RANDOMPOINTS PACKAGE FOR *MACAULAY2*

SANKHANEEL BISUI, ZHAN JIANG, SARASIJ MAITRA, THÁI THÀNH NGUYỄN, AND KARL SCHWEDE

Abstract. We present RandomPoints, a package in *Macaulay2* designed mainly to identify rational and geometric points in a variety over a finite field. We provide different strategies such as linear intersection and projection to obtain such points. We also present methods to obtain non-vanishing minors of a given size in a given matrix, by evaluating the matrix at a point.

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 RationalPoints [Stab], which we took inspiration from, which aims to find *all* the rational points of a variety; our aim here is to find one or more points quickly, even if it is not rational. We note also that the package Cremona [Staa] can find rational points on projective varieties, as can built-in the function randomKRationalPoint [GS], our methods frequently appear to be faster and apply in the affine setting as well.

We develop functions that apply different strategies to generate random rational and geometric points on the given variety. We also provide functions that will expedite the process of determining properties of the singular locus of X.

We provide the following core functions:

- randomPoints: This tries to find a point in the vanishing set of an ideal. (Section 2)
- projectionToHypersurface and genericProjection: These functions provide customizable projection. (Section 4)
- findANonZeroMinor and extendIdealByNonZeroMinor: The first function finds a submatrix of a given matrix that is nonsingular at a point of a given ideal. The second, adds said submatrix to an ideal, which is useful for computing partial Jacobian ideals. (Section 5.1)

All polynomial rings considered here will be over finite fields. In the subsequent sections, we explain the core and helper functions and describe the strategies that we have implemented.

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. The authors are also grateful to the reviewers for suggesting and providing preliminary codes to speed up computations, thereby improving the efficacy of the package substantially. Frank-Olaf Schreyer is also an author on this package, as he provided code related to the MultiplicationTable decomposition strategy and suggested using a probabilistic approach to compute dimension.

Key words and phrases. RandomPoints, Macaulay2.

Schwede was supported by NSF Grants #1801849, FRG #1952522 and a Fellowship from the Simons Foundation.

2. Our primary purpose: randomPoints

We start with the core function in this package: `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.

2.1. **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` or `LinearIntersection`
.

- `Default` performs a sequence of the different strategies below, aimed at finding a point quickly. It begins begins with brute force and moves to linear intersections with particularly simple linear forms, and gradually ramps up the randomness.
- `BruteForce` simply tries random points and sees if they are on the variety.
- `LinearIntersection` intersects with a random linear space.

Notice that the speed, or success, varies depending on the strategy (see also Section 3).

Example 2.1. Consider the following example.

```
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 : time randomPoints(J,Strategy=>BruteForce, PointCheckAttempts=>10)
         -- used 0.00186098 seconds

o4 = {}

o4 : List

i5 : time randomPoints(J)
         -- used 0.0205099 seconds

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

o5 : List

i6 : time randomPoints(J, Strategy=>LinearIntersection)
         -- used 0.0334881 seconds

o6 = {{0, 10, 48}}
```

`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. In fact, `ExtendField => true` will also tell Macaulay2 to use linear spaces defined over a field extension, which can improve randomness properties. This sometimes can slow computation, and other times can substantially speed it up when the variety has few rational points. For some applications, points over extended fields may also have better randomness properties.

`DecompositionStrategy`: Within the `LinearIntersection` strategy, one can also specify the option `DecompositionStrategy`. Valid values are `Decompose` and `MultiplicationTable`, the latter of which is currently only implemented for homogeneous ideals. The point is, after intersecting the linear space and obtaining an ideal defining a set of (possibly thickened) points, we need to find the minimal associated primes. By default we use Macaulay2's built-in `decompose` command. We also have implemented a `MultiplicationTable` algorithm, as provided by Frank-Olaf Schreyer, which utilizes the action of a variable on the residue fields of these points computed in more than one way. This method is frequently faster for rings with smaller numbers of variables.

The `Default` strategy switches back and forth between `Decompose` and `MultiplicationTable` for homogeneous ideals (starting with one the function thinks will be fastest). Setting this to `Decompose` in the default strategy will force only `Decompose` to be used; setting it to `MultiplicationTable` will force only `MultiplicationTable` to be used (if the ideal is homogeneous).

`Homogeneous`: Setting this true specifies that the origin (corresponding to the irrelevant ideal) is not a valid point.

