FASTLINALG PACKAGE FOR *MACAULAY2*

BOYANA MARTINOVA, MARCUS ROBINSON, KARL SCHWEDE, AND YUHUI YAO

ABSTRACT. In this article, we present `FastLinAlg.m2`, a package in *Macaulay2* designed to introduce new methods focused on computations in function field linear algebra. Some key functionality that our package offers includes: finding a submatrix of a given rank in a provided matrix (when present), verifying that a ring is regular in codimension n, recursively computing the ideals of minors in a matrix, and finding an upper bound of the projective dimension of a module.

1. INTRODUCTION

We start with some motivation. Suppose that $I = (f_1, \ldots, f_m) \subseteq k[x_1, \ldots, x_n]$ is a prime ideal. The corresponding variety $X := V(I)$ is singular if and only if I plus the ideal generated by the determinants of the $n - \dim X$ submatrices of the Jacobian matrix

$$\mathrm{Jac}(X) = \left(\begin{array}{c} \frac{\partial f_i}{\partial x_j} \end{array} \right),$$

generate the unit ideal. Unfortunately, even for relatively small values of m and n, the number of such submatrices is prohibitive. Suppose for instance that $n = 10, m = 15$ and $\dim X = 5$. Then there are

$$\binom{10}{5} \cdot \binom{15}{5} = 756756$$

such submatrices.

We cannot reasonably compute all of their determinants (this computation is often time intensive). This package attempts to fix this in several ways.

(a) We try to compute just a portion of the determinants, in a relatively smart way.
(b) We offer some tools for computing determinants which are sometimes faster.

Of course, computing the singular locus is not the only potential application. This technique has been applied to the related problem of showing that the singular locus has a certain codimension (for example, checking that a variety is R1 in order to prove normality). We provide a function for giving a better upper bound on projective dimension of a non-homogeneous module. Finally, this package has also been applied in the `RationalMaps` Macaulay2 package.

We provide the following functions:

○ `getSubmatrixOfRank`, this tries to find a submatrix of a given rank, Section 3.
○ `isRankAtLeast`, this uses `getSubmatrixOfRank` to try to find lower bounds for the rank of a matrix, Section 4.
○ `Rn`, tries to verify if an integral domain is regular in dimension n, Section 5.
○ `projDim`, tries to find upper bounds for the projective dimension of a non-homogeneous module, Section 6.

- `recursiveMinors`, computes the ideal of minors of a matrix via a recursive cofactor algorithm, as opposed to the included non-recursive cofactor algorithm, Section 7.

The latest version of this package is available at:

`https://github.com/kschwede/M2/blob/master/M2/Macaulay2/packages/FastLinAlg.m2`

2. Finding interesting submatrices

A lot of the speedups available in the package come down to finding interesting square submatrices of a given matrix. For example, it is often useful to compute a square submatrix whose determinant has small degree as this is less likely to vanish.

2.1. **How are the submatrices chosen?** Consider the following matrix defined over $\mathbb{Q}[x, y]$.

$$\begin{bmatrix} x & xy & 0 \\ xy^2 & x^6 & 0 \\ 0 & x^2y^3 & xy^4 \end{bmatrix}$$

Suppose we want to choose a submatrix of size 2×2. Consider the monomial order `Lex` where $x < y$. We find, in the matrix, the nonzero element of smallest order. In this case, that is x. We choose this element to be in part of our submatrix.

$$\begin{bmatrix} \mathbf{x} & xy & 0 \\ xy^2 & x^6 & 0 \\ 0 & x^2y^3 & xy^4 \end{bmatrix}$$

Hence we discard that row and column containing this term and find that the next smallest element with respect to our monomial order is xy^4.

$$\begin{bmatrix} x^6 & 0 \\ x^2y^3 & xy^4 \end{bmatrix} \qquad \begin{bmatrix} x^6 & 0 \\ x^2y^3 & \mathbf{xy^4} \end{bmatrix}$$

Since we are only looking for a 2×2 submatrix, we stop here. We have selected the submatrix with rows 0 and 2 and columns 0 and 2.

$$\begin{bmatrix} x & 0 \\ 0 & xy^4 \end{bmatrix}$$

The determinant of which is x^2y^4. This happens to be the smallest 2×2 minor with respect to the given monomial order (which frequently happens, although it is certainly not always the case).

