RANDOMRATIONALPOINTS PACKAGE FOR *MACAULAY2*

SANKHANEEL BISUI, ZHAN JIANG, SARASIJ MAITRA, THAI NGUYEN, AND KARL SCHWEDE

Abstract. In this article, we present `RandomRationalPoints`, a package in *Macaulay2* designed mainly to identify rational and geometric points in a variety over a finite field. We provide different strategies like linear intersection and projection to obtain such points. We also present methods to obtain non-vanishing minors of some given size in a given matrix, by evaluating the matrix at a point. This, in conjunction with the `FastLinAlg` package, should aid in determining properties of the singular locus.

1. Introduction

Let I be an ideal in a polynomial ring $k[x_1, \dots, x_n]$ over a finite field k. Let $X := V(I)$ denote the corresponding set of rational points in affine n-space. Finding one such rational point or geometric point (geometric meaning a point in some finite field extension), in an algorithmically efficient manner is our primary motivation.

There is an existing package called `RandomPoints`, which we took inspiration from, which aims to find *all* the rational points of a variety; our aim here is to find one point quickly, even if it is not rational. We provide some functions that apply different strategies to generate random rational points for the given variety. We also provide functions that will expedite the process of determining properties of the singular locus of X when combined with the functions provided in the package `FastLinAlg`.

We provide the following core functions:

- `projectionToHypersurface` and `genericProjection`: These functions provide customizable projection. (Section 3)
- `randomPoints`: This tries to find a point in the vanishing set of an ideal. (Section 4)
- `findANonZeroMinor`: This finds a submatrix of a given matrix that is nonsingular at a point of a given ideal. (Section 5.1)
- `extendIdealByNonZeroMinor`: This extends an ideal by a minor produced by `findANonZeroMinor`. (Section 5.2)

All polynomial rings considered here will be over finite fields. We first briefly describe some helper functions that will be used extensively in the core functions above. In the subsequent sections, we explain briefly the core functions. We shall also briefly mention the strategies that we have implemented in the execution of both helper and core functions.

Acknowledgements: The authors would like to thank David Eisenbud and Mike Stillman for useful conversations and comments on the development of this package. The authors began work on this package at the virtual Cleveland 2020 Macaulay2 workshop.

Date: December 21, 2020.

Key words and phrases. RandomRationalPoints, Macaulay2.

2. A helper function: randomCoordinateChange

`randomCoordinateChange`: This function takes a polynomial ring as an input and outputs the coordinate change map. i.e. given a polynomial ring, this will produce a linear automorphism of the ring. This function checks whether the map is an isomorphism by computing the Jacobian.

In some applications, a full change of coordinates is not desired, as it might cause code to run very slowly. A binomial change of coordinates might be preferred, or we might only replace some monomials by other monomials. Refinements of `randomCoordinateChange` are controlled with the following options.

- `Replacement`: Setting `Replacement => Full` will mean that coordinates are replaced by a general degree 1 form. If `Replacement => Binomial`, the coordinates are only changed to binomials, which can be much faster for certain applications. If `Homogeneous => false`, then there will be constant terms, and we view $mx + b$ as a binomial.
- `MaxCoordinatesToReplace`: The user can specify that only a specified number of coordinates should be non-monomial (assuming `Homogeneous => true`). This option is passed to `randomCoordinateChange` by other functions that call it.
- `Homogeneous`: Setting `Homogeneous => false` will cause degree zero terms to be added to modified coordinates (including monomial coordinates).

Example 2.1.

```
i3 : R = ZZ/11[x,y,z];

i4 : randomCoordinateChange(R)

             ZZ
o4 = map(R,--[x, y, z],{4x + 5y - 5z, 3x - 4y - 3z, 4x})
             11

i5 : matrix randomCoordinateChange(R, MaxCoordinatesToReplace=>1)

o5 = | x -x-4y-5z y |

i6 : matrix randomCoordinateChange(R, MaxCoordinatesToReplace=>1, Homogeneous=>false)

o6 = | x-3 z-5 -x+3y-4z+2 |

i7 : matrix randomCoordinateChange(R, MaxCoordinatesToReplace=>1, Replacement=>Binomial)

o7 = | y x+4z z |
```

3. genericProjection, projectionToHypersurface

Both of these functions provide customizable projection techniques. We describe them here.