`Replacement => Monomial, Binomial, Trinomial, Full`: When intersecting with a random linear space, it is frequently much faster to use a linear space defined by relatively sparse equations (ie, equations that do not involve all variables). Specifying `Monomial` will mean linear forms such as $ax + b$ are used (for constants a and b), involving only one variable. Binomial means forms like $ax + by + c$, using two variables. Trinomial means forms like $ax + by + cz + d$. `Full` means all variables will have coefficients.

`DimensionFunction => Function`: Our current implementation does not need to know the dimension of $V(I)$. However, there are places where we try to verify the dimension of an ideal before we decompose the ideal. You can pass this function `dim` (the default) or our probabilistic `dimViaBezout` or any other dimension function you might prefer.

`PointCheckAttempts => ZZ`: When calling `randomPoints` with a `BruteForce` strategy, this denotes the number of trials for brute force point checking. It also controls how many linear spaces to simultaneously study in the `LinearIntersection` strategy.

Example 2.2. We re-compute Example 2.1 this time specifying more attempts.

```
i7 : time randomPoints(J, Strategy => BruteForce, PointCheckAttempts => 10000)
-- used 1.16294 seconds

o7 = {{-43, 25, 29}}
```

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

2.2. **Comments on performance and implementation.** When working over very small fields, especially with hypersurfaces, frequently `BruteForce` is most efficient. This is not surprising as there may not be many points to check. However, if the field size is larger, `BruteForce` will perform poorly. Even for some simple examples, it could not provide any rational points if the number of trials is not large enough. Other strategies work differently on different examples, and the same strategy can sometimes work very quickly even if it typically works very slowly.

The current version of the `LinearIntersection` strategy no longer computes the dimension of the algebraic set. Instead, it first finds a point defined by linear equations. If the point is on the algebraic set, we are done. If not, we throw away one of the forms and so now have a line and we see if this line intersects our algebraic set. We continue in this way until we find a point. This appears to avoid a number of bottlenecks in our previous implementation since Macaulay2 is relatively fast at identifying when a linear space and a variety do *not* intersect.

Example 2.3. We begin with an example over a small field.

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

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

o3 : Ideal of R

i4 : time randomPoints(I, Strategy => BruteForce, PointCheckAttempts => 20000)
        -- used 0.00311884 seconds

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

o4 : List

i5 : time randomPoints(I, Strategy => Default)
        -- used 0.081349 seconds

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

Example 2.4. Now we work over a larger field.

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

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

o7 : Ideal of S

i8 : time randomPoints(J, Strategy => BruteForce, PointCheckAttempts => 2000000)
        -- used 17.7988 seconds

o8 = {{15, 67, -27, -103, 56, 66, -23, 28, -50, 13}}

o8 : List

i9 : time randomPoints(J, Strategy => Default)
        -- used 0.0864013 seconds

o9 = {{0, 0, 0, 0, 34, 76, 51, 0, 1, 0}}
```

Example 2.5. Finally, we can allow our functions to extend our field.

```
i11 : time randomPoints(J, Strategy => Default, ExtendField => true)
           -- used 0.144332 seconds

                3      2                        3      2
o11 = {{0, - a  + 62a  - 47a - 76, 0, 0, 13a  - 18a  + 63a - 31, 0, 0,
               3      2                   3      2
        - 20a  - 82a  + 35a - 19,  55a  - 64a  - 8a - 50, 1}}