If we chose a different monomial order, we get a different submatrix, with a different determinant. For example,

Lex, $x > y$: We yield the submatrix with rows 0 and 1 and columns 0 and 1, whose determinant is $x^7 - xy^3$

GRevLex, $x < y$: We yield the submatrix with rows 0 and 2 and columns 0 and 1, whose determinant is x^3y^3

For any of these strategies, in this package we randomize the order of the variables before choosing a submatrix.

The strategies we implement in this package are a bit more complicated, however. If we have a matrix whose entries are not monomial, then we could reasonably either pick the submatrix of smallest entries with respect to our monomial order (`LexSmallest` or `GrevLexSmallest`), or the submatrix whose entries have the smallest terms (`LexSmallestTerm` or `GRevLexSmallestTerm`). Both of these strategies are implemented.

For example, consider the matrix

$$\begin{bmatrix} x^2 + y^2 & 0 & xy + 2x \\ y^4 - x & 0 & 3x^5 \\ x^3 & x^4y^5 - y^8 & 0 \end{bmatrix}$$

In this case, if we are choosing the entries with smallest terms, we first replace each entry in the matrix with the smallest term. For example, if we are using Lex with $x < y$ we would obtain:

$$\begin{bmatrix} x^2 & 0 & 2x \\ -x & 0 & 3x^5 \\ x^3 & x^4y^5 & 0 \end{bmatrix}$$

Then we proceed as before. Notice that if there is a tie, it is broken randomly. We also have the strategies GRevLexLargest and LexLargest available to the user.

Finally, we have two random strategies available. Random and RandomNonzero.

Random: With this strategy, a random submatrix is chosen.

RandomNonzero: With this strategy, a random nonzero element is chosen in each step following the method used by the other strategies. This guarantees a submatrix where no row or column is zero which can be very useful when dealing with relatively sparse matrices.

Remark 2.1. Different strategies work differently on different examples. When you have a non-homogeneous matrix, with some entries that have constant terms, those entries will always be chosen first when picking smallest terms, regardless of the monomial order. On the other hand, for homogeneous matrices, choosing the smallest term is frequently very effective.

2.1.1. *Modifying the underlying matrix when using* GRevLex*. Finally, when using GRevLex* orders, we periodically change the underlying matrix by replacing terms of small order with terms of larger order in order to avoid computing the same submatrix. For example, in the following matrix, after a couple iterations, we might replace the x^2 term with

$$x^2 \cdot (\text{a random degree 1 polynomial}).$$

It might look something like the following.

$$\begin{bmatrix} x^2 & 0 & xy \\ y^4 & 0 & x^5 \\ x^3 & x^4y^5 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} x^2(2x - 7y) & 0 & xy \\ y^4 & 0 & x^5 \\ x^3 & x^4y^5 & 0 \end{bmatrix}.$$

This forces the algorithm to make different choices to be made. After several iterates are done, the matrix is reset again to its original form.

2.2. **The strategy options.** The core features included in the package allow the user to choose which strategy should be used when selecting submatrices. This is done by setting a Strategy option to one of the following.

○ StrategyDefault: This strategy uses LexSmallest, LexSmallestTerm, GRevLexSmallest, GRevLexSmallestTerm, Random, and RandomNonzero with equal probability.

○ StrategyDefaultNonRandom: This uses LexSmallest, LexSmallestTerm, GRevLexSmallest, GRevLexSmallestTerm, Random, and RandomNonzero with equal probability.

○ StrategyLexSmallest: chooses 50% of the submatrices using LexSmallest and 50% using LexSmallestTerm.

○ StrategyGRevLexSmallest: chooses 50% of the submatrices using GRevLexSmallest and 50% using GRevLexLargest.

○ StrategyRandom: chooses submatrices by using 50% Random and 50% RandonNonzero.

