Appendix A

Macaulay2

There are several computer algebra systems dedicated to algebraic geometry and commutative algebra computations, such as Singular (more popular among algebraic geometers), CoCoA (which is more popular with european commutative algebraists, having originated in Genova, Italy), and Macaulay2. There are many computations you could run on any of these systems (and others), but we will focus on Macaulay2 since it's the most popular computer algebra system among US based commutative algebraists.

Macaulay2, as the name suggests, is a successor of a previous computer algebra system named Macaulay. Macaulay was first developed in 1983 by Dave Bayer and Mike Stillman, and while some still use it today, the system has not been updated since its final release in 2000. In 1993, Daniel Grayson and Mike Stillman released the first version of Macaulay2, and the current stable version if Macaulay2 1.16.

Macaulay2, or M2 for short, is an open-source project, with many contributors writing packages that are then released with the newest Macaulay2 version. Journals like the *Journal of Software for Algebra and Geometry* publish peer-refereed short articles that describe and explain the functionality of new packages, with the package source code being peer reviewed as well.

The National Science Foundation has funded Macaulay2 since 1992. Besides funding the project through direct grants, the NSF has also funded several Macaulay2 workshops — conferences where Macaulay2 package developers gather to work on new packages, and to share updates to the Macaulay2 core code and recent packages.

A.1 Getting started

A Macaulay2 session often starts with defining some ambient ring we will be doing computations over. Common rings such as the rationals and the integers can be defined using the commands `QQ` and `ZZ`; one can easily take quotients or build polynomial rings (in finitely many variables) over these. For example,

```
i1 : R = ZZ/101[x,y]

o1 = R
```

```
o1 : PolynomialRing
```

and

```
i1 : k = ZZ/101;

i2 : R = k[x,y];
```

both store the ring $\mathbb{Z}/101$ as R, with the small difference that in the second example Macaulay2 has named the coefficient field k. One quirk that might make a difference later is that if we use the first option and later set k to be the field $\mathbb{Z}/101$, our ring R is *not* a polynomial ring over k. Also, in the second example we ended each line with a `;`, which tells Macaulay2 to run the command but not display the result of the computation — which is in this case was simply an assignment, so the result is not relevant.

We can now do all sorts of computations over our ring R. For example, we can define an ideal in R, as follows:

```
i3 : I = ideal(x^2,y^2,x*y)

              2   2
o3 = ideal (x , y , x*y)

o3 : Ideal of R
```

Above we have set I to be the ideal in R that is generated by x^2, y^2, xy. The notation `ideal( )` requires the usage of `^` for powers and $*$ for products; alternatively, we can define the exact same ideal with the notation `ideal" "`, as follows:

```
i3 : I = ideal"x2,y2,xy"

              2   2
o3 = ideal (x , y , x*y)

o3 : Ideal of R
```

Now we can use this ideal I to either define a quotient ring $S = R/I$ or the R-module $M = R/I$, as follows:

```
i4 : M = R^1/I

o4 = cokernel | x2 y2 xy |
                          1
o4 : R-module, quotient of R

i5 : S = R/I

o5 = S

o5 : QuotientRing
```

It's important to note that while R is a ring, R^1 is the R-module R — this is a very important difference for Macaulay2, since these two objects have different types. So S defined above is a ring, while M is a module. Notice that Macaulay2 stored the module M as the cokernel of the map

$$R^3 \xrightarrow{\begin{bmatrix} x^2 & y^2 & xy \end{bmatrix}} R\,.$$

When you make a new definition in Macaulay2, you might want to pay attention to what ring your new object is defined over. For example, now that we defined this ring S, Macaulay2 has automatically taken S to be our current ambient ring, and any calculation or definition we run next will be considered over S and not R. If you want to return to the original ring R, you must first run the command `use R`.

If you want to work over a finitely generated algebra over one of the basic rings you can define in Macaulay2, and your ring is not a quotient of a polynomial ring, you want to rewrite this algebra as a quotient of a polynomial ring. For example, suppose you want to work over the second Veronese in 2 variables over our field k from before, meaning the algebra $k[x^2, xy, y^2]$. We need 3 algebra generators, which we will call a, b, c, corresponding to x^2, xy, and y^2:

```
i6 : U = k[a,b,c]

o6 = U

o6 : PolynomialRing

i7 : f = map(R,U,{x^2,x*y,y^2})
                 2        2
o7 = map(R,U,{x , x*y, y })

o7 : RingMap R <--- U

i8 : J = ker f
             2
o8 = ideal(b  - a*c)

o8 : Ideal of U

i9 : T = U/J

o9 = T

o9 : QuotientRing
```

Our ring T at the end is isomorphic to the 2nd Veronese of R, which is the ring we wanted. Note the syntax order in `map`: first target, then source, then a list with the images of each algebra generator.

