07 Nov 2023

ENOUGH POLYNOMIALS AND LINEAR ALGEBRA TO IMPLEMENT KYBER

I was once talking with a mathematician and trying to explain elliptic curve cryptography. Eventually, something clicked and they went "oh, that! I think there was a chapter about it in the book. You made a whole field out of it?"

Yes, in cryptography we end up focusing on a very narrow slice of the general math we use. I think that's good, and makes for good engineering, because we need to make computer programs that are *very good* at the thing they do, and it being narrow enough lets us get decently good at it.[1]

Anyway, over the years of implementing RSA and elliptic curves, we collectively learned quite a bit about modular arithmetic, large finite fields, and group logic. We did it wrong for many years, and now I think we mostly know how to do it right.[2]

Except now the post-quantum algorithms are coming, and they use lattices and matrices and polynomials. So, how much linear algebra and polynomial algebra do you need to know to implement these post-quantum cryptography primitives? Turns out, surprisingly little! The word "lattice" doesn't even show up in the spec except in the name.

If you want to learn math, this is not the article for you. If you want to implement ML-KEM (FIPS 203, formerly known as Kyber), you're about to be all set.

Note that all this is explained pretty clearly and in more detail in Section 2.4 of FIPS 203 (DRAFT). The FIPS spec actually does a great job of avoiding formalisms and speaking to engineers. Consider this a friendlier, even more pragmatic summary.

Polynomials

A polynomial is an element[3] of the ring R_q and looks like this

$$f = f_0 + f_1X + f_2X^2 + \cdots + f_{255}X^{255}$$

but you don't even need to know that. As far as you're concerned as an implementor, an ML-KEM polynomial is an array of 256 coefficients. Each coefficient is an integer modulo q, where q is 3329. An array of coefficients is said to be in $\mathbb{Z}_q^{256}$ because it's made of 256 coefficients, each in $\mathbb{Z}_q$, the integers modulo q.

$$f_0 + f_1X + f_2X^2 + \cdots + f_{255}X^{255} \in R_q$$
$$\downarrow$$
$$(f_0, f_1, f_2, \ \ldots, f_{255}) \in \mathbb{Z}_q^{256}$$

Each coefficient fits in a uint16, so you can write a type for a polynomial like `[256]uint16`.

To add or subtract two polynomials (or ring elements) you go coefficient by coefficient (`c[0] = a[0] + b[0]`, and so on). You never multiply ring elements directly in ML-KEM.

Modular arithmetic

As we said, each coefficient is an integer modulo 3329, so while it fits in a uint16, you do need to apply the right constant-time modular arithmetic to it.

You might be used to doing limb-based modular arithmetic for very large moduli. This is analogous but much easier because the field size is just 12 bits.

For addition and subtraction, you do a classic conditional subtraction.[4] You will find the conditional subtraction useful for ByteDecode$_{12}$, too.

Multiplication fits in a uint32. Then you have a choice of Montgomery or Barrett reduction. I find Barrett simpler and fast enough. (Note that the inner product of the Barrett reduction is 37 bits, so you need a uint64 intermediate value. Ask me how I know!) You can't use % because division is not always constant time in hardware.

The field is so small that you can exhaustively test your add, subtract, and multiply implementations in a fraction of a second. Do that.

NTT

One of the scariest-sounding parts of the ML-KEM specification is the *Number-Theoretic Transform*. Good news, you don't need to understand this one either.

What you need to know is that it's a different way to represent a polynomial. Each polynomial, or element of R_q, can be mapped to an element of T_q (the NTT domain) and back. The function doing the mapping is called the NTT, and the one mapping back is the inverse NTT (or NTT^{-1}). An element of T_q is called an "NTT representative", is denoted by a letter with a hat like $\hat{f}$, and is stored just like a polynomial: as 256 integers modulo q.

Technically, an NTT representative is a sequence of 128 polynomials each of them with two coefficients, but you don't need to think about that and you can represent both elements in R_q and in T_q with the same data structure, such as `[256]uint16`. Still, it would be a good idea to assign them separate types in your type system so you don't mix them up.

If you used Montgomery reduction and the Montgomery domain before, you're already familiar with the concept of mapping values to a different domain where some operation (again multiplication) is faster. Just like with the Montgomery domain, the data structure you use to represent elements is the same in and out of the domain, but they have semantically different types.

You can implement both NTT and NTT^{-1} without understanding the math behind them. Note that there is a complicated term in NTT and NTT^{-1} called *zeta*. You're expected to precompute that for its 128 possible values.