The user can create their own custom strategy. One creates a `HashTable` which has the keys `LexLargest`, `LexSmallestTerm`, `LexSmallest`, `GRevLexSmallestTerm`, `GrevLexSmallest`, `GRevLexLargest`, `Random`, `RandomNonzero` each with value an integer (the values need not sum to 100). If one value is twice the size of another, that strategy will be employed twice as often. For example, `StrategyDefaultNonRandom` was created by the command:

```
StrategyDefaultNonRandom = new HashTable from {
    LexLargest => 0,
    LexSmallestTerm => 25,
    LexSmallest => 25,
    GRevLexSmallestTerm => 25,
    GRevLexSmallest => 25,
    GRevLexLargest => 0,
    Random => 0,
    RandomNonzero => 0
};
```

3. FIND A SUBMATRIX OF A GIVEN RANK: `getSubmatrixOfRank`

This method examines the submatrices of an input matrix and attempts to find one of a given rank. If a submatrix with the specified rank is found, a list of two lists is returned. The first is the list of rows, the second is the list of columns, which describe the desired submatrix of desired rank. If no such submatrix is found, the function will return `null`.

The option `MaxMinors` allows the user to control how many minors to consider. If left `null`, the number considered is based on the size of the matrix. This method will choose the indicated amount of minors using one of the strategy options described below. If one of the chosen submatrices has the desired rank, the function will terminate and return its rows and columns. This process continues until a submatrix is found or `MaxMinors` submatrices have been unsuccessfully checked. The strategy can be controlled using the `Strategy` option as described above, the default value is `StrategyDefaultNonRandom`.

3.1. **Examples of `getSubmatrixOfRank`.** In the following example, we first create a 3×4 rational matrix, M. We execute two calls to `getSubmatrixOfRank`, the first has no Strategy parameter and the second utilizes `StrategyGRevLexSmallest`. Note that these calls return different indices, but both find valid rank 3 submatrices. We then create a larger 9×10 matrix over the 3-dimensional space of integers modulo 103. We display the time needed for the `rank` function to return, followed by the time elapsed during a call to `getSubmatrixOfRank` when searching for a rank 7 submatrix. We repeat these calculations on a new matrix with the same parameters, execting the call to `getSubmatrixOfRank` first, where we search for a rank 9 submatrix, and then calling `rank`. In both trials we found that `getSubmatrixOfRank` significantly outperformed `rank`.

```
i1 : loadPackage "FastLinAlg";

i2 : R = QQ[x,y];

i3 : M = random(R^{2,2,2},R^4)

o3 = {-2} | x2+2/3xy+9/2y2     3/10x2+2/3xy+1/5y2 2x2+5/3xy+7/5y2     4/3x2+1/3xy+10/9y2 |
     {-2} | 3/2x2+2/3xy+2y2    1/2x2+3/2xy+3/4y2  6x2+5xy+4y2         9/5x2+1/5xy+7/2y2  |
     {-2} | 1/4x2+1/7xy+5/6y2 7/5x2+4xy+4/5y2     10/9x2+3/7xy+5/9y2 5/2x2+xy+7/6y2      |
              3         4
o3 : Matrix R  <--- R
```

```
i4 : getSubmatrixOfRank(3,M)

o4 = {{2, 0, 1}, {0, 1, 3}}

o4 : List

i5 : getSubmatrixOfRank(3, M, Strategy=>StrategyGRevLexSmallest)

o5 = {{0, 2, 1}, {1, 2, 0}}

o5 : List

i6 : Q = ZZ/103[x,y,z];

i7 : N = random(Q^{7,7,7,7,7,7,7,8,8},Q^10);
                9        10
o7 : Matrix Q  <--- Q

i8 : elapsedTime rank N;
     -- 17.701 seconds elapsed

i9 : elapsedTime getSubmatrixOfRank(7,N);
     -- 0.0373561 seconds elapsed

i10 : O = random(Q^{7,7,7,7,7,7,7,8,8},Q^10);
                9        10
o10 : Matrix Q  <--- Q

i11 : elapsedTime getSubmatrixOfRank(9,O);
-- 0.76357 seconds elapsed

i12 : elapsedTime rank O;
      -- 14.6581 seconds elapsed
```

In one of the core examples from the `RationalMaps` package, before using this package a function would look at several thousands of submatrices (randomly) typically before finding a submatrix of the desired rank whereas this package finds one after looking at fewer than half a dozen. Using this package sped up the computation of that example by more than one order of magnitude, see [BHSS19, Page 7], the non-maximal linear rank example.

