10-29-2018

A Transparent Square Root Algorithm to Beat Brute Force for Sufficiently Large Primes of the Form p = 4n + 1

Michael R. Spink

ROCHESTER INSTITUTE OF TECHNOLOGY
College of Science
School of Mathematical Sciences

A Transparent Square Root Algorithm to Beat Brute Force for Sufficiently Large Primes of the Form $p = 4n + 1$

by

Michael R. Spink

October 29, 2018

Thesis submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in
APPLIED AND COMPUTATIONAL MATHEMATICS

1 Committee Signature Page

Dr. Manuel Lopez Date
School of Mathematical Sciences, Rochester Institute of Technology
Thesis Advisor

Dr. Anurag Agarwal Date
School of Mathematical Sciences, Rochester Institute of Technology
Committee Member

Dr. James Marengo Date
School of Mathematical Sciences, Rochester Institute of Technology
Committee Member

Dr. Matthew Hoffman Date
School of Mathematical Sciences, Rochester Institute of Technology
Director of the MS program

2 Abstract

Finding square roots in the modular integers is a well known problem that is the basis for many modern cryptosystems. For primes of the form $p = 4n + 3$, given $C \in \mathbb{Z}_p^\times$, finding solutions to $x^2 \equiv C \pmod{p}$ is deterministic. For primes of the form $p = 4n + 1$, no known deterministic computation exists for determining x given C. Tonelli (later improved by Shanks,) Cipolla, and Pocklington, among others, found sophisticated algorithms to perform this task. Brute force is a transparent approach, but offers no insights into the problem. In this thesis, we produce a transparent approach to this problem, visualized using a model built on Symplectic Geometry. One of the insights from viewing the problem in this way is a conjecture on the distribution of quadratic residues, which we exploit in our algorithm. Even though the conjecture is not essential to the workings of the algorithm, it gives it an edge over brute force for large enough primes. Finally, we follow this with examples of the algorithm's execution.

3 Acknowledgements

Firstly, I would like to thank my thesis advisor, Dr. Manuel Lopez for being by my side mathematically, time wise, and just being someone to talk to about PiRIT or whatever else is the topic of the day. Without your help there is no way this thesis would be done. I would also like to thank Dr. Anurag Agarwal and Dr. Jim Marengo for being co-advisers on this thesis and providing me with an endless number of Putnam problems to whittle away at when wanting to get away from actual coursework.

Secondly, I would also like to thank the Director of the MS program, Dr. Matthew Hoffman, the heads of the School of Mathematical Sciences, Dr. Tamas Wiandt and Dr. Matthew Coppenbarger, the Dean of the College of Science, Dr. Sophia Maggelakis, and the staff members of the School of Mathematical Sciences, Tina Williams, Ginny Gross, Corinne Teravainen, and Anna Fiorucci for their support while I was working on this thesis and my work as a whole at RIT. I would also like to thank Dr. David Barth-Hart and Kesavan Kushalnagar for their support and comments that helped direct my efforts in various ways.

Most importantly, I want to thank my parents, my siblings, and my family as a whole for always being there for me. I would also like to thank my friends at home and at RIT, everyone in PiRIT, and everyone I have forgotten or haven't found a bucket to dump you in for your support and for helping me even in the worst of my days.

Thank you.

Contents

1 Committee Signature Page **i**

2 Abstract **ii**

3 Acknowledgements **iii**

4 Introduction/Preliminaries **1**

4.1 The square root problem and number theory 1

4.2 Relevant Symplectic Geometry 7

5 Previous Work **11**

6 Geometrical solutions to the square root problem **15**

6.1 Transforming the problem into symplectic geometry 15

6.2 Using Ω to solve for the square root 19

6.3 The Front-Loading Conjecture 23

7 An Algorithm to find the square root **27**

7.1 The Preprocessing steps 28

7.2 Crux of the Algorithm: The Repeated Multiplication 32

7.3 Comparing our algorithm to Brute Force 35

7.4 Improving the Design of the Algorithm 40

7.5 Simple working examples of repeated multiplication 44

8 Future work **48**

9 Conclusion **49**

10 Appendix 1: Code **50**

11 References **59**

List of Tables

1 Table of determinants using Ω mod 13 21
2 Table of determinants using Ω mod 17 21
3 How ρ and ξ is computed for small $4n + 1$ primes 24
4 How the cases of preprocessing partition $\mathbb{Z}_p^\times$ 29
5 Comparing runtimes for $1 \pmod 8$ primes 36
6 Comparing runtimes of other primes 36
7 Comparing iterations for various $4n + 1$ primes 38
8 The runtime effects of just using Euler's Criterion 40
9 The runtime effects of just using a Collatz approach to preprocessing 41

List of Figures

1 Commutivity diagram for our manifold. 17
2 Illustration that $\sqrt{8} \equiv 5 \pmod{17}$ 20
3 A graph of QR_{101}. 23
4 How ρ is affected as p increases 25
5 How ξ is affected as p increases 26
6 How ξ affects ρ 26
7 The general flow of the algorithm 28
8 Running 10,000 random trials for $p = 73529$ to see runtime 35
9 The most time consuming steps of the algorithm 37
10 Running 10,000 random trials for $p = 73529$ to see iteration count 38
11 Average number of iterations for brute force vs. repeated multiplication 39

List of Symbols and Notation

p	Prime number, usually of form $p = 4n + 1$
$\mathbb{Z}_p^\times$	The group of units mod p, i.e $\{1, 2, \cdots, p-2, p-1\}$
C	Potential Quadratic Residue to find the square root of
x	Solution to the Square Root Problem, if it exists
λ	Shifted Solution to the Square Root Problem, if it exists
QR_p, NR_p	Sets of Quadratic Resides and Quadratic Nonresidues, respectively
$\|G\|$	cardinality of set or group G.
$ord(m)$	order of element m in $\mathbb{Z}_p^\times$
$< \alpha >$	subgroup generated by element $\alpha \in G$
g	Primitive Root for $\mathbb{Z}_p^\times$
a, b	Integers in the decomposition of $p = a^2 + b^2$
V	Vector space in which a symplectic manifold is set
Ω	Two form for manifold
$\tilde{\Omega}$	Dual mapping for Ω
π	Symplectomorphism to create our symplectic manifold
R	Table of Determinants for mod p.
"Quadrant"	For $p = 4n + 1$, each of the interval of integers $[1, n][n+1, 2n][2n+1, 3n][3n+1, 4n]$
ρ	Ratio between Quadratic Residues in Quadrant II to those in Quadrant I
ξ	Difference between Quadratic Residues in Quadrant II to those in Quadrant I
Q, s	Coefficients in decomposition of $p - 1 = Q * 2^s$
$\mathcal{A}$	Set of integers α_1 in $\mathbb{Z}_p^\times$ such that $\alpha_1^Q \equiv 1 \pmod p$
$\mathcal{B}$	Set of integers α_2 in $\mathbb{Z}_p^\times$ such that $\alpha_2^Q \equiv -1 \pmod p$
y_i	Perfect integer squares to use in algorithm
χ	Representation of the current value being looked at in the algorithm
Ψ	Representation of the integer multiple to invert in the algorithm.
κ	Boolean representation of the need to compute $\sqrt{-1}$ in the algorithm.
$[x]$	x rounded to the nearest integer

4 Introduction/Preliminaries

For this thesis we assume a working knowledge of discrete math and basic matrix operations. The rest of the groundwork for this thesis will be laid out in this section.

4.1 The square root problem and number theory

We begin by defining the problem that we are examining in this thesis.

Definition 1. Square root Problem [16]
Consider $\mathbb{Z}_p^\times$ for an odd prime integer p. Solutions to the modular equation

$$x^2 \equiv C \pmod{p} \tag{1}$$

for x where $x, C \in \mathbb{Z}_p^\times$ and C is known, solves the square root problem mod p.

We can equivalently write this as

$$x \equiv \sqrt{C} \pmod{p} \tag{2}$$

to emphasize the name of the problem.
We also now lay out some basic facts that we will be referencing throughout this thesis.

Definition 2. Quadratic Residue mod p [6, pp. 98]
The variable C in the square root problem is known as a quadratic residue mod p whenever a solution exists.

We will sometimes call these QRs for short.

Definition 3. Quadratic Non Residues mod p.[6, pp. 98].
Values in $\mathbb{Z}_p^\times$ that are not quadratic residues mod p are known as non-residues mod p.

We sometimes call these NRs for short. We can also use the following shorthand when referring to these values as a set. As will be seen in Theorem 3, only the QR's form a group, due to a lack of closure in the NR's.

Definition 4. Sets of Reciprocity [1]

Let p be prime. Then we define the following sets:

$$QR_p = \{y \in \mathbb{Z}_p^\times | \exists x \in \mathbb{Z}_p^\times, x^2 \equiv y \pmod{p}\} \tag{3}$$

$$NR_p = \{y \in \mathbb{Z}_p^\times | \forall x \in \mathbb{Z}_p^\times, x^2 \not\equiv y \pmod{p}\} \tag{4}$$

We do know the cardinality of these sets.

Theorem 1. Cardinality of the sets of Reciprocity [5, pp 121].

Let p be prime. Then

$$|QR_p| = |NR_p| = \frac{p-1}{2} \tag{5}$$

Proof. The well-known Lagrange's Theorem (see Theorem 2) implies $x^2 \equiv C \pmod{p}$ has at most two solutions. If $x_0 \in \mathbb{Z}_p^\times$ is such that $x_0^2 \equiv C \pmod{p}$, then $(-x_0)^2 \equiv C \pmod{p}$. Either $p = 2$ and thus $\mathbb{Z}_2^\times = \{1\}$ or p is odd, thus $x_0 \neq -x_0$, and it follows that if $x^2 \equiv C \pmod{p}$ has a solution, then it has exactly two solutions. □

Put another way, we can establish the function $f : \mathbb{Z}_p^\times \to Z_p^\times$ such that $f(x) \equiv x^2 \pmod{p}$ and note that f is a two-to-one function. The ending of this proof establishes the following result

Lemma 1. Number of solutions to the square root problem

Let p be prime and let $x^2 \equiv C \pmod{p}$ have a solution. Then $x^2 \equiv C \pmod{p}$ has exactly two solutions.

From this, we know that the sets of Reciprocity partition $\mathbb{Z}_p^\times$.

Corollary 1. Partition of $\mathbb{Z}_p^\times$ by sets of Reciprocity.

Let p be prime. Then NR_p and QR_p partition $\mathbb{Z}_p^\times$

Proof. Clearly by Definition 4 these two sets are disjoint and are subsets of $\mathbb{Z}_p^\times$. Now we note that $|Z_p^\times| = \phi(p) = p - 1$. Consider $NR_p \bigcup QR_p$. Since these two sets have $\frac{p-1}{2}$ elements, the entire union must have $p - 1$ elements. This proves the corollary. □

We also can look at the elements of QR_p individually in the group. In this discussion, we use the language of groups without invoking group theory in its entirety. Let us first look at the order of these elements themselves.

Definition 5. Order of an element in a group [9, pp. 105]
Let G be a group and $g \in G$ over multiplication with identity e. Then the order of g, denoted $ord(g)$, is the least integer k such that

$$g^k = e. \tag{6}$$

When $k = |G|$ in the above definition, we have a special element of a group. We will be using these quite a bit in this thesis.

Definition 6. Primitive root/Generator mod p [6, pp. 79].
Let p be prime, and G be a group. Then $g \in G$ is a primitive root if powers of g generate every element in G.

We denote by $< b >$ the group of elements generated by $b \in \mathbb{Z}_p^\times$ [6, pp. 79].We state Lagrange's Theorem here, bridging the gap between the order of an element and the order of a group.

Theorem 2. Lagrange's Theorem [9, pp. 129]
Let G be a group, and $g \in G$. Then $ord(g) \mid |G|$

Proof. See [9, pp. 129] □

Returning to the discussion of residues, a property of QR's/NR's that we will need is as follows:

Theorem 3. Product of Residues [5, pp 124]
Let p be prime and let $w, x \in QR_p$ and $y, z \in NR_p$. Then:

$$wx \in QR_p, \quad wy \in NR_p, \quad yz \in QR_p \tag{7}$$

Proof. Let g be a primitive root mod p. Then $C \in QR_p$ iff $C \equiv g^{2k} \pmod{p}$ and $C \in NR_p$ iff $C \equiv g^{2l+1}$, with $k, l \in \mathbb{N}$. Then the conclusion follows the parity properties of the addition of natural numbers. □

We can also now determine that the Quadratic residues form a group.

Corollary 2. Existence of the Group of Quadratic Residues
Let p be prime. Then QR_p forms a subgroup of $\mathbb{Z}_p^\times$ over standard multiplication.

Proof. Theorem 3 gives us closure over multiplication. Due to the fact that this is over standard multiplication, 1 will be the identity of the group. Trivially, $1 \in QR_p$. As $1 \in QR_p$ and due to Theorem 3, all QR's m have inverses $m^{-1} \in QR_p$. Lastly, we are granted associativity, as it is inherited from the larger group, $\mathbb{Z}_p^\times$. $\square$

We can quickly determine if a is a quadratic residue by Euler's Criterion.

Theorem 4. Euler's Criterion [6, pp. 68-69]
Let p be a prime, and $m \in \mathbb{Z}_p^\times$. $m \in QR_p$ iff

$$m^{\frac{p-1}{2}} \equiv 1 \pmod{p} \tag{8}$$

From here, it is easy to find the quadratic character of specific elements in $\mathbb{Z}_p^\times$. For Example:

Corollary 3. Quadratic Character of -1 [5, pp. 126]
Let p be an odd prime. Then $-1 \in QR_p$ iff $p = 4n+1$

From this and the closure of QRs, we have the following result.

Corollary 4. Quadratic residue property of $4n+1$ primes
Let $p = 4n+1$ be prime, and let $a \in \mathbb{Z}_p^\times$. $a \in QR_p$ iff $-a \in QR_p$
Similarly, if $p = 4n+3$, $a \in QR_p$ iff $-a \in NR_p$

Alternatively, one may use the Law of Quadratic Reciprocity, the properties of the Legendre Symbol, and Gauss' Lemma [6, pp. 68-69,101-103] to determine the quadratic character of an integer $m \in \mathbb{Z}_p^\times$. Often, however, this involves knowing the prime factorization of m, which is a challenging task. Further, finding the square root of m, if there is one, is more challenging. This is what we set out to do.
Now, we discuss some known results for specific primes to aid in finding the square root. The set of prime numbers $\mathbb{P}$ can be partitioned into three cases: $p = 2$, those that are of the form $p = 4n+1$, and those of the form $p = 4n+3$. The case of $p = 2$ is uninteresting with regards to this problem. As the following theorem shows, we will only need to concentrate on on those primes of the form $4n+1$.

Theorem 5. Finding the square root for $4n + 3$ primes [8].
Let $p \equiv 3 \pmod 4$, be prime. The square root, x, of quadratic residue C is:

$$x \equiv \pm C^{\frac{p+1}{4}} \pmod p \tag{9}$$

It is worth noting that either C or $-C$ is a quadratic residue, but not both, in accordance with Corollary 4
There are similar formulas for increasingly restrictive properties on primes of the form $p = 4n+1$, but there is no similarly simple formula in general. Trivially, one could square every value of x from 1 to $\frac{p-1}{2}$, but this is inefficient for large primes. This is the brute-force method for finding the solution to the square root problem.
Thus, to do better than brute force to solve square root problems, we exploit properties of primes of the form $p = 4n+1$. The two properties that we will be using of these often called Pythagorean Primes are as follows:

Theorem 6. Pythagorean property of $4n + 1$ primes [14]
Let $p = 4n+1$ be prime. Then we can write p uniquely as the sum of two squares over the integers, that is

$$p = a^2 + b^2 \tag{10}$$

This is called the Pythagorean Property of $4n + 1$ primes because it resembles the Pythagorean Theorem.Fermat was the first to conjecture about this, and Euler proved it.

