Mersenne 31 Polynomial Arithmetic

Tomer Solberg
tomer@ingonyama.com

Yuval Domb
yuval@ingonyama.com

Abstract

This note aims to provide a comprehensive yet concise introduction and tutorial on the efficient implementation of field and polynomial arithmetic within the M31 field, specifically in the context of Circle STARKs [UH24]. By addressing both the advantages and challenges associated with this field choice, the note seeks to equip practitioners with the knowledge necessary to optimize their cryptographic systems effectively.

1 Mersenne 31 Field Arithmetic

M31 is the name of the number $p = 2^{31} - 1$. This is a prime whose field $\mathbb{F}_p$ enjoys some favorable properties for arithmetic calculations. [1]

First note that this number fits in a 32-bit int variable. Addition, subtraction and multiplication are the same as the integer operations, except they must be followed by a modular reduction operation to take care of overflows.

1.1 Base field arithmetic

1.1.1 Modular reduction

Modular reduction is the first place where M31 shines. Given a result x of an arithmetic operation which overflows the 31-bits of a field element, one can perform a modular reduction by breaking the result into a low part x_l (the 31 LSBs) and a high part x_h (the remaining MSBs) so that $x = 2^{31} \cdot x_h + x_l$. The reduced value is then simply

$$x_r = x_h + x_l. \tag{1.1}$$

In the special case of Mersenne prime, the Montgomery form of a number is indeed identical to the number itself. This is because the Montgomery form of a is $aR \mod p$, and by choosing $R = p + 1$ which is a power of 2, we get that $aR = a \mod p$.

1.1.2 Redundant 32 bits form

For many systems, it is advantageous to work with units of 32 bits, taking x_l as the 32 LSBs and x_h as the remaining MSBs, so that $x = 2^{32} \cdot x_h + x_l$. The reduced value can then be calculated by

$$x'_r = 2x_h + x_l \tag{1.2}$$

[1] Field arithmetic is implemented in Icicle

Note that this is not yet the fully reduced value x_r, as one must then check again for overflows and subtract p until the result is smaller than p. If x comes from a product of two M31 elements, then $2x_h \leq 2^{31} - 4$ and $x_l < 2^{32}$. This means that $x'_r \leq 3 \cdot 2^{31} - 4 < 3p$, so up to 2 subtractions of p might be required. This procedure is what we get when using Mongomery form with $R = 2^{32}$ rather than $R = 2^{31}$.

It is possible to work in the redundant form presented above without performing a final reduction until the final stages of the program.
Adding two numbers in the redundant form can be done as:

```
# 32b addition
tmp[32:0] = in0[31:0] + in1[31:0]   # max: 2^33-2 = 2^32(1)+(2^32-2)
tmp[32:0] = 2 * tmp[32] + tmp[31:0] # max: 2(1)+(2^32-2) = 2^32(1)+(0)
out[31:0] = 2 * tmp[32] + tmp[31:0] # max: 2(1)+0 = 2
```

Subtracting two numbers in the redundant form can be done as (note that this requires 2 carry bits):

```
# 32b subtraction (2 carry bits)
tmp[32:0] = 3p + in0[31:0] - in1[31:0] # max: 3(2^31-1)+(2^32-1)-0
                                               = 2^32(2)+(2^31-4)
out[31:0] = 2 * tmp[33:32] + tmp[31:0] # max: 2(2)+(2^31-4) = 2^31
```

Negating a number in the redundant form can be done as:

```
# 32b negation
tmp[32:0] = 3p - in0[31:0]          # max: 3(2^31-1)-0 = 2^32(1)+(2^31-3)
out[31:0] = 2 * tmp[32] + tmp[31:0] # max: 2(1)+(2^31-3) = 2^31-1
```

Multiplying two numbers in the redundant form can be done as:

```
# 32b multiplication
tmp[63:0] = in0[31:0] * in1[31:0]      # max: 2^64-2^33+1 = 2^32(2^32-2)+1
tmp[33:0] = 2 * tmp[63:32] + tmp[31:0] # max: 2(2^32-2)+1 = 2^32(1)+(2^32-3)
out[31:0] = 2 * tmp[32] + tmp[31:0]    # max: 2(1)-(2^32-3) = 2^32-1
```

Reducing from the 32 bit redundant form to M31 can be done as:

```
# 32b reduction to 31b
tmp[31:0] = in0[31] + in1[30:0] # max: 1 + 2^31-1 = 2^31
tmp[30:0] = in0[31] + in1[30:0] # max: 1 + 0 = 1
out[30:0] = (tmp[30:0] == 0x7fff_ffff) ? 0 : tmp[30:0]
```

where it is important to check for the all-ones condition, which is not necessary when full reduction is performed at every step.

Working in the redundant form is safe for any program containing additions and multiplications. Taking advantage of special operations like multiplication by a power of 2 and inversion as detailed in the next subsections, requires prior reduction to canonical form.

1.1.3 Multiplying by a power of 2

A special case of multiplication occurs when multiplying by a power of 2. For integers, multiplying by a power of 2 is simply a bit shift. Since our modular reduction adds the overflowing bits back to the LSB, we get that for M31 prime field, multiplying by a power of 2 is simply a cyclic bit shift, with a cycle of 31 bits. This also applies to division by a power of 2, where the shift takes the opposite direction.