4. Finding lower bounds for matrix ranks: `isRankAtLeast`

This method is a direct implementation of `getSubmatrixOfRank`. This function returns a boolean indicating whether the rank of an input matrix, M, is greater than or equal to an input integer, n. In order to do so, the function first performs some basic checks to ensure a rank of n is possible given M's dimensions, then executes a call to `getSubmatrixOfRank`. If `getSubmatrixOfRank` returns a matrix, then this function will return true. However, if `getSubmatrixOfRank` does not return a matrix, a conclusive answer can not be reached. As such, the method will then evaluate the rank of M and return the appropriate boolean value.

`isRankAtLeast` is efficient when `getSubmatrixOfRank` returns quickly, however may be costly if the results are inconclusive and a rank evaluation is necessary. As such, the described implementation is not optimized. In order to lead to time improvements, we developed a multithreaded version

of this function that concurrrently evaluates the rank of M and invokes `getSubmatrixOfRank`. Once a thread has terminated, the method cancels the other and returns the appropriate value. During the implementation of this functionality, we discovered that *Macaulay2* becomes unstable when cancelling threads and thus do currently not allow users to invoke the multithreaded version. However, this functionality is included in the package and can be made easily accessible once the stability issue is resolved.

4.1. **Example of `isRankAtLeast`.** The following example first creates and displays a smaller 3×3 matrix, M. We call `isRankAtLeast` to determine if its rank is at least 3. We then create a larger 9×9 matrix, N, and call `isRankAtLeast` to determine if its rank is at least 7. Directly calling `rank N` on a matrix of this size would take multiple seconds, whereas `isRankAtLeast` returns in a fraction of the time.

```
i1 : loadPackage "FastLinAlg";

i2 : R = QQ[x,y,z];

i3 : M = random(R^{2,2,2},R^3)

o3 = {-2} | 7/2x2+2/3xy+y2+2/9xz+7/10yz+3/5z2     x2+5xy+1/2y2+2xz+7/9yz+7/5z2
     {-2} | 8/7x2+1/10xy+y2+2/3xz+3yz+1/2z2        3x2+7/8xy+1/9y2+3xz+10yz+5z2
     {-2} | 1/10x2+5/2xy+3/4y2+2/3xz+3/2yz+10/7z2 3x2+10/7xy+7/2y2+7/10xz+2yz+1/4z2
     --------------------------------------------------------------------------------
     6/7x2+9/7xy+3/10y2+2/5xz+2/5yz+5/4z2 |
     2/9x2+5/4xy+3y2+1/2xz+yz+7/8z2       |
     8/9x2+3/4xy+y2+10xz+9yz+4/5z2        |
              3      3
o3 : Matrix R  <--- R

i4 : isRankAtLeast(3,M)

o4 = true

ii5 : rank M

oo5 = 3

i6 : N = random(R^{6,6,6,6,6,6,6,7,7},R^9);
              9       9
o6 : Matrix R  <--- R

i7 : elapsedTime isRankAtLeast(7,N)
-- 0.0654172 seconds elapsed

o7 = true
```

5. Regular in codimension n: `Rn`

Using the `getSubmatrixOfRank` routines, we provide a function for checking if a variety is regular in codimension n, or Rn. The default strategy is `Strategy=>Default`.

The function `Rn(ZZ, Ring)` returns `true` if it verifies that the ring is regular in codimension n. Note, this only works if the ring is equidimensional, as it is using a Jacobian criterion. If it cannot

make a determination, it returns `null`. If it ended up computing all minors of the matrix, and it still doesn't have the desired codimension, it will return `false` (note this will likely only occur for small matrices).

5.1. **Example of** `Rn`. We begin with an example of a 3 dimensional ring that is regular in codimension 1, but not in codimension 2. It is generated by 12 equations in 7 variables.

```
i3 :  T = ZZ/101[x1,x2,x3,x4,x5,x6,x7];

i4 :  I =  ideal(x5*x6-x4*x7,x1*x6-x2*x7,x5^2-x1*x7,x4*x5-x2*x7,x4^2-x2*x6,x1*x4-x2*x5,
x2*x3^3*x5+3*x2*x3^2*x7+8*x2^2*x5+3*x3*x4*x7-8*x4*x7+x6*x7,x1*x3^3*x5+3*x1*x3^2*x7
+8*x1*x2*x5+3*x3*x5*x7-8*x5*x7+x7^2,x2*x3^3*x4+3*x2*x3^2*x6+8*x2^2*x4+3*x3*x4*x6
-8*x4*x6+x6^2,x2^2*x3^3+3*x2*x3^2*x4+8*x2^3+3*x2*x3*x6-8*x2*x6+x4*x6,x1*x2*x3^3
+3*x2*x3^2*x5+8*x1*x2^2+3*x2*x3*x7-8*x2*x7+x4*x7,x1^2*x3^3+3*x1*x3^2*x5+8*x1^2*x2
+3*x1*x3*x7-8*x1*x7+x5*x7);

o4 : Ideal of T

i5 : S = T/I; dim S

o6 = 3

i7 : time Rn(1, S)
     -- used 0.150734 seconds

o7 = true

i8 : time Rn(2, S)
     -- used 2.12777 seconds

i9 : time singularLocus S;
     -- used 8.29746 seconds

i10 : time dim o9
-- used 23.2483 seconds

o10 = 1
```

