AN INTRODUCTION TO MACAULAY2

ABRAHAM MARTÍN DEL CAMPO

This document gives a basic introduction to `Macaulay2`, an open source software for algebraic computation. You can download it and find installation instructions in their website: `http://www2.macaulay2.com/Macaulay2/`.

1. First steps

Once `Macaulay2` is initiated, you can input any command and execute it by pressing enter.

```
i1 : 5^2
o1 = 25
```

In `Macaulay2`, every object has a class (a data type) so that the software knows how to deal with it. `Macaulay2` normally displays the type of the output value on a second labeled output line, except for the simplest types (e.g. integers and Boolean values). For instance, any letter is understood as a symbol unless it has been assigned some value.

```
i2 : a
o2 = a
o2 : Symbol
```

Assignment is done by =, and the value of an object is displayed by typing its name. Assignment can also be done in parallel. Two dashes -- are used by `Macaulay2` to indicate a comment and any text following them is disregarded when executing the line.

```
i3 : a = 2*100 -- this is a comment
o3 = 200
i4 : a
o4 = 200
i5 : (a,a') = (3^3, 3/4)
           3
o5 = (27, -)
           4
o5 : Sequence
i6 : a
o6 = 27
i7 : a'
     3
o7 = -
     4
o7 : QQ
```

Date: February 2, 2018.

Comparison is done by the symbols == and !=, returning `true` or `false`. Note that comparison can only be done between comparable elements, otherwise we get an error message.

```
i8 : a == a'
o8 = false
i9 : a != a'
o9 = true
i10 : a == w
stdio:11:3:(3): error: no method for binary operator == applied to
   objects:
--            27 (of class ZZ)
--     ==     w (of class Symbol)
```

If you would like to suppress the printing environment in a statement but keeping the result, then the statement should be terminated by `;`

```
i11 : 5!;
```

The last output can always be accessed by typing two o's as in `oo`. This is particularly useful when working on Macaulay2 directly from the terminal in an interactive computation, and you forgot to store the result in a variable.

```
i12 : oo
o12 = 120
```

The values of the lines before the last one can also be accessed by typing `ooo` and `oooo`. Alternatively, the symbol labelling an output line can be used to retrieve the value; for instance, if we want to access the output of line 11 (i.e. 5 factorial), we would type `o11` as in the following example.

```
i13 : 3/5 - 7/11

         2
o13 = - --
        55
o13 : QQ

i14 : o11 + 3
o14 = 123
i15 : factorial = oooo
o15 = 120
```

Strings are delimited by quotation marks `"`, but you can also convert any value into a string with the function `toString`.

```
i16 : text = "how are you doing?"
o16 = how are you doing?
i17 : nonumber = toString(o11+3)
o17 = 123
```

In the last example, we stored the number `123` (coming from adding 3 to the output in line 11) in the variable `nonumber`, but considering it as a string and not as a number. Thus, it can be concatenated horizontally with other strings using `|`, or vertically using `||`.

```
i18 : s = "Hi everyone"
o18 = Hi everyone
i19 : s | " - " | text | nonumber
o19 = Hi everyone - how are you doing? 123
i20 : s || " - " || text
o20 = Hi everyone
       -
      how are you doing?
```

The best way to learn `Macaulay2` is to read the online documentation either through their web site or through the local copy included in the installation folder. You can access this local copy by typing `help`.

```
i21 : help matrix
o21 = matrix -- make a matrix
      **********************
      Synopsis
      ========
        * Optional inputs:
            * Degree => ...,  -- create a matrix from a doubly-nested
   list of ring elements or matrices

      Description
      ===========
   ...
```

You could display the same information in your web browser instead of the terminal window using the command `viewHelp` instead.

```
i22 : viewHelp matrix
```

The command `apropos` helps you find a function you may be searching but you forgot parts of its name. The output will be a list with all functions that included the searched part included in their name.

```
i23 : apropos "dimension"
o23 = {AllCodimensions, CodimensionLimit}
o23 : List
i24 : apropos "dim"
o24 = {AllCodimensions, codim, CodimensionLimit, dim, pdim, RadicalCodim1}
o24 : List
```

