.second, entry.first);
    }
}
```

- When overloading $\texttt{+=}$ you should do this by overloading the member function.
- When overloading $\texttt{+}$ you have a choice of writing a standalone function or a member function according to taste.
- You can overload $\texttt{++}$ when it makes sense. It probably doesn’t for Polynomials.
The payoff

```cpp
Polynomial computeSPolynomial(const Polynomial& p1, const Polynomial& p2) {
    auto m1 = p1.leadMonomial();
    auto c1 = p1.leadMonomialCoefficient();
    auto m2 = p2.leadMonomial();
    auto c2 = p2.leadMonomialCoefficient();
    auto m = lcm(m1, m2);
    return (1 / c1) * m.quotient(m1) * p1 + (-1 / c2) * m.quotient(m2) * p2;
}
```

Methods such as `reduce` to perform 1-step reduction and `buchberger` to implement Buchberger’s algorithm were similarly simple to implement.
The biggest obstacle to testing is that so far polynomials are really tedious to create. You have to create lots of Monomial objects and call add repeatedly.

We would like the following test to work:

```java
static void testSPolynomial() {
    Polynomial f1 = parsePolynomial(2, "2x0^2-4x0+x1^2-4x1+3");
    Polynomial f2 = parsePolynomial(2, "x0^2-2x0+3x1^2-12x1+9");
    Polynomial S = computeSPolynomial(f1, f2);
    assert(S == BigRational(1,2)*parsePolynomial(2,"-15 + 20 x1 - 5 x1 ^ 2"));
}
```
Mini-languages and parsers

- It is quite normal to want to write a mini-language. It is a good design pattern.
- For very mini-languages its easiest to parse them using regular expressions we’ll demo this for polynomials.
- For slightly more grown up languages you can use parser generator software (Bison and Yacc are two well known parser generators).
- You might want to use a mini language so that two programs can communicate with each other.
- Note that for “data” there are already lots of standard formats such as comma delimited files and XML. These are no good for us because there is an obviously correct format for polynomials.
Regular expressions

- The regular expression D.g matches Dog and Dig
- Do*g matches Dg and Dog and Doooooog
- Do+g matches Dog and Doooooog but not Dg
- Do?g matches Dog and Dg but not Dooog
- i.e. * means 0 or more, + means 1 or more, ? means 0 or 1
- D[iu]g matches Dig and Dug but not Dog
- D[^i]g matches Dog and Dug but not Dig
- [iu] and [^i] are called character classes and represent sets of characters. The symbol ^ in this context means "not".
- \d is shorthand for the character class [0123456789]. It means digit
- \s is shorthand for white space.
Matching terms in a polynomial

- The regular expression

\s*\[+-]\?[\^+-]+

can be used to match a term in a polynomial.

- It means (Optional space)(Optional + or −)(Anything other than + or −, but not completely blank)

- We can use the `regexp` package in C++ to find all the matches to this pattern in a string and so split a polynomial into separate terms.
We don’t simply want to test whether strings match patterns.
We want to read off components of the match.
Use round brackets to surround terms you want to read.

\( \texttt{s*([+-]?([-+-]+)}\)

This means match the terms in a polynomial and report back to me the sign and the remainder of the term. Simply ignore leading space.
Polynomial parsePolynomial(int dim, const string& polynomial) {

    Polynomial poly(dim);

    regex sumRegex("\s*([-+]?)\([^+-]+\)\); // Use regex_search to find the first match in the string. Then use [0] to read off the desired entries from the match object. [0] is always the entire matched string.
    smatch match;
    string s = polynomial;
    while (regex_search(s, match, sumRegex)) {
        string sign = match[1];
        string monomialString = match[2];
        //... real code calls parseMonomial rather than prints out
        std::cout << "Sign = " << sign << "\n";
        std::cout << "Monomial = " << monomial << "\n";

        s = match.suffix().str(); // standard way to loop through regex
    }

    return poly;
}
Monomial parseMonomial( int dim, const string& monomial, int& coefficient ) {
    regex monomialRegex("\s*\d*\s*((x\d+\s*(\^\s*\d+)?\s*)*)\s*");
    smatch match;
    if (!regex_match(monomial, match, monomialRegex)) {
        std::cout << "Invalid monomial " << monomial << "\n";
        assert(false);
    }
    string coeff = match[1];
    coefficient = 1;
    if (coeff.size() > 0) {
        coefficient = stoi(coeff);
    }
    string terms = match[2];

    regex termRegex("x(\d+)\s*(\^\s*(\d+))?\s*");
    vector<int> powers(dim, 0);
    // loop through matching patterns
    string s = terms;
    while (regex_search(s, match, termRegex)) {
        string index = match[1];
        int i = stoi(index);
        assert(i >= 0 && i < dim);
        string power = match[3];
        if (power == ")" | "\n") {
            powers[i] = 1;
        } else {
            powers[i] = stoi(power);
        }
        s = match.suffix().str(); // standard way to loop through regex
    }
    return Monomial(powers);
}
Working with regexp

- The \ character must be escaped as \\ in C++ strings, so all the \ characters end up doubled up.
- Use `regex_search` to find multiple matches
- Use `regex_match` to check that an entire string matches a pattern
- Use `stoi` (string to integer) and other similar functions to perform basic parsing of numbers.
- Tip: start just making sure you can match the pattern using `regex_match` and then move on to capturing etc.
- Tip: write unit tests - I didn’t write `parsePolynomial` or `parseMonomial` all in one go.
The benefits of regexp

- The best way to get programs to communicate is through text files
- Regular expressions provide a quick and easy way to manipulate text files
- TeXstudio (my tex editor) provides search and replace based on regular expressions as do many other text editors.
- Regular expressions are available in most modern computer languages. In some languages (e.g. Perl) they are very central to the whole system
Mini-languages

- Writing a mini-language to parse polynomials isn’t that difficult (two methods) but provides huge usability benefits.
- Regular expressions are themselves written using a mini-language (see http://www.cplusplus.com/reference/regex/ECMAScript/ for more details on the syntax of this language).
- I wrote a risk management system whose key selling point was that I implemented a mini-language for executing complex queries (SQL).
- LaTeX, HTML, R, Singular, ...
Monomial ordering

- We’d like to be able to use different monomial orderings
- When you create a `std::map<Key,Value>` you can specify how it orders data by massing in an object of type `std::map<Monomial,BigRational>::key_compare`

- The class `key_compare` is defined inside `map` which is why we have the extra `::`. We saw this sort of thing with `iterator` classes too.
- We’ve created a class `MonomialOrdering` which extends this class (and makes it’s name more meaningful)
- The `map` uses the `key_compare` object polymorphically to compare objects.
The monomial ordering declaration