Don Zagier came up with a method to find this decomposition of p, which we have implemented in Section 10. We call this the Zagier Method henceforth. Shailesh A Shirali expanded on this idea, and developed an algorithm to constructively find $p = a^2 + b^2$. Biman Bagchi proved that the algorithm always terminates, yielding the answer. [14]

Corollary 4 indicates that $-1 \in QR_p$ for $p = 4n + 1$. Now, as a result of Theorem 6, we will have concrete expressions for $\sqrt{-1} \pmod p$ if $p = 4n + 1$. The Zagier Method will provide the integers a and b needed and the Extended Euclidean Algorithm will let us efficiently compute the expression for $\sqrt{-1}$ given by the following Lemma.

Lemma 2. Finding $\sqrt{-1} \pmod p$
Given prime p and integers a, b such that $p = a^2 + b^2$. Then

$$\sqrt{-1} \equiv ab^{-1} \pmod p \tag{11}$$

and

$$\sqrt{-1} \equiv a^{-1}b \pmod p \tag{12}$$

Proof. Modulo p, we have

$$a^2 + b^2 \equiv 0 \quad \to \quad a^2 \equiv -b^2 \tag{13}$$

Consider ab^{-1} When we square this, we get

$$(ab^{-1})^2 \equiv a^2(b^2)^{-1} \equiv -b^2(b^2)^{-1} \equiv -1 \tag{14}$$

The proof for $a^{-1}b$ is similar. □

Ideally, the general solution the square root problem would come from a solution to the discrete log problem. Since this is unlikely due to the "difficult" nature of the problem - it is in the complexity class NP, among other classes, while the decision portion of the problem is indeed in class P, due to the ease of using Euler's Criterion [6, pp. 68-69]. We are left with a brute force approach to find the numerical answer to the square root problem, or a less than transparent approach like Tonelli-Shanks', Cipolla's, or Pocklington's Algorithms. We say this as these methods rely on the Discrete Logarithm (Tonelli-Shanks) or Quadratic extensions (the other two) [2]. We discuss these three algorithms in more detail in Section 5.

We approach the problem using $\sqrt{-1} \pmod p$ determined above in addition to the distribution of quadratic residues mod p. First, let's examine this distribution in geometric terms, specifically, on the surface of a torus.

4.2 Relevant Symplectic Geometry

For our problem, we need to build a Symplectic Manifold. We pull everything in this section and follow the conventions of Silva [3] unless otherwise stated. Our geometry will not be couched in differential notation. Darboux's Theorem will account for the difference in construction, avoiding any discrepancy. To this end, we note only what we see as paramount in creating a symplectic manifold.

We start with a given vector space V. In our case, V will be $\mathbb{R}^2$. A symplectic manifold is based on a symplectic vector space, which is based on a skew symmetric bilinear map. We we call this map $\Omega : V \times V \to \mathbb{R}$

Definition 7. Bilinear Map [3]
Consider map $\Gamma : X \times Y \to Z$, and let $\Gamma_{x_0} = \Gamma(x_0, y)$, where x_0 is fixed and $\Gamma_{y_0} = \Gamma(x, y_0)$ where y_0 is fixed. Γ is bilinear if Γ_{x_0} is linear for every $x_0 \in X$ and Γ_{y_0} is linear for every $y_0 \in Y$.

Definition 8. Skew Symmetric Map [3]
Map $\Gamma : X \times X \to Z$ is skew symmetric if

$$\Gamma(u, v) = -\Gamma(v, u) \quad \forall u, v \in X \times X \tag{15}$$

In our case, Ω will turn out to be

$$\Omega(u, v) = det \begin{pmatrix} -\vec{u}- \\ -\vec{v}- \end{pmatrix} = \vec{u}^T \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \vec{v} \tag{16}$$

and we see that

$$\Omega(\begin{pmatrix} a \\ b \end{pmatrix}, \begin{pmatrix} c \\ d \end{pmatrix}) = \begin{pmatrix} a & b \end{pmatrix} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a & b \end{pmatrix} \begin{pmatrix} d \\ -c \end{pmatrix} = ad - bc = det \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{17}$$

Further, note that this map is linear by the properties of determinant and matrices, so our Ω will be bilinear. We will reference this later.
We also want Ω to be symplectic. To do so we need the concept of a Dual Map.

Definition 9. Dual Map [3]
Let $\Omega : V \times V \to \mathbb{R}$ be a skew symmetric bilinear map. Then $\tilde{\Omega}$, called a dual map, is the mapping from V into its dual, denoted V^*. Further, this map is

$$\tilde{\Omega}(\vec{u}) = \Omega(\vec{u}, \_) \in Hom(V, \mathbb{R}) \tag{18}$$

Since Ω's co domain is $\mathbb{R}$, the dual of V are the functionals that map into $\mathbb{R}$. Following our example, one could define $\tilde{\Omega}$ as follows.

$$\tilde{\Omega}(\vec{u}) = \det \begin{pmatrix} u_1 & u_2 \\ x & y \end{pmatrix} \tag{19}$$

We now put this map into the vector space properly.

Definition 10. Symplectic Map/ Symplectic Linear Structure/Symplectic Vector Space [3]
Let $\Gamma : V \times V \to \mathbb{R}$ be a skew symmetric bilinear map. Γ is symplectic, if $\tilde{\Gamma} : V \to V^*$ is bijective. We call (Ω, V) a symplectic vector space, and Ω a symplectic linear structure on V.

This does bring up a property of Symplectic Vector Spaces that will confirm our use of $V = \mathbb{R}^2$

Theorem 7. Even degree Manfolds in Symplectic Linear Structures [3]
Let $\Omega : V \times V \to \mathbb{R}$ be a symplectic linear structure on V. Then $dim(V)$ is even.

We can relate two symplectic vector spaces, or a symplectic vector space with itself with a Symplectomorphism.

Definition 11. Symplectomorphism [3]
Let (V, Ω) and (V', Ω') be symplectic vector spaces in vector spaces V, V'. These two spaces are symplectomorphic if there exists a linear mapping $\pi : V \to V'$, the symplectomorphism, such that

$$\Omega'(\pi(x)) = \Omega(x) \forall x \in V \tag{20}$$

Now that we can create these spaces to be symplectic, let us now put things in this space.

Definition 12. Manifold [18]
A Manifold M is a topological space that is locally Euclidean at every point in the space.

This means that for manifold M and every $x \in M$ there is an open neighborhood U and a bijective homeomorphism $h : U \to V$ where V is a neighborhood of the origin in $\mathbb{R}$

We want the manifold to maintain the properties of the original symplectic vector space . Relevantly to us, we want to ensure that the manifold maintains the local topology of $\mathbb{R}^2$ as we are using $V = \mathbb{R}^2$.

Definition 13. Symplectic Manifold [3]
A Manifold M in space V, symplectically paired with 2-form Ω is a Symplectic Manifold if at every point p on the manifold, there corresponds a tangental plane, denoted T_pM, and on this tangental plane the map Ω_p is a symplectic linear structure in V. We denote the manifold (M, Ω)

For our purposes, we want all of these $\Omega_p = \Omega$ to carry out the geometric interpretation of the square root problem that we will create in Section 6.1. Darboux's Theorem will allow us to view the answer to one square root problem in relation to other square root problems for the same prime. We present this theorem without differential notation.

Theorem 8. Darboux's Theorem [3]
Let (M^{2n}, Ω) be a symplectic manifold. For every point in the manifold, there is an open neighborhood diffeomorphic to an open neighborhood of the origin in $\mathbb{R}^{2n}$ and every pair of points on the manifold have diffeomorphic open neighborhoods.

The standard differential 2-form on $\mathbb{R}^{2n}$ is

$$\Omega_0 = \sum_{j=1}^{n} dx_j \wedge dy_j [12] \tag{21}$$

and Darboux's Theorem implies that for every point on the manifold (M^{2n}, Ω), there exists a coordinate chart ϕ such that

$$\phi^* \Omega_0 = \Omega. \tag{22}$$

This ϕ helps M^{2n} keep the smoothness of V. [12].

This theorem implies that every point on the manifold behaves just like every other point on the manifold. This gives validity to Brute Force being used as an approach to the answer the square root problem. At some points, namely those corresponding to where C is a perfect integer square, the answer to the square root problem is readily available by arithmetic. However, this is a relatively few number of points compared to the entirety of the torus. Thus, our approach is to go point to point to gather enough global information to be able to trace back from one of these points to the solution of our given square root problem. As we will see, the important question is what portion of this global information is valuable to obtain.

5 Previous Work

In this section, we focus mainly on the history on the square root problem and relevant related topics, instead of focusing on symplectic geometry. We used this geometry as it is common to present the addition and multiplication tables for $\mathbb{Z}_p$ as embedded in a torus. Further, the companion matrix that we will use to translate the square root problem into this context requires us to solve $det(A - \lambda I)$, and later $det(A + \lambda I)$, provides us with a natural skew symmetric 2-form, which segues nicely into using this geometry. More knowledge about symplectic geometry can be found in [3].

Let us return to the square root problem, which falls into a well developed area of number theory known as quadratic reciprocity. Many of the results, however, strictly address whether of not C is a quadratic residue mod p. This is how the Legendre and Jacobi symbols are defined [5, pp. 123]. For example:

Theorem 9. [5, pp. 128] Quadratic Character of 2
2 is a Quadratic residue mod p iff $p = 8n + 1$ or $p = 8n + 7$.

The Law of Quadratic Reciprocity is also extremely useful in this regard.

Theorem 10. The Law of Quadratic reciprocity [5, pp. 130-131].
Let p and q be distinct odd primes. Then

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)(q-1)}{4}} \tag{23}$$

where

$$\left(\frac{p}{q}\right) = \begin{cases} 1, & p \in QR_q \\ -1, & \text{otherwise} \end{cases} \tag{24}$$

A considerable amount of work deals with the distribution of the quadratic residues as a whole. An example of this work is on consecutive values in $\mathbb{Z}_p^{\times}$. Walum cites M. Dunton for the following:

Theorem 11. Small Consecutive QR result [17]
Let $p \geq 11$ be prime. Then $\mathbb{Z}_p^\times$ has a $q \in QR_p > 0$ such that $q + 1 \in QR_p$

Walum extends from just QR's to k^{th} power residues [17]. Further, for a prime p, there are known formulas for the number of sequences of QR, followed by QR, sequences of QR, followed by NR, sequences of NR followed by QR, and lastly, sequences of NR followed by NR, all approximately $\frac{p}{4}$ [7]. Note that these cover the entirety of $\mathbb{Z}_p^\times$. There are also upper bounds found by Davenport for the number of sequences of QR, QR, QR and NR, NR, NR [7]. Finally, Peralta cites Hudson for showing that the sequences of QR, QR, NR, QR and QR, NR, QR, QR is nonzero for large primes [7]. Peralta worked to try to generalize these results [7].
Another body of work is into finding the least NR mod p. Trivially the least QR mod p is 1. Gauss in 1798 bounded the least NR by the following:

Theorem 12. Gauss' upper bound for the least Nonresidue modulo p. [7]
Let $p = 8n + 1$ be prime. Then the least nonresidue mod p is less than or equal to $2\sqrt{p} + 1$

However, this bound can be improved. Peralta cites Burgess for one such bound [7]. It is possible that the bound is $\frac{3(\ln p)^2}{2}$, however, this is dependent on the Generalized Reimann Hypothesis [10].
There are three well publicized algorithms that aid us in solving the square root problem. The first relies on the discrete log problem, known as the Tonelli-Shanks Algorithm [2], and the second two rely on implicit quadratic extensions, known as Cipolla's Algorithm and Pocklington's Algorithm [2]. We begin with the first of these, as it is the most well known. Tonelli-shanks relies on finding an i such that for an integer t such that

$$t^{(2^i)} \equiv 1 \pmod{p}[15] \tag{25}$$

In trying to solve Equation 1, this t is initalized to $x^Q \pmod{p}$, if we decompose p-1 as such:

$$p - 1 = Q * 2^s \tag{26}$$

Often, however, this initialization is indeed 1, and we can easily find the square root. Indeed, Turner proves that this initialization immediately works in $\frac{2}{3}$ of cases [16]. This relies on a proof that Q many elements satisfy

$$x^Q \equiv 1 \pmod p \tag{27}$$

for 4n+1 primes. Turner presents this proof in a probabilistic sense [16]. We will expand on this with an alternative proof in Section 7.1. Further, [15] presents the runtime of this algorithm as an argument in terms of the amount of multiplications. This will not be our way to assess runtime due to the difficulty to arithmetically determine how long the algorithm will run. See Section 7.4 for more information. Note however, the fact that finding this i can be difficult to compute by hand, as well as computing large powers of x. Further, there are primes of specific forms that Tonelli-Shanks can be considerably slower, This is where we want to improve on this approach.

Cipolla in our view further lessens this transparency. In his attempt to solve Equation 1 do so, it finds a t such that

$$t^2 - a \in NR_p \tag{28}$$

(or $t^2 - a \equiv 0 \pmod p$), and after doing so, performing arithmetic to find the square root [15]. This approach loses a few things: the first is decidability. A precondition for Cipolla is that $x \in QR_p$. The second is that it appears that there is no good strategy to picking t other than brute force and repeatedly computing the Legendre Symbol [15]. We hope to fix both of these things with our algorithm in Section 7 Once again, [15] computes the runtime of this algorithm in terms of modular multiplication.

Lastly, we touch on Pocklington's algorithm. Pocklington focuses mainly on primes of the form $8n+1$, as they use the following result to make things easier in the event we have a prime of the form $4n+1$:

Theorem 13. Solution to the square root problem for $8n + 5$ primes [8]
Let $p = 8n + 5$ be prime. Then if $C^{2n+1} \equiv 1 \pmod{p}$,

$$x \equiv \pm C^{n+1} \pmod{p} \tag{29}$$

Otherwise

$$x \equiv \frac{(4C)^{n+1}}{2} \quad \text{or} \quad \frac{p + (4C)^{n+1}}{2} \tag{30}$$

depending on the parity of $(4C)^{n+1}$

If $p = 4n + 3$, Pocklington uses Theorem 9. So if $p = 8n + 1$, Pocklington finds constants α_1, β_1 such that

$$\alpha_1^2 + C\beta_1^2 \in NR_p[8] \tag{31}$$

From there, Pocklington iterates on a two recurrence relations for α, β:

$$\alpha_n = \frac{(\alpha_1 + \beta_1\sqrt{-C})^n + (\alpha_1 - \beta_1\sqrt{-C})^n}{2} \tag{32}$$

$$\beta_n = \frac{(\alpha_1 + \beta_1\sqrt{-C})^n - (\alpha_1 - \beta_1\sqrt{-C})^n}{2\sqrt{-C}} \tag{33}$$

and performs some arithmetic once $\alpha_k \equiv 0$. We find that this is further removed from Cipolla and has the same problems that Cipolla did: decidability and transparency. The recurrence relations indeed look daunting at a glance. However, this does give us a simple way to handle some of the $4n + 1$ primes [8].

There have been other algorithms to find the square root. For example, in [13] uses ideas from Elliptic curves to find an algorithm that runs in $\mathcal{O}(log^9 p)$, which is faster than brute force, and [11] shows that methods can be derived from continued fractions. We do not emphasize Elliptic curves here, which are outside the purview of knowledge that we view as "basic" relative to this problem, furthering this issue of transparency.
We now begin deriving our algorithm from ideas that will help us understand the difficulty of improving on the brute force approach.