A.2 Asking Macaulay2 for help

As you're learning how to use Macaulay2, you will often find yourself needing some help. Luckily, Macaulay2 can help you directly! For example, suppose you know the name of a command, but do not remember the syntax to use it. You can ask `?command`, and Macaulay2 will show you the different usages of the command you want to know about.

```
i10 : ?primaryDecomposition

primaryDecomposition -- irredundant primary decomposition of an ideal

  * Usage:
        primaryDecomposition I
  * Inputs:
      * I, an ideal, in a (quotient of a) polynomial ring R
  * Optional inputs:
      * MinimalGenerators => a Boolean value, default value true, if false, the
        components will not be minimalized
      * Strategy => ..., default value null,
  * Outputs:
      * a list, containing a minimal list of primary ideals whose intersection
        is I

Ways to use primaryDecomposition :
==================================

  * "primaryDecomposition(Ideal)" -- see "primaryDecomposition" -- irredundant
    primary decomposition of an ideal
  * "primaryDecomposition(Module)" -- irredundant primary decomposition of a
    module
  * "primaryDecomposition(Ring)" -- see "primaryDecomposition(Module)" --
    irredundant primary decomposition of a module

For the programmer
==================

The object "primaryDecomposition" is a method function with options.
```

If instead you'd rather read the complete Macaulay2 documentation on the command you are interested in, you can use the `viewHelp` command, which will open an html page with the documentation you asked for. So running

```
i11 : viewHelp "primaryDecomposition"
```

will open an html page dedicate to the method `primaryDecomposition`, which includes examples and links to related methods.

A.3 Basic commands

Many Macaulay2 commands are easy to guess, and named exactly what you would expect them to be named. Often, googling "Macaulay2" followed by a few descriptive words will easily land you on the documentation for whatever you are trying to do.

Here are some basic commands you will likely use:

- `ideal`$(f_1, \ldots, f_n)$ will return the ideal generated by $f_1, \ldots, f_n$. Here products should be indicated by $*$, and powers with $\widehat{}$. If you'd rather not use $\widehat{}$ (this might be nice if you have lots of powers), you can write `ideal`$(f_1, \ldots, f_n)$ instead.
- `map`$(S, R, f_1, \ldots, f_n)$ gives a ring map $R \to S$ if R and S are rings, and R is a quotient of $k[x_1, \ldots, x_n]$. The resulting ring map will send $x_i \mapsto f_i$. There are many variations of `map` — for example, you can use it to define R-module homomorphisms — but you should carefully input the information in the required format. Try `viewHelp map` in Macaulay2 for more details
- `ker`(f) returns the kernel of the map f.
- `I + J` and `I*J` return the sum and product of the ideals I and J, respectively.
- `A = matrix`$\{\{a_{1,1}, \ldots, a_{1,n}\}, \ldots, \{a_{m,1}, \ldots, a_{m,n}\}\}$ returns the matrix

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ & \ddots & \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$$

If you are familiar with any other programming language, many of the basics are still the same. For example, some of the commands we will use return lists, and we might often need to do operations on lists. As with many other programming languages, a list is indicated by { } with the elements separated by commas.

```
i6 : w = {ZZ, 3, ideal"xy3"}
                          3
o6 = {ZZ, 3, ideal(x*y )}

o6 : List
```

As in most programming languages, Macaulay2 follows the convention that the first position in a list is the 0th position.

The method `primaryDecomposition` returns a list of primary ideals whose intersection is the input ideal, and `associatedPrimes` returns the list of associated primes of the given ideal or module. Operations on lists are often intuitive. For example, let's say we want to find the primary component of an ideal with a particular radical.

```
i1 : R = QQ[x,y];

i2 : I = ideal"x2,xy";

o2 : Ideal of R

i3 : prim = primaryDecomposition I
                              2
o3 = {ideal x, ideal (y, x )}

o3 : List

i4 : L = select(prim, Q -> radical(Q) == ideal"x,y")
                   2
o4 = {ideal (y, x )}

o4 : List
```