You can see that the function `Rn` verified that S was regular in codimension 1 in a fraction of a second. When we ran `Rn(2, S)` nothing was returned, indicating that nothing was found. Computing the jacobian ideal however took more than 8 seconds and verifying that it had dimension 1 took more than 23 seconds.

5.2. **The** `Strategy` **option.** Let us look in slightly more detail at the same example but this time using some different strategies. For instance, you might think that it might be just as effective to choose random matrices, and sometimes it is.

```
i11 :  time Rn(1, S, Strategy=>StrategyRandom, Verbose=>true)
  Rn: ring dimension =3, there are 17325 possible minors, we will compute up to 324 of them.
  Rn: About to enter loop
  Rn:  Loop step, about to compute dimension.  Submatrices considered: 7, and computed = 7
  Rn:  partial singular locus dimension computed, = 2
  Rn:  Loop step, about to compute dimension.  Submatrices considered: 9, and computed = 9
  Rn:  partial singular locus dimension computed, = 2
  Rn:  Loop step, about to compute dimension.  Submatrices considered: 11, and computed = 11
  Rn:  partial singular locus dimension computed, = 2
```

```
Rn:  Loop step, about to compute dimension.  Submatrices considered: 14, and computed = 14
Rn:  partial singular locus dimension computed, = 2
Rn:  Loop step, about to compute dimension.  Submatrices considered: 18, and computed = 18
Rn:  partial singular locus dimension computed, = 2
Rn:  Loop step, about to compute dimension.  Submatrices considered: 24, and computed = 24
Rn:  partial singular locus dimension computed, = 2
Rn:  Loop step, about to compute dimension.  Submatrices considered: 31, and computed = 31
Rn:  partial singular locus dimension computed, = 2
Rn:  Loop step, about to compute dimension.  Submatrices considered: 40, and computed = 40
Rn:  partial singular locus dimension computed, = 2
Rn:  Loop step, about to compute dimension.  Submatrices considered: 52, and computed = 52
Rn:  partial singular locus dimension computed, = 2
Rn:  Loop step, about to compute dimension.  Submatrices considered: 67, and computed = 67
Rn:  partial singular locus dimension computed, = 2
Rn:  Loop step, about to compute dimension.  Submatrices considered: 87, and computed = 87
Rn:  partial singular locus dimension computed, = 1
Rn:  Loop completed, submatrices considered = 87, and computed = 87.
     singular locus dimension appears to be = 1
     -- used 0.758961 seconds

o11 = true
```

You can see that the `StrategyRandom` option looked at 87 submatrices in this particular example. Note it does not check to see if we have obtained the desired outcome with each submatrix considered, it does this periodically with the space between checks increasing. The `considered` values on each line tells how many submatrices have been selected. Computed tells how many were not repeats (this will be very low with a random strategy).

Running `Rn(1, S, Strategy=>StrategyRandom, Verbose=>true)` 50 times yielded:

  (a) 61.3 average number of submatrices matrices considered.
  (b) a median value of 40 or 52 sumatrices considered.
  (c) a minimum value of 7 submatrices considered (one time).
  (d) a maximum value of 248 submatrices considered (one time).

Note 7 is the minimum number of submatrices that will be considered when checking regular in codimension 1 for this variety (the minimum number depends on the dimension).

On the other hand, the default strategy `Rn(1, S, Strategy=>StrategyDefaultNonRandom, Verbose=>true)` run 50 times yields

  (a) 12.1 average number of submatrices considered.
  (b) a median value of 7 or 9 submatrices considered.
  (c) a minimum value of 7 submatices considered (25 times).
  (d) a maximum value of 40 submatrices considered (one time).