3.1. genericProjection.

This function finds a random (somewhat, depending on options) generic projection of the ring or ideal. The typical usages are as follows:

- `genericProjection(n, I)`,
- `genericProjection(n, R)`,
- `genericProjection(I)`,
- `genericProjection(R)`

where I is an ideal in a polynomial ring, R can denote a quotient of a polynomial ring and $n \in \mathbb{Z}$ is an integer specifying how many dimensions to drop. Note that this function makes no attempt to verify that the projection is actually generic with respect to the ideal.

This gives the projection map from $\mathbb{A}^N \mapsto \mathbb{A}^{N-n}$ and the defining ideal of the projection of $V(I)$. If no integer n is provided then it acts as if $n = 1$.

Example 3.1.

```
i1 : R=ZZ/5[x,y,z,w];

i2 : I = ideal(x,y^2,w^3+x^2);

i3 : genericProjection(2,I)

            ZZ                                      2          2
o6 = (map(R,--[z, w],{- x - 2y - z, - y - 2z}), ideal(z  - z*w - w ))
            5

o6 : Sequence
```

Alternately, instead of I, we may pass it a quotient ring. It will then return the inclusion of the generic projection ring into the given ring, followed by the source of that inclusion. It is essentially the same functionality as calling `genericProjection(n, ideal R)` although the format of the output is slightly different.

Example 3.2.

```
i1 : R=ZZ/5[x,y,z,w];

i2 : I = ideal(x,y^2,w^3+x^2);

i3 : genericProjection(2,R/I)

                             ZZ                         ZZ
                             --[z, w]                   --[z, w]
               R             5                          5
o3 = (map(----------------,--------,{2y - w, - y - z}), --------)
             2   3    2        4                           4
        (x, y , w  + x )      z                           z

o3 : Sequence
```

This method works by calling `randomCoordinateChange` (see Section 2) before dropping some variables. It passes the options `Replacement`, `MaxCoordinatesToReplace`, `Homogeneous` to that function.

The user can also specify the same options as before to be passed on to the `randomCoordinateChange` function.

Example 3.3.

```
i2 : R=ZZ/5[x,y,z,w];

i3 : I = ideal(x,y^2,w^3+x^2);

o3 : Ideal of R

i4 :genericProjection(2,I, Replacement=>Binomial)

           ZZ                                 4
o4 = (map(R,--[z, w],{x + 2z, - 2y + w}), ideal w )
           5

o4 : Sequence
```

3.2. `projectionToHypersurface`. This function creates a projection to a hypersurface. The typical usages are as follows:

– `projectionToHypersurface I`,
– `projectionToHypersurface R`

where I is an ideal in a polynomial ring, R is a quotient of a polynomial ring. The output is a list with two entries, the generic projection map, and the ideal if `I` was provided, or the ring if `R` was provided.

It differs from `genericProjection(codim I - 1, I)` as it only tries to find a hypersurface equation that vanishes along the projection, instead of finding one that vanishes exactly at the projection. This can be faster, and can be useful for finding points. If we already know the codimension is `c`, we can set `Codimension=>c` so the function does not compute it.

Example 3.4. `i2 : R=ZZ/5[x,y,z];`

```
i3 : I = ideal(random(3,R)-2, random(2,R))

               3       2     3             2         2       2       2
o3 = ideal (- x  - 2x*y  - 2y  - x*y*z + y z - 2x*z  + 2y*z  - 2, x  -
-----------------------------------------------------------------------
        2
2x*y - y  + x*z - y*z)

o3 : Ideal of R

i4 : projectionToHypersurface(I)

           ZZ                             4    3      2 2     4
o4 = (map(R,--[y, z],{x + 2y, 2y}), ideal(y  + y z - 2y z  - 2z  - y -
           5
-----------------------------------------------------------------------
z))

o4 : Sequence
```

```
i5 : projectionToHypersurface(R/I)


                                                     R
o5 = (map(-------------------------------------------------------------
    3       2     3             2        2       2       2
(- x  - 2x*y  - 2y  - x*y*z + y z - 2x*z  + 2y*z  - 2, x  - 2x
------------------------------------------------------------------
                                    ZZ
                                    --[y, z]
                                    5
--------------------,---------------------------------------------
      2              4 2    3 3       5     6     3       2     3
*y - y  + x*z - y*z) y z  + y z  - 2y*z  + 2z  + 2y  + 2y*z  + 2z
------------------------------------------------------------------
                            ZZ
                            --[y, z]
                            5
---,{x - z, y}), -------------------------------------------------)
                  4 2    3 3       5     6     3       2     3
- 1              y z  + y z  - 2y*z  + 2z  + 2y  + 2y*z  + 2z  - 1

o5 : Sequence
```