```cpp
/**
*   When constructing a polynomial you must choose a monomial ordering. By
*   default lexicographicOrder is selected.
*/
class MonomialOrdering : public std::map<Monomial,BigRational>::key_compare {
public:
    virtual ~MonomialOrdering() {}  
    virtual bool operator()(const Monomial& x, const Monomial& y) const = 0; 
};
```

This is an interface class. It's just that the function is called `operator()`.
A MonomialOrdering is a `std::map<Monomial,BigRational>::key_compare`
Implementing the standard monomial ordering

```cpp
class LexicographicOrder : public MonomialOrdering {
public:
    bool operator()(const Monomial& x, const Monomial& y) const {
        return x < y;
    }
};

static LexicographicOrder lex;

MonomialOrdering& lexicographicOrder() {
    return lex;
}
```

- We have a *global variable* called `lex`. This is a variable defined outside any function.
- It is marked as `static` so only the file `monomial.cpp` knows about this variable.
Calling operator()

Monomial m1({ 1, 2, 3 });
Monomial m2({ 2, 3, 4 });
MonomialOrdering& isSmaller = lexicographicOrder();
assert( isSmaller(m1, m2) );

- We can use isSmaller as though it were a function
- Objects which overload `operator()` are often called functors or function objects. Confusing if you are a category theorist.
- Function objects which return bool values are called predicates.
The constructors of polynomial

class Polynomial {
public:
    explicit Polynomial(int dimension,
                         const MonomialOrdering& ordering=lexicographicOrder()) :
        dimension(dimension),
        ordering(ordering),
        coefficients(ordering) {} 
    /**< Automatic conversion of monomials into polynomials */
    Polynomial(Monomial m,
               const MonomialOrdering& ordering = lexicographicOrder()) :
        dimension(m.getDimension()),
        ordering(ordering),
        coefficients(ordering) {
        coefficients[m] = 1;
    }
    ... 
private:
    int dimension;
    const MonomialOrdering& ordering;
    std::map<Monomial, BigRational> coefficients;
Summary

- We’ve implemented a sophisticated Polynomial class
- We’ve overloaded all kinds of operators to achieve this
- We’ve learned how to use the map data type
- We’ve learned about iteration, and auto
- We’ve learned how to write a mini-language using regular expressions
- We’ve learned how to use predicates to control the behaviour of maps
- We’ve seen how to supply default arguments
- The standard template library = templates, containers, iterators, predicates. It’s the hardest subject in C++ programming.
I wrote methods `reduce`, `buchberger` and `isEmpty` building on each in turn.

I then wrote the following test:
static void testIsEmpty() {
    {
        Polynomial f1 = parsePolynomial(2, "2x0^2-4x0+x1^2-4x1+3");
        Polynomial f2 = parsePolynomial(2, "x0^2-2x0+3x1^2-12x1+9");
        vector<Polynomial> ideal({ f1, f2 });
        assert(!isEmpty(ideal));
    }
    {
        Polynomial f1 = parsePolynomial(2, "x0 + x1 + 3");
        Polynomial f2 = parsePolynomial(2, "2 x0 + 2 x1 + 5");
        vector<Polynomial> ideal({ f1, f2 });
        assert(isEmpty(ideal));
    }
    {
        Polynomial f1 = parsePolynomial(2, "x0 + x1 + 3");
        Polynomial f2 = parsePolynomial(2, "2 x0 + 2 x1 + 6");
        vector<Polynomial> ideal({ f1, f2 });
        assert(!isEmpty(ideal));
    }
    {
        Polynomial f1 = parsePolynomial(2, "x0^2 - 2 x0 + 1 ");
        Polynomial f2 = parsePolynomial(2, "x0 - 1");
        vector<Polynomial> ideal({ f1, f2 });
        assert(!isEmpty(ideal));
    }
}
C++ Summary

Sadly we haven’t covered the whole of C++
- Inheritance
- Destructors and the rule of three
- Template programming
- Lambda functions
- Further C++11 features
- ...

For a huge list of possible books see:

I would particularly emphasize the book “Effective C++”. Ultimately programming is a craft and one learns more by doing than by reading.
Here’s a list of the most important things we’ve touched on in the course:

- Functional programming
- Rule based programming
- Procedural programming
- Object oriented programming
- Text manipulation
- Visualization
- Unit testing
- Source control
Exercise

- Write your own rational number class
  - Use regular expressions to parse a rational number
  - Implement some operators.
- Write a sparse matrix class using map.

(Of course in practice you should use libraries like Boost which provide excellent matrix and rational number classes already.)
Thanks

- Thank you for your attention.