The method `select` returns a list of all the elements in our list with the required properties. In this case, if we actually want the primary ideal we just selected, as opposed to a list containing it, we need to extract the first component of our list L.

```
i5 : L_0
                  2
o5 = ideal (y, x )

o5 : Ideal of R
```

A.4 Graded rings

Polynomial rings in Macaulay2 are graded with the standard grading by default, meaning that all the variables have degree 1. To define a different grading, we give Macaulay2 a list with the grading of each of the variables:

```
i1 : R = ZZ/101[a,b,c,Degrees=>{{1,2},{2,1},{1,0}}];
```

We can check whether an element of R `isHomogeneous`, and the function `degree` applied to an element of R returns the least upper bound of the degrees of its monomials:

```
i2 : degree (a+b)
o2 = {2, 2}
o2 : List

i3 : isHomogeneous(a+b)
o3 = false
```

A.5 Complexes and homology in Macaulay2

There are two different ways to do computations involving complexes in Macaulay2: using `ChainComplexes`, or the new (and still under construction) `Complexes` package. To use `Complexes`, you must first load the `Complexes` package, while the `ChainComplexes` methods are automatically loaded with Macaulay2.

A.5.1 Chain Complexes

To create a new chain complex by hand, we start by setting up R-module maps.

```
i1 : R = QQ[a,b];

i2 : d1 = map(R^1, R^2, {{a,b}})

o2 = | a b |
             1       2
o2 : Matrix R  <--- R

i3 : d2 = map(R^2, R^1, {{-b},{a}})

o3 = | -b |
     | a  |
             2       1
o3 : Matrix R  <--- R
```

Keep in mind that the syntax of map is a bit funny: we write `map(target,source,matrix)`. To make sure we set up the next map in a way that is composable with d_1, we can use the methods `source` and `target`:

```
i3 : d1 = map(source d0, R^1, {{-b},{a}})

o3 = | -b |
     | a  |
             2       1
o3 : Matrix R  <--- R
```

We can also double check our maps do indeed map a complex, by checking the composition $d_1 \circ d_2$:

```
i4 : d1 * d2 == 0

o4 = true
```

So now we are ready to set up our new chain complex.

```
i5 : C = new ChainComplex

o5 = 0

o5 : ChainComplex

i6 : C#0 = target d1

      1
o6 = R

o6 : R-module, free

i7 : C#1 = target d2

      2
o7 = R

o7 : R-module, free

i8 : C#2 = source d2

      1
o8 = R

o8 : R-module, free
```

Given a chain complex `C`, we can ask Macaulay2 what our complex is by simply running the name of the complex:

```
i9 : C

      1      2      1
o9 = R  <-- R  <-- R

     0      1      2

o9 : ChainComplex
```

Or we can ask for a better visual description of the maps, using `C.dd`:

```
i10 : C.dd

            1         2
o10 = 0 : R  <----- R  : 1
                0
```

```
           2         1
      1 : R  <----- R  : 2
                0

o10 : ChainComplexMap
```

We can also set up the same complex in a more compact way, by simply feeding the maps we want in order. Macaulay2 will automatically place the first map with the target in homological degree 0 and the source in degree 1.

```
11 : D = chainComplex(d1,d2)

       1      2      1
o11 = R  <-- R  <-- R

      0      1      2

o11 : ChainComplex
```

Notice this is indeed the same complex.

```
i12 : D.dd

           1               2
o12 = 0 : R  <----------- R  : 1
                | a b |

           2              1
      1 : R  <---------- R  : 2
                | -b |
                | a  |

o12 : ChainComplexMap
```

We can also ask Macaulay2 to compute the homology of our complex:

```
i13 : HH D

o13 = 0 : cokernel | a b |

      1 : subquotient (| b  |, | -b |)
                      | -a |  | a  |
      2 : image 0

o13 : GradedModule
```

Or we could simply ask for the homology in a specific degree:

```
i14 : HH_0 D

o14 = cokernel | a b |
                                1
o14 : R-module, quotient of R
```

A.5.2 The Complexes package

To use this functionality, you must first load the `Complexes` package.

```
i15 : needsPackage "Complexes";

o15 = Complexes

o15 : Package
```

We can use our maps from above to set up a complex with the same maps. We feed a list of the maps we want to use to the method `complex`.

```
i16 : F = complex({d1,d2})

       1      2      1
o16 = R  <-- R  <-- R

      0      1      2

o16 : Complex
```