A list of expressions can be formed with braces. The number of elements is obtained by placing the symbol # before the list.

```
i25 : L = {1, 2, s}
o25 = {1, 2, Hi everyone}
o25 : List
i26 : #L
o26 = 3
```

The elements in a list are internally numbered by integer numbers starting from 0. You can access an element by placing the symbol # after the list followed by the element number.

```
i27 : L#0
o27 = 1
```

`Macaulay2` creates a matrix from a nested list of lists, which the softwares interprets as a list of rows, each of which is in turn a list of ring elements. Once you declare a matrix, you can use basic matrix operations.

```
i28 : M = matrix {{1,2,3},{4,5,6},{7,8,0}}
o28 = | 1 2 3 |
      | 4 5 6 |
      | 7 8 0 |
               3        3
o28 : Matrix ZZ  <--- ZZ

i29 : M + 3*M
o29 = | 4  8  12 |
      | 16 20 24 |
      | 28 32 0  |
               3        3
o29 : Matrix ZZ  <--- ZZ

i30 : M^2
o30 = | 30 36 15 |
      | 66 81 42 |
      | 39 54 69 |
               3        3
o30 : Matrix ZZ  <--- ZZ

i31 : trace M
o31 = 6

i32 : transpose M
o32 = | 1 4 7 |
      | 2 5 8 |
      | 3 6 0 |
               3        3
o32 : Matrix ZZ  <--- ZZ
```

```
i33 : det M
o33 = 27
```

You can see the entries of the matrix specifying the coordinates. However, you cannot modify the values of an entry unless you work with a mutable matrix.

```
i34 : M_(0,1) -- this is the entry in first row and second column
o34 = 2

i35 : M_(0,1) = 0
stdio:38:9:(3): error: no method for assignment to binary operator _
   applied to objects:
--            | 1 2 3 | (of class Matrix)
--            | 4 5 6 |
--            | 7 8 0 |
--      _     (0, 1) (of class Sequence)

i36 : M = mutableMatrix M
o36 = | 1 2 3 |
      | 4 5 6 |
      | 7 8 . |
o36 : MutableMatrix

i37 : M_(0,1) = 0
o37 = 0

i38 : M
o38 = | 1 . 3 |
      | 4 5 6 |
      | 7 8 . |
o38 : MutableMatrix
```

If you want to protect the information of the matrix again, you can pass from mutable to regular matrix

```
i39 : M = matrix M
o39 = | 1 0 3 |
      | 4 5 6 |
      | 7 8 0 |
               3        3
o39 : Matrix ZZ  <--- ZZ
```

You may notice that `Macaulay2` interprets matrices not only as a list of lists, but also as a map between modules. We need to take this in consideration when operating with matrices. For instance, the matrix `M` above has integer entries; thus, it is regarded as a map between

$\mathbb{Z}$-modules and so, the elements of its inverse should live in the fraction field otherwise, it regards `M` as not invertible.

```
i40 : inverse M
stdio:43:1:(3): error: matrix not invertible

i41 : promote(M, QQ)
o41 = | 1 0 3 |
      | 4 5 6 |
      | 7 8 0 |
               3        3
o41 : Matrix QQ  <--- QQ

i42 : inverse oo
o42 = | 16/19  -8/19 5/19  |
      | -14/19 7/19  -2/19 |
      | 1/19   8/57  -5/57 |
               3        3
o42 : Matrix QQ  <--- QQ
```

2. Basic Programming in M2

We now explain a few basic things about the programming tools in `Macaulay2`. Let us start by creating a matrix entry by entry using a `for` loop. We will begin by declaring a mutable integer matrix of size 3×3.

```
i1 : M = mutableMatrix(ZZ, 3,3)
o1 = 0
o1 : MutableMatrix

i2 : for i from 0 to 2 list
        for j from 0 to 2 list
           M_(i, j) = 3*i+j
o2 = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}}
o2 : List
```