1.1.4 Inversion

Method 1 Inversion in M31 field can be done using Fermat's little theorem, which states that for every number $x \in \mathbb{F}_p$ we get that $x^{-1} = x^{p-2}$. This is an easy to implement method, but it is not the most efficient one. Raising to a power m can be done using the bit decomposition of m, which we denote $(m_{k-1}m_{k-2}\dots m_1m_0)$, where in our case as $m < p$ we have $k = 31$. Then

$$x^m = ((((x^{m_{k-1}})^2 \cdot x^{m_{k-2}})^2 \cdot x^{m_{k-3}})^2 \cdots x^{m_1})^2 \cdot x^{m_0} \tag{1.3}$$

This method requires s squarings and t multiplications by x, where s is the position of the MSB of m and t is one less than the total number of ones in the bit decomposition of m. Since p is a 31-bit number of all ones, $p-2$ contains but a single 0, and in total inversion takes $t + s = 30 + 29 = 59$ multiplication operations.

Method 2 The literature suggests a possibly more efficient algorithm for finding the inverse modulo a Mersenne prime [CP06]. The method appears in algorithm 1

Algorithm 1 Finding $x^{-1} \mod p$, where $p = 2^q - 1$

```
1:  function INV(x)
2:      (a, b, y, z) ← (1, 0, x, p)
3:      while do
4:          Find e such that 2^e | y          ▷ e is the number of LSB zeros in y
5:          y ← y/2^e
6:          a ← 2^{q-e} a mod p               ▷ Cyclic right-shift of a by e bits
7:          if y == 1 then
8:              return a
9:          end if
10:         (a, b, y, z) ← (a + b, a, y + z, y)
11:     end while
12: end function
```

The correctness of this algorithm comes from the fact that for every iteration of the loop, the value of $y/a \mod p$ remains constant and equal to the initialized value x, so when $y = 1$ then $a = x^{-1}$.

This algorithm notably requires only addition and bit shifts and not multiplications. However, we were unable to find an analytic calculation of how many iterations it requires. Empirical results with lower values of $q = (7, 13, 17, 19)$ suggest that at the worst case the algorithm needs $q\log(q)$ iterations, while the average case is about half of that. This means that while the complexity is the same for these two methods, it is possible that the second method has a smaller constant for most cases if not all of them.

1.2 Complex Field Extension

The Complex field extension is $C(\mathbb{F}_p) = \mathbb{F}_p[x]/(x^2+1)$, and we denote its elements as $(a,b) \equiv a + bi$ where $a, b \in \mathbb{F}_p$ and $i^2 = -1$. It is often convenient to refer to a and b as the Real and Imaginary parts of the Complex field element (a,b).

addition and subtraction in $C(\mathbb{F}_p)$ is done using term by term base field addition and subtraction

$$(a,b) \pm (c,d) = (a \pm c, b \pm d). \tag{1.4}$$

The multiplication law is

$$(a,b) \cdot (c,d) = (ac - bd, ad + bc), \tag{1.5}$$

and it can be done with 3 base field multiplications using Karatsuba

$$(a,b) \cdot (c,d) = (ac - bd, (a+b)(c+d) - ac - bd)\,. \tag{1.6}$$

One can also consider multiplication by a base field element as a term by term multiplication

$$(a,b) \cdot c = (ac, bc). \tag{1.7}$$

Inversions are done by

$$(a,b)^{-1} = (a,-b) \cdot (a^2 + b^2)^{-1}, \tag{1.8}$$

where the $()^{-1}$ on the RHS is a base field inversion.

1.2.1 Complex arithmetic on the circle-curve

The Complex field extension of the M31 field is useful as unlike the base field it contains roots of unity of 2^k, for k up to 31. These roots of unity all appear on the circle $a^2+b^2 = 1$, where some arithmetic operations simplify.

Squaring a point on the circle-curve can be done with just 2 base field multiplications (recall that multiplying by 2 is just a cyclic bit shift)

$$(a,b) \cdot (a,b) = (2a^2 - 1, 2ab). \tag{1.9}$$

Inversion on the circle-curve is just a flip of the sign of b

$$(a,b)^{-1} = (a,-b). \tag{1.10}$$

The set of all points on the circle-curve form a cyclic group of size $p + 1 = 2^{31}$. Since they all lie on the unit-circle, it is often convenient to identify them using a single variable corresponding to their angle $\theta \in 0, .., p$. In this context we can define all the usual trigonometric relations, such as

$$\omega = (a,b) \equiv e^{i\theta} \tag{1.11}$$

$$\cos\theta = a \tag{1.12}$$

$$\sin\theta = b \tag{1.13}$$

$$\omega^2 = (\cos 2\theta, \sin 2\theta) \tag{1.14}$$

$$\cos 2\theta = 2\cos^2\theta - 1 = 2a^2 - 1 \tag{1.15}$$

$$\sin 2\theta = 2\cos\theta\sin\theta = 2ab \tag{1.16}$$

$$... \tag{1.17}$$

1.2.2 Roots of unity

Few of the roots of unity also take a special form. The 4th roots of unity are $\omega_4 = i = (0, \pm 1)$, and the 8th roots of unity are $\omega_8 = (\pm 2^{15}, \pm 2^{15})$. Unfortunately, 16th roots of unity do not have a nice form. However, there are many 2^{31}-th roots of unity, and we can look for one with a relatively nice form. For example, we can take $\omega_{2^{31}} = (2, b)$. Requiring that this is on the circle gives $4 + b^2 = 1$, or $b = \sqrt{-3} = \pm 879471824$. Explicit checking shows that this is indeed a 2^{31}-th root-of-unity. Multiplying by this root of unity can be done with 2 base field multiplications (plus 2 bit shifts and 2 addition/subtractions):

$$(2, b) \cdot (c, d) = (2c - bd, 2d + bc). \tag{1.18}$$

1.3 Quad Field Extension

The quad field extension is $Q(\mathbb{F}_p) = C(\mathbb{F}_p)[x]/(x^2 - 2 - i)$. We can denote its elements as $[a, b] \equiv a + bu$ where $a, b \in C(\mathbb{F}_p)$ and $u^2 = 2 + i$. We use here $[\cdot]$ rather than $(\cdot)$ to distinguish that the elements inside are from the Complex field rather than the base field. We can also denote its elements directly over $\mathbb{F}_p$ as $(a, b, c, d) = a + bi + cu + dui$. This field is used for security reasons as its elements are 124-bit. The polynomial $x^2 - 2 - i$ is arbitrary and any irreducible polynomial over $C(\mathbb{F}_p)$ could be used. This particular choice was made by Starkware as evident from their code [Sta24].