In this particular example, `Strategy=>StrategyLexSmallest` appears to be optimal. Note that larger matrices tend to create even larger disparities.

5.3. **Notes on implementation.** As mentioned above, this function computes minors (based on the passed `Strategy`) option until either it finds that the singular locus has the desired dimension, or until it has considered too many minors. By default, it considers up to:

$$10 \cdot (\text{Minimum number of minors needed}) + 8 \cdot \log_{1.3}(\text{possible minors}).$$

This was simply chosen by experimentation. Note that if you are trying to show your singular locus has a certain codimension, you will need a minimum number of minors. The 10 multiplying it is because our default strategy uses multiple strategies, but only one might work well on a given

matrix. You can provide an alternate function of $x =$ (minors needed) and $y =$ (possible minors) passing it to the option `MaxMinors`. You can also pass `MaxMinors` a number.

These matrices are considered in a loop. We begin with computing a constant number of minors, by default $2 \cdot$ (Minimum number of minors needed) $+ 3$ and check whether the output has the right dimension. You can provide a different function of $x =$ (minors needed) via the option `MinMinorsFunction`. After that, we compute additional minors, checking periodically (based on an exponential function, 1.3^k minors considered before the next reset) whether our minors define a subset of the desired desired codimension. You can provide your own function via the option `CodimCheckFunction`. If in this loop, a submatrix is considered again, it is not recomputed, but the counter is still increased.

5.4. **Other options.** This function also includes other options including the option `ModP` which handles switching the coefficient field for a field of characteristic $p > 0$ (which you specify with `ModP => p.`)

One can also control how determinants are computed with the `DetStrategy` option, valid values are `Bareiss`, `Cofactor` and `Recursive`.

6. PROJECTIVE DIMENSION: `projDim`

In April of 2019, it was pointed out in a thread on github

https://github.com/Macaulay2/M2/issues/936

that the command `pdim` sometimes provides an incorrect value (an overestimate) for projective dimension for non-homogeneous modules over polynomial rings. There it was also suggested that this could be addressed by looking at appropriate minors of the matrices in a possibly non-minimal resolution, but that in practice these matrices have too many minors to compute. We have implemented a function `projDim` that tries to address this by looking at only *some* minors. Our function does not solve the problem as it also gives only an upper bound on the projective dimension. However, this upper bound is more frequently correct.

The idea is as follows. Take a free resolution of a module M over a polynomial ring R

$$0 \longleftarrow M \longleftarrow F_0 \xleftarrow{d_1} F_1 \longleftarrow \ldots \xleftarrow{d_{n-1}} F_{n-1} \xleftarrow{d_n} F_n \longleftarrow 0.$$

Each d_i is given by a matrix. The term F_n is unnecessary (ie, d_n splits) happens exactly when the rank F_n minors of d_n generate the unit ideal. In that case, we know or projective dimension is at most $n-1$. However, we can continue in this way, we can compute the (rank $F_{n-1} -$ rank F_n)-minors of d_{n-1}, and see if they generate the unit ideal. Our algorithm of course only computes a subset of those minors.

6.1. **Example of `projDim`.** In the below example, we take a monomial ideal of projective dimension 2, compute a non-homogeneous change of coordinates, and observe that `pdim` returns an incorrect answer that `projDim` corrects.

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

i2 : I = ideal(x^4,x*y,w^3, y^4);

i3 : pdim module I

o3 = 2

i4 : f = map(R, R, {x+x^2+1, x+y+1, z+z^4+x-2, w+w^5+y+1});