4. randomPoints

`randomPoints` is a function to find rational or geometric points in a variety. The typical usages are as follows:

– `randomPoints(I)`,
– `randomPoints(n, I)`

where n is a positive integer denoting the number of desired points, and I is an ideal inside a polynomial ring. If n is omitted, then it is assumed to be 1.

4.1. **Options.** The user may also choose to provide some additional information depending on the context which may help in faster computations, or whether a point is found at all.

`Strategy => •`: Here the • can be `Default`, `BruteForce`, `LinearIntersection`, `GenericProjection` or `HybridProjectionIntersection`.

- `Default` performs a sequence of the different strategies below, aimed at finding a point quickly.
- `BruteForce` simply tries random points and sees if they are on the variety.
- `GenericProjection` projects to a hypersurface, via `projectionToHypersurface` and then uses a `BruteForce` strategy.
- `LinearIntersection` intersects with an appropriately random linear space.
- `HybridProjectionIntersection` does a generic projection, followed by a linear intersection.

Notice that the speed, or success, varies depending on the strategy.

Example 4.1.

```
i2 : R = ZZ/101[x,y,z];

i3 : J = ideal(x^3+y^2+1,z^3-x^2-y^2+2);

o3 : Ideal of R

i4 : randomPoints(J)
-- used 0.0181549 seconds

o4 = {{32, 37, -30}}

o4 : List

i5 : time randomPoints(J,Strategy=>BruteForce)
-- used 0.0146932 seconds

o5 = {}

o5 : List

i6 : randomPoints(J,Strategy=>GenericProjection)
-- used 0.0146932 seconds

o6 = {{2, -30, -8}}

o6 : List
```

`ProjectionAttempts => ZZ:` When calling the Strategy `GenericProjection` or `HybridProjectionIntersection` from `randomPoints`, this option denotes the number of trials before giving up. This option is also passed to `randomPoints` by other functions.

`MaxCoordinatesToReplace => ZZ:` (see Section 2)

`Codimension => ZZ:` (see Section 2)

`ExtendField => Boolean:` Intersection with a general linear space will naturally find scheme theoretic points that are not rational over the base field. Setting `ExtendField => true` will tell the function that such points are valid. Setting `ExtendField => false` will tell the function ignore such points. This sometimes can slow computation, and other times can substantially speed it up when the variety has few rational points. In some cases, points over extended fields may also have better randomness properties for applications.

`IntersectionAttempts => ZZ:` This option is used by `randomPoints` in some strategies to determine the maximum number of attempts to intersect with a linear space when looking for random rational points.

`PointCheckAttempts => ZZ:` When calling `randomPoints`, and functions that call it, with a `BruteForce` strategy or `GenericProjection` strategy, this denotes the number of trials for brute force point checking.

Example 4.2. We re-compute Example 4.1 this time specifying more attempts.

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

o1 : PolynomialRing

i2 : J = ideal "x3+y2+1,z3-x2-y2+2";

o2 : Ideal of R

i3 : randomPoints(J,Strategy=>BruteForce,PointCheckAttempts=>20000)
-- used 0.84679 seconds

o3 = {{-39, -43, 28}}

o3 : List
```

`MaxCoordinatesToTrivialize`: When calling `randomPoints` and performing an intersection with a linear space, this is the number of defining equations of the linear space of the form $x_i - a_i$. Having a large number of these will provide faster intersections.

`NumThreadsToUse => ZZ`: When calling `randomPoints`, and functions that call it, with a `BruteForce` strategy, this denotes the number of threads to use in brute force point checking.

4.2. **Comments on performance.** When working over very small fields, frequently `BruteForce` is most efficient. This is not surprising as there may not be many points to check, and a surprisingly large percentage of points can be rational in hypersurfaces and complete intersections. However, if the field size is larger, `BruteForce` will perform poorly. Other strategies work differently on different examples, and even the same strategy can sometimes work very quickly even if it typically works very slowly.

Example 4.3.