addition and subtraction in $Q(\mathbb{F}_p)$ is done using term by term Complex field addition and subtraction

$$[a, b] \pm [c, d] = [a \pm c, b \pm d]. \tag{1.19}$$

Or equivalently term by term as quadruples of the base field. The multiplication law is

$$[a, b] \cdot [c, d] = [ac + (2 + i)bd, ad + bc], \tag{1.20}$$

and it can be done with 3 Complex field multiplications using Karatsuba

$$[a, b] \cdot [c, d] = [ac + (2 + i)bd, (a + b)(c + d) - ac - bd]\,. \tag{1.21}$$

Here, each Complex field multiplication can be done with 3 base field multiplications, to get a total of 9 base field multiplications per quad field multiplication.

One can also consider multiplication of a quad field element by a Complex field element as a term by term multiplication of Complex field elements

$$[a, b] \cdot c = [ac, bc]. \tag{1.22}$$

Inversions are done by

$$[a, b]^{-1} = [a, -b] \cdot (a^2 - (2 + i)b^2)^{-1}, \tag{1.23}$$

where the $()^{-1}$ on the RHS is a Complex field inversion.

1.3.1 Toom-Cook

Another option for the multiplication is by using Toom-Cook's trick. This reduces the number of base field multiplications from 9 to 7, but increase the cost by many additional additions, subtractions, and multiplications by constant. Benchmarking is required to determine which method is the fastest.

Consider an element $x = a_0 + a_1 i + a_2 u + a_3 ui$ in the quad field. Since $i = u^2 - 2$, we can rewrite this as $x = \alpha_0 + \alpha_1 u + \alpha_2 u^2 + \alpha_3 u^3$ where

$$\alpha_0 = a_0 - 2a_1,\ \alpha_1 = a_2 - 2a_3,\ \alpha_2 = a_1,\ \alpha_3 = a_3. \tag{1.24}$$

Now, say we wish to multiply this with another element $y = \beta_0 + \beta_1 u + \beta_2 u^2 + \beta_3 u^3$. We can write the result as

$$z(u) = x(u) \cdot y(u) = \sum_{i=0}^{6} \gamma_i u^i. \tag{1.25}$$

We see that we need 7 equations for the 7 variables γ_i. For this we evaluate the elements as polynomials of u on 7 points $u = \{0, 1, -1, 2, -2, 3, \infty\}$. We get the following set of linear equations

$$\begin{pmatrix} z(0) \\ z(1) \\ z(-1) \\ z(2) \\ z(-2) \\ z(3) \\ z(\infty) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & -1 & -1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 32 & 64 \\ 1 & -2 & 4 & -8 & 16 & -32 & 64 \\ 1 & 3 & 9 & 27 & 81 & 243 & 729 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \gamma_0 \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_5 \\ \gamma_6 \end{pmatrix} \tag{1.26}$$

We can invert this matrix to get

$$\begin{pmatrix} \gamma_0 \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_5 \\ \gamma_6 \end{pmatrix} = \frac{1}{120} \begin{pmatrix} 120 & 0 & 0 & 0 & 0 & 0 & 0 \\ -40 & 120 & -60 & -30 & 6 & 4 & -1440 \\ -150 & 80 & 80 & -5 & -5 & 0 & 480 \\ 50 & -70 & -5 & 35 & -5 & -5 & 1800 \\ 30 & -20 & -20 & 5 & 5 & 0 & -600 \\ -10 & 10 & 5 & -5 & -1 & 1 & -360 \\ 0 & 0 & 0 & 0 & 0 & 0 & 120 \end{pmatrix} \begin{pmatrix} z(0) \\ z(1) \\ z(-1) \\ z(2) \\ z(-2) \\ z(3) \\ z(\infty) \end{pmatrix} \tag{1.27}$$

The evaluations of $z(u)$ can each be calculated as a multiplication of 2 base field elements by $z(u) = x(u) \cdot y(u)$. This gives explicitly

$$z(0) = \alpha_0 \beta_0, \tag{1.28}$$
$$z(\pm 1) = (\alpha_0 \pm \alpha_1 + \alpha_2 \pm \alpha_3)(\beta_0 \pm \beta_1 + \beta_2 \pm \beta_3), \tag{1.29}$$
$$z(\pm 2) = (\alpha_0 \pm 2\alpha_1 + 4\alpha_2 \pm 8\alpha_3)(\beta_0 \pm 2\beta_1 + 4\beta_2 \pm 8\beta_3), \tag{1.30}$$
$$z(3) = (\alpha_0 + 3\alpha_1 + 9\alpha_2 + 27\alpha_3)(\beta_0 + 3\beta_1 + 9\beta_2 + 27\beta_3), \tag{1.31}$$
$$z(\infty) = \alpha_3 \beta_3 \tag{1.32}$$