6 Geometrical solutions to the square root problem

In this section, we build a useful symplectic manifold that will allow us to look at the square root problem in a non number theoretic way. The Ω in this manifold is then used to attempt to find a robust solution to the square root problem. Our approach is to gather global information on the manifold help solve Equation 1, rather than the indiscriminate approach to the manifold used by brute force. Our algorithm, discussed in Section 7, will benefit if a conjecture we discover from looking at the manifold in this manner is true. We call this conjecture the Front-Loading Conjecture. As will be seen, our algorithm will terminate successfully regardless of truth of the conjecture.

6.1 Transforming the problem into symplectic geometry

Let p be an odd prime and consider Equation 1: Consider shifting the problem by simply letting $x = \lambda - 1$:

$$(\lambda - 1)^2 \equiv C \pmod{p} \tag{34}$$

This expands to

$$\lambda^2 - 2\lambda - C + 1 = \lambda(\lambda - 2) - (C - 1) \equiv 0 \pmod{p} \tag{35}$$

so a companion matrix A for this polynomial is

$$A = \begin{pmatrix} 2 & C-1 \\ 1 & 0 \end{pmatrix} \tag{36}$$

Note however how the characteristic and minimal polynomials of this is the same as the expanded form of the problem. Since we want to solve $det(A - \lambda I) \equiv 0 \pmod{p}$ for this matrix, we want to create a symplectic manifold that uses the determinant as the skew symmetric 2-form.

To build this manifold, we begin with a smooth vector space, $V = \mathbb{R}^2$. Note that $dim(\mathbb{R}^2) = 2$, which satisfies Theorem 7. Keep in mind that we will be using the integer lattice over this space in due time. Our basis sets will be as follows:

$$\{u_i\} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix} \in V \right\} \tag{37}$$

$$\{v_i\} = \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix} \in V \right\} \tag{38}$$

We now build our bilinear map as in Equation 16.

$$\Omega(\vec{u}, \vec{v}) = det \begin{bmatrix} -\vec{u}- \\ -\vec{v}- \end{bmatrix}. \tag{39}$$

As shown in Equation 17, this choice of Ω is skew symmetric. Based on the properties of the determinant, this Ω is bilinear, or more simply, linear in $\mathbb{R}$. Further, note how the output of Ω is real valued by the definition of the matrix determinant. Now, fix

$$\vec{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \tag{40}$$

Then to keep $\tilde{\Omega}$ similar to that of Ω, we define

$$\tilde{\Omega}(\vec{u}) = \det \begin{pmatrix} u_1 & u_2 \\ x & y \end{pmatrix} \tag{41}$$

where x and y will be determined when something is plugged into this determinant function. Note that is in the dual of $\mathbb{R}^2$, as u determines the top row of this array.

For example, let $\vec{u} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$, then

$$\tilde{\Omega}(\vec{u}) = \det \begin{pmatrix} 1 & 2 \\ x & y \end{pmatrix} \tag{42}$$

and

$$\tilde{\Omega}(\vec{u})[\begin{bmatrix} 1 \\ 3 \end{bmatrix}] = \det \begin{pmatrix} 1 & 2 \\ 1 & 3 \end{pmatrix} = 1 \in \mathbb{R} \tag{43}$$

We denote the input of $\tilde{\Omega}$ in square brackets. We know that this output is bijectively related to the choice of x and y. This means that (V, Ω) is a symplectic vector space. Now, we just need a manifold that has $\mathbb{R}^2$'s smoothness. To do this, we will use a symplectomorphism. Let $V' = V$. Consider $\pi : V \times V \to V' \times V'$

$$\pi(a,b) = \pi(c,d) \quad \text{if} \quad p|(a-c) \quad \text{and} \quad p|(b-c) \tag{44}$$

This creates an equivalence relation $\mathcal{R}$ on $V \times V$. Let $\Omega' = \Omega$ on these equivalence classes. We have gone from $V \times V$ to $V' \times V'$ to $\mathbb{R}$. We can also illustrate this (and other symplectomorphisms) by the commutivity diagram in Figure 1.

$$\begin{tikzcd}
V \times V \arrow[rr, "\Omega"] \arrow[dr, "\Pi"'] & & \mathbb{R} \\
& V' \times V' \arrow[ur, "\Omega' \text{ on the equivalency classes in } V' \times V'"'] &
\end{tikzcd}$$

Figure 1: Commutivity diagram for our manifold.

In essence, we are taking V and creating a cylinder out of it in “both” directions. Applied to a specific prime, this will turn into a representation of $\mathbb{R}$, or in terms of the integer lattice, $\mathbb{Z}_p$. This torus will be the manifold that we will be working on. This is also the classical geometric picture of the “donut” torus, yet made out of the real plane. This torus, combined with Ω, will be our symplectic manifold. Since this manifold maintains the properties of its underlying vector space, this torus remains smooth. Thus, tangential planes exist at all points on this plane. Thus, This manifold is symplectic. Note that we only care about the integer lattice on the torus. If there exists a tangental plane at all real points on the torus, then clearly there is a tangental plane at every point on the integer lattice on the torus.

Now that we have our torus and have shifted our problem, we look at what steps to take in our attempts to solve the square root problem. We can change the prime p to determine the specific manifold (and the points that are in equivalence classes), the value of C (the specific problem to solve), and the potential solution to the problem, λ. For us, fix p to fix the specific manifold to work with. This leaves us with (C, λ) as points on our torus. Pick an arbitrary point on the torus, and call it the origin. Let the C axis rotate counterclockwise around the torus from the origin. Since C are the problem to solve, C will range from 1 to $p-1$ Let the λ axis rotate around the “center” of the torus. Since x and $-x$ are both solutions if one is, λ will range from 1 to $\frac{p-1}{2} - 1$. With this in mind, we have all possibilities for all of the square root problems. Those that fit certain criteria (to be discussed) are solutions to the square root problem. Since there are $\frac{p-1}{2}$ quadratic residues, there will be this many points that fit this criteria.

6.2 Using Ω to solve for the square root

We will be finding the square root through a few related ideas here. First, we will be using Ω directly to find if a solution is correct. We then holistically look at the entire torus in this lens to improve on the brute force method.

Let $A = \begin{pmatrix} 2 & C-1 \\ 1 & 0 \end{pmatrix}$ as described above. If we fix C and let it be a quadratic resiude, we want to find where $det(A - \lambda I) \equiv 0 \pmod p$. That is, this determinant is a multiple of p. For example, mod 13, say we want to find the square root of 3. Thus $A = \begin{pmatrix} 2 & 2 \\ 1 & 0 \end{pmatrix}$ Consider the matrix $A = \begin{pmatrix} 2-5 & 2 \\ 1 & 0-5 \end{pmatrix}$. The determinant of this matrix is 13, which is a multiple of 13. Since this $A - \lambda I$ has been shifted at the outset of the problem (see equation 34), we have that $x + 1 = 5$, so $x = 4$. Indeed, $4^2 \equiv 3 \pmod{13}$. Notice how this also works if you look for $\lambda = 10$, or $x = 9$.

To make this problem easy to compute by hand, we can convert the determinant into an similar problem about finding the area of the parallelogram between the vectors of the columns of A. Begin with A, so that the parallelogram in question is bounded between $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} C-1 \\ 0 \end{pmatrix}$. Shift the first of these vectors to the right by 1 at each iteration, and the second of these vectors up by 1 at each iteration. We are now shifting the vectors in question up and to the right. Due to this shift, we are looking at $det(A + \lambda I) \equiv 0 \pmod p$ instead of the traditional $det(A - \lambda I) \equiv 0$. This is valid to due the fact that we are working in the modular integers. We do this for the convenience of arithmetic and to keep the picture in Quadrant I of our vector space, $\mathbb{R}^2$. Due to this shift, we similarly shift our solution to $x = \lambda + 1$ This may make the square root easy to identify, however, this is still infeasible to use for large primes, as one would still have to scan though all possible areas to find the one that works. Though, perhaps the use of Pick's Theorem would make things easier to compute by hand. We can animate this in Geogebra. Figure 2 illustrates that $\sqrt{8} = 5 \pmod{17}$. We identify $\lambda = 4$, so $x = 5$. The figure also shows the original columns of A. See `https://youtu.be/61SjGAkMvKY` for a sample run through of these areas. Notice how $\lambda = 3$ corresponds to the mod 13 case we mentioned above.

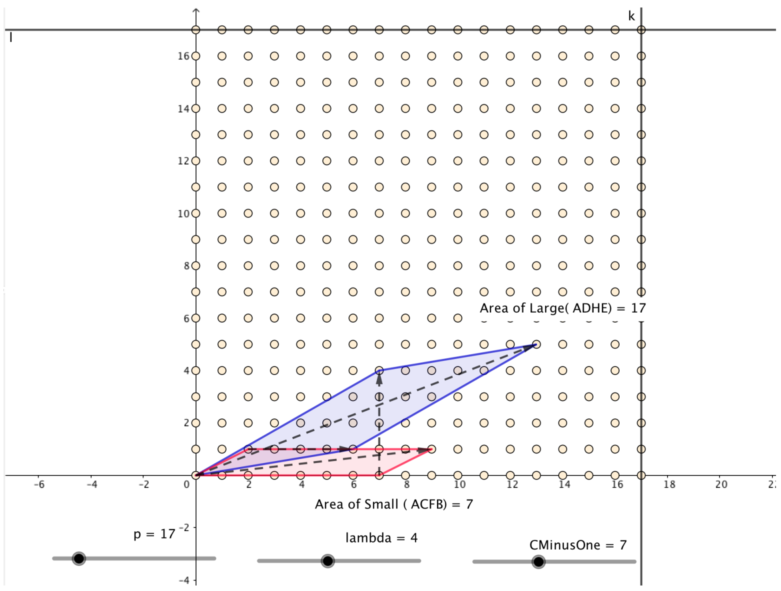


Figure 2: Illustration that $\sqrt{8} \equiv 5 \pmod{17}$

While all of the points on the torus look locally identical via Theorem 8, we can solve a specific square root problem this way. Thus we want to look at the torus as a whole, rather than around one rotation about the torus. This is the benefit of hosting all possible problem and solution on the torus. which says that every point on We show the table for all possible square root problems mod 13 and 17 below in Tables 1 and 2. Do note that this shows values of C, not $C - 1$.

From this table, we see a few trends. Firstly, we see that since these are $4n+1$ primes, Theorem 4 holds. Thus we can restrict C to just run from 1 to $\frac{p-1}{2}$ Second, the top left of Table 2 is Table 1. In other words, as $p \to \infty$, this table just builds on the largest prime not exceeding p, which recursively builds on smaller primes all the way down to easily computable values.

We can represent these tables as a system of recurrence relations. Let the table be R and $R_{i,j}$ is the value in the i^{th} row from the **top** and the j^{th} column from the **left**. The top left entry of the table is R_{11}.

$$\begin{cases} \delta_1 = 3, \quad R_{1,1} = 0 \\ R_{(i+1),j} = R_{i,j} + \delta_i, \quad R_{i,(j+1)} = R_{i,j} - 1 \\ \delta_i = \delta_{i-1} + 2 \end{cases} \tag{45}$$

		Values of λ					
		0	**1**	**2**	**3**	**4**	**5**
Values of C	**1**	**0**	3	8	15	24	35
	2	-1	2	7	14	23	34
	3	-2	1	6	**13**	22	33
	4	-3	**0**	5	12	21	32
	5	-4	-1	4	11	20	31
	6	-5	-2	3	10	19	30
	7	-6	-3	2	9	18	29
	8	-7	-4	1	8	17	28
	9	-8	-5	**0**	7	16	27
	10	-9	-6	-1	6	15	**26**
	11	-10	-7	-2	5	14	25
	12	-11	-8	-3	4	**13**	24

Table 1: Table of determinants using Ω mod 13

		Values of λ							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Values of C	**1**	**0**	3	8	15	24	35	48	63
	2	-1	2	7	14	23	**34**	47	62
	3	-2	1	6	13	22	33	46	61
	4	-3	**0**	5	12	21	32	45	60
	5	-4	-1	4	11	20	31	44	59
	6	-5	-2	3	10	19	30	43	58
	7	-6	-3	2	9	18	29	42	57
	8	-7	-4	1	8	**17**	28	41	56
	9	-8	-5	**0**	7	16	27	40	55
	10	-9	-6	-1	6	15	26	39	54
	11	-10	-7	-2	5	14	25	38	53
	12	-11	-8	-3	4	13	24	37	52
	13	-12	-9	-4	3	12	23	36	**51**
	14	-13	-10	-5	2	11	22	35	50
	15	-14	-11	-6	1	10	21	**34**	49
	16	-15	-12	-7	**0**	9	20	33	48

Table 2: Table of determinants using Ω mod 17

We see from this that the entire table can be generated from the top left entry. Looking exclusively at the top row, we can generate this top row more succinctly by combining these recurrences:

$$a_n = a_{n-1} + (2(n-1)+3) = a_{n-1} + (2n+1), \qquad a_0 = 0 \tag{46}$$

When we solve this recurrence, we get

$$a_n = n(n+2) = n^2 + 2n. \tag{47}$$

If we want to go down to row C to find a multiple of p, we see that

$$kp + C - 1 = a_n. \tag{48}$$

So

$$n^2 + 2n - (kp + C - 1) = 0 \tag{49}$$

Taking modulo p, we have returned to the original problem of solving a quadratic quickly, now specifically

$$n^2 + 2n - (C-1) \equiv 0 \pmod{p}, \tag{50}$$

which is what we were trying to avoid. Notice how this matches Equation 35. Thus, let's restrict this idea to the primes of the form $p = 4n + 1$ to try to take advantage of the symmetry of $\mathbb{Z}_p^\times$. This will form the basis of the Front-Loading Conjecture and the Repeated Multiplication portion of our algorithm.

6.3 The Front-Loading Conjecture

Let us now focus on $4n+1$ primes and return to examining Figures 1 and 2. The other thing of note is that the the bolded answers seem to be concentrated in the top and bottom of the chart. Since the rows of the table are concerned with the quadratic residues, it appears that the quadratic residues mod p are biased toward these ends of $\mathbb{Z}_p$.

We now make this more concrete and state this as a principal idea of this thesis:

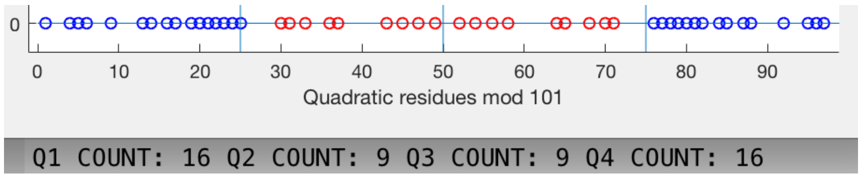
Conjecture 1. The Front-Loading Conjecture.

Let $p = 4n+1$ be prime. We partition $\mathbb{Z}_p^\times$ into four distinct regions, from $[1, n]$, $[n+1, 2n]$, $[2n+1, 3n]$,$[3n+1, 4n]$, respectively called quadrants I,II,III, IV. Furthermore, we call the union of quadrants I, IV the "RICH" regions, and the union of quadrants II and III the "POOR" regions. Then there are more quadratic residues in the RICH regions than the POOR ones.

Note that this is conjecture. However, we strongly believe it to be true. We present a heuristic argument below.

We illustrate this point with a few examples. Firstly, we show that this does not hold for primes of the form $p = 4n+3$, even taking Corollary 4 into account. Take $p = 19 = 4*4+3$ for example. By Theorem 1, there are 9 QR's. Creating a list of them yields $1, 4, 5, 6, 7, 9, 11, 16, 17$. Put into their quadrants, the respective counts is that are 2 QR's in Quadrant I, 3 QR's in Quadrant II, 2 QR's in Quadrant III, and 2 QR's in Quadrant 4, contradicting the claim.