```
i2 : R = ZZ/7[x_1..x_10]

o2 = R

o2 : PolynomialRing

i3 : I = ideal(random(2, R), random(3,R));

o3 : Ideal of R

i4 : time randomPoints(I,Strategy=>BruteForce, PointCheckAttempts=>20000)
 -- used 0.0102075 seconds

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

o4 : List

i5 : time randomPoints(I,Strategy=>Default)
```

```
 -- used 0.085741 seconds

o5 = {{1, -2, 2, 2, -2, 0, 0, -1, -1, 3}}

o5 : List

i6 : S = ZZ/211[x_1..x_10]

o6 = S

o6 : PolynomialRing

i7 : J = ideal(random(2, S), random(3, S));

o7 : Ideal of S

i8 : time randomPoints(J,Strategy=>BruteForce, PointCheckAttempts=>2000000)
-- used 25.3479 seconds

o8 = {{-36, 26, -63, -7, -48, 78, -44, -22, 105, 39}}

i9 : time randomPoints(J,Strategy=>Default)
-- used 0.102918 seconds

o9 = {{0, -42, -53, -33, -65, -27, -105, -37, 99, 64}}

o9 : List

i10 : time randomPoints(J,Strategy=>LinearIntersection)
-- used 0.0169781 seconds

o10 = {{-51, -11, -34, -33, -65, 19, -86, 50, -44, -97}}

o10 : List

i11 : time randomPoints(J,Strategy=>GenericProjection)
-- used 0.364902 seconds

o11 = {{31, -15, 53, 41, 87, 9, -21, -43, -12, 85}}

o11 : List

i12 : time randomPoints(J,Strategy=>LinearIntersection, ExtendField => true)
      -- used 0.0612306 seconds

               5      4      3      2                                       5
o12 = {{54, 14a  - 48a  - 56a  + 56a  - 13a + 48, -101, 56, -42, -27, - 26a  -
-----------------------------------------------------------------------------
   4      3      2
43a  - 70a  - 96a  + 50a - 103, -29, 0, -37}}
```

5. findANonZeroMinor, extendIdealByNonZeroMinor

As mentioned in the introduction, the two functions in this section will provide further tools for computing singular locus, in addition to those available in the package `FastLinAlg`.

5.1. `findANonZeroMinor`: The typical usage of this function is as follows:

– `findANonZeroMinor(n,M,I)`

where I is an ideal in a polynomial ring over `QQ` or `ZZ/p` for p prime; M is a matrix over the polynomial ring and $n \in \mathbb{Z}$ denotes the size of the minors of interest.

The function outputs the following:
– randomly chosen point P in $V(I)$ which it finds using `randomPoints`.
– the indexes of the columns of M that stay linearly independent upon plugging P into M,
– the indices of the linearly independent rows of the matrix extracted from M in the above step,
– a random $n \times n$ sub-matrix of M that has full rank at P.

The user may also provide the following additional information:

`Strategy => Symbol`: To specify which strategy to use when calling `randomPoints` (see Section 4.1).

`Verbose => Boolean`: Set the option `Verbose => true` to turn on verbose output. This may be useful in debugging or in determining why a computation is running slowly.

`Homogeneous => Boolean`: (see Section 2)

`MinorPointAttempts => ZZ`: This controls how many points at which to check the rank of the matrix.

Example 5.1.

```
i3 : R = ZZ/5[x,y,z]

o3 = R

o3 : PolynomialRing

i4 : I = ideal(random(3,R)-2, random(2,R))

              3     2         2    3    2               2        2       2
o4 = ideal (2x  - 2x y + 2x*y  + y  + x z - 2x*y*z + y z - 2x*z  + 2y*z
-------------------------------------------------------------------
   3                      2
- z  - 2, - 2x*y - x*z - z )

o4 : Ideal of R

i5 :  M = jacobian(I)

o5 = {1} | x2+xy+2y2+2xz-2yz-2z2   -2y-z |
     {1} | -2x2-xy-2y2-2xz+2yz+2z2 -2x   |
     {1} | x2-2xy+y2+xz-yz+2z2     -x-2z |

             3       2
o5 : Matrix R  <--- R
```

```
i6 : findANonZeroMinor(2,M,I, Strategy => GenericProjection)

o6 = ({-2, 1, 1}, {0, 1}, {0, 1}, {1} | x2+xy+2y2+2xz-2yz-2z2   -2y-z |)
{1} | -2x2-xy-2y2-2xz+2yz+2z2 -2x    |

o6 : Sequence
```

5.2. `extendIdealByNonZeroMinor`: The typical usage is

– `extendIdealByNonZeroMinor(n,M,I)`

where n, M, I are same as before. This function finds a submatrix of size $n \times n$ using `findANonZeroMinor`; it extracts the last entry of the output, finds its determinant and adds it to the ideal I, thus extending I.

As before, we can have the following additional options:

`Strategy => Symbol`: This specifies which strategy to use when calling `randomPoints`.
`Homogeneous => Boolean`: This is passed in calls to `randomPoints`.
`Verbose => Boolean`: This turns on or off verbose output (see Section 5.1).
`MinorPointAttempts => ZZ`: This controls how many points at which to check the rank of the matrix.

Example 5.2.

```
i2 : R = ZZ/5[x,y,z]

o2 = R

o2 : PolynomialRing

i3 : I = ideal(random(3,R)-2, random(2,R))

                3    2       2     3     2               2       2     3
o3 = ideal (- 2x  + x y - x*y  - 2y  - 2x z + 2x*y*z + y z + y*z  - 2z
-------------------------------------------------------------------
        2            2                 2
- 2, - 2x  - 2x*y + 2y  + x*z - 2y*z - z )

o3 : Ideal of R

i4 : M = jacobian(I)

o4 = {1} | -x2+2xy-y2+xz+2yz    x-2y+z   |
     {1} | x2-2xy-y2+2xz+2yz+z2 -2x-y-2z |
     {1} | -2x2+2xy+y2+2yz-z2   x-2y-2z  |

              3       2
o4 : Matrix R  <--- R

i5 : extendIdealByNonZeroMinor(2,M,I, Strategy => LinearIntersection)

                3    2       2     3     2               2       2     3
o5 = ideal (- 2x  + x y - x*y  - 2y  - 2x z + 2x*y*z + y z + y*z  - 2z
```

```
-----------------------------------------------------------------
        2            2                2   3    2        2    3
- 2, - 2x  - 2x*y + 2y  + x*z - 2y*z - z , x  + x y + 2x*y  - y  +
-----------------------------------------------------------------
  2       2    3
2x z + y*z  - z )

o5 : Ideal of R
```

One use for this function can be in showing that certain rings are (R_1) (regular in codimension 1). Consider the following example which is (R_1) where computing the dimension of the singular locus takes around 30 seconds as there are 15500 minors of size 4×4 in the associated 7×12 Jacobian matrix. However, we can use this function to quickly find interesting minors.

Example 5.3.

```
i2 : T = ZZ/101[x1,x2,x3,x4,x5,x6,x7];

i3 : 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);

o3 : Ideal of T

i4 : M = jacobian I;

             7       12
o4 : Matrix T  <--- T

i5 : i = 0;

i6 : J = I;

o6 : Ideal of T

i7 : elapsedTime(while (i < 10) and dim J > 1 do (
                                i = i+1;
                                J = extendIdealByNonZeroMinor(4, M, J)));
-- 0.903328 seconds elapsed

i8 : dim J

o8 = 1

i9 : i

o9 = 5
```

In this particular example, there tend to be about 5 associated primes when adding the first minor to J, and so one would expect about 5 steps as each minor computed most likely will eliminate one of those primes.

There is some similar functionality obtained via heuristics (as opposed to actually finding rational points) in the package `FastLinAlg`.

Email address: sbisui@tulane.edu

Tulane University

Email address: zoeng@umich.edu

University of Michigan

Email address: sm3vg@virginia.edu

University of Virginia

Email address: tnguyen11@tulane.edu

Tulane University

Email address: schwede@math.utah.edu

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