Note that the division by 120 can be done by first dividing by 8 and then by 15. Since 8 is a power of 2, this division is a simple bit rotation. The division by 15 can be performed by multiplying by its inverse. The inverse of 15 in M31 field can be calculated by noting that (mod$p$)

$$1 = 2p + 1 = 2^{32} - 1 = (2^{16} + 1)(2^{16} - 1) \tag{1.33}$$

$$= (2^{16} + 1)(2^8 + 1)(2^4 + 1)(2^4 - 1) = 65537 \cdot 257 \cdot 17 \cdot 15 \tag{1.34}$$

So $15^{-1} = (2^{16} + 1)(2^8 + 1)(2^4 + 1)$. Dividing by 15 can then be done by 3 bit rotations and 3 additions. Having calculated the coefficients γ_i, we can now perform a polynomial modular reduction to get $z = \delta_0 + \delta_1 u + \delta_2 u^2 + \delta_3 u^3$. Note that $(u^2 - 2)^2 = i^2 = -1$, so we take $u^4 = 4u^2 - 5$. This gives $u^6 = 11u^2 - 20$ so

$$z = \underbrace{(\gamma_0 - 5\gamma_4 - 20\gamma_6)}_{\delta_0} + \underbrace{(\gamma_1 - 5\gamma_5)}_{\delta_1} u + \underbrace{(\gamma_2 + 4\gamma_4 + 11\gamma_6)}_{\delta_2} u^2 + \underbrace{(\gamma_3 + 4\gamma_5)}_{\delta_3} u^3 \tag{1.35}$$

From here we can return to the canonical form of $z = d_0 + d_1 i + d_2 u + d_3 ui$ by

$$d_0 = \delta_0 + 2\delta_2,\ d_1 = \delta_2,\ d_2 = \delta_1 + 2\delta_3,\ d_3 = \delta_3. \tag{1.36}$$

We can summarize this procedure: To multiply 2 quad field elements $x = (a_0, a_1, a_2, a_3)$ and $y = (b_0, b_1, b_2, b_3)$:

1. Calculate the evaluations

$$z(0) = (a_0 - 2a_1)(b_0 - 2b_1), \tag{1.37}$$

$$z(\pm 1) = (a_0 - a_1 \pm (a_2 - a_3))(b_0 - b_1 \pm (b_2 - b_3)), \tag{1.38}$$

$$z(\pm 2) = (a_0 + 2a_1 \pm 2(a_2 + 2a_3))(b_0 + 2b_1 \pm 2(b_2 + 2b_3)), \tag{1.39}$$

$$z(3) = (a_0 + 7a_1 + 3(a_2 + 7a_3))(b_0 + 7b_1 + 3(b_2 + 7b_3)), \tag{1.40}$$

$$z(\infty) = a_3 b_3 \tag{1.41}$$

2. The result is $z = (d_0, d_1, d_2, d_3)$ where

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \frac{15^{-1}}{8} \begin{pmatrix} -90 & 100 & 100 & 5 & 5 & 0 & -600 \\ -30 & 0 & 0 & 15 & 15 & 0 & -600 \\ 30 & 10 & -55 & 25 & -7 & -3 & 1080 \\ 10 & -30 & 15 & 15 & -9 & -1 & 360 \end{pmatrix} \begin{pmatrix} z(0) \\ z(1) \\ z(-1) \\ z(2) \\ z(-2) \\ z(3) \\ z(\infty) \end{pmatrix} \tag{1.42}$$

The matrix in 1.42 looks daunting, but we can represent it with only additions (or subtractions) and bit shifts. For example, multiplying by 9 requires one bit shift (to multiply by 8) and one addition (to add the original value with the shifted value). Multiplying by 7 can be done similarly with a subtraction instead of an addition. We propose here an optimization for calculating the 4 rows of eq. 1.42, however there may still be further optimizations that one can take.

$$d_0 = \frac{15^{-1}}{8} \cdot 5\Big(z(2) + z(-2) + 2z(0) + 20\big(z(1) + z(-1) - z(0) - 6z(\infty)\big)\Big) \tag{1.43}$$

This requires 12 add/subs, and 10 bit shifts (recall that multiplying by 15^{-1} takes 3 additions and 3 shifts). When calculating this we can store the values of $z(1)+z(-1)$, $z(2)+z(-2)$, $6z(\infty)$ for later use. Next:

$$d_1 = \frac{1}{8}\Big(z(2) + z(-2) - 2\big(z(0) + 20z(\infty)\big)\Big) \tag{1.44}$$

This requires 3 add/subs, and 3 bit shifts. the value $20z(\infty)$ is calculated by subtracting $z(\infty)$ from the stored value $6z(\infty)$ and then bit shifting the result to multiply by 4. This value is also stored. Next:

$$d_2 = \frac{15^{-1}}{8}\Bigg(10\Big(z(1)+z(-1)+3\big(z(0)+36z(\infty)\big)\Big) - 65z(-1) - 7\big(z(2)+z(-2)\big) + 32z(2) - 3z(3)\Bigg) \tag{1.45}$$

This requires 15 add/subs, and 12 bit shifts. we use here $36z(\infty) = 20z(\infty) + 2^4 z(\infty)$ and $65 = 64 + 1$. The last row is

$$d_3 = \frac{1}{8} \cdot \Big(z(2) + z(-1) - 2z(1) + 24z(\infty) + 15^{-1}\big(10z(0) - 9z(-2) - z(3)\big)\Big) \tag{1.46}$$

This requires 10 add/subs, and 6 bit shifts. This means that in total the second step of the Toom Cook calculation takes 40 add/subs and 31 bit shifts. The first step takes another 26 add/subs and 8 bit shifts. So the entire Toom Cook multiplication takes 7 base field multiplications, 66 base field add/subs, and 39 base field bit shifts.