Now we look at $p = 17 = 4*4+1$, and create the same list. The QRs are 1,2,4,8,9,13,15,16. Since $n = 4$, there are 3 QR's in Quadrant I, 1 in Quadrant II, 1 in Quadrant 2, and 3 in Quadrant IV. Graphically, we can illustrate this for $p = 101$ in Figure 3. The QR count is below the image.



```
Q1 COUNT: 16 Q2 COUNT: 9 Q3 COUNT: 9 Q4 COUNT: 16
```

Figure 3: A graph of QR_{101}.

However, as p increases, it becomes difficult to see the individual Quadratic residues. This also illustrates Corollary 4. So for this brief discussion, we ignore Quadrants III and IV. We now define a constant to quantify the strength of this conjecture for a prime p.

Definition 14. Degree of Front-Loading modulo p.

The constant

$$\rho := \frac{\text{Number of QR's in Quadrant II}}{\text{Number of QR's in Quadrant I}} \tag{51}$$

is the Degree of Front-Loading mod p. This tells us how biased the quadratic residues mod p are.

For ease of discussion, also define the difference between the numbers of QR's as ξ Formally,

Definition 15. Difference between QR counts.

The constant

$$\xi = \text{Number of QR's in Quadrant I} - \text{Number of QR's in Quadrant II} \tag{52}$$

difference in QR's from Quadrant I to that of Quadrant II.

We directly compute ρ and ξ for a few small primes in Figure 3 and carry it to primes less than 10,000 in Figures 4 and 5.

		Values of interest			
		QR I	QR II	ρ	ξ
Primes	**5**	1	0	N/A	1
	13	2	1	.5000	1
	17	3	1	.3333	2
	29	5	2	.4000	3
	37	5	4	.8000	1
	41	7	3	.4286	4
	53	8	5	.6250	3
	61	9	6	.6667	3
	73	10	8	.8000	2
	89	14	8	.5714	6
	97	13	11	.8462	2
	⋮	⋮	⋮	⋮	⋮
	233	32	26	.8125	6
	617	80	74	.9250	6
	73529	9275	9107	.9819	168

Table 3: How ρ and ξ is computed for small $4n + 1$ primes

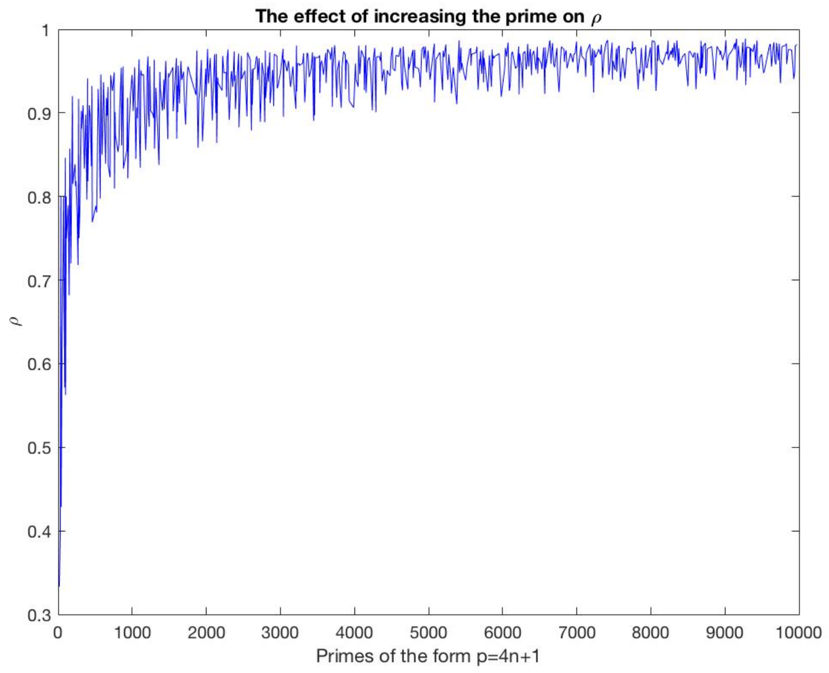


Figure 4: How ρ is affected as p increases

We see from Figure 4 that the values of ρ appear to be asymptotic to $\rho = 1$,yet $\lim_{p\to\infty} \rho = 1$ indicates that Front-Loading holds, however, ρ is not monotonically increasing, despite being well above $\rho = 0.8$ for much of this figure. This means that the quadratic residues are relatively close in number for large primes. Ther However, this asymptotic nature seems to indicate that techniques from analytic number theory may be helpful in proving this conjecture. Further, there does not seem to be any sharpness in the graph, thus it is difficult to compute ρ algebraically given p.

We also see from Figure 5 that less can be stated about the pure difference in QR's other than ξ appears to be increasing in a general sense. Importantly to the case of Front-Loading, ξ appears to be nonegative for all primes, supplemented by the general linear increase of ξ as p increases. However, ρ and ξ appear to be uncorrelated, as shown in Figure 6. Do note again that most of the ρ coordinates are larger than 0.8.

These constants further the argument that the quadratic residues are not uniformly distributed in $\mathbb{Z}_p^\times$, but this will aid us in selecting perfect integer squares quickly in our algorithm below. We will see that we believe that a lower ρ, but larger ξ will help run the algorithm in fewer iterations. Throughout Section 7, this discussion will continue relative to the algorithm.

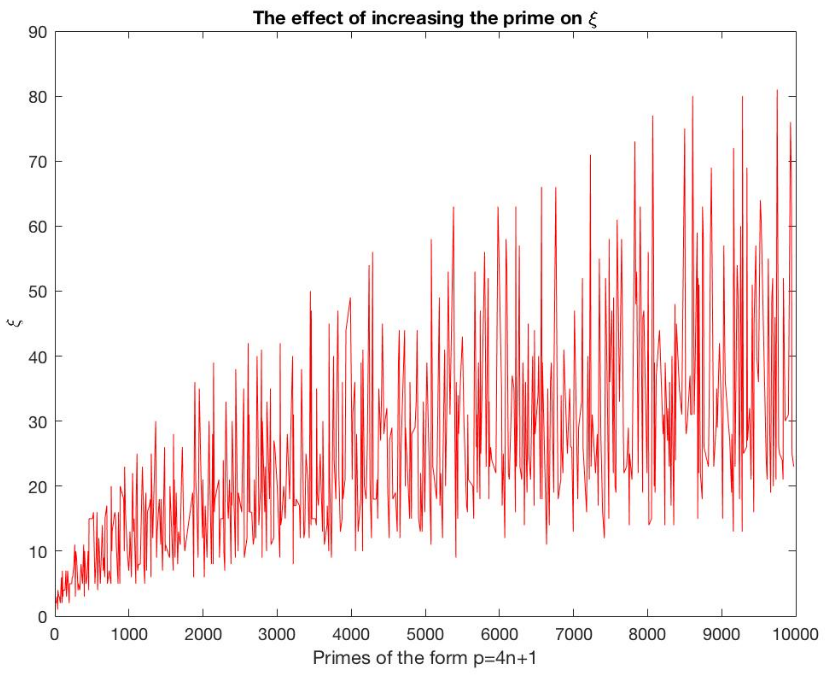


Figure 5: How ξ is affected as p increases

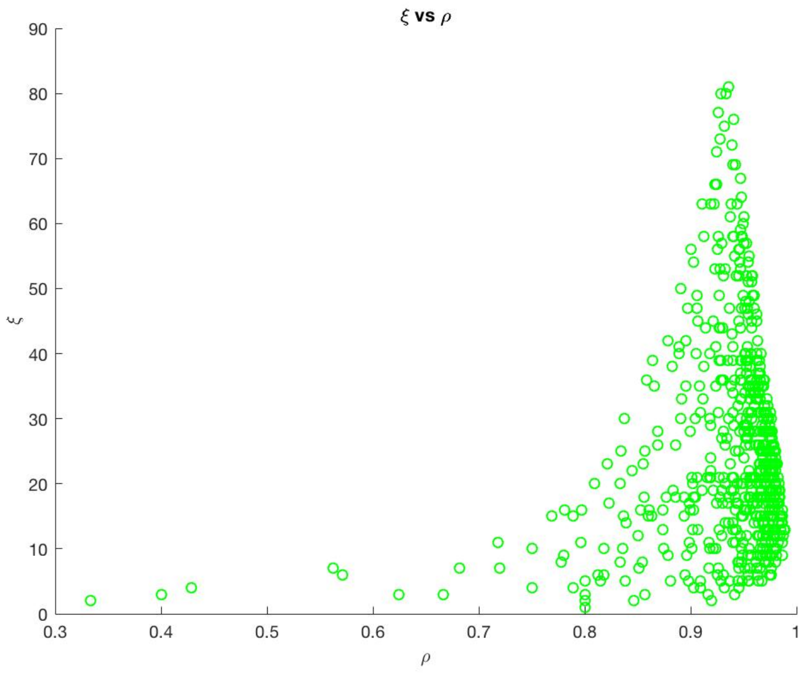


Figure 6: How ξ affects ρ

7 An Algorithm to find the square root

We now will motivate and discuss the algorithm that we developed. The reader can find a working MATLAB implementation in Appendix I of this thesis (Section 10). MATLAB has been used due to its ease of use, portability, and easy to use tools to compute detailed runtime statistics to aid in debugging/comparison. Let p be prime, and $C \in \mathbb{Z}_p^\times$. We want to determine whether or not C is in QR_p or in NR_p. If it is in QR_p, we want to find the two square roots of C. We will be heavily using Theorem 3 and Corollary 4 from section 4.1 of this thesis.

Our algorithm is broken into two parts: Preprocessing and Perfect Square Multiplication. The goal of preprocessing is to determine if a potential QR has an easy deterministic solution, or if it is a NR. The remaining cases imply $p = 8n + 1$ and are difficult to assess. We will repeatedly multiply by carefully selected perfect integer squares to determine the square root of these cases. The Front-Loading Conjecture, if it is true, will aid in selecting these integer squares quickly. See Figure 7 for a visual representation of this.

Our algorithm beats brute force for large enough primes. We will be examining this in terms of runtime and iteration count. Further, it is more transparent than the algorithms presented in Section 5, as it uses neither quadratic extensions nor the discrete log problem. These benefits allow our algorithm to stand out and be useful in solving the square root problem, and we discuss potential improvements to the algorithm's design. We also discuss a replacement to the Preprocessing step that allows the algorithm to avoid overflow inaccuracies. Importantly, implementing the algorithm in a lower level language such as Python or C will improve the effectiveness of the algorithm by getting around the dictionary and conversion ineffectiveness of MATLAB.

Now, we discuss the two major components of our algorithm in more detail, beginning with Preprocessing.

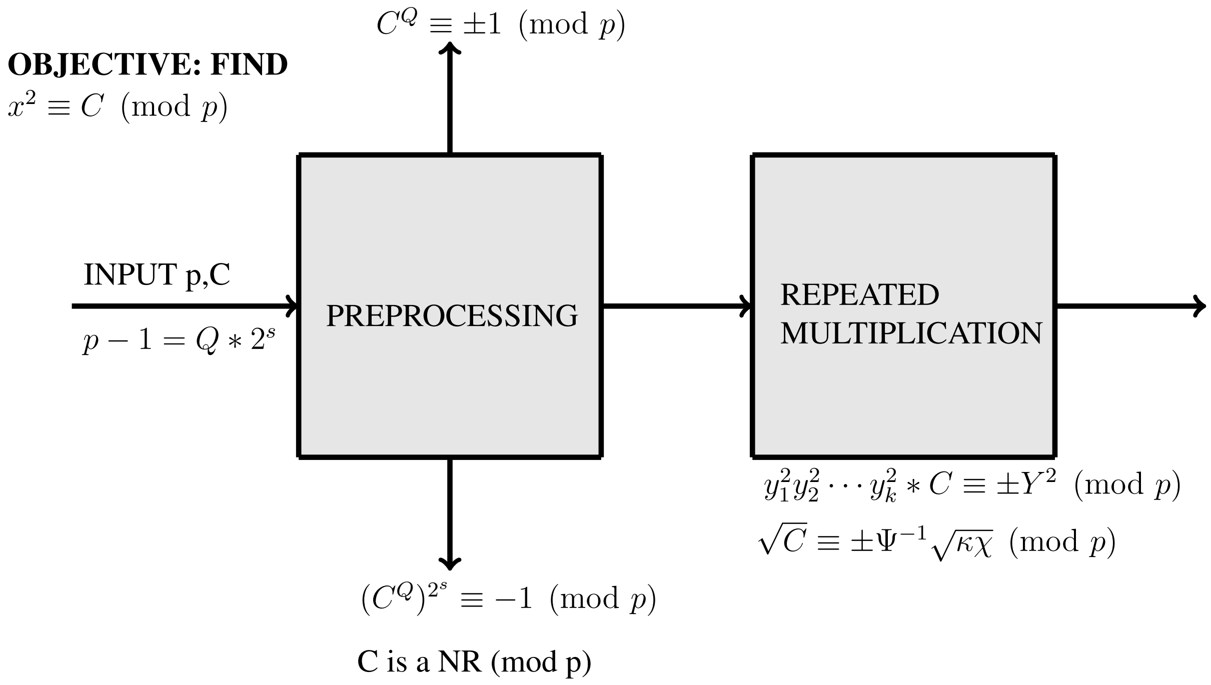

$C^Q \equiv \pm 1 \pmod p$
OBJECTIVE: FIND
$x^2 \equiv C \pmod p$
INPUT p,C
$p-1 = Q*2^s$
PREPROCESSING
REPEATED MULTIPLICATION
$y_1^2 y_2^2 \cdots y_k^2 * C \equiv \pm Y^2 \pmod p$
$\sqrt{C} \equiv \pm \Psi^{-1}\sqrt{\kappa\chi} \pmod p$
$(C^Q)^{2^s} \equiv -1 \pmod p$
C is a NR (mod p)

Figure 7: The general flow of the algorithm

7.1 The Preprocessing steps

For preprocessing, we want to decide the quadratic character of potential quadratic residue C. As we will see in the remainder of Section 7, finding an efficient decidability condition in the repeated multiplication step of the algorithm proves difficult, so this preprocessing proves important. We could simply just use Euler's Criterion, but to lower the strain on our machine we do this in stages. We find

$$p - 1 = Q * 2^s, \quad \text{where } Q \text{ is odd} \tag{53}$$

Q is easy to find as we are dividing by 2 as many times as possible. We then examine C^Q. If this is ± 1, we conclude that C is a quadratic residue. We can compute $\sqrt{C}$ as follows

$$C^Q \equiv C * C^{Q-1} \equiv \pm 1 \pmod p \tag{54}$$

$$\sqrt{C}C^{\frac{Q-1}{2}} = \pm\sqrt{\pm 1} \pmod p \tag{55}$$

$$\sqrt{C} = \pm(C^{\frac{Q-1}{2}})^{-1}\sqrt{\pm 1} \pmod p \tag{56}$$

We note that this is a deterministic method to find the square root in these cases. We now prove that this correct. We need to confirm that no NR will fall into this case.