The matrix defined is regarded as a mutable list, but we can declare it a matrix as well.

```
i3 : M
o3 = | . 1 2 |
     | 3 4 5 |
     | 6 7 8 |
o3 : MutableMatrix

i4 : matrix M
o4 = | 0 1 2 |
     | 3 4 5 |
```

```
     | 6 7 8 |

              3        3
o4 : Matrix ZZ  <--- ZZ
```

In the previous code, the output of each `for` loop is a list, but we could had written `do` instead to return just the output and discard it afterwards.

```
i5 : SQ = for i from 1 to 4 list i^2
o5 = {1, 4, 9, 16}
o5 : List
i6 : #SQ
o6 = 4

i7 : SQ2 = for i from 1 to 4 do i^2
i8 : #SQ2
stdio:10:1:(3): error: expected a list, sequence, hash table, or string
```

A shorter way to type `for` loops in `Macaulay2` is with the functions `apply` and `scan`, that starts from a list and a function to specify what to do with each element of the list.

```
i9 : Sq = apply({1,2,3,4}, i -> i^2)
o9 = {1, 4, 9, 16}
o9 : List
i10 : #Sq
o10 = 4

i11 : Sq2 = scan({1,2,3,4}, i -> i^2)
i12 : #Sq2
stdio:14:1:(3): error: expected a list, sequence, hash table, or string

i13 : SQ == Sq
o13 = true

i14 : SQ2 == Sq2
o14 = true
```

The operator `->` above indicates that we are creating a function, in this case, on the elements of the list. Our next example will be a function to compute the factorial $n!$ of an integer n.

```
i15 : factrl = n -> (
          prod := 1; -- initializing the output variable
          for i from 1 to n do
              prod = i*prod;
          return prod;
          )
o15 = factrl
o15 : FunctionClosure
```

```
i16 : factrl(5)
o16 = 120

i17 : factrl(-3)
o17 = 1
```

The second line in our function declares a local variable `prod` by initializing it using a colon before the equal sign `:=`. This is so that `Macaulay2` knows only to use the values of `prod` within a function call to `factrl` and nowhere else in your code.

Another feature of our function is that it does not check for the negativity of our input. We will fix it with the use of a conditional statement.

```
i18 : factorial = n -> (
          if n < 0 then return "n must be positive";
          prod := 1;
          scan(n, i-> prod = (i+1)*prod);
          return prod;
          )
o18 = factorial
o18 : FunctionClosure

i19 : factorial(5)
o19 = 120

i20 : factorial(-3)
o20 = n must be positive
```

We could had used a `while` loop to implement $n!$.

```
i21 : factorial = n -> (
          if n < 0 then return "n must be positive";
          count := 1;
          prod := 1;
          while count <= n do (
                prod = count * prod;
                count = count + 1;
                );
          return prod
          )

o21 = factorial
o21 : FunctionClosure

i22 : factorial(5)
o22 = 120
```

```
i23 : factorial(-3)
o23 = n must be positive
```

We show now a recursive implementation of $n!$.

```
i24 : factorial = n -> (
          if n == 1 then return 1;
          return(n*factorial(n-1))
          )
o24 = factorial
o24 : FunctionClosure

i25 : factorial 5
o25 = 120
```

Finally, we mention a few useful combinatorial functions that we could use for indexing over permutations and combinations.

```
i26 : subsets {1,2,3}
o26 = {{}, {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}}
o26 : List

i27 : subsets({1,2,3},2)
o27 = {{1, 2}, {1, 3}, {2, 3}}
o27 : List

i28 : permutations {a,b,c}
o28 = {{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a}}
o28 : List

i29 : permutations 3
o29 = {{0, 1, 2}, {0, 2, 1}, {1, 0, 2}, {1, 2, 0}, {2, 0, 1}, {2, 1, 0}}
o29 : List

i30 : partitions 3
o30 = {Partition{3}, Partition{2, 1}, Partition{1, 1, 1}}
o30 : List

i31 : partitions(4,2)
o31 = {Partition{2, 2}, Partition{2, 1, 1}, Partition{1, 1, 1, 1}}
o31 : List
```