Counting the number of base field operations in the Karatsuba method of the quad field, we get 9 multiplications, as well as 29 add/subs and 2 bit shifts. We can therefore expect that Toom Cook method will do better if a base field multiplication is more expensive than about 18.5 add/subs and 18.5 bit shifts.

2 Discrete Circle-Curve Transform

As we had seen previously in Section 1.2.1, the circle-curve over the M31 field consists of 2^{31} Complex pairs that are also a cyclic multiplicative group with unity $(1, 0)$. As such, it is possible to construct a Radix-2 Cooley-Tukey (CT) FFT up to that size that can transform polynomials with Complex coefficients. This Complex FFT can also analyze Real-valued sequences since they are a subset of the Complex sequences (see Chapter 2 of [Dom24]). In this section, we discuss the Discrete Circle-Curve Transform (DCCT), an alternative transform that uses a different basis, achieving slightly better performance for the case of Real-valued sequences. DCCT has a close relationship to the Discrete Cosine Transform [Str99] and is largely based on [UH24].

The standard univariate polynomial (basis) in Complex Discrete Fourier Transform (DFT) analysis is of the form

$$\{1, z, z^2, ..., z^{N-1}\} \tag{2.1}$$

$$|z| = 1 \tag{2.2}$$

where for Real-valued sequences of length N, it suffices to consider a basis of length $\frac{N}{2}$

$$\{1, z, z^2, ..., z^{\frac{N}{2}-1}\} \tag{2.3}$$

$$|z| = 1 \tag{2.4}$$

An alternative for the above is using the following bivariate polynomial basis of length N

$$\{1, x, x^2, ..., x^{\frac{N}{2}-1}, y, yx, yx^2, ..., yx^{\frac{N}{2}-1}\} \tag{2.5}$$

$$x^2 + y^2 = 1 \tag{2.6}$$

Note that any bivariate polynomial $\mathbb{F}_p[x, y]$ of individual degree $\frac{N}{2} - 1$ can be reduced to this form by repeatedly substituting the unit-circle constraint $y^2 = 1 - x^2$.[2]

The construction is general in the sense that it works for any Complex field that has a $2N$'th root-of-unity. The requirement for order $2N$ is necessary for the existence of an inverse transform.

Let us begin by defining ω, the $2N$'th root-of-unity and define its cartesian representation as[3]

$$\omega = \left(x = \cos\frac{\pi}{N}, y = \sin\frac{\pi}{N}\right) \tag{2.7}$$

The DCCT is a linear mapping between a length-N sequence of Real-valued evaluations over the unit-circle odd-subdomain $\omega \left\langle \omega^2 \right\rangle$ and Real-valued coefficients of a polynomial in a ring isomorphic to (2.5). Figure 1 depicts the DCCT evaluation domain for $N = 2^4$.

Let us describe a polynomial from the ring (2.5) as a univariate function of the angle of $\omega \left\langle \omega^2 \right\rangle$ (since we are over a unit-circle domain) and analyze it in a Decimation-in-Time (DIT) fashion

$$f(\theta) = \sum_{n=0}^{\frac{N}{2}-1} c_n(\cos\theta)^n + \sin\theta \sum_{n=0}^{\frac{N}{2}-1} c_{n+\frac{N}{2}}(\cos\theta)^n \tag{2.8}$$

$$\theta = \frac{(2k+1)\pi}{N} \quad \forall k \in [0, 1, .., N-1] \tag{2.9}$$

The idea is to recursively split $f(\theta)$ such that each split is a sum of an even function and odd function, where the odd function is a product of an odd function and another even function, namely

$$f = f^{\text{even}}_{\text{left}} + f^{\text{odd}}_{\text{right}} \cdot f^{\text{even}}_{\text{right}} \tag{2.10}$$

This can be achieved using the following recursive chain

$$f(\theta) \to f^{(1)}(\cos\theta) \to f^{(2)}(\cos 2\theta) \to f^{(3)}(\cos 4\theta) \to ... \tag{2.11}$$

We will now illustrate the DIT recursion by presenting it for size $N = 2^4$ using a step-by-step construction.
In the first step we split $f(\theta)$ as

$$f(\theta) = f^{(1)}_0(\cos\theta) + \sin\theta \cdot f^{(1)}_1(\cos\theta) \tag{2.12}$$