We cam read off the maps and the homology in our complex using the same commands as we use with `chainComplexes`, although the information returned gets presented in a slightly different fashion.

```
i17 : HH F

o17 = cokernel | a b | <-- subquotient (| b  |, | -b |) <-- image 0
                                        | -a |  | a  |
      0                                                          2
                              1

o17 : Complex

i18 : F.dd

           1               2
```

```
o18 = 0 : R  <----------- R  : 1
                | a b |

           2              1
      1 : R  <---------- R  : 2
                | -b |
                | a  |

o18 : ComplexMap
```

If we want to set up our complex starting in a different homological degree, we can do the following:

```
i19 : G = complex({d1,d2}, Base => 7)

       1      2      1
o19 = R  <-- R  <-- R

      7      8      9

o19 : Complex

i20 : H = complex({d1,d2}, Base => -13)

       1       2       1
o20 = R   <-- R   <-- R

      -13     -12     -11

o20 : Complex
```

A.5.3 Maps of complexes

Suppose we are given two complexes `C` and `D` and a map of complexes `f : C` $\longrightarrow$ `D`. The routine `map` can be used to define f using `chainComplexes`: it receives the target `D`, the source `D`, and a function `f` that returns f_i when we compute `f(i)`.

```
i1 : R = QQ[a,b];

i2 : c1 = map(R^0,R^1,0);

                    1
o2 : Matrix 0 <--- R

i3 : c2 = map(R^1, R^2, {{a,b}});
```

```
             1       2
o3 : Matrix R  <--- R

i4 : c3 = map(R^2, R^1, {{-b},{a}});

             2       1
o4 : Matrix R  <--- R

i5 : c4 = map(R^1, R^0, 0);

             1
o5 : Matrix R  <--- 0

i6 : C = chainComplex(c1,c2,c3,c4);

i7 :
     d1 = map(R^0,R^1,0);

                     1
o7 : Matrix 0 <--- R

i8 : d2 = id_(R^1);

             1       1
o8 : Matrix R  <--- R

i9 : d3 = map(R^1, R^0, 0);

             1
o9 : Matrix R  <--- 0

i10 : d4 = map(R^0, R^0, 0);

o10 : Matrix 0 <--- 0

i11 : D = chainComplex(d1,d2,d3,d4)

             1      1
o11 = 0 <-- R  <-- R  <-- 0 <-- 0

      0     1      2      3     4

o11 : ChainComplex

i12 :
```

```
      f0 = map(R^0, R^0, 0);

o12 : Matrix 0 <--- 0

i13 : f1 = map(R^1, R^1, matrix{{0_R}});

              1       1
o13 : Matrix R  <--- R

i14 : f2 = map(R^2, R^1, {{b},{-a}});

              2       1
o14 : Matrix R  <--- R

i15 : f3 = map(R^1, R^0, 0);

              1
o15 : Matrix R  <--- 0

i16 : f4 = map(R^0, R^0, 0);

o16 : Matrix 0 <--- 0

i17 : f = map(C,D,i -> if i==0 then f0 else(
      if i==1 then f1 else (
      if i==2 then f2 else (
      if i == 3 then f3 else (
      if i==4 then f4)))))

o17 = 0 : 0 <----- 0 : 0
               0

           1         1
      1 : R  <----- R  : 1
                0

           2              1
      2 : R  <---------- R  : 2
                | b  |
                | -a |
           1
      3 : R  <----- 0 : 3
                0

      4 : 0 <----- 0 : 4
```

```
              0

o17 : ChainComplexMap
```

Here's what we can do if we prefer to write a list with the maps in `f`:

```
i18 : f = map(C,D,i -> {f0,f1,f2,f3,f4}_i)

o18 = 0 : 0 <----- 0 : 0
              0

           1         1
      1 : R  <----- R  : 1
                0

           2              1
      2 : R  <---------- R  : 2
                | b  |
                | -a |

           1
      3 : R  <----- 0 : 3
                0

      4 : 0 <----- 0 : 4
              0

o18 : ChainComplexMap
```

If we prefer to do the same with the `Complexes` package, one advantage is that `map` *does* receive `(target, source, list of maps)`.

```
i42 : C = complex({c1,c2,c3,c4});

i43 : D = complex({d1,d2,d3,d4});

i44 : f = map(C,D,{f0,f1,f2,f3,f4})

           2              1
o44 = 2 : R  <---------- R  : 2
                | b  |
                | -a |

o44 : ComplexMap
```