Addition and subtraction of NTT representatives works the same as with polynomials. You just can't mix and match them. (If you have a weak type system or generics, you can literally use the same functions.)

The whole reason to have the NTT is that multiplication is faster in the NTT domain. Indeed, there's a MultiplyNTTs algorithm that you can blindly implement. It has a γ term that you're expected to precompute like the *zeta* of NTT.

ML-KEM is special in that the NTT is part of the wire format, not just a behind-the-scenes optimization, because encryption and decryption keys are serialized and deserialized directly in their NTT representation.[5]

Matrices and vectors

Aside from coefficients, polynomials, and NTT representatives, the other main types you need to deal with are vectors and matrices.

A vector is, for our purposes, an array of k elements of R_q or T_q. k is 2, 3, or 4 depending on the parameter set. You can represent a vector as a `[k][256]uint16`. Vectors are denoted by bold lowercase letters like $\mathbf{v}$ or $\hat{\mathbf{v}}$.

$$\mathbf{v} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{k-1} \end{bmatrix}$$

Vectors are a special case of a $k \times 1$ matrix. That's $\mathrm{rows} \times \mathrm{columns}$, so a vector is a matrix with k rows and one column. The only other matrix we encounter is $\mathbf{A}$, which is a $k \times k$ matrix. I recommend storing it in a `[k*k][256]uint16` array, rather than `[k][k][256]uint16`, because the latter leads to indexing like `A[column][row]` to be consistent with the vector type, which is backwards compared to the notation $\mathbf{A}[\mathrm{row}, \mathrm{column}]$.

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,k-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k-1,0} & a_{k-1,1} & \cdots & a_{k-1,k-1} \end{bmatrix}$$

Applying a function like NTT or ByteEncode to a vector or a matrix just applies it to every element.

Vector addition is coordinate-wise (`c[0] = a[0] + b[0]`, and so on).

There are only two kinds of matrix multiplications in ML-KEM: matrix by vector ($\hat{\mathbf{A}} \circ \hat{\mathbf{v}}$), and transposed vector by vector ($\hat{\mathbf{v}}^\intercal \circ \hat{\mathbf{v}}$). They might look intimidating but they are both very simple. (They all have hats because as mentioned in the NTT section, we only do multiplications in the NTT domain.)

I'm not going to do a better job of explaining the general concept of matrix multiplication and dot product than other online resources, but I'll tell you how it works in the two cases you'll encounter. The rule to keep in mind is that the result of multiplying two matricies is another matrix. More specifically, the result of multiplication of a matrix $m \times n$ by a matrix $n \times p$ is a matrix of dimensions $m \times p$.

Matrix by vector multiplication ($\hat{\mathbf{A}} \circ \hat{\mathbf{v}}$, also known as transforming a vector by a matrix) is $k \times k \circ k \times 1 = k \times 1$, so it produces a vector. To do a matrix by vector multiplication, you go row-by-row in the matrix, and for each row you multiply every element by the corresponding vector element, and add all those products together. Each result vector element is the "dot product" (sum of corresponding products) of one matrix row by the vector.

Here's an example for $k = 3$, as in ML-KEM-768.

$$\hat{\mathbf{A}} \circ \hat{\mathbf{v}} = \begin{bmatrix} \hat{a}_{0,0} & \hat{a}_{0,1} & \hat{a}_{0,2} \\ \hat{a}_{1,0} & \hat{a}_{1,1} & \hat{a}_{1,2} \\ \hat{a}_{2,0} & \hat{a}_{2,1} & \hat{a}_{2,2} \end{bmatrix} \circ \begin{bmatrix} \hat{v}_0 \\ \hat{v}_1 \\ \hat{v}_2 \end{bmatrix} = \begin{bmatrix} \hat{a}_{0,0}\hat{v}_0 + \hat{a}_{0,1}\hat{v}_1 + \hat{a}_{0,2}\hat{v}_2 \\ \hat{a}_{1,0}\hat{v}_0 + \hat{a}_{1,1}\hat{v}_1 + \hat{a}_{1,2}\hat{v}_2 \\ \hat{a}_{2,0}\hat{v}_0 + \hat{a}_{2,1}\hat{v}_1 + \hat{a}_{2,2}\hat{v}_2 \end{bmatrix}$$

Transposed vector by vector ($\hat{\mathbf{v}}^\intercal \circ \hat{\mathbf{v}}$, also known as inner product) is $1 \times k \circ k \times 1 = 1 \times 1$, so in practice it's just a fancy way to say it's a single dot product, and it produces a "scalar" (a single element). You just multiply elements coordinate-wise and add them all together.