i5 : pdim module f I
```

```
o5 = 3

i6 : time projDim module f I
     -- used 3.43851 seconds

o6 = 2

i7 : time projDim(module f I, MinDimension=>2)
     -- used 0.0503165 seconds

o7 = 2
```

6.2. **Options.** As you can see in the previous example, setting `MinDimension` will can substantially speed up the computation as it won't try to determine if the projective dimension is actually 1.

The option `MaxMinors` can either be set to be a number (the number of a minors computed at each step). Or it can be set to be a list of numbers (one for each step in the above algorithm). Finally, it can be set to be a function of the dimension d of the polynomial ring R and the number t of possible minors. This is the default option, and the function is: $5 * d + 2 * \log_{1.3}(t)$. The option `Strategy` is also available and it works as above with the default value being `StrategyDefault`.

7. COMPUTING IDEALS OF MINORS: `recursiveMinors`

Macaulay2 is a unique because it contains a `minors` method that returns the ideal of minors of a certain size, n, in a given matrix, a necessary step in locating singularities. However, the current implementation's default is to evaluate determinants using the Bareiss algorithm, which is efficient when the entries in the matrix have a low degree and few variables, but very slow otherwise. The current minors method also allows users to compute determinants using cofactor expansion, but this strategy performs some unnecessary calculations, causing it to be quite costly as well. We improved the current cofactor expansion method to find the determinants of minors by adding recursion and multithreading throughout. We also eliminated said unnecessary calculations by ensuring that only the required determinants are being computed at each step of the recursion, rather than all possible determinants of the given size.

In order to do so, we programmed a method in *Macaulay2*'s software that recursively finds all $n \times n$ minors by first computing the 2×2 minors and storing them in a hash table. Then we use the 2×2 minors to compute the necessary 3×3 minors, and so forth. This process is repeated recursively until the minors of size $n \times n$ are evaluated. At each step, we only compute the determinants that will be needed when performing a cofactor expansion on the following size minor.

To allow for further time improvements, we also utilized *Macaulay2*'s existing parallel programming methods to multithread our code so different computations at each step of the recursion can occur simultaneously in separate threads. We divide the list of all determinants to be evaluated into different available threads and wait for them to finish before consolidating the results in a hash table and proceeding with the recursion. In order to more effectively utilize *Macaulay2*'s multithreading methods, we also created a nanosleep method that waits a given number of nanoseconds, rather than full seconds. This function has already been incorporated into the software.

7.1. **Example of `recursiveMinors`.** Below, we first create a simple matrix, M, of one dimensional polynomials with rational coefficients and execute the `recursiveMinors` method to find the ideal of all 3×3 minors. As can be seen, the result is equivalent to the output of the minors method when called with the same parameters. We then create a new, larger matrix, N, with two dimensional rational coefficients and return the computation time for `recursiveMinors` and minors utilizing

both the Bareiss and Cofactor strategies. The `recursiveMinors` method finished executing approximately six times faster than the Bareiss algorithm and almost seven times faster than the Cofactor expansion, while yielding the same results.

```
i1 : loadPackage "FastLinAlg";

i2 : allowableThreads = 8

i3 : R = QQ[x];

i4 : M = random(R^{2,2,2}, R^4)

o4 = {-2} | x2     3x2    5/8x2 7/10x2 |
     {-2} | 3/4x2 2x2    7/4x2 9x2     |
     {-2} | x2     2/9x2 1/2x2 4/3x2   |
              3       4
o4 : Matrix R   <--- R

i5 : recursiveMinors(3,M)

            1403 6   449 6      292 6   517 6
o5 = ideal (----x , ---x , - ---x , ---x )
             60      240         45    144

o5 : Ideal of R

i6 : recursiveMinors(3,M) == minors(3,M)

o6 = true

i7 : Q = QQ[x,y];

i8 : N = random(Q^{5,5,5,5,5,5}, Q^7);
              6       7
o8 : Matrix Q   <--- Q

i9 : elapsedTime minors(5,N, Strategy => Bareiss);
-- 3.0094 seconds elapsed

o9 : Ideal of Q

i10 : elapsedTime minors(5,N, Strategy => Cofactor);
-- 3.76846 seconds elapsed

o10 : Ideal of Q

i11 : elapsedTime recursiveMinors(5,N);
-- 0.590152 seconds elapsed;

o11 : Ideal of Q

i12 : recursiveMinors(5,N) == minors(5,N)

o12 = true
```

References

[BHSS19] C. J. Bott, S. H. Hassanzadeh, K. Schwede, and D. Smolkin: *Rationalmaps, a package for macaulay2*, arXiv:1908.04337V1.

Email address: u1056124@utah.edu

Email address: robinson@math.utah.edu

Email address: schwede@math.utah.edu

Email address: ywei018@gmail.com

Department of Mathematics, University of Utah, 155 S 1400 E Room 233, Salt Lake City, UT, 84112