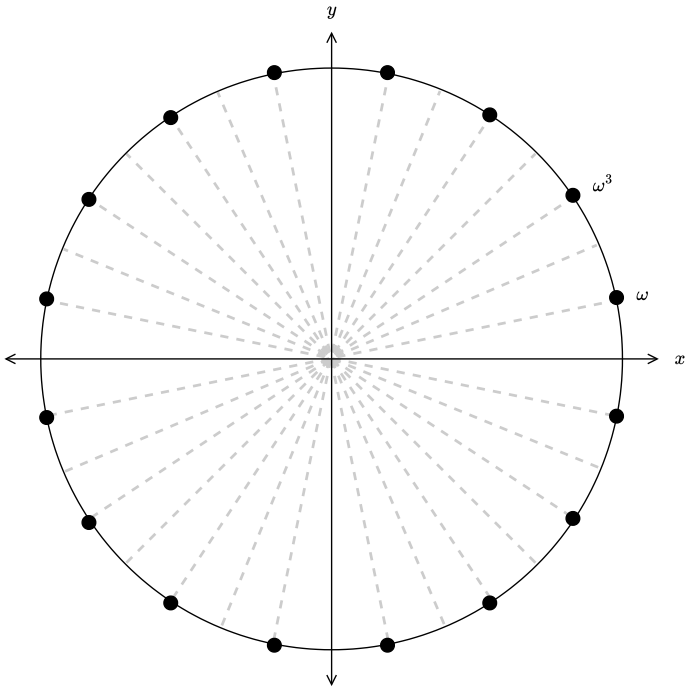

Figure 1: DCCT domain for $N = 2^4$

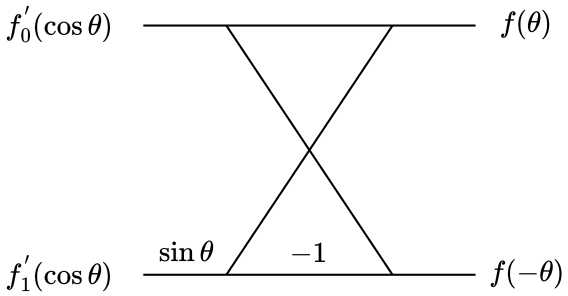

Figure 2: First step butterfly

and note that $f_i^{(1)}(\cos\theta)$ and that $\sin\theta$ is are even and odd in θ, respectively. This means that calculating $f(\pm\theta)$ can be achieved using a CT butterfly (see Figure 2).

[2] A different way to present this basis is as the ring $\mathbb{F}_p[x,y]/(x^2+y^2-1, x^{\frac{N}{2}}-1)$

[3] We define the trigonometric relations in the usual way as a mapping from the unit-circle angle, measured in radians, to the Cartesian coordinates of the corresponding point on the unit-circle.

In the subsequent steps we continue to split $f_i^{(1)}(\cos\theta)$ recursively as

$$\begin{aligned} f_i^{(1)}(\cos\theta) &= f_{i,0}^{(2)}(\cos 2\theta) + \cos\theta \cdot f_{i,1}^{(2)}(\cos 2\theta) \\ f_{i,j}^{(2)}(\cos 2\theta) &= f_{i,j,0}^{(3)}(\cos 4\theta) + \cos 2\theta \cdot f_{i,j,1}^{(3)}(\cos 4\theta) \\ f_{i,j,k}^{(3)}(\cos 4\theta) &= f_{i,j,k,0}^{(4)} + \cos 4\theta \cdot f_{i,j,k,1}^{(4)} \end{aligned} \tag{2.13}$$

where the terms $f_{i,j,k,l}^{(4)}$ on the final step are constant. The twiddle factors follow the recursive structure as

$$\cos 4\theta \to \cos 2\theta \to \cos\theta \to \sin\theta \tag{2.14}$$

and are depicted in Figure 3 where $\sin\theta$ corresponds to the angles of all black dots, $\cos\theta$ corresponds to the black dots of the upper half-circle, and $\cos 2\theta$ and $\cos 4\theta$ correspond to the white and grey dots, respectively. Clearly, all twiddle factors are subsets of the cyclic group $\langle\omega\rangle$.

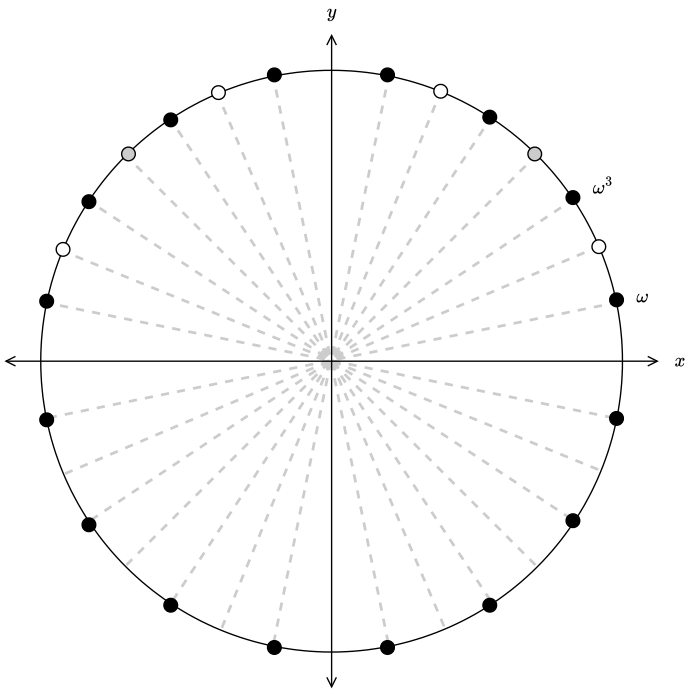

Figure 3: Twiddle factors for $N = 2^4$

Clearly, the number of distinct twiddles (dots) halves per stage due to periodicity. It is noteworthy, that going from $\cos\theta$ to $\cos 2\theta$ is equivalent to the non-linear translation $x \to 2x^2 - 1$ by elementary trigonometric relations.

The complete DIT DCCT and unnormalized IDCCT (inverse DCCT) for $N = 2^4$ are illustrated in Figures 4 and 5, respectively.