3. Rings and Gröbner bases

To calculate with objects such as ideals and polynomials, a polynomial ring has to be defined first. An advantage is that `Macaulay2` uses mathematical notation to define polynomial

rings. When typing a polynomial, the software automatically consider it as a ring element, and the ring is displayed in the second output line.

```
i43 : R = QQ[x,y,z]
o43 = R
o43 : PolynomialRing

i44 : f = (x+y)^3
       3     2         2    3
o44 = x  + 3x y + 3x*y  + y
o44 : R
```

The definition of a polynomial ring consist of a ground ring (e.g. `ZZ`, `QQ`, `RR`, or `CC`), and a list (or sequence) of variables. `Macaulay2` works also with fields of positive characteristic and with specific monomial orderings.

```
i45 : R1=ZZ/32003[x,y,z]
o45 = R1
o45 : PolynomialRing

i46 : R2 = QQ[x..z, MonomialOrder=> Lex]
o46 = R2
o46 : PolynomialRing
```

In the example above, `R1` is a ring over a ring or field of characteristic 32003 and the variables x, y, z, while `R2` is a ring over the rationals $\mathbb{Q}$ also in the variables x, y, z. A nice feature is that we were able to add the variables a sequence starting from x and ending in z. This is particularly useful when working with many variables.

Another difference between these two rings is the monomial order. The ring `R2` is specified to use the Lexicographic order, while `R1` uses the default in `Macaulay2` which is the graded reverse lexicographic. The list of posible monomial orders can be accessed by

```
i47 : viewHelp MonomialOrder
```

After declaring a ring, `Macaulay2` has some built in functions to retrieve useful information about it. The original description of the ring can be recovered with `describe`. The number of variables is provided by the command `numgens`, while the command `gens` provides the list of variables of the ring. You could also use `vars` to obtain the variables not as a list but as a matrix (with one row).

```
i48 : describe R2
o48 = QQ[x..z, Degrees => {3:1}, Heft => {1},
                    MonomialOrder => {MonomialSize => 32}, DegreeRank => 1]
                    {Lex => 3}
                    {Position => Up}

i49 : numgens R2
o49 = 3
```

```
i50 : gens R2
o50 = {x, y, z}
o50 : List

i51 : vars R2
o51 = | x y z |
              1       3
o51 : Matrix R  <--- R
```

Defining a ring makes it the current working ring, so each time we define a ring we switch to a new ring. If you write a polynomial, it will automatically be regarded as an element of the last ring. You can switch back to a previously defined ring with the command `use`. Working with multiple rings is described carefully in the `Macaulay2` documentation in the section *substitution and maps between rings*.

```
i52 : f = x^3+y^3+(x-y)*x^2*y^2+z^2
       3 2    3    2 3    3    2
o52 = x y  + x  - x y  + y  + z
o52 : R2

i53 : use R
o53 = R
o53 : PolynomialRing

i54 : f = x^3+y^3+(x-y)*x^2*y^2+z^2
       3 2    2 3    3    3    2
o54 = x y  - x y  + x  + y  + z
o54 : R
```

Acknowledgment

This tutorial relies on some introductory notes written for `Singular` and `Maple` by Luis Garcia-Puente and Chris Hillar respectively, the book "Computations in Algebraic Geometry with Macaulay 2" (eds. Eisenbud, Grayson, Stillman, and Sturmfels,) and the introductory part of the Macaulay2 documentation.

Centro de Investigación en Matemáticas, A.C., Jalisco s/n, col. Valenciana, Guanajuato, 36023 Guanajuato, México

E-mail address: `abraham.mc@cimat.mx`

URL: `http://www.cimat.mx/~abraham.mc`