i12 : coefficientRing ring first first o11

o12 = GF 1982119441
```

```
i13 : log_211 1982119441

o13 = 4
```

In this case, we found a degree 4 point.

Remark 2.2.1 (Comments on the probability of finding a point)**.** In the case of an absolutely irreducible hypersurface in $\mathbb{A}^n_{\mathbb{F}_q}$ (defined by f say), there is significant discussion in the literature estimating lower bounds of number of rational points (see for instance, [Sch, LW54, GL02, CM06]) all of which point to the fact that there is "good probability" of finding a rational point in this case when we intersect with a random line. Heuristically, we can make the following rough estimation. We expect that each equation $f = \lambda$ for $\lambda \in \mathbb{F}_q$ has approximately the same number of solutions. Since each point on $\mathbb{F}_q^n$ solves exactly one of these equations, we expect that $f = 0$ has approximately q^{n-1} solutions, or in other words, our hypersurface has q^{n-1}-points. Now, a random line L has q points. We want to find the probability that one of these points is rational for $V(f)$. We would expect that if these points are randomly distributed, then the probability that our line contains one of those points $1-(1-\frac{1}{q})^q$ which tends to $1-e^{-1} \approx 0.63$ for q large. Alternately, one can use the proof of [BS05, Proposition 2.12] for a more precise statement. For each point of L, we see that the probability that the chosen point does not lie in the intersection, $L \cap V(f)$, is $1-\frac{1}{q}$. We then exhaust this search over all the points on L to get the probability that there is indeed a successful intersection is $1-(1-\frac{1}{q})^q$. As q gets larger, this value tends to $1-e^{-1} \approx 0.63$.

Of course, there are schemes over $\mathbb{F}_q$ with no rational points at all, even for plane curves.

Remark 2.2.2 (Projecting to a hypersurface first)**.** Suppose $X \subseteq \mathbb{A}^n$ is an algebraic set. In a number of existing algorithms, one first does a generic (or even not very generic) projection $h : \mathbb{A}^n \to \mathbb{A}^m$ and so that $h(X)$ is a hypersurface (at least set theoretically). Then one finds a point $x \in h(X)$ (say as above), and computes $h^{-1}(\{x\})$ which is a linear space in $\mathbb{A}^n$ that typically intersects X in a rational point. For example, this is done in `randomKRationalPoint` in core Macaulay2. Note that projecting to a hypersurface *still is performing an intersection with a linear space*, since $h^{-1}(\{x\})$ is linear, but it tries to choose the linear space intelligently.

However, in our experience, doing this generic projection first yields slower results. First, one has to compute the dimension. There are also numerous cases where computing this hypersurface $h(X)$ can be quite slow. This particularly appears in cases when one is computing successive minors to identify the locus where some variety is nonsingular.

On the other hand, instead of using a truly random linear space to intersect with, in the default strategy, we normally choose a linear space whose defining equations are particularly noncomplicated (i.e., with many linear or binomial terms). Such simple linear spaces are the ones implicitly considered in `randomKRationalPoint` for instance since that generic projection is so simple. In practice, our approach seems to give at least as good performance compared to projecting to a hypersurface, without the chance of the code getting hung up on the generic projection or dimension computations. We also do successive intersections in a way that avoids computing the dimension as described above in Section 2.2.

3. Other functions `dimViaBezout` and `randomCoordinateChange`

`dimViaBezout`: We thank the reviewer for pointing out that in most of the computations, computing the codimension of the given ideal, is a significant bottleneck. While we have avoided most dimension computations in our current implementation, we have also implemented a probibalistic dimension computation of $V(I)$. This function takes as input an ideal I in a polynomial ring over a field and intersects $V(I)$ with successively higher dimensional random linear spaces until there is an intersection. For example, if $V(I)$ intersect a random line has a point, then we expect that $V(I)$ contains a hypersurface. If there was no intersection, this function tries a 2-dimensional linear space, and so on. This can speed up a number of computations. The function also takes in optional inputs as described below:

- `DimensionIntersectionAttempts`: Our function actually estimates dimension several times and then averages the result (rounding down) since we tend to overestimate the dimension by the above. By default it does this 3 times unless the `Homogeneous` flag is set, in which case it is done 5 times.
- `MinimumFieldSize`: If the ambient field is smaller than this integer value, it will automatically be replaced with an extension field. For instance, there are relatively few linear spaces over a field of characteristic 2, and this can cause incorrect results to be returned to the user. The user may set the `MinimumFieldSize` to ensure that the field being worked over is big enough. If this is not set, the program tries to choose a reasonable minimum field size based on the ambient ring.
- `Homogeneous`: If the ideal is homogeneous, we can use homogeneous linear spaces to compute dimension. However, in practice, this appears slower than the non-homogeneous strategy due to the fact that we must saturate the result. Hence this is turned off by default.

Example 3.1. We illustrate the speed difference in this example.

```
i2 : S = ZZ/101[y_0..y_9];

i3 : I=ideal random(S^1,S^{-2,-2,-2,-3})+(ideal random(2,S))^2;

o3 : Ideal of S

i4 : time dimViaBezout I
        -- used 0.837359 seconds

o4 = 5

i5 : time dim I
        -- used 36.8496 seconds

o5 = 5

i6 : time dimViaBezout(I, DimensionIntersectionAttempts=>1)
        -- used 0.280803 seconds

o6 = 5
```

As you can see doing a single intersection attempt is about three times faster, and it usually gives the right answer (more than 99% of the time in this particular example, but in others doing the computation in triplicate avoids returning incorrect answers).

`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. This is controlled with the following options.

- `Replacement`: This works like the `Replacement` option for `RandomPoints`.

- `MaxCoordinatesToReplace`: The user can specify that only a specified number of coordinates should be non-monomial (assuming `Homogeneous => true`).
- `Homogeneous`: Setting `Homogeneous => false` will cause degree zero terms to be added to modified coordinates (including monomial coordinates).

Example 3.2. We demonstrate some of these options.