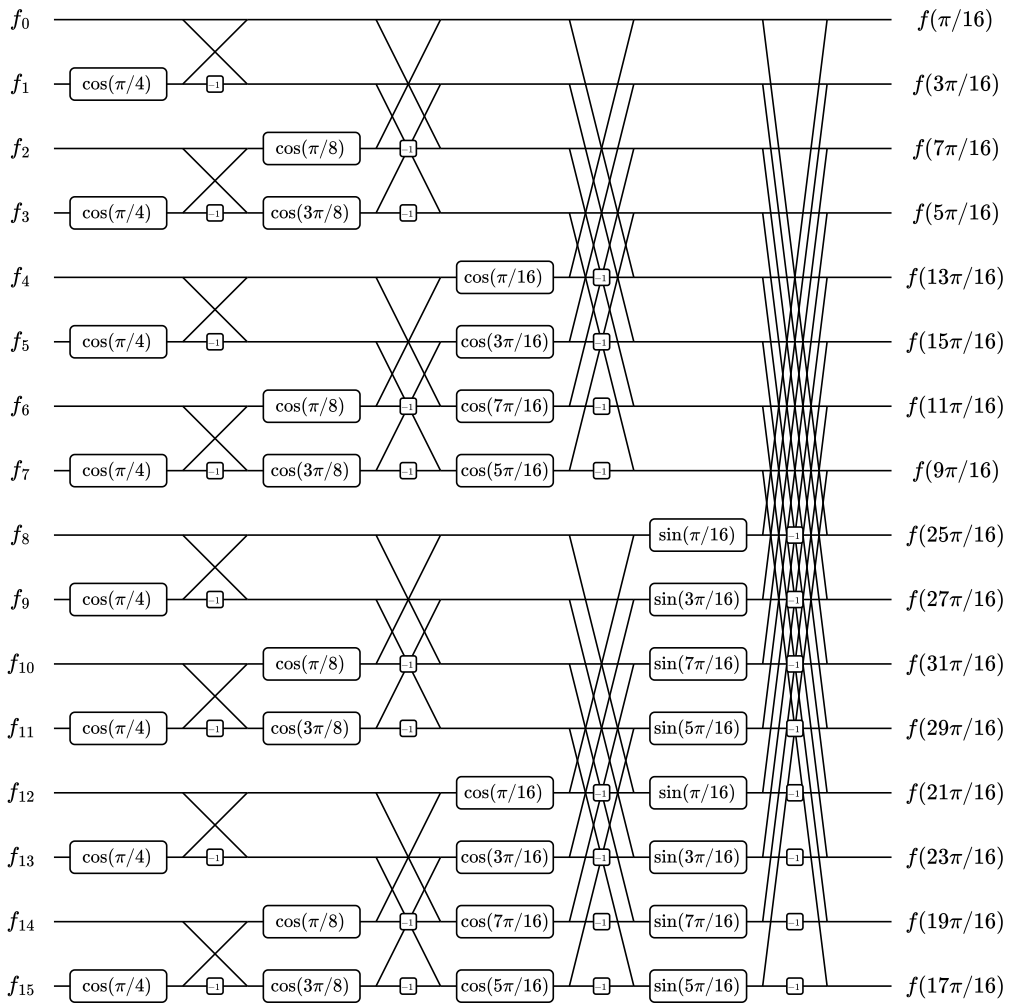

Figure 4: DCCT diagram for $N = 2^4$

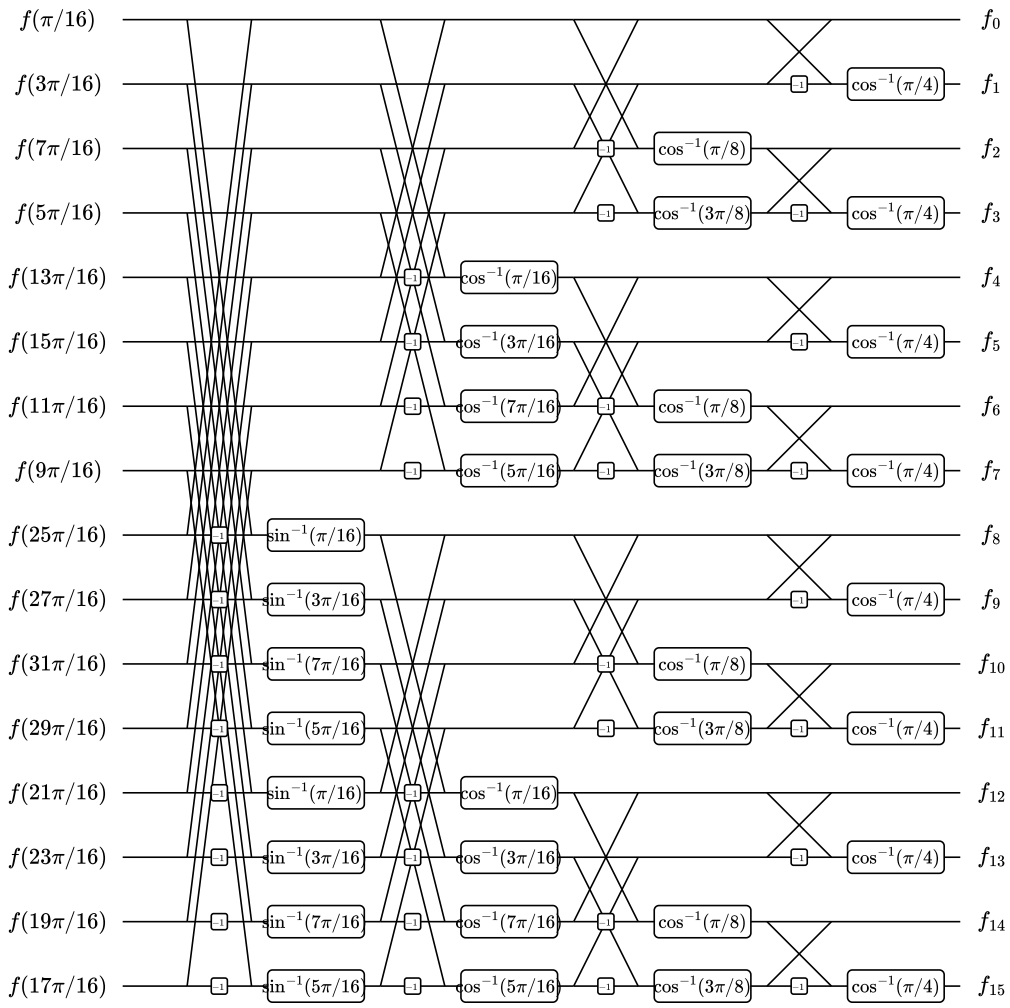

Figure 5: Unnormalized IDCCT diagram for $N = 2^4$

The ordering of the twiddle factors in the transform presented in Figures 4 and 5 follows the following construction:

```
def twiddle_order(n):
    idx = np.array([0])
    for i in np.arange(1, n + 1):
        idx = np.concatenate((idx, 2**i - 1 - idx))
    return idx
```

The ordering of the DCCT output in Figure 4 follows the following algorithm:

```
def output_order(n):
    idx = np.array([0])
    for i in np.arange(n):
```

```
        idx = np.concatenate((idx, 2**i + idx[::-1]))
    return idx
```

Note that other orders of the transform are possible and can be viewed as input and output permutations of the transform presented above.

3 Circle-Curve FRI

Circle-Curve FRI is very similar to univariate FRI [BSBHR18] but uses the DCCT mechanics instead of the standard NTT. We assume here that the polynomial in question, which we denote as $f^{(0)}(x,y)$,[4] has at most $M = 2^m$ degrees of freedom and that it can be defined as a coefficients vector over a DCCT basis of size M, as described in the previous section. In what follows, we implicitly assume that the polynomial $f^{(0)}(x,y)$ is represented by its over-sampled evaluation over the domain D of size $N = 2^b M$ where b is termed the *blowup factor*. The domain $D = \omega \langle \omega^2 \rangle$ where ω is a $2N$'th root-of-unity. As in DCCT, all subsequent polynomial evaluations $f^{(i)}$ are defined over subdomains of $\langle \omega \rangle$ and their sizes can be understood from the context. Let us refer to a commitment to the polynomial evaluation of $f^{(i)}$ as $[f^{(i)}]$.
The Prover's commit phase consists of the following steps:

1. Init: Send the commitment $[f^{(0)}]$.
2. Round 1:
   - Split the polynomial as $f^{(0)}(x,y) = f_0^{(1)}(x) + y f_1^{(1)}(x)$,
   - Receive a random challenge α_1 from the Verifier,
   - Calculate $f^{(1)}(x) = f_0^{(1)}(x) + \alpha_1 f_1^{(1)}(x)$,
   - Send the commitment $[f^{(1)}]$.
3. Rounds $j = 2, \ldots, r$:
   - Split the polynomial as $f^{(j-1)}(x) = f_0^{(j)}(2x^2 - 1) + x f_1^{(j)}(2x^2 - 1)$,
   - Receive a random challenge α_j from the Verifier,
   - Calculate $f^{(j)}(x) = f_0^{(j)}(x) + \alpha_j f_1^{(j)}(x)$,
   - Send the commitment $[f^{(j)}]$.
     * In the last round $j = r$, instead of the commitment $[f^{(r)}]$, the prover sends all the evaluations of the polynomial $f^{(r)}$, so that the verifier can check that it is indeed a polynomial with at most 2^{m-r} degrees of freedom.

Note that here the evaluations $f^{(j-1)}$ at the points x are defined by the evaluations of $f^{(j)}$ at the points $2x^2 - 1$. Denoting $x = \cos(k\theta)$ and $2x^2 - 1 = \cos(2k\theta)$, this is exactly the DCCT recursion depicted in equation (2.13).

[4] Note that here we choose to describe the polynomial as bivariate over the Cartesian coordinates and not as a univariate polynomial over the unit-circle angle.

In the query phase, the Verifier sends a random sample $Q_0 \in D$ and from it the Prover generates a list $Q_j = 2Q_{j-1}^2 - 1$. These are then back-propagated through the commitments where at each stage the commitments are opened to check that indeed

$$f^{(j-1)}(Q_{j-1}) = f_0^{(j)}(Q_j) + Q_{j-1} f_1^{(j)}(Q_j) \tag{3.1}$$

$$f^{(j)}(Q_j) = f_0^{(j)}(Q_j) + \alpha_j f_1^{(j)}(Q_j) \tag{3.2}$$

4 Acknowledgements

We would like to thank the Starknet foundation for supporting this work.

References

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (icalp 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[CP06] R. Crandall and C.B. Pomerance. *Prime Numbers: A Computational Perspective*. Lecture notes in statistics. Springer New York, 2006.

[Dom24] Yuval Domb. Ntt 201 - foundations of ntt hardware design, 2024. `https://github.com/ingonyama-zk/papers/blob/main/ntt_201_book.pdf`.

[Sta24] Starkware. M31 quad extension, 2024. `https://github.com/starkware-libs/stwo/blob/8d16c060b513ab2da4184e9abb5dd720abdb38a0/crates/prover/src/core/backend/simd/qm31.rs#L116`.

[Str99] Gilbert Strang. The discrete cosine transform. *SIAM Review*, 41(1):135–147, 1999.

[UH24] Shahar Papini Ulrich Haböck, David Levit. Circle starks. Cryptology ePrint Archive, Paper 2024/278, 2024. `https://eprint.iacr.org/2024/278.pdf`.