Here's an example, again for $k = 3$.

$$\hat{\mathbf{u}}^\intercal \circ \hat{\mathbf{v}} = \begin{bmatrix} \hat{u}_0 & \hat{u}_1 & \hat{u}_2 \end{bmatrix} \circ \begin{bmatrix} \hat{v}_0 \\ \hat{v}_1 \\ \hat{v}_2 \end{bmatrix} = \hat{u}_0\hat{v}_0 + \hat{u}_1\hat{v}_1 + \hat{u}_2\hat{v}_2$$

Note that you never actually apply a transposition to a vector or a matrix in memory. Transposed vectors are only used on the left hand side of a dot product as explained above, and the transposed matrix $\hat{\mathbf{A}}^\intercal$ in K-PKE.Encrypt can just be generated pre-transposed by inverting column and matrix in the XOF input to SampleNTT. (There is actually a bug in the first ML-KEM draft around this, which will probably be resolved by denoting $\hat{\mathbf{A}}^\intercal$ as $\hat{\mathbf{B}}$, which will make it clear you're not actually doing any transposition.)

That's it, this is all the linear algebra you need to implement ML-KEM! If you got this far, you might want to follow me on Bluesky at @filippo.abyssdomain.expert or on Mastodon at @filippo@abyssdomain.expert.

The picture

Walking around Trastevere, thinking about Certificate Transparency, I stopped to listen to an excellent jazz street performance. Great city. 🎷

My awesome clients—Sigsum, Protocol Labs, Latacora, Interchain, Smallstep, Ava Labs, Teleport, and Tailscale—are funding all my

work for the community and through our retainer contracts they get face time and unlimited access to advice on Go and cryptography.

Here are a few words from some of them!

Latacora — Latacora bootstraps security practices for startups. Instead of wasting your time trying to hire a security person who is good at everything from Android security to AWS IAM strategies to SOC2 and apparently has the time to answer all your security questionnaires plus never gets sick or takes a day off, you hire us. We provide a crack team of professionals prepped with processes and power tools, coupling individual security capabilities with strategic program management and tactical project management.

Protocol Labs — In case you're around Istanbul next week, for DevConnect or other purposes, make sure to check our LabWeek, which is packed with events. Among the notable ones, we have a libp2p day in case you're interested in powering your peer to peer communication using the same technology as most decentralized networks in the world. If you're more into Zero Knowledge proofs or decentralized storage, we have you covered too, and don't miss the League of Entropy summit in case you're into randomness!

Teleport — For the past five years, attacks and compromises have been shifting from traditional malware and security breaches to identifying and compromising valid user accounts and credentials with social engineering, credential theft, or phishing. Teleport Identity Governance & Security is designed to eliminate weak access patterns through access monitoring, minimize attack

surface with access requests, and purge unused permissions via mandatory access reviews.

Ava Labs — We at Ava Labs, maintainer of AvalancheGo (the most widely used client for interacting with the Avalanche Network), believe the sustainable maintenance and development of open source cryptographic protocols is critical to the broad adoption of blockchain technology. We are proud to support this necessary and impactful work through our ongoing sponsorship of Filippo and his team.

---

1. Sometimes a cryptography engineer will be math trained, and you see it because they will pull tricks no one else does to optimize stuff, or they'll prototype things in Sage faster than I can understand them. Sometimes they are also a good engineer, and we get math solutions for engineering problems, like Decaf. ↩

2. How to do elliptic curves right can be its own issue, but it involves fiat-crypto, Montgomery domain, complete formulas, prime order curves or cofactor removal via Decaf/Ristretto, bytes based APIs with well-defined decoding/encoding, and compressed points. That's how filippo.io/nistec, filippo.io/edwards25519, and gitlab.com/yawning/secp256k1-voi are made. ↩

3. Math-trained folks will correctly object that polynomial and ring elements are not the same thing, the same way integers and finite field elements are not, but for the purposes of this

article we'll use polynomial and ring element interchangeably. ↩

4. Conditional subtraction is where you subtract q and then add it back if the subtraction underflowed. For subtraction, you add q to the difference, then apply the conditional subtraction. ↩

5. "The role of the NTT" from the Kyber Round 3 submission is a good read about all this. ↩