Lemma 3. Quadratic character of C if $C^Q \equiv \pm 1 \pmod p$
Let $p = 4n + 1 = Q * 2^s + 1$. Then if $C^Q \equiv \pm 1 \pmod p$, then $C \in QR_p$

Proof. Let $p = 4n + 1 = Q * 2^s + 1$, and $C \in NR_p$. Then $Q = \frac{p-1}{2^s}$. Since $p = 4n + 1, s \geq 2$. Assume that $\alpha = C^Q \equiv \pm 1 \pmod p$. We can repeatedly square α $s - 1$ times. so we have

$$\alpha^{(2^{s-1})} \equiv (C^Q)^{2^{s-1}} \equiv C^{(Q*2^{s-1})} \equiv C^{\frac{p-1}{2}} \equiv 1 \tag{57}$$

And now we apply Euler's Criterion (Theorem 4) to get the desired result. □

Now, if $C^Q \equiv 1 \pmod p$, then $C(C^{Q-1}) \equiv 1 \pmod p$, so $\sqrt{C}C^{\frac{p-1}{2}} \equiv \pm 1 \pmod p$.

On an implementation note, in the case where $C^Q \equiv -1$, we use the method of Zagier [14] to easily compute $p = a^2 + b^2$, and thus $\sqrt{-1}$ via the extended Euclidean Algorithm [1].

If C^Q is not ± 1, then we will repeatedly square $\pmod p$ this value, up to $s - 1$ times. If we get 1 at any point, say $s = k$, then we know that C is a QR by the same logic as in the proof to Lemma 3. However, $(C^Q)^{2^k} = C(C^{Q*2^k-1})$. This exponent will be odd if $2 \leq k \leq s - 1$, so we cannot perform the same trick we did above to easily find the modular square root. From this, we can easily find $\sqrt{-1} \equiv (C^Q)^{2^{k-2}} \pmod p$ as a direct consequence to Lemma 1
This establishes four possible outcomes to this preprocessing. We state how this partitions $\mathbb{Z}_p^\times$ in Table 4

Case Number	Qualifier	Number of Instances
1	$C^Q \equiv 1 \pmod p$	Q
2	$C^Q \equiv -1 \pmod p$	Q
3	$C^{\frac{p-1}{2}} \equiv -1 \pmod p$	$\frac{p-1}{2}$
4	none of the above	$\frac{p-1}{2} - 2Q$

Table 4: How the cases of preprocessing partition $\mathbb{Z}_p^\times$

We now prove each of these instance numbers. Trivially, Case 3 follows Theorem 1. Cases 1 and 2 require some justification. We state this here.

Theorem 14. $2Q$ many instances of C satisfy $C^Q \equiv \pm 1 \pmod{p}$
Let $p = 4n + 1 = Q * 2^s + 1$, and $a \in \mathbb{Z}_p^\times$. Define $\mathcal{A}$ as follows:

$$\mathcal{A} = \{C \in \mathbb{Z}_p^\times | C^Q \equiv 1 \pmod{p}\} \tag{58}$$

Similarly, define

$$\mathcal{B} = \{C \in \mathbb{Z}_p^\times | C^Q \equiv -1 \pmod{p}\} \tag{59}$$

Then $|\mathcal{A}| = |\mathcal{B}| = Q$

Proof. Let $p = 4n + 1$ and $p - 1 = Q * 2^s$. Thus, $|\mathbb{Z}_p^\times| = Q * 2^s$ Further, let g be a primitive root mod p. Now let $a = g^{2^s}$. By Theorem 2, $ord(a) = Q$, and thus, $| < a > | = Q$.
Let $\gamma \in \mathcal{A}$. Thus, $\gamma^Q \equiv 1 \pmod{p}$. Thus, γ belongs to the unique cyclic subgroup of order Q in cyclic group $\mathbb{Z}_p^\times$. This unique subgroup of order Q is $< a >$. Further, consider a^r for $r \geq 1$. Then consider

$$(a^r)^Q = (a^Q)^r = 1^r = 1 \tag{60}$$

due to Definition 5. Since the powers of a define $< a >$ and $a^r \in \mathcal{A}$, we can say that $< a >= \mathcal{A}$, and thus $|\mathcal{A}| = Q$.
Now, similarly define $b = g^{2^{s-1}}$. By Theorem 2, $ord(b) = 2Q$, and thus $| < b > | = 2Q$. Note that since $b^2 \equiv a \pmod{p}$, $\mathcal{A} = \{b^{2j} | 1 \leq j \leq Q\}$

$$((b^{2j-1})^2) \equiv (a^{2j-1})^Q \tag{61}$$

We can thus see that every element that in $< b >$ that is not in $< a >$ is the square root of an element in $< a >$. But since this element $\hat{b}$ is not in $\mathcal{A}$, and thus $\hat{b}^Q \equiv \sqrt{1}$, but $\hat{b}^Q \not\equiv 1$ or it would be in $< a >$. Now that the square root problem has exactly two solutions if a solution exists by Lemma 1, and $(-1)^2 = 1$, so $\hat{b}^Q \equiv -1$ and thus $\hat{b} \in \mathcal{B}$.
In terms of set notation, we say that $\mathcal{B} =< b > -\mathcal{A}$.
Since $| < b > | = 2Q$ and $|\mathcal{A}| = Q$, and now $|\mathcal{B}| = Q$ as expected. □

We further observe that since every other element of $< b >$ goes to $\mathcal{B}$ and $\mathcal{A}$, we see that $\mathcal{A}$ and $\mathcal{B}$ are disjoint, as expected by their definitions. Do note that this is an expansion to Turner's probabilistic proof for $|\mathcal{A}| = Q$ [16].

Thus we see that we can find a deterministic solution for $2Q$ of the quadratic residues mod p. Thus there are $\frac{p-1}{2} - 2Q$ quadratic residues unaccounted for. We can deduce when $2Q$ is maximal, among primes of the form $4n + 1$.

Theorem 15. Size of $2Q$ for $5 \pmod 8$ primes
Let $p = 8n + 5 = Q * 2^s + 1$. Then $2Q = \frac{p-1}{2}$

Proof. Let $p = 8n + 5 = Q * 2^s + 1$. Via Algebra, we have,

$$p - 1 = 2^2(2n + 1) = 2^2(Q) \tag{62}$$

and we are done □

This means that all quadratic residues for $p = 8n + 5$ primes fall into these cases and have a deterministic solution to the square root problem. See [8] or Section 5 for Pocklington's deterministic algorithm for these primes.

Applying this line of thinking to primes of the form $4n + 3$ produces a deterministic algorithm for finding the modular square root.

Corollary 5. Variation on the Cardinality of the sets of Reciprocity for $p = 4n + 3$ primes
Let $p = 4n + 3 = Q * 2, +1$, where $Q = \frac{p-1}{2}$. Then only cases 1 and 2 in Table 4

Proof. $Q = 2n + 1 = \frac{p-1}{2}$ □

Thus, we can incorporate deterministic results for all primes of the form $8n + 1$ This paints a more complete a more complete picture of prime modular behavior than Turner's Result, as turner ignores the case of $C^Q \equiv -1 \pmod p$. Thanks to Zagier-Shirali's algorithm we have a methodology to hand the case of $C^Q \equiv -1 \pmod p$.

7.2 Crux of the Algorithm: The Repeated Multiplication

We now come up with a generic, yet easy to follow by hand algorithm to compute the square root of a quadratic residue. We do not care whether or not the specific case would have been absorbed in preprocessing. We do however, for this section, make the following two preconditions:

1. Prime p is of the form $p = 4n + 1$
2. Exclude case 3 in Table 4

Our goal is to repeatedly multiply C by perfect integer squares until we arrive at a perfect integer square or its negative modulo p. We provide pseudocode here. We use phrasing from the Front-Loading Conjecture (See Section 6.3)

```
1. INPUT: $p = 4n + 1, C \in QR_p$
2. Initialize $\chi = C, \Psi = 1, \kappa = 1,$
3. While $\chi$ is not a perfect integer square and $-\chi \pmod{p}$ is not a perfect integer square:
    (a) if $\chi$ is not in Quadrant I or II update $\chi \equiv -\chi \pmod{p}, \kappa = -\kappa$
    (b) Select perfect integer square $y_i^2$ such that $\pm\chi y_i^2 \pmod{p}$ has yet to have been assigned
        to $\chi$ and update $\chi \equiv \chi * y_i^2 \pmod{p}, \Psi \equiv \Psi * y_i \pmod{p}$
4. if $-\chi \pmod{p}$ was the perfect integer square, update $\chi \equiv -\chi \pmod{p}, \kappa = -\kappa$
5. Return
```

$$\sqrt{C} \equiv \pm\Psi^{-1}\sqrt{\kappa\chi} \pmod{p} \tag{63}$$

In full detail, we write the computation of $\sqrt{C}$ as

$$(y_1^2 y_2^2 \cdots y_{k-1}^2 y_k^2)C \equiv \sqrt{\chi}^2 \pmod{p} \tag{64}$$

$$(y_1 y_2 \cdots y_{k-1} y_k)\sqrt{C} \equiv \pm\sqrt{\chi}\sqrt{\pm 1} \pmod{p} \tag{65}$$

$$\sqrt{C} \equiv \pm(y_1 y_2 \cdots y_{k-1} y_k)^{-1}\sqrt{\chi}\sqrt{\pm 1} \tag{66}$$

Thus, when this occurs, we can arithmetically find the square root. We now prove that this will always occur given a Quadratic residue C.

Theorem 16. Termination of repeated multiplication

Let $p = 4n + 1$ and $C \in QR_p$. Then there exist a finite sequence of integers y_i such that

$$(y_1^2 y_2^2 \cdots y_{k-1}^2 y_k^2)C \equiv \sqrt{\chi}^2 \pmod{p} \tag{67}$$

We prove this in a few steps. We need a lemma first

Lemma 4. QR closure by integer squares.

Let $p = 4n + 1$, $C_1, C_2 \in QR_p$. Then there exists an integer y_i such that

$$y_i^2 C_1 \equiv C_2 \pmod{p} \tag{68}$$

Proof. From Theorem 3, we know that QR_p is closed. Thus there exists an x such that

$$xC_1 \equiv C_2 \pmod{p}, x, C_1, C_2 \in QR_p. \tag{69}$$

Namely, $x = C_1^{-1}C_2$. By Definition 4, since $x \in QR_p$, there exists a y such that $y^2 \equiv x \pmod{p}$. We substitute this into Equation 69 to get the desired result. □

We now can prove that this overall process will terminate.

Proof. Let $p = 4n + 1$ and $C \in QR_p$. Note that $1 \in QR_p$. Via Lemma 4, one can pick any QR of choice that has not been visited, and find a perfect integer square to visit that QR. Due to Theorems 1 and 3 (namely the fact that $|QR_p|$ is finite), one must visit $\chi = 1$ at some point and the algorithm will terminate. □

Due to the fact that the algorithm only modular multiplication, this algorithm avoids all of the advanced calculations and much of the trial and error that is present in Tonelli-Shanks, Cipolla, and Pocklington's algorithms [8, 15]. Further, the algorithm can be sped up since we are always looking at the value of χ or it's negative as allowed to us by Corollary 4. However, this is dependent on the strategy that one uses to pick y_i^2. The strategy that we have implemented is predicated on the fact that Quadrant I contains more perfect integer squares and the Front-Loading Conjecture. We state at the outset of this discussion:

Conjecture 2. Optimal selection of y_i^2

Select the least integer k such that

$$y_i^2 = \left[\sqrt{\frac{kp}{\chi}}\right]^2, \quad k \in \mathbb{N} \tag{70}$$

corresponds to a $\chi y_i^2 \pmod p$ has yet to be visited in the proceedings of the algorithm and this χy_i^2 is in Quadrants I or IV, where $[x]$ in this case rounds x to the nearest integer.

It is important to note that this χy_i^2 has yet to be visited. If this was not the case, the algorithm would infinitely loop. To avoid this, k can increase as needed, creating a "moving goalposts" approach to finding a perfect integer square that will work. Take $p = 17, C = 8$ for example. The integer square that would be found is $y_1 = 1$, which would obviously cause a loop as χ would never leave 8. Other examples could create a loop that does not include C itself. Illustrating this is a case when $y_m^2 y_{m+1}^2 \cdots y_n^2 \equiv 1 \pmod p$ There are other strategies, but we will discuss them in Section 7.4.

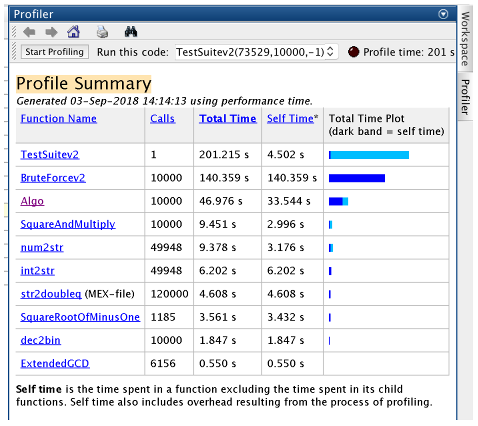
We want to maintain that we can pick y_i^2 quickly. This is the place of Front-Loading. Because there are more Quadratic residues in the 'RICH' half of $\mathbb{Z}_p^\times$, if we pick a y_i^2 randomly, we are more likely to land in Quadrants I and IV. Further, this likelihood increases as ξ, ρ increase. Further, This choice of y_i^2 will help terminate the algorithm quickly, as this will put the corresponding $\pm\chi y_i^2 \pmod p$ as close to $0 \pmod p$ as possible. As the perfect integer squares are concentrated about $0 \pmod p$, this choice of y_i^2 will help facilitate that .

Lastly, we would like to address this constant shifting to quadrants I and II.The larger χ is, the closer $\frac{kp}{\chi}$ is to $k \ \ \forall k$. This will decrease the speed in which we can select y_i^2, as candidate values of k will need to be larger than that of the same k, but with a smaller χ. This shifting is once again allowed to us by Corollary 4 However, on the note of Quadrant II, by how we are choosing y_i^2, once we leave Quadrant II, we will never return. This implies that if we ever use Quadrants II or III, then C is in one of these quadrants.

We noted that since the number of QR's are finite, however this will form the basis of the argument that the number of iterations should be at most $\lceil \frac{n}{2} \rceil$, allowing for the "moving goalposts" to counterbalance the Front-Loading Conjecture.

7.3 Comparing our algorithm to Brute Force

In this section, we discuss the overall effectiveness of our approach and compare it to brute force. We will do this in two ways. In terms of sheer runtime, we will be using MATLAB's profiling tool, as using the tic/toc and cputime functions can be buggy or misleading in our experience. We avoid measuring our algorithm in terms of Big-Oh notation, deferring to practical observables from implemented computation. This is the major portion of the algorithm, as the Square and Multiply algorithm and the Extended Euclidean Algorithm are sublinear in this respect. What you will see in this analysis is a function that we have written called TestSuitev2. This function takes in a prime, a number of random elements in $\mathbb{Z}_p^\times$ to test against brute force, and a trigger quantifying if we only want to test QRs (1), NRs(0), or a random mixture of the two (-1), in this order. We present sample output in Figure 8.



Profile Summary

Generated 03-Sep-2018 14:14:13 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
TestSuitev2	1	201.215 s	4.502 s	
BruteForcev2	10000	140.359 s	140.359 s	
Algo	10000	46.976 s	33.544 s	
SquareAndMultiply	10000	9.451 s	2.996 s	
num2str	49948	9.378 s	3.176 s	
int2str	49948	6.202 s	6.202 s	
str2doubleq (MEX-file)	120000	4.608 s	4.608 s	
SquareRootOfMinusOne	1185	3.561 s	3.432 s	
dec2bin	10000	1.847 s	1.847 s	
ExtendedGCD	6156	0.550 s	0.550 s	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 8: Running 10,000 random trials for $p = 73529$ to see runtime

Due to the segmented nature of preprocessing, we break this into primes of the form $p = 8n+1$ and everything else. We now present $\frac{p-1}{2}$ mixed trials in Tables 5 and 6

This shows that it appears that our algorithm does not run well for small primes, but does well for large primes. In the first case, it appears that our algorithm does better for primes larger than approximately 3500, and approximately 1000 for the second case. If a more concrete number could be found, we would be happy. We then have the following statement

Conjecture 3. Beating brute force with our algorithm.

Let $w \approx 3500$. Our algorithm beats brute force in terms of runtime for a prime p if $p = 4n+1 > w$. This is an empirical result.

p	AlgoRuntime	Brute-force Runtime
17	.010s	.001s
97	.016s	.002s
617	.047s	.018s
977	.081s	.043s
1361	.115s	.080s
2377	.260s	.246s
3041	.378s	.414s
3313	.484s	.482s
4721	.594s	1.023s

Table 5: Comparing runtimes for $1 \pmod 8$ primes

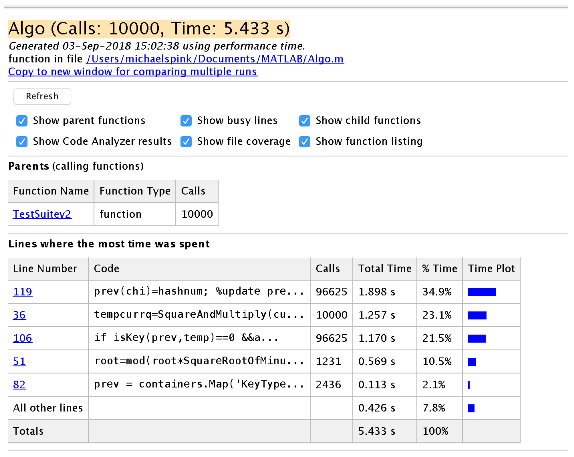
p	AlgoRuntime	Brute-force Runtime
13	.009	.001s
101	.013s	.002s
283	.017s	.006s
503	.026s	.013s
1061	.054s	.050s
1999	.090s	.175s
2029	.096s	.184s
6053	.333s	1.646s

Table 6: Comparing runtimes of other primes

Thus, the sheer number of possibilities appear to hamper brute force compared to the algorithm and this selection of y_i^2. This shows that the algorithm is not perfect for every $4n + 1$ prime, but does indeed beat brute force if p is large enough.

We believe that there are two major things that slow the algorithm down. These are the square and multiply algorithm and dictionary use in MATLAB. Recall that the Square and Multiply algorithm should be running sublinearly. If we return to examining Figure 8, let us look at the most time consuming steps of the algorithm in Figure 9 (this was for $p = 73529 \equiv 1 \pmod 8$). Note that we call our dictionary of previously visited values "prev".

We see of the five most consuming steps are visiting a value χ, the Square And Multiply Algorithm, checking if we have visited χ or its negative, and creating the dictionary itself. These steps relating to the dictionary should run in constant time, which does not seem to happen here. Thus, we think that implementing our algorithm in a lower level language such as Python, Java, or C would alleviate these issues. We state this as a conjecture here



Algo (Calls: 10000, Time: 5.433 s)

Generated 03-Sep-2018 15:02:38 using performance time.

function in file /Users/michaelspink/Documents/MATLAB/Algo.m

Copy to new window for comparing multiple runs

Refresh

[✓] Show parent functions [✓] Show busy lines [✓] Show child functions

[✓] Show Code Analyzer results [✓] Show file coverage [✓] Show function listing

Parents (calling functions)

Function Name	Function Type	Calls
TestSuitev2	function	10000

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
119	`prev(chi)=hashnum; %update pre...`	96625	1.898 s	34.9%	▬
36	`tempcurrq=SquareAndMultiply(cu...`	10000	1.257 s	23.1%	▬
106	`if isKey(prev,temp)==0 &&a...`	96625	1.170 s	21.5%	▬
51	`root=mod(root*SquareRootOfMinu...`	1231	0.569 s	10.5%	■
82	`prev = containers.Map('KeyType...`	2436	0.113 s	2.1%	▏
All other lines			0.426 s	7.8%	■
Totals			5.433 s	100%	

Figure 9: The most time consuming steps of the algorithm

Conjecture 4. Lower level language implementation.

Implementing our algorithm in a lower level language will reduce the value of w in Conjecture 3

When it comes to the Square and Multiply algorithm, most of the runtime is spent converting a number into binary. This is done via the $str2double$ command (we have implemented str2doubleq to make things a little better [4])

Supplementing this argument is an argument on the average number of iterations. This is a rather nebulous term to define. We do this for brute force and our algorithm below. In general, we are incrementing this iteration counter whenever a for/while loop, well, loops. We ignore preprocessing for our algorithm because, again, it should run sublinearly due to the square and Multiply Algorithm and repeated squaring. Recall that C is our candidate Quadratic residue. For this discussion, we assume that C is indeed a quadratic residue.

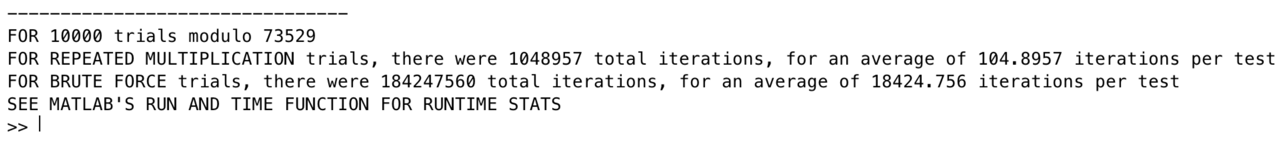
Definition 16. Iteration for brute force

We define one iteration for brute force as every integer that we square and compare to C.

Definition 17. Iteration for our algorithm

We define one iteration for our algorithm as every instance we compute an invalid y_i^2, and every time we update our bookkeeping for Ψ.

Note that we are not counting the bookkeeping if we need to flip κ and χ to $-\kappa$ and $-\chi$ as an iteration, as this is done in constant time. Further, we note that these iterations usually involve three modular operations and two dictionary pulls (ideally in $\mathcal{O}(1)$ time) each, but this does not change the argument in the grand scheme of things. We are using our own built function to compute this. We also have built our own function that performs just the repeated multiplication step of the algorithm. As before, we show one sample output in Figure 10,



```
-------------------------------
FOR 10000 trials modulo 73529
FOR REPEATED MULTIPLICATION trials, there were 1048957 total iterations, for an average of 104.8957 iterations per test
FOR BRUTE FORCE trials, there were 184247560 total iterations, for an average of 18424.756 iterations per test
SEE MATLAB'S RUN AND TIME FUNCTION FOR RUNTIME STATS
>> |
```

Figure 10: Running 10,000 random trials for $p = 73529$ to see iteration count

before extrapolating this and showing results for various primes in Table 7. We do not care of the form of the prime other than it must be of the form $4n + 1$.

p	Total Algo iters	Average Algo iters	Total Brute-force iters	Total Brute-force iters
13	6	1	8	1.3333
17	6	0.75	35	4.375
97	184	3.8333	1200	25
101	275	5.5	1067	21.34
617	2581	8.3799	48927	158.8539
977	6240	12.7869	116519	238.7684
1999	44716	44.7608	505815	506.3213
1361	8061	11.8544	232521	341.9426
2377	38752	32.6195	693200	583.5017
3041	36637	24.1033	1168862	768.9882
4721	65538	27.7703	2784188	1179.7407
6053	128021	42.307	4691574	1550.421

Table 7: Comparing iterations for various $4n + 1$ primes

This shows a few things. Most importantly, the total number of iterations seems to be far less than that of brute force. This forms our heuristic argument that if implemented in a lower level language, our algorithm should be able to beat brute force for smaller primes, as well as larger ones. We graph these average iteration count for brute force and repeated multiplication in Figure 11 to strengthen this. Notice how small the average number of iterations grows incredibly slowly for repeated multiplication.

Further, we expected brute force to be average out to about $\frac{p+1}{4}$ iterations due to the definition of expected value. If check for $C = 1$, this comes down to $\frac{p-1}{4} = n$. This is heuristically confirmed by Table 7. We hypothesized without proof that repeated multiplication would take about $\lceil \frac{n}{2} - \sqrt{\frac{n}{2}} \rceil$

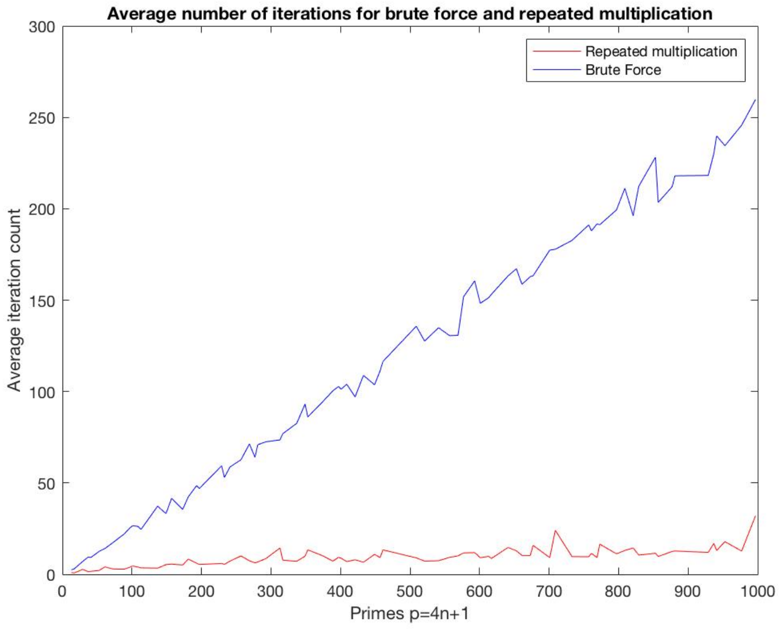


Figure 11: Average number of iterations for brute force vs. repeated multiplication

number of updates to Ψ. This was based on the number of non perfect integer square quadratic residues, with the front loading conjecture allowing us to select y_i^2 without much issue. However, especially for large primes, our algorithm seems to perform much better than this.

Taking these two measurement methods into account, measuring by runtime shows that our algorithm beats brute force in terms of runtime for large primes. Conjecture 4 will seem to be our out to make our algorithm beat brute force in all cases. Measuring in terms of iterations seems much more promising to showing that our algorithm is better than brute force.

The question now becomes if there are ways to improve the design itself of the algorithm. We will see that we do not believe that there is a trivial way to do so, so our algorithm is ideal to find the modular square root when it exists, and to decide upon the quadratic character of a number quickly as well.

7.4 Improving the Design of the Algorithm

Further, there are two ways to try improve the algorithm that we see from a design perspective. The first way to improve the algorithm's design is to attempt to better incorporate the preprocessing steps into the design of the algorithm. Let us first try to fix preprocessing before trying to replace it. As the iteration comparisons are related to repeated multiplication exclusively, we will use runtime to make the assertions we will for this improvement.

Perhaps, it would be faster to just compute Euler's Criterion and run repeated multiplication on the QR's. This however, is not shown by the empirical data. We show this in Table 8, which takes both decidability and removing these easy cases by using a mixture of QR's and NR's.

p	Algo Runtime	Euler's Criterion Runtime
17	.008	.009
97	.017	.014
101	.013	.014
617	.049	.059
977	.090	.104
1361	.119	.141
6053	.338	1.153

Table 8: The runtime effects of just using Euler's Criterion

This gap seems small for small primes, but grows as p increases. We believe that ruling out the easy cases of QR's rather than making the effort of engaging repeated multiplication gives our preprocessing the edge here.

A less trivial attempt that we have made is an approach inspired by the Collatz Conjecture. One can find its implementation in Section 10 for those looking for details on how it works. Basically, it is built on many subcases and driving C down to a perfect integer square. However, empirical data is not on its side either. See Table 9.

For smaller primes, this approach seems close to/sometimes beating brute force. However, as p increases, the regular version of preprocessing takes the lead. However, all is not lost for this Collatz Preprocessing. One major advantage that it has to our preprocessing is that overflow is less likely here. For extremely large primes, on the order of 10^{10}, is some computations might be rounded incorrectly, especially in computing $C^Q \pmod{p}$. To this end, this Collatz interpretation can deal with larger primes easier, as it never allows values to leave $\mathbb{Z}_p^\times$. .

p	Algo Runtime	Collatz Version Runtime
17	.012	.009
97	.017	.012
101	.012	.014
617	.050	.042
977	.092	.087
1361	.116	.120
6053	.343	.949

Table 9: The runtime effects of just using a Collatz approach to preprocessing

We also came up with a method that performs a depth first search between integers that have the quadratic character, however, this renders the transparency argument moot, the runtime won't probably be improved much due to having to generate the graph and then perform depth first search and we found it difficult to implement efficiently.

The reader may have noticed that the preprocessing appears to be independent of the repeated multiplication.This is due to the fact that none of the decidability conditions we discovered for the repeated multiplication itself are feasible at a fast runtime or number of iterations. The first of these attempts is to find an upper bound for y_i^2. We note that

Theorem 17. Consecutive set of QR_p
Let $p = 4n + 1$ be prime. Then $QR_p = \{1^2, 2^2, 3^2 \cdots (2n-1)^2, (2n)^2\}$

Proof. Let $p = 4n + 1$ and $1 \leq \alpha, \beta \leq 2n$. Further, assume $\alpha^2 \equiv \beta^2 \pmod{p}$. Then

$$\alpha^2 - \beta^2 = kp, k \in \mathbb{Z} \tag{71}$$

Thus,

$$p \mid (\alpha - \beta)(\alpha + \beta) \tag{72}$$

By the prime property $p \mid (\alpha - \beta)$ or $p \mid (\alpha + \beta)$.
Assume $p \mid (\alpha + \beta)$. Due to the fact that $\alpha, \beta \leq 2n$ means that this can be at most $p - 1$. By this and the lower bounds of α, β, we can say that

$$1 + 1 \leq |\alpha + \beta| < p \tag{73}$$

And note that no multiple of p lies in this interval. This contradicts the fact that $p \mid (\alpha + \beta)$. Let us now consider the other possibility. By the same logic, we can say

$$0 \leq |\alpha - \beta| < p \tag{74}$$

Thus, the only way for p to divide $|\alpha - \beta|$ if if

$$|\alpha - \beta| = 0, \quad \rightarrow \quad \alpha = \beta \tag{75}$$

this means that for all QR's generated by α^2 are uniquely defined for $1 \leq \alpha \leq 2n$. This provides $2n$ total QR's in this interval. Theorem 1, also gives that there are $2n$ QR's total. Thus, by the pigeonhole principle, we have the expected result. $\square$

This means that the maximum y_i^2 should be bounded above by $(2n)^2$. However, in order to effectively implement this, one would have to ensure that they check all possible y_i^2 for validity. Note how we have not done that in our strategy of picking y_i^2. Perhaps the algorithm just skips over the squares it needs to find one that works, which could result in false negatives in terms of decidability, and enforcing this would amount to brute force. Further, to get to get to this value of y_i, k will have to be quite large. In effect, the algorithm will have to bounce to most of the NR's to avoid hitting values in QR_p that we can bounce to. In essence, why not just scan through the set $\{1^2, 2^2 \cdots (2n-1)^2, (2n)^2\}$ until one finds the multiplicative inverse? This results in the same expected number of iterations as what we computed above. Thus, this condition is not feasible to implement.

To this end, what if we put an upper bound on the number of updates to Ψ? We discussed above how we expected the worst number of updates (iterations to account for Front-Loading and shifting of k) was $\lceil \frac{n}{2} - \sqrt{\frac{n}{2}} \rceil$. If we have a non residue, the maximum amount of times the algorithm could update was $\frac{n}{2}$ times. However, this does not account for Front-Loading, and even if we could, ρ could vary widely enough with p that the adjustment we could make to this $\frac{n}{2}$ is not consistent across all primes. The other problem with this approach is that getting to this bound is going to become more and more difficult as the number of updates increases. Recall that Front-Loading betters the quadratic residues after all. We run into the same problem bonding y_i^2, and thus this fix is not feasible to implement either.

The last obvious idea that could be implemented into the repeated multiplication is to find a non residue, and if it is hit during the repeated multiplication step conclude that C is a non residue mod p. There are two major problems with this. The first is that knowing just one nonresidue that the algorithm is trying to hit is equivalent to the previous "solution" to the problem. The second is that finding this nonresidue can be costly, and any hope of it not being so depends on the Generalized Riemann Hypothesis [10]. Mixing expanding this search to more than one nonresidue while running the algorithm is difficult to implement. Perhaps a better way to do this exists with quantum computers. This method is also not feasible to implement.

Thus, we believe that our form of preprocessing is justified. Let us turn our attention to the last thing that we can change: the strategy with which we select y_i^2. However, coming up with another non trivial way to select this has proven to be difficult. Thus, we present a greedy strategy, and a thorough one.

Let us discuss this thorough one first. We were discussion earlier the bound for y_i^2 that we would need to search all possible integer squares to ensure we haven't missed one that could work. Here we try the same thing in order to directly minimize the value of y_i^2. We search through all possible QR's that would put χy_i^2 in Quadrants I and IV, starting from $0 \pmod{p}$ and working outwards. We know that the multiplicative inverse of C is in this group, however this searching procedure is comparable to brute force. We could also just use the least y_i^2 that puts us in Quadrants I or IV, but we run into the same searching problem that the above technique runs into, with added problems due to having to check those that would not put us in the right quadrant. Thus, we believe that our strategy for picking y_i^2 is a valid one. This allows our algorithm to remain justified as is from a design standpoint.

7.5 Simple working examples of repeated multiplication

In this section we provide two examples of the execution of the repeated squares portion of our algorithm. We leave many of the implementation details, namely, how we are computing integer squares to multiply by, to be explored by the reader by looking at Section 7 and Section 10.
Importantly, we are ignoring preprocessing as this is rather self explanatory to trace other than notes to which case these instances fall into. Recall from Section 7.2 the meanings of χ, Ψ, κ.

1. FIND THE SQUARE ROOTS OF $26 \pmod{37}$

   - NOTE: if preprocessing applied, this would fall into case one, as

     $$37 - 1 = 9 * 2^2 \tag{76}$$

     So Q=9. And we see that

     $$26^9 \equiv 1 \pmod{37} \tag{77}$$

     So

     $$\sqrt{26} \equiv (26^4)^{-1} \equiv 10 \pmod{37} \tag{78}$$

     We will be ignoring this and proceeding anyway into the repeated multiplication step of the algorithm.

   - Note that

     $$37 = 1^2 + 6^2 = 4(9) + 1 \tag{79}$$

     It is easily verifiable that 37 is prime. If necessary, we also now know that

     $$\sqrt{-1} \equiv 6 * 1^{-1} \equiv 6 \pmod{37} \tag{80}$$

   - This means that our four quadrants are [1,9],[10,18],[19,27][28,36]. So we will be trying to force the current value between 28 and 9 $\pmod{37}$ each iteration

- Begin with χ=26; Ψ=1; κ=1; . Note that $26, 11$ are not perfect integer squares.

  We see that 26 is in Quadrant III, so we look at $-26 \equiv 11 \pmod{37}$ and multiply sign by -1.

  We have: χ=11; Ψ=1; κ=-1;

- Observe that since

$$2^2 = \left[\sqrt{\frac{2*37}{11}}\right]^2 \tag{81}$$

  we can write

$$2^2 * 11 \equiv 44 \equiv 7 \pmod{37} \tag{82}$$

  . This is in Quadrant I, so we do not have to look at $-7 \pmod{37}$ other than the fact that 7 and 30 are not perfect integer squares. We multiply Ψ by 2.

  We have: χ=7; Ψ=2; κ=-1;

- Observe that since

$$2^2 = \left[\sqrt{\frac{37}{7}}\right]^2 \tag{83}$$

  we can write

$$2^2 * 7 = 28 \pmod{37} \tag{84}$$

  . This is in Quadrant IV, so we look at $-28 \equiv 9 \pmod{47}$. 9 is indeed a perfect integer square. We multiply Ψ by 2 and κ by -1.

  We have χ=9; Ψ=4; κ=1;

- Our loop terminates, and our algorithm now computes the solution:

$$16 * 26 \equiv 9 \pmod{37} \tag{85}$$

$$4\sqrt{26} \equiv \pm 3 \pmod{37} \tag{86}$$

$$\sqrt{26} \equiv \pm 4^{-1} 3 \pmod{37} \tag{87}$$

$$\sqrt{26} \equiv \pm 27 \pmod{37} \tag{88}$$

And indeed one can use the square and multiply algorithm to confirm that $10^2 \equiv 100 \equiv 26 \pmod{37}$

2. FIND THE SQUARE ROOTS OF $5 \pmod{73529}$.

- Note that if preprocessing applied, it would fall into case 2, as

$$73529 - 1 = 9191 * 2^3 \tag{89}$$

So

$$5^{9191} \equiv -1 \pmod{73529} \tag{90}$$

So Q=9191. And we see that

$$\sqrt{5} \equiv (5^{4595})^{-1}\sqrt{-1} \equiv 4782 \pmod{73529} \tag{91}$$

We will be ignoring this and proceeding anyway into the repeated multiplication step of the algorithm.

- Note that $73529 = 77^2 + 260^2 = 4(18382) + 1$. It can be verified that 73529 is prime. If necessary, we also know that

$$\sqrt{-1} = 260 * 77^{-1} \equiv 260 * 12414 \equiv 65893 \tag{92}$$

Note the use of the Extended Euclidean Algorithm.

- This means that our four quadrants are [1,18382],[18383,36764],[36765,55146],[55147,73528]. So we will be trying to force the current value to be between 55147 and 18382 each iteration.

- Begin with χ=5; Ψ=1; κ=1. Note that 5 is clearly in Quadrant I, and neither 5 nor 73524 are perfect integer squares.

- Observe that since

$$121^2 = \left[\sqrt{\frac{73529}{5}}\right]^2 \tag{93}$$

we can write

$$121^2 * 5 \equiv 73205 \pmod{73529} \tag{94}$$

This is in Quadrant IV, so we look at $-73205 \equiv 18^2 \pmod{73529}$. 324 is indeed a perfect integer square. We multiply Ψ by 121 and κ by -1.
We have χ=324; Ψ=121; κ=-1;

- Our loop terminates, and our algorithm now computes the solution:

$$121\sqrt{5} \equiv 18\sqrt{-1} \pmod{73529} \tag{95}$$

$$\sqrt{5} \equiv 54691 * 18 * 65893 \equiv 68747 \pmod{73529} \tag{96}$$

And indeed one can use the square and multiply algorithm to confirm that $68747^2 \equiv 5 \pmod{73529}$

8 Future work

Our work is not perfect, as simple running through the possible areas of our algorithm could still be refactored to run faster. To this end, we present a few open questions that are relevant to our algorithm presented here. We discuss most of these in Section 7.3 and Section 7.4

1. In the event that are multiple possible integer squares to multiply a current value by, is there a better way to select one that will terminate the algorithm faster?

2. In the repeated multiplication step of the algorithm, is there a more feasibly implementable NR decidability condition?

3. Will running our algorithm in a lower level language drastically decrease the runtime of the algorithm with respect to brute force? Particularly, we contemplate this in regards to the speed of MATLAB's map.container class. Ideally, dictionaries should run in $\mathcal{O}(1)$ time

4. Do certain primes or values of C run the repeated multiplication section of the algorithm faster? We think this may depend on ξ and ρ.

We now look at our work more globally. Similar to how we looked at the partition of QR_p using our algorithm, we are curious if repeatedly applying Ω to a given initial 2×1 vector will create anything meaningful. That is, we want to look at the orbits of this repeated use of Ω. With respect to this iteration, we are curious if a dynamic programming approach could be useful in the case where multiple square roots need to be found. Lastly, could shifting the problem another way yield better results?

Another idea for an algorithm that we had was just directly using the tables in Section 6.2 was to see if we can get any entry in the table from just the top left entry, which is always 0. We wonder if this is feasible at a runtime that beats brute force. Are there any other strategies about information to grab as we move about the symplectic manifold?

Since we mentioned these throughout this thesis, could more results from Symplectic/differential geometry, analytic number theory, or discrete math expand any results we have presented here?

Lastly, we present the obvious request of "prove the front loading conjecture please." We strongly believe it to be true, but have yet to complete a proof for it.

9 Conclusion

In closing, we think that this interpretation of the square root problem will help make the problem more tractable by the fact that it should now be much easier to compute by hand. This transparency arises from transforming the problem into a geometric problem built upon a simple shifting of the problem.

This approach pulls from so many areas of math rather than just number theory, making things more usable to aid in solving the problem at hand, which could solve much harder problems such as RSA Factoring. and many other applications of quadratic reciprocity.

While the algorithm not perfect, we hope that implementing the algorithm and all of its children functions in a lower level language can bring the run time down to one comparable to that of running brute force, as shown by the vast improvement in terms of iterations, as well as potentially using dictionaries in this lower level language could make the algorithm better for smaller primes. Formally proving the front loading conjecture would also open so many windows for the problem, as it could make decidability conditions other than Euler's Criterion viable in the algorithm or increasing the number of usable primes before machine overflow.

All things considered, we are very proud of our approach and algorithm and hope to improve upon it/see it improved upon to make a very difficult problem have the sense of purpose.

10 Appendix 1: Code

This appendix gives the source code that we used for our MATLAB simulations. To keep this appendix from being ridiculously long, we only include essentials. Go to my Github page for this code and miscellaneous MATLAB testing files. In all of the following, the comment 'EOF' denotes the end of the file in question.

What follows is code for the algorithm described in Section 7. This is the version that utilizes preprocessing to solve 2 easier QR cases and determine NR's, and the repeated multiplication step of the algorithm.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                              %%
%% File: Algo.m                                 %%
%% Internal Filename: algov11.m                 %%
%%                                              %%
%% Author: Michael R. Spink                     %%
%% Author: Manuel Lopez                         %%
%%                                              %%
%% This file runs the algorithm we developed    %%
%% WITH careful square selection                %%
%%                                              %%
%%  input p, a prime to work with (Z_p)         %%
%%  input c, a value in Z_p for the program     %%
%%           to find its square root,           %%
%%           if it exists                       %%
%%                                              %%
%%  output root, one of two square roots,       %%
%%               0 if NR                        %%
%%                                              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function root =Algo(p,C)

%%%%%%%%%%%%%%%%%%%% INITIALIZE VALUES %%%%%%%%%%%%%%%%%%
n=(p-1)/4; % compute n, p=4n+1
curr=C; %set current value
%%%%%%%%%%%%%%%%%% PREPROCESSING %%%%%%%%%%%%%%%%%%%%%%%%%%
%write p-1=Q*2^s
Q=p-1;
s=0;
while mod(Q,2)==0
    Q=Q/2;
    s=s+1;
end
tempcurrq=SquareAndMultiply(curr,(Q-1)/2,p); %compute ans if in path 1/2
currq=mod(tempcurrq^2*curr,p);%% curr Curr^Q
twopow=1; % power of two currenrtly at

%% COMPUTE IF LIFE IN THE FAST LANE
if currq==1 % curr^odd=1  // PATH ONE
    [root,~,~]=ExtendedGCD(tempcurrq,p); % compute root
    root=mod(root,p);
    return
elseif currq==p-1 %curr^odd =-1 // PATH TWO
    if mod(p,4)==3
        root=0;
        return
    end
```

```
        [root,~,~]=ExtendedGCD(tempcurrq,p);
        root=mod(root*SquareRootOfMinusOne(p),p);
        return
    else %life is hard
        if mod(p,8)==5
            root=0;
            return
        end
        rootminusone=currq; %sqrt(-1) is two iterations back
        previous=currq;
        while(twopow<s)
            currq=mod(currq^2,p); %repeated squaring
            if currq==1 %if QR
                break;
            end
            rootminusone=previous; %update values if QR
            previous=currq;
            twopow=twopow+1;
        end
        if currq ==p-1 % IF NR, PRINT RESULTS
            root=0;
            return
        end
    end
    %% END PREPROCESSING %%%%%%%%%%%%%%

    %%%%%%%% REPEATED MULTIPLICATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %% INITALIZE VALUES FOR MULTIPLICATION
    chi=C; %value that we are currently at
    kappa=1; %compute number of sign switches
    psi=1; %compute multiple
    %create previous visited list
    prev = containers.Map('KeyType','int32','ValueType','int32');
    prev(chi)=1; %% visit curr
    hashnum=2; %to try to speed up lookup
    while true  % MAIN ALGO LOOP
        if floor(sqrt(chi))==sqrt(chi) %check if integer square
            break; %WE'RE DONE HERE
        end
        temp=p-chi;
        if floor(sqrt(temp))==sqrt(temp) %check if negative is integer square
            chi=p-chi; %FLIP
            kappa=kappa*-1; %UPDATE
            break; %WE'RE DONE HERE
        end
        if chi>2*n % FLIP IF IN Q3,Q4
            chi=p-chi;
            kappa=kappa*-1;
        end
        %% DETERMINE VALUE TO MULTIPLY BY
        pmult=1; %what target are we aiming for?
        while(true)
            sq=(round(sqrt((pmult*p)/chi)))^2; %compute possible square
            temp=mod(sq*chi,p); %compute temp
              %check if we've been here before
            if temp>3*n|| temp<=n
                if isKey(prev,temp)==0 && isKey(prev,p-temp)==0
                    break
                end
            end
            % other wise update to next target and increment
            pmult=pmult+1;
        end
        chi=temp;% determine next value
        if(chi>2*n) %flip if need be
            chi=p-chi;
            kappa=kappa*-1;
        end
        psi=mod(psi*sqrt(sq),p);  %update
        prev(chi)=hashnum; %update previously visited values
        hashnum=hashnum+1;
    end
    if kappa==1
         rootminusone=1;
    end
    [inv,~,~]=ExtendedGCD(psi,p); %compute multiplicative inv
    inv=mod(inv,p); %invert
    root=mod(inv*kappa*sqrt(chi)*rootminusone,p);  %compute root
    return

    %%% EOF
```

The algorithm relies on implementations for the Extended Euclidean Algorithm for quick modular inversion (we use a matrix method presented to me by Dr. Anurag Agarwal [1]), the square and multiply algorithm in preprocessing and testing (we use [4] to speed up the algorithm on MATHLAB 2018a), and decomposing a $4n + 1$ prime into the sum of two squares to find the square root of -1 in the style of Zagier when path 2 is hit [14]. In testing variations of the algorithm that do not use preprocessing, we compute the $p = a^2 + b^2$ decomposition, yet save inverting one of these variables until we have to, combined with other parts of the problem. See Section 7 for more information.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                         %%
%% File: SquareRootOfMinusOne.m                            %%
%%                                                         %%
%% Author: Michael R. Spink                                %%
%%                                                         %%
%% This file computes the square root of                   %%
%% -1 modulo a given 4n+1 prime                            %%
%%                                                         %%
%% INPUTS:                                                 %%
%%    @input p                                             %%
%%            a is any 4n+1 prime                          %%
%%                                                         %%
%%    Returns:                                             %%
%%      @return root                                       %%
%%                a=  sqrt(-1) (mod p)                     %%
%%                                                         %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function root=SquareRootOfMinusOne(p)

x=1;
y=1;
z=(p-1)/4;
% X=['x=',num2str(x),' y=', num2str(y),' z=', num2str(z)];
%disp(X)
while(true)
    if z+x<y
        %disp('OPTION 1')
        tempx= x+2*z;
        tempy= y-z-x;
        tempz= z;
    else
        %disp('OPTION 2')
        tempx= 2*y-x;
        tempy= z+x-y;
```

```
            tempz= y;
        end
        x=tempx;
        y=tempy;
        z=tempz;
        %X=['x=',num2str(x),'  y=', num2str(y),'  z=', num2str(z)];
        %disp(X)
        %pause(3)
        if y==z
            b=2*y;
            a=x;
            %X=[num2str(p),'=',num2str(a),'^2+',num2str(b),'^2'];
            %disp(X)
            break
        end
end
  [binv,~,~]=ExtendedGCD(mod(b,p),p);
root=mod(a*mod(binv,p),p);
%X=['root=',num2str(root)];
%disp(X)
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                    %%
%% File: extendedGCD.m                                %%
%%                                                    %%
%% Author: Michael R. Spink                           %%
%%                                                    %%
%% This file computes the gcd of two                  %%
%%  positive integers using a matrix method           %%
%%  to reduce workload on myself and                  %%
%%  potentially runtime if I have                     %%
%%  implemented this well in the time that            %%
%%   I have. Written for CS462                        %%
%%                                                    %%
%% INPUTS:                                            %%
%%    @input a                                        %%
%%             a is any positive integer              %%
%%    @input b                                        %%
%%             b is any positive integer              %%
%%                                                    %%
%%    Preconditions:                                  %%
%%           ideally you want a>b, though             %%
%%            this should be fine otherwise           %%
%%            I have not tested this though           %%
%%                                                    %%
%%    Returns:                                        %%
%%       @return d                                    %%
%%               d=gcd(a,b)>0 by assumption           %%
%%       @return s                                    %%
%%            s=integer associated with a             %%
%%       @return t                                    %%
%%            t=integer associated with b             %%
%%         NOTE: d=sa+tb                              %%
%%                                                    %%
%%                                                    %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



 function[s,t,d] = ExtendedGCD(a,b)
A=zeros(2,3);  %% INITIALIZE MATRIX
%% SET INITIAL VALUES OF MATRIX
A(1,1)=a;
A(2,1)=b;
A(1,2)=1;
A(2,3)=1;
%%
while(A(2,1)~=0 &&A(1,1)~=0)  %% MAIN LOOP; while recursion can occur;
     %%% sanity check to reduce runtime
```

```
    if(A(2,1)==A(1,1))
         %%% SET APPROPRIATE VALUES
        d=A(2,1);
        s=A(2,2);
        t=A(2,3);
        return; %% RETURN TO BASE
    end
    if(A(2,1)<A(1,1)) %% CHECK WHICH VALUE IN MATRIX IS BIGGER
          q=floor(A(1,1)/A(2,1)); %% COMPUTE q in division algo; a=bq+r
          A(1,1)=mod(A(1,1),A(2,1)); %% ''CHEAT'' to compute r;
                                     %%; this is in pseudocode given
          A(1,2)=A(1,2)-q*A(2,2);  %% update s
          A(1,3)=A(1,3)-q*A(2,3);  %% update t
    elseif(A(2,1)==0&&A(1,1)==0) %% second sanity check to reduce runtime
        %%% but really, more like error checking

        %% SET VALUES
        d=A(2,1);
        s=A(2,2);
        t=A(2,3);
        return; %% RETURN TO BASE
    else              %%% (A(2,1)>A(1,1)
        q=floor(A(2,1)/A(1,1)); % compute q in division algo
        A(2,1)=mod(A(2,1),A(1,1)); %% ''cheat to find r. Is in code given
        A(2,2)=A(2,2)-q*A(1,2);  %% update s
        A(2,3)=A(2,3)-q*A(1,3); %% update t
    end
end

if(A(1,1)==0) %% determine which row to use; use row 2 here
    %% SET VALUES
    d=A(2,1);
    s=A(2,2);
    t=A(2,3);
else  %% use row 1r
    %% SET VALUES
    d=A(1,1);
    s=A(1,2);

    t=A(1,3);

end

%%%%% FOR TESTING PURPOSES

 %%% EOF
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                              %%
%% File: SquareAndMultiply.m                    %%
%%                                              %%
%% Author: Michael R. Spink                     %%
%%                                              %%
%% This file runs the                           %%
%%  square and multiply algorithm (a^b mod n)%%
%%                                              %%
%%    Originally written for CS461              %%
%%                                              %%
%% INPUTS:                                      %%
%%    @input a                                  %%
%%            a is any positive integer         %%
%%    @input b                                  %%
%%           b is any positive integer (exp)    %%
%%    @input n                                  %%
%%            n is any positive integer (mod)   %%
%%                                              %%
%%    Returns:                                  %%
%%       @return a                              %%
%%              a=  result of algo              %%
%%                                              %%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function a = SquareAndMultiply(a,b,n)
b=dec2bin(b); %convert to binary
len=length(b);
orig=a;
i=2;
while(i<len+1)
    a=mod((a^2),n);
    currbit=str2double(b(i));
    if(currbit == 1)
        a=mod((a*orig),n);
    end
    i=i+1;
end

%%%%%%%%%%%%%%%%%% EOF
```

We also include the Collatz interpretation of preprocessing from Section 7.4 here.

```

 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 %%                                                 %%
 %% File: Collatz_QR_NR_02.m                        %%
 %%                                                 %%
 %% Author: Manuel Lopez                            %%
 %% Author: Michael R. Spink                        %%
 %%                                                 %%
 %% This file tries to determine if a value         %%
 %%   is a QR or NR mod p. Better than Euler?       %%
 %%                                                 %%
 %%  input p, a prime to work with (Z_p)            %%
 %%  input C, a value in Z_p for the program        %%
 %%           to find its square root,              %%
 %%           if it exists                          %%
 %%                                                 %%
 %%  output tst, 1 if C is a QR,                    %%
 %%              p-1 if NR                          %%
 %%                                                 %%
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function tst =Collatz_QR_NR_02(p,C)
if C > p/2 && C < p %% IF IN Q3, Q4
  C = p-C; %FLIP
end
d = C;
tst = p-1;
if mod(p,8)==1 % if 1 mod 8
   for i=1:(p-1)/4
      if floor(d/2)==d/2
         d = d/2;
      else
         d = (p-d)/2;
      end
      if d == C
         tst = p-1;
         break;
      elseif floor(sqrt(d))==sqrt(d)
         tst = 1;
         break;
      end
   end
elseif mod(p,8)==5
    if floor(C/2)==C/2
       e = C/2;
    else
       e = (p-C)/2;
    end
    if floor(sqrt(e))==sqrt(e)
       tst = p-1;
    end
    for i=1:(p-1)/4
        if floor(e/2)==e/2
            d = e/2;
        else
            d = (p-e)/2;
        end
        if floor(sqrt(d))==sqrt(d)
            tst = 1;
            break;
        end
        if floor(d/2)==d/2
            e = d/2;
        else
            e = (p-d)/2;
        end
        if floor(sqrt(e))==sqrt(e)
            tst = p-1;
            break;
        end
    end
end
```

The following code is the brute force algorithm that we compared to in Section 7.3

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                        %%
%% File: BruteForce.m                                     %%
%% Internal Filename: BruteForcev2.m                      %%
%%                                                        %%
%% Author: Michael R. Spink                               %%
%% Author: Manuel Lopez                                   %%
%%                                                        %%
%% This file brute forces the square root                 %%
%% problem for given values                               %%
%%                                                        %%
%%  input p, a prime to work with (Z_p)                   %%
%%  input c, a value in Z_p for the program               %%
%%           to find its square root,                     %%
%%           if it exists                                 %%
%%                                                        %%
%%  output root, one of two square roots,                 %%
%%               0 if NR                                  %%
%%                                                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function root = BruteForcev2(p,c)
%Was going to exclude c=1, root=1, but it's faster this way somehow
A=randperm((p-1)/2,(p-1)/2); %randomize elements in Z_p
for root=A %pick element
        temp=root; %create temporary
        temp=mod(temp^2,p); %square and mod
        if temp==c %check if winner
            return
        end
end
root=0; %is NR, report as such
return

%%%% EOF
```

Lastly, here is one of the functions that we used to get the results seen in Section 7.3. This file tested total runtime. The function that tested for iterations requires slight changes to this file, the algorithm, and the brute force file. Note the use of internal filenames for my files.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                     %%
%% File: TestSuite.m                                   %%
%% Internal Filename: TestSuitev2.m                    %%
%%                                                     %%
%% Author: Michael R. Spink                            %%
%% Author: Manuel Lopez                                %%
%%                                                     %%
%% This file tests brute force and our                 %%
%%   algorithm to compare iteration counts             %%
%%   and the time it takes these                       %%
%%   files to run                                      %%
%%                                                     %%
%%  input p, a 4n+1 prime to test                      %%
%%  input number, the number of                        %%
%%                   trials to run                     %%
%%  input QROnlyTrigger, kind of test to run           %%
%%                          0- NR's only               %%
%%                          1- QR's only               %%
%%                          else- random mixture       %%
%%                                                     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [] =TestSuitev2(p,number,QROnlyTrigger)
clc
X=['RUNNING: TestSuitev2(',num2str(p),',',num2str(number),',' ...
    ,num2str(QROnlyTrigger),')'];
disp(X) %display what user inputted.
disp('------------------------------------')
wrong=0; %number of wrong answers
answers=[]; %array of answers to be compared against
wrongseeds=[]; %wrong answers
trials=0; %number of trials.
ar=randperm(p-1,number); %choose number random integers on [1,p-1]
disp('BRUTE FORCE') %run brute force
for i=1:numel(ar)
    C=ar(i);
    if QROnlyTrigger==0 %ensure NR
        while SquareAndMultiply(C,(p-1)/2,p)==1
            C=mod(C+1,p);
        end
    end %ensure QR
    if QROnlyTrigger==1
        C=mod(C^2,p);
    end
    elm=BruteForcev2(p,C);
    if elm==0
        W=[num2str(C),' is an NR (mod ',num2str(p),').'];
        disp(W)
    else
        W=[num2str(elm),'^2=',num2str(C),'(mod ',num2str(p),').'];
        disp(W)
    end
    answers=[answers, elm];
end
disp('----------------------------------------')
disp('ALGORITHM RESULTS')
for i=1:numel(ar)
    C=ar(i);
    if QROnlyTrigger==0
        while SquareAndMultiply(C,(p-1)/2,p)==1
            C=C+1;
        end
    end
    if QROnlyTrigger==1
        C=mod(C^2,p);
    elseif QROnlyTrigger==0
        if answers(i)~=0
```

```
                continue
            end
        end
        elm=algov11(p,C);
        %elm=JustAlgov2(p,C);
        if elm==0
            W=[num2str(C),' is an NR (mod ',num2str(p),').'];
            disp(W)
        else
            W=[num2str(elm),'^2=',num2str(C),'(mod ',num2str(p),').'];
            disp(W)
        end
        if elm~= answers(i) && p-elm ~=answers(i)
            wrong=wrong+1;
            wrongseeds=[wrongseeds, ar(i),p-ar(i)];
        end
        trials=trials+1;
end
%Report results
disp('')
disp('--------------------------------------------')
disp('')
if QROnlyTrigger==1
    X=['QR ONLY TEST MOD ', num2str(p)];
    disp(X)
elseif QROnlyTrigger==0
    X=['NR ONLY TEST MOD ', num2str(p)];
    disp(X)
else
     X=['RANDOM TEST MOD ', num2str(p), ', QR/NR AREs MIXED'];
     disp(X)
end
W=['NUMBER OF TRIALS: ', num2str(trials)];
disp(W)
Z=['NUMBER OF WRONG ANSWERS: ',num2str(wrong), '. THEY ARE: '];
disp(Z)
disp(wrongseeds)
disp('SEE MATLAB''S RUN AND TIME FUNCTION FOR RUNTIME STATS')
%%%% EOF
```

11 References

[1] Agarwal, A. [Lecture notes] (Fall 2017). MATH 771: Mathematics of Cryptography Lectures. Lectures presented at Rochester Institute of Technology, Rochester NY.

[2] Bernstein, D. (2001). Faster Square Roots in Annoying Finite Fields; Draft. Retrieved from `https://www.researchgate.net/publication/2381439_Faster_Square_Roots_in_Annoying_Finite_Fields`

[3] Cannas da Silva, A. (2006). Lectures on Symplectic Geometry. Retrieved from `https://people.math.ethz.ch/~acannas/Papers/lsg.pdf`

[4] Guy, Q. (2012, October 10). Fast String to Double Conversion. Retrieved August 16, 2018, from Mathworks: File Exchange website: `https://www.mathworks.com/matlabcentral/fileexchange/28893-fast-string-to-double-conversion?s_tid=mwa_osa_a`

[5] Jones, G. A., & Jones, J. M. (2005). Springer Undergraduate Mathematics Series: Elementary Number Theory (8th ed.). Springer.

[6] LeVeque, W. J. (1977). Fundamentals of Number Theory (2015 ed.). New York, NY: Dover Publications.

[7] Peralta, R. (1992). On the Distribution of Quadratic Residues and Nonresidues Modulo a Prime Number. Mathematics of Computation, 58(197), 433-440. `https://doi.org/10.1090/S0025-5718-1992-1106978-9`

[8] Pocklington, H.C. (1917). The Direct Solution of the Quadratic and Cubic Binomial Congruences with Prime Moduli. Proceedings of the Cambridge Philosophical Society, XIX, 57-59. Retrieved from `https://archive.org/stream/proceedingsofcam1920191721camb/proceedingsofcam1920191721camb_djvu.txt`

[9] Printer, C. C. (1990). A Book of Abstract Algebra (2nd ed.). New York, NY: Dover.

[10] Quadratic Nonresidue. (n.d.). Retrieved August 26, 2018, from Wolfram Mathworld website: http://mathworld.wolfram.com/QuadraticNonresidue.html

[11] Quadratic Residue. (n.d.). Retrieved August 16, 2018, from Wolfram Mathworld website: http://mathworld.wolfram.com/QuadraticResidue.html

[12] Schlenk, F. (2018), Symplectic Embedding Problems, Old and New, Bulletin of the AMS, 55(2), p.139-182 https://doi.org/10.1090/bull/1587

[13] Schoof, R. (1985). Elliptic Curves Over Finite Fields and the Computation of Square Roots mod p. Mathematics of Computation, 44(170), 483-494. https://doi.org/10.2307/2007968

[14] Shirali, S. A. (2003). 6: On Fermat's Two Squares Theorem. In S. A. Shirali (Author) & C. S. Yogananda (Ed.), Number Theory (pp. 30-33). Retrieved from https://books.google.com/books?id=BkBSfFjT1BgC&pg=PA30&lpg=PA30&dq#v=onepage&q&f=false

[15] Tornaría, G. (2002). Square Roots Modulo p. Latin American Symposium on Theoretical Informatics, 430-434. https://doi.org/10.1007/3-540-45995-2_38

[16] Turner, S. M. (1994). Square Roots mod p. The American Mathematical Monthly, 101(5), 443-449. https://doi.org/10.2307/2974905

[17] Walum, H. (1982). On the distribution of quadratic residues modulo a prime. Journal of Number Theory, 15(2), 248-251. https://doi.org/10.1016/0022-314X(82)90029-4

[18] Wilson, J. (2012). [Manifolds] [Lecture notes]. Retrieved August 19, 2018, from http://www.math.lsa.umich.edu/~jchw/WOMPtalk-Manifolds.pdf