NTRU Post-Quantum Encryption demo

Dmytro Husan

January 30, 2022

1 Example from lecture

Let us try to re-create example from lecture We use the following common parameters:

```
N = 7; p = 3; q = 41
```

Select random polynomial:

$$f = x^6 - x^4 + x^3 + x^2 - 1$$

Check that polynomials f_p and f_q with the property $f \cdot f_p = 1(\text{mod} p)$ and $f \cdot f_q = 1(\text{mod} q)$ exist.

1.1 balancedmod(f(x),q,N)

This is auxiliary helper function. It reduces every coefficient of a polynomial $f \in \mathbb{Z}[x]$ modulo q with additional balancing, so the result coefficients are integers in interval $[-q/2, +q/2]$. More specifically:

- for an odd q coefficients belong to $[-\frac{q-1}{2}, +\frac{q-1}{2}]$
- for an even q coefficients belong to $[-\frac{q}{2}, +\frac{q}{2} - 1]$

Finally the resulting polynomial is fit into $\mathbb{Z}[x]$ and returned.

```
def balancedmod(f,q):
    g = list(((f[i] + q//2) % q) - q//2 for i in range(N))
    Zx.<x> = ZZ[]
    return Zx(g)
```

Example:

$$\text{balancedmod}(1 + 31x + 32x^2 + 33x^3 - x^4, 32) = -x^4 + x^3 - x + 1$$

1.2 multiply(f(x), g(x))

The following function performs multiplication operation specific for NTRU, which works like a traditional polynomial multiplication with additional reduction of the result by $x^N - 1$

```
def convolