FASTER SQUARE ROOTS MODULO A PRIME ON THE TI-89

Joseph Fadyn
Southern Polytechnic State University
1100 South Marietta Parkway
Marietta, Georgia 30060
jfadyn@spsu.edu

INTRODUCTION

We consider the problem of solving the quadratic congruence:

$$x^2 \equiv a \pmod{p},$$

where p is an odd prime number using the TI-89. In [10], I produced a technique to accomplish this task in connection with solving the quadratic congruence:

$$ax^2 + bx + c \equiv 0 \pmod{p}$$

This technique was formalized as TI-89 program sqrtmdp() in [9] and was used in [9] and [11] to assist in solving cubic and quartic congruences modulo a prime p. The purpose of this paper is to substantially improve the efficiency of the program sqrtmdp. First we review some of the ideas of the program sqrtmdp: In [10] we considered the problem of solving $x^2 \equiv \delta \pmod{p}$. To quote from [10]:

"The Euler criterion says that if gcd(δ,p) =1 then δ is a quadratic residue modulo p if and only if $\delta^{\frac{(p-1)}{2}} \equiv 1 \pmod{p}$. In the case that δ is a quadratic residue modulo p, let r denote a square root of δ modulo p. There are three cases to consider:

Case 1: $p \equiv 3 \pmod 4$. Then if p = 4n + 3, we have:

$r \equiv \pm\delta^{n+1} \pmod p$

Case 2: $p \equiv 5 \pmod 8$. Then if p = 8n + 5, we have:

$r \equiv \pm\delta^{n+1} \pmod p$ or $r \equiv \pm 2^{2n+1}\,\delta^{n+1} \pmod p$.

Case 3: $p \equiv 1 \pmod 8$. In this case we use Shank's Algorithm. We use the following version of Shanks Algorithm adopted with minor changes from Robbins [12]:

Let x be a solution to $x^2 \equiv \delta \pmod p$, let n, k be integers such that $p - 1 = 2^n k$ where $n \geq 1$ and k is odd, and let q be a quadratic *nonresidue* modulo p. The x can be found by repeating the loop:

1. Set $r \equiv \delta^k \pmod p$ and $t \equiv \delta^{\frac{(k+1)}{2}} \pmod p$.
2. Find the least i such that $r^{2^i} \equiv 1 \pmod p$.
3. If i = 0 then the solutions are $x \equiv \pm t \pmod p$.
4. If i > 0, set $u \equiv q^{k(2^{n-i-1})} \pmod p$, and go to step 2 and replace t by tu and r by ru^2."

We begin to code our modified square root program, sqrtmdpm( ) as follows:

sqrtmdpm(dd,p)

Prgm : DelVar r, rot, bt, at, ts : ClrIO: mod(dd,p) -> dd : If mod(dd, p) = 0 Then: Disp "Root Is:" Disp "0" : 0 -> rot[1]: 1 -> numrots : Goto stp6 : EndIf

One way to improve our original program is to shorten the time it takes to compute large powers of a base modulo p. The original program, sqrtmdp( ), uses the basic technique of modular exponentiation (see program mdexp( ) in [10]). The use of an addition chain for the exponent can improve this performance somewhat. The basic idea of an addition chain, quoted according to Bernstein [2] follows:

An **addition chain of length** ℓ is a sequence of $\ell + 1$ integers, where the first integer is 1 and each subsequent integer is a sum of two earlier integers. In other words, it is a sequence $c_0, c_1, \ldots, c_\ell$ such that $c_0 = 1$ and, for each $k \in \{1, 2, \ldots, \ell\}$, there exist $i, j \in \{0, 1, \ldots, k-1\}$ such that $c_k = c_i + c_j$.

For example, $1, 2, 3, 5, 7, 14, 28, 56, 63$ is an addition chain of length 8, because $2 = 1 + 1$, $3 = 2 + 1$, $5 = 3 + 2$, $7 = 5 + 2$, $14 = 7 + 7$, $28 = 14 + 14$, $56 = 28 + 28$, and $63 = 56 + 7$.

Using chains to compute powers. Short addition chains are a model of fast algorithms for modular exponentiation. For each addition chain $c_0, c_1, \ldots, c_\ell$, there is an algorithm that computes the c_ℓth power of its input with ℓ multiplications, by computing successively the c_1st power, the c_2nd power, etc.

For example, given m, and given an integer x between 0 and $m - 1$, one can compute successively

$$\begin{aligned}
x^2 \bmod m &= (x \cdot x) \bmod m, \\
x^3 \bmod m &= ((x^2 \bmod m) \cdot x) \bmod m, \\
x^5 \bmod m &= ((x^3 \bmod m) \cdot (x^2 \bmod m)) \bmod m, \\
x^7 \bmod m &= ((x^5 \bmod m) \cdot (x^2 \bmod m)) \bmod m, \\
x^{14} \bmod m &= ((x^7 \bmod m) \cdot (x^7 \bmod m)) \bmod m, \\
x^{28} \bmod m &= ((x^{14} \bmod m) \cdot (x^{14} \bmod m)) \bmod m, \\
x^{56} \bmod m &= ((x^{28} \bmod m) \cdot (x^{28} \bmod m)) \bmod m, \\
x^{63} \bmod m &= ((x^{56} \bmod m) \cdot (x^7 \bmod m)) \bmod m,
\end{aligned}$$

Obtaining x^{63} mod m with 8 multiplications modulo m.

Our modular exponentiation program, mdexpch( ) uses a classical algorithm first proposed by Brauer [3] and explained in Section 3 of Bernstein's paper [2]:

mdexpch(a, n, m): Prgm : DelVar x: mod(a,m) -> x[1]: mod(x[1]^2,m) -> x[2]: mod(x[1]*x[2],m) -> x[3]: mod(x[2]^2,m):
mod(x[2]^2,m) -> x[4]: mod(x[2]*x[3], m) -> x[5]: mod(x[3]^2,m) -> x[6]:
mod(x[3]*x[4],m) ->x[7]: If n < 8 Then x[n] ->pp: Goto stp4: EndIf: baseconc(n,8):
1 ->pp: i-1 -> i: If 2*c[i] < 7 and 4* c[i] < 7 then 8*c[i] -> t1: mod(x[1]*x[7], m) -> pp: Goto stp2: EndIf: If 2*c[i] < 7 Then 4*c[i] -> t1:mod(x[t1-7]*x[7],m) ->pp: 8*c[i] -> t1: mod(pp^2,m) ->pp: Goto stp2: EndIf: If 2*c[i] >7 Then 2*c[i] -> t1: mod(x[t1-7]*x[7],m) -> pp: mod(pp^2,m) -> pp: mod(pp^2,m) -> pp: 8*c[i] -> t1: EndIf: i – 1 -> i: While i > 1: t1 + c[i] -> t1: If c[i] = 0 Then Goto stp3: Else: mod(pp*x[c[i]], m) ->pp: EndIf: Lbl stp3: For j,1,3: mod(pp^2,m) ->pp: EndFor: 8*t1 ->t1: I – 1 -> I EndWhile: t1 + c[1] -> t1: If c[1] = 0 Then: Goto stp4 Else: mod(pp*x[c[1]], m) -> pp: Goto stp4: EndIf: Lbl stp4: Disp pp: EndPrgm

The program mdexpch( ) calls baseconc( ) which is a base conversion program:

baseconc(a, b): Prgm: ClrIO: DelVar c: 1 -> i : while a ≠ 0 diva(a,b) -> d: d[1,1] -> a: d[1,2] -> c[i] : i + 1 -> i: EndWhile: EndPrgm
Here the function diva(a,b) is a simple TI-89 implementation of the division algorithm: diva(a,b): [floor(a/b), a-b*floor(a/b)]
We move on to the simple case where p ≡ 3 (mod 4). We can improve the speed in this case by building in the Euler criterion into our computations. This will save one modular exponentiation and replace it by two simple multiplications. We continue with the coding of sqrtmdpm as follows:
If mod(p,4) = 3 Then: (p-3)/4 -> k: mdexpch(dd,k,p): pp -> ak: mod(ak*dd,p) -> akone: If mod(akone*ak,p) = 1 Then: Disp "Two Roots Exist": Else: Disp "No Root Exists": 0 -> numrots: Goto stp6: EndIf: Disp "Roots Are:": Disp akone: akone -> rot[1]: Disp "And:" : Disp p – akone: p – akone -> rot[2]: 2 -> numrots: Goto stp6: EndIf
Now we consider the case p ≡ 5 (mod 8). Here we use Atkin's Algorithm [1], which is outlined in [4] as follows:

Algorithm 3. Atkin algorithm when $p \equiv 5 \pmod 8$
Require: $a \in \mathbf{F}_q$ such that $\zeta(a) = 1$.
Ensure: x satisfying $x^2 = a$.
1: $b \leftarrow (2a)^{\frac{q-5}{8}}$.
2: $i \leftarrow 2ab^2$.
3: $x \leftarrow ab(i-1)$.
4: **return** x

We continue coding sqrtmdpm:

If mod(p,8) = 5 Then: mdexpch(2*dd,(p-50/8,p): mod(2*dd*pp^2,p) -> i: mod(dd*pp*(i – 1),p) ->x : If mod(x^2,p) = dd Then Disp "Two Roots Exist": Disp "Roots Are:": Disp x: x -> rot[1]: Disp "And:" Disp p – x: p – x -> rot[2]: 2 -> numrots: Goto stp6: Else: Disp "No Root Exists": 0 -> numrots: Goto stp6: EndIf: EndIf

For the case p ≡ 9 (mod 16) we use a modified form of Atkin's algorithm first proposed by Muller [14] and later improved upon by Kong [5]. As stated in [4], we have:

Algorithm 4. Modified Atkin algorithm when $p \equiv 9 \pmod{16}$
Require: $a \in \mathbf{F}_q$ such that $\zeta(a) = 1$.
Ensure: x satisfying $x^2 = a$.
1: $b \leftarrow (2a)^{\frac{q-9}{16}}$.
2: $i \leftarrow 2ab^2$.
3: $r \leftarrow i^2$.
4: If $r = -1$ then $x \leftarrow ab(i-1)$.
5: Else
5.1 Select d randomly such that $\zeta(d) = -1$.
5.2 $u \leftarrow bd^{\frac{q-9}{8}}$.
5.3 $i \leftarrow 2u^2d^2a$.
5.4 $x \leftarrow uda(i-1)$.
6: **return** x

In order to skip the test that a is a quadratic residue (which would require an additional modular exponentiation), I will modify the algorithm above to build this test into steps 1-4. Otherwise, step 5 would always need to be executed and this step can be time consuming. This modification uses the fact that 2 is a quadratic residue if and only if p is congruent to 1 or 7 modulo 8. We continue our coding of sqrtmdpm():

If mod(p,16) = 9 Then: mdexpch(2*dd, (p-9)/16, p): pp -> b: mod(b^2,p) -> bt: mod(bt^2, p) -> bt: mod(bt^2,p) -> bt: mod((2*dd)^2,p) -> at: mod(at^2,p) ->at: mod(at*bt,p) -> ts: If mod(p,8)=1 or mod(p,8)=7 Then: If ts = 1 Then Goto stpy: Else: Disp "No Root Exists": 0 -> numrots: Go to stp6: EndIf: If ts = -1 Then: Goto stpy: Else: Disp "No Root Exists": 0 -> numrots: Goto stp6: EndIf: EndIf: Lbl stpy: mod(2*dd*b^2,p) -> i: mod(i^2,p) -> r: If r = p-1 Then: mod(dd*b*(i-1),p) -> x: Disp "Square Root Is:" : Disp x: Goto stp6: Else: quadnrn(p): mdexpch(q,(p-9)/8,p): mod(b*z,p) -> u: mod(2*u^2*q^2*dd,p) -> i: mod(u*q*dd*(1-i),p) -> x: Disp "Square Root Is:" : Disp x: Goto stp6: EndIf: EndIf:

Step 5 in the algorithm requires us to find a quadratic nonresidue, q, for the prime p. One way to do this is to pick random values for q and use Euler's criterion. I have attempted to improve this by using some ideas from a recent (2007) doctoral thesis by Sze [15]. For example, the least quadratic nonresidue must be a prime number. The result is my TI-89 program quadnrn( ) which is called by the code above:

quadnrn(p): Prgm: If mod(p,3)=2 Then 3 -> q: Goto stp3: EndIf: If mod(p,5) = 2 or mod(p,5) = 3 then 5 ->q: Goto stpe: EndIf: maxtwo((p-1)/2): n -> tcap: (p-1)/2/2^n -> s: 2 -> q: Lbl stpc: If q = 2 and (mod(p,8) = 1 or mod(p,8) = 7) Then: 3 -> q: EndIf: If q = 3 and (mod(p,12=1 or mod (p,12) = 11) Then 5 -> q: EndIf: If q = 5 and (mod(p,10) = 1 or mod(p, 10) = 9) Then 7 -> q: EndIf: mdexpch(q, (s-1)/2, p): pp -> z: mod(q*z,p) -> xsp: mod(xsp*z,p) -> xs: 0 -> t: While xs ≠ p-1: mod(xs^2,p) -> xs: t+1 -> t: EndWhile: If t < tcap then nextprim(q) -> q: Goto stpc: Else: Disp "q = ": Disp q: EndIf: Lbl stpe: EndPrgm

Here nextprim() finds the next prime and is the TI-89 function:
nextprim(n): Func: Loop: n+1 ->n: If isPrime(n): Return n: EndLoop: EndFunc
This function can be found in the TI-89 manual. The program maxtwo() is listed later in this paper. There remains the difficult case that $p \equiv 1 \pmod{16}$, so that $p - 1 = 2^T * k$, where k is odd and $T \geq 4$. After searching the literature, I have decided to compare five algorithms which can be used in this case. All of these algorithms are probabilistic in the sense that they all require a quadratic non residue as a starting point. These algorithms are: Shanks-Tonelli, Smart, Moving Window Sign Testing, Cipolla, and Peralta. Shank's Algorithm is outlined on the first page of this paper. The Smart algorithm (proposed in 1999—see [8]) is an improvement on Shank's algorithm and is outlined in a 2005 paper by Wang et. al. [6] as follows:

3.1 Smart algorithm

Smart algorithm

Input: A nonzero QR $x \in GF(p^m)$.

Output: A square root $\sqrt{x} \in GF(p^m)$.

Preparation:

1. Factorize $(p^m - 1)/2$ as shown in Eq.(2).
2. Find an appropriate QNR $\theta \in GF(p^m)$ and compute $a = \theta^s$.

Procedure:

Step1 : Compute $b = x^{(s-1)/2}$ and set $t_0 = 0, k = 0$.

Step2 : Iteratively compute the following expressions as k increases by 0 to $T-1$:

$$t_{k+1} = t_k + 2^k c_k, \tag{8a}$$

where

$$c_k = \begin{cases} 0 & \text{if } \left((a^{t_k} b)^2 \cdot x\right)^{2^{T-1-k}} = 1 \\ 1 & \text{if } \left((a^{t_k} b)^2 \cdot x\right)^{2^{T-1-k}} = -1 \end{cases}. \tag{8b}$$

Step3 : Output the square root $\sqrt{x} = (a^{t_T} b)x$.

The moving window sign testing (MWST) algorithm is an improvement on the Smart algorithm and was proposed by Wang et. al. in [6] as follows:

Moving window-sign testing algorithm over $GF(p^m)$ ($m = 1$ or 2)

Input: A nonzero QR $x \in GF(p^m)$.

Output: A square root $\sqrt{x} \in GF(p^m)$.

Preparation:

1. Factorize $(p^m - 1)/2$ as shown in Eq.(2).
2. Execute the proposed QR test as shown in Fig.2 (a) and store the intermediate values $x_0, x_1, \cdots, x_t$ and $x^{(s+1)/2}$.
3. Find an appropriate QNR $c \in GF(p^m)$, compute $c_0, c_1, \cdots, c_T$ as shown in Fig.2 (b) and store them in the memory.

Main Procedure:

1. Check whether x_0 is equal to 1 or not. If $x_0 = 1$, then output $\sqrt{x} = x^{(s+1)/2}$ and input another element. If not, execute **Procedure** 2 and **Procedure** 3 in order.
2. Let $\tau_0 = T-1$, $\mu = 1$ and $k = 1$, and then repeatedly execute **Step**$1-3$ until k becomes t.

Step 1 : Compute $\sigma = x_{t-k} \prod_{i=0}^{\mu-1} c_{\tau_i}$. If $\sigma = -1$, then $\tau_\mu = T-1$ and $\mu = \mu + 1$. If not, the values of τ_μ and μ are not modified.

Step 2 : For $0 \leqslant i < \mu$, let $\tau_i = \tau_i - 1$.

Step 3 : Let $k = k + 1$.

3. Output the value of $\sqrt{x} = x^{(s+1)/2} \prod_{i=0}^{\mu-1} c_{\tau_i}$.

Here is Wikipedia's [7] summary of Cipolla's algorithm:

Inputs:

- p, an odd prime,
- $n \in \mathbf{F}_p$, which is a square.

Outputs:

- $x \in \mathbf{F}_p$, satisfying $x^2 = n$.

Step 1 is to find an $a \in \mathbf{F}_p$ such that $a^2 - n$ is not a square. There is no known algorithm for finding such an a, except the trial and error method. Simply pick an a and by computing the Legendre symbol $(a^2 - n \mid p)$ one can see whether a satisfies the condition. The chance that a random a will satisfy is $(p - 1) / 2p$. With p large enough this is about $1 / 2$. Therefore, the expected number of trials before finding a suitable a is about 2.

Step 2 is to compute x by computing $x = \left(a + \sqrt{a^2 - n}\right)^{(p+1)/2}$ within the field $\mathbf{F}_{p^2} = \mathbf{F}_p(\sqrt{a^2 - n})$. This x will be the one satisfying $x^2 = n$.

If $x^2 = n$, then $(-x)^2 = n$ also holds. And since p is odd, $x \neq -x$. So whenever a solution x is found, there's always a second solution, $-x$.

Finally, Peralta's Algorithm from his original 1986 paper [13]:

Algorithm 2:

Input: a prime P congruent to 1 mod 4 and a quadratic residue x mod P.
Output: $y \in \mathbb{Z}_P$ such that $y^2 \equiv x \bmod P$.
1) Choose r at random in $\mathbb{Z}_P^*$.
2) If $r^2 \equiv -x \bmod P$, go to 1.
3) Compute $(r + \sqrt{-x})^Q = u + v\sqrt{-x}$.
4) If either u or v is 0, go to 1.
5) Compute $(u + v\sqrt{-x})^{2^i}$ for $i = 1, 2, \cdots$ by repeated squaring until $(u + v\sqrt{-x})^{2^i} = 0 + w\sqrt{-x}$ for some w. Let $(u + v\sqrt{-x})^{2^{i-1}} = a + b\sqrt{-x}$. Then $a^2 - b^2 x \equiv 0 \bmod P$. Output $a/b \bmod P$.

Now I will code each of these five algorithms on the TI-89:

shanks(dd,p): Prgm: mdexpch(dd, (p-1)/2,p) If mod(pp,p) = p-1 Then: Disp "No Square Root Exists": Stop: End If: 1 -> n: p-1 -> h: While fPart(h/2) = 0: h/2 ->h: n+1 -> n: Endwhile: n-1 ->n: (p-1)/2^n -> k: 2 -> q: mdexpch(q, (p-1)/2,p): While pp = 1: q + 1 -> q: mdexpch(q, (p-1)/2, p): EndWhile: mdexpch(dd, k, p): pp -> r: mdexpch(dd, (k+1)/2,p): pp -> tLbl stp1: 0 -> i: mdexpch(r, 2^I, p) While pp ≠ 1: I + 1 -> i: mdexpch(r, 2^I,p): EndWhile: If I = 0 Then: Disp "Square Roots Are:" Disp mod(t,p): Disp "And:" : Disp mod(-t,p): Stop: Else: k*2^(n-i-1) -> v: mdexpch(q, v, p): pp -> u: mod(t*u,p) -> t: mod(r*u^2,p) -> r: Goto stp1: EndIf: EndPrgm

smart(x,p): Prgm: ClrIO: mod(x,p) -> x: mdexpch(x, (p-1)/2,p): If mod(x,p) = 0 Then: Disp "Root Is:" : 1 -> numrots: Disp "0": 0 -> rot[1]: Goto stp6: ElseIf pp = p-1 Then: Disp "No Root Exists": 0 _> numrots: Goto stp6: Else: Disp "Two Roots Exist": 2 -> numrots: EndIf: DelVar a,t,b,k,tk,ten,ck,r: maxtwo((p-1)/2): n -> t: (p-1)/2/2^t -> s: qnrmdp(p): mdexpch(q,s,p): pp -> a: mdexpch(x, (s-1)/2,p): z -> b: 0 -> k: 0 -> tk: While k ≤t-1: mdexpch(a, tk, p): pp -> atk: mod((atk*b)^2*x,p) -> ten: mdexpch(ten, 2^(t-1-k),p): If pp = 1 Then: 0 -> ck: Else: 1 -> ck: EndIf: tk+2^k*ck -> tk: k+1 -> k: EndWhile: mdexpch(a,tk,p): pp -> ten: mod(ten*b*x,p) -> r: Disp "A Square Root Is:" : Disp r: Lbl stp6: EndPrgm

The smart algorithm code above calls the program maxtwo() whose listing follows:

maxtwo(k): Prgm: If k < 0 Then: Disp "K must be >0": Stop: EndIf: If k = 1 Then: 0 -> n: Disp n: Stop: EndIf: If mod(k,2) = 1 Then: k-1 -> k: EndIf: 0 ->n: While fPart(k/2) =0: k/2 -> k: n+1 -> n: EndWhile: Disp n: EndPrgm

mwstn(x,p): Prgm: DelVar z,e,c,tao,t: maxtwo((p-1)/2): n -> tcap: (p-1)/2/2^n -> s: mdexpch(x,(s-1)/2,p): pp -> z: mod(x*z,p) -> xsp: mod(xsp*z,p) -> xs: If xs = 1 Then: Disp "A square Root Is:" : Disp xsp: Goto stpe: EndIf: 0 -> t: xs -> e[1]: 1 -> i: While e[i] ≠ p-1: mod(e[i]^2,p) -> e[i + 1]: i + 1 -> i: t + 1 ->t: EndWhile: t -> tsn: If t = tcap Then: Disp "No Square Root Exists": Goto stpe: EndIf: xsp -> xspn: tcap -> tcapn: quadnrn(p): mdexpch(q,s,p): DelVar c: pp -> c[1]: 1 -> i: While c[i] ≠ p – 1: mod(c[i]^2,p) -> c[i+1]: i+1 -> i: EndWhile: tcapn -> tao[1]: 2 -> mu: 1 -> k: 0 -> flag: While k ≤ tsn: mod(e[tsn-k+1]*∏(c[tao[I]],i,1,mu-1),p) -> ho:1-> i: For i,1,mu-1: tao[i] – 1 -> tao[i]: EndFor: If ho = p-1 Then: tcapn -> tao[mu]: mu + 1 -> mu: EndIf: k = 1 -> k: EndWhile: mod(xspn*∏(c[tao[i]],i,1,mu-1),p) -> rt: Disp "A Square Root is:" : Disp rt: Lbl stpe: EndPrgm

cipollas(dd,p): Prgm: DelVar r, rot: ClrIO: mod(dd,p) -> dd: mdexpch(dd,(p-1)/2,p): If mod(dd,p)=0 Then: Disp "Root Is:" 1 -> numrots: Disp "0": 0 -> rot[1]: Goto stp6: ElseIf z = p-1 Then: Disp "No Root Exists": 0 -> numrots: Goto stp6: Else: Disp "Two Roots Exist" : 2 -> numrots: EndIf:1 -> q: mdexpch(mod(q^2-dd,p),(p-1)/2,p): While pp = 1: q+1 -> q: mdexpch(mod(q^2-dd,p),(p-1)/2,p): EndWhile: mdexps(q,1,mod(q^2-dd,p),(p+1)/2,p): Disp "A Square Root Is:" : Disp pp[1,1]: Lbl stp6: EndPrgm

The above algorithm calls the program mdexps() which follows:

mdexps(t,s,a,e,n): Prgm: DelVar pp,mm: [1,0] -> pp: mod(t,n) -> tt: mod(s,n) -> ss: [tt,ss] -> mm: While e ≠ 0: diva(e,2) -> d: If d[1,2] = 0 Then: Goto stp10: EndIf: multr(pp,mm,a,n): [xx,yy] -> pp: Lbl stp10: d[1,1] -> e: sqrr(mm,a,n): zz -> mm: EndWhile: Disp pp: EndPrgm

Programs called by mdexps() include multr() and sqrr(). Listings follow:

multr(u,z,a,p): Prgm: ClrIO: DelVar xx, yy: Disp mod([u[1,1]*z[1,1]+a*z[1,2]*u[1,2],u[1,1]*z[1,2]+u[1,2]*z[1,1]]],p): mod(u[1,1]*z[1,1]+ a*z[1,2]*u[1,2],p) -> xx: mod(u[1,1]*z[1,2]+u[1,2]*z[1,1],p) -> yy: Disp [xx,yy]: EndPrgm

sqrr(u,a,p): Prgm: ClrIO: DelVar zz: mod([[u[1,1]^2+a*u[1,2]^2,2*u[1,1]*u[1,2]]],p) -> zz: Disp zz: EndPrgm

peraltac(dd,p): Prgm: DelVar r, rot: ClrIO: mod(dd,p) -> dd: mdexpch(dd,(p-1)/2,p): pp ->z: If mod(dd,p)=0 Then: Disp "Root Is:" : 1 -> numrots: Disp "0": 0 -> rot [1]: Goto stp6: ElseIf z = p-1 Then: Disp "No Root Exists": 0 -> numrots: Goto stp6 Else: Disp "Two Roots Exist": 2 -> numrots: EndIf: maxtwo(p): (p-1)/2^n -> de: Mod(-dd,p) -> m: Lbl stpr: mod(rand(1000000),p) -> x: If x = 0 or mod(x^2,p) = m or x = 1 Then: Goto stp r: EndIf: mdexprc(x,1,-dd,de,p): If mod(pp[1,1])*pp[1,2],p) = 0 Then: Goto stpr: EndIf: pp[1,1] -> s: pp[1,2] -> r: Lbl stpy: multr([s,r],[s,r],m,p): If mod(xx,p) = 0 Then: Goto stpe Else: xx -> s: yy -> r: Goto stpy: EndIf: Lbl stpe: mod(s*modinv(r,p),p) -> ro: Disp "A Square Root Is:" : Disp ro: Lbl stp6: EndPrgm

The program mdexprc() called in the peraltac() program above is coded as follows:

mdexprc(t,s,a,e,n): Prgm: DelVar x, pp, mm: [[1,0]] -> pp: mod(t,n) -> tt: mod(s,n) -> ss: [[tt,ss]] -> mm: randMat(7,2) -> x: tt -> x[1,1]: ss -> x[1,2]: sqrr(mm, e, n): zz[1,1] -> x[2,1]: zz[1,2] -> x[2,2]: multr(mm,zz,a,n): xx -> x[3,1]: yy -> x[3,2]: sqrr([[x[2,1],x[2,2]]]],a,n): zz[1,1] -> x[4,1]: zz[1,2] -> x[4,2]: multr(mm,zz,a,n): xx -> x[5,1]: yy -> x[5,2]: sqrr([[x[3,1], x[3,2]]], a, n): zz[1,1] -> x[6,1]: zz[1,2] -> x[6,2]: multr(mm,zz,a,n): xx -> x[7,1]: yy -> x[7,2]: baseconc(e,8): i – 1 -> i: if 2*c[i] < 7 and 4*c[i] < 7 Then: 8*c[i] -> t1: multr(mm, [[xx,yy]], a, n): [[xx,yy]] -> pp: Goto stp2: EndIf: If 2*c[i] < 7 Then: 4*c[i] -> t1: multr([[x[t1-7,1],x[t1-7,2]]], [[xx,yy]], a, n) : [[xx,yy]] -> pp: 8*c[i] -> t1: sqrr(pp,a,n): zz -> pp: Goto stp2: EndIf: If 2*c[i] > 7 Then: 2*c[i] -> t1: multr([[x[t1-7,1], x[t1-7,2]]], [[xx,yy]], a, n): [[xx,yy] -> pp: sqrr(pp,a,n): zz -> pp: sqrr(pp,a,n): zz -> pp: 8*c[i] -> t1: EndIf: Lbl stp2: i – 1 -> i: While i > 1: t1 + c[i] -> t1: If c[i] = 0 Then: Goto stp3: Else: multr(pp,[[x[c[i],i],x[c[i],2]]],a,n): [[xx,yy]] -> pp: EndIf: Lbl stp3: For j,1,3: sqrr(pp,a,n): zz -> pp: EndFor: 8*t1 -> t1: i – 1 -> i: EndWhile: t1 + c[1] -> t1: If c[1] = 0 Then: Goto stp4: Else: multr(pp, [[x[c[1],1], x[c[1],2]]],a, n): [[xx,yy]] -> pp: Goto stp4: EndIf: Lbl stp4: Disp pp: EndPrgm

Extensive testing (with 100 randomly generated primes p congruent to 1 modulo 16) shows that Shanks algorithm and the Smart algorithm are almost always inferior to the moving-window sign testing algorithm. The mwst algorithm is generally faster when $T \leq 30$ in the representation: $p - 1 = 2^T * k$, where k is odd and $T \geq 4$. This is likely because the Main Procedure in the mwst algorithm must be executed T times and this can be fairly expensive if $T > 30$ in which case Peralta's algorithm is generally better. In addition, there are some primes for which Cipolla's algorithm is usually superior. The optimal choice of the algorithm also sometimes depends on the size of the prime. The exact choice I have come up with is included in our final coding of sqrtmdpm():

maxtwo(p-1): If n ≤ 30 and p ≥ 10^18 Then: mwstn(dd,p): ElseIf 20 ≤n and n≤32 and p ≤10^15 Then: cipollas(dd,p): ElseIf n < 20 Then mwstn(dd,p): Else: peraltac(dd,p): EndIf: Lbl stp6: EndPrgm

Results: In comparing algorithms by testing 60 random primes from 5 to 50 decimal digits in length, the determination of "no square root" had a time decrease of 13.4% from our original algorithm, whereas the determination of a square root had a time decrease of 68.3% from our original algorithm. So on average, time for determination of a square root when one exists is roughly multiplied by ⅓ . In none of these cases did the original algorithm take less time to compute a root than our new algorithm. Of course, extreme examples are easy to construct which involve "annoying" primes where finding square roots is notoriously time-consuming. The most extreme time differences occur when $p - 1 = 2^T * k$, where k is odd and T is "large". For example, if T=20, one prime p is p = 104857600002154823681, and if we run: sqrtmdp(6598745687, p) it takes 235 seconds to compute a square root, which is 256081105603345690282. On the other hand sqrtmdpm(6598745687, p) computes a root in only 25 seconds.

References

1. A.O.L. Atkin, *Probabilistic Primality Testing,* summary by F. Morain, Research Report 1770, INRIA, pp. 159-163, 1992.
2. Bernstein, Daniel, *Pippenger's Exponentiation Algorithm*, Unpublished Draft, Dept. of Math., The University of Illinois at Chicago, Chicago, Illinois.
3. Brauer, Alfred, *On Addition Chains*, Bulletin of the American Mathematical Society, 45, 1939, 736-739.
4. Dong-Guk Han, Dooho Choi, and Howon Kim, Member, IEEE, *Improved Computation of Square Roots in Specific Finite Fields,* IEE Transactions On Computers, Vol. 58, No. 2, February 2009.
5. F. Kong, Z. Cai, J. Yu, D. Li, *Improved Generalized Atkin Algorithm for Computing Square Roots in Finite Fields,* Information Processing Letters, vol. 98, no. 1, pp. 1-5, 2006.
6. Feng Wang, Yasuyuki Nogami, and Yoshitaka Morikawa, *A High-Speed Square Root Computation In Finite Fields With Application To Elliptic Curve*

Cryptosystem, Memoirs of the Faculty of Engineering, Okayama University, Vol. 39, pp. 82-92, January 2005.

7. http://en.wikipedia.org/wiki/Cipolla's_algorithm
8. I. Blake, G. Seroussi, and N. Smart: Elliptic Curves in Cryptography, LMS 265, Cambridge University Press, 1999.
9. J.N. Fadyn, *Solving Cubic Congruences Modulo a Prime on The TI-89,* Proceedings of the ICTCM 2011.
10. J.N. Fadyn, *Solving Quadratic Congruences Modulo a Prime on The TI-89,* Proceedings of the ICTCM 2010.
11. J.N. Fadyn, *Solving Quartic Congruences Modulo a Prime on The TI-89,* Proceedings of the ICTCM 2012.
12. Neville Robbins, *Beginning Number Theory, Second Edition,* Jones and Bartlett Publishers, 2006.
13. Peralta, Rene C., *A Simple and Fast Probabilistic Algorithm For Computing Square Roots Modulo A Prime Number,* IEEE Transactions On Information Theory, Vol. IT-32, No. 6, November 1986.
14. S. Muller, *On The Computation of Square Roots in Finite Fields*, J. Design, Codes and Cryptography, vol. 31, pp. 301-312, 2004.
15. Tsz Wo Sze, *On Solving Univariate Polynomial Equations Over Finite Fields And Some Related Problems,* Doctoral Dissertation, University of Maryland, College Park, 2007.