```
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

                      ZZ
o4 : RingMap R <--- --[x, y, z]
                      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 |
```

4. Other functions: `genericProjection`, `projectionToHypersurface`

We include two functions providing customizable projections. We describe them here.

4.1. **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)`

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 4.1. We project a curve in 4-space to one in 2-space.

```
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
o3 = (map(R,--[z, w],{- x - 2y - z, - y - 2z}), ideal(z  - z*w - w ))
              5
```

Alternatively, 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.

This method works by calling `randomCoordinateChange` (Section 3) before dropping variables. It passes the options `Replacement`, `MaxCoordinatesToReplace`, `Homogeneous` to that function.

4.2. `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, respectively, R is a quotient of a polynomial ring. The output is a list with two entries: the generic projection map and the ideal (respectively the ring).

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. The same approach was used in the `point` command in the `Cremona` package [Staa]. If we already know the codimension is `c`, we can set `Codimension=>c` so the function does not compute it.

5. An application: `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.

Besides the options from `randomPoints` which are automatically passed to that function, the user may also provide the following additional information:

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

Example 5.1. We demonstrate this function.

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

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   |
```

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. It has the same options as `findANonZeroMinor`.

One can use this function to show that rings are (R_1), that is, regular in codimension 1.

Example 5.2. Consider the following 3-dimensional example (which is (R_1), regular in codimension 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.

```
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; J = I;

o6 : Ideal of T

i7 : elapsedTime(while (i < 10) and dim J > 1 do (
                                i = i + 1;
                                J = extendIdealByNonZeroMinor(4, M, J)));
           -- 0.640164 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 expects about 5 steps as each minor will typically eliminate one of those primes.

There is some similar functionality computing partial Jacobian ideals obtained via heuristics (as opposed to actually finding rational points) in the package `FastLinAlg`, see [MRSY]. That package now uses the functionality contained here in `RandomPoints` in some of its functions.

References

[BS05] H-C Graf v Bothmer and F-O Schreyer. A quick and dirty irreducibility test for multivariate polynomials over $\mathbb{F}_q$. *Experimental Mathematics*, 14(4):415–422, 2005.

[CM06] Antonio Cafure and Guillermo Matera. Improved explicit estimates on the number of solutions of equations over a finite field. *Finite Fields and Their Applications*, 12(2):155–185, 2006.

[GL02] Sudhir R Ghorpade and Gilles Lachaud. Number of solutions of equations over finite fields and a conjecture of Lang and Weil. In *Number theory and discrete mathematics*, pages 269–291. Springer, 2002.

[GS] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at http://www.math.uiuc.edu/Macaulay2/.

[LW54] Serge Lang and André Weil. Number of points of varieties in finite fields. *American Journal of Mathematics*, 76(4):819–827, 1954.

[MRSY] Boyana Martinova, Marcus Robinson, Karl Schwede, and Yuhui (Wei) Yao. FastLinAlg: faster linear algebra operations. Version 1.0. A *Macaulay2* package available at `https://github.com/Macaulay2/M2/tree/master/M2/Macaulay2/packages`.

[Sch] Frank Schreyer. randomKRationalPoint: A core *Macaulay2* function. Available at `http://www2.macaulay2.com/Macaulay2/doc/Macaulay2-1.16/share/doc/Macaulay2/Macaulay2Doc/html/_random__K__Rational__Point.html`.

[Staa] Giovanni Staglianò. Cremona: rational maps between projective varieties. Version 5.0. A *Macaulay2* package available at `https://github.com/Macaulay2/M2/tree/master/M2/Macaulay2/packages`.

[Stab] Nathaniel Stapleton. RationalPoints: A *Macaulay2* package. Version 0.95. A *Macaulay2* package available at `https://github.com/Macaulay2/M2/tree/master/M2/Macaulay2/packages`.

Email address: `sbisui@tulane.edu`

Department of Mathematics, Tulane University, New Orleans, LA 70118

Email address: `zoeng@umich.edu`

Department of Mathematics, University of Michigan, Ann Arbor, MI 48109

Email address: `sm3vg@virginia.edu`

Department of Mathematics, University of Virginia, Charlottesville, VA 22904

Email address: `tnguyen11@tulane.edu`

Department of Mathematics, Tulane University, New Orleans, LA 70118

Email address: `schwede@math.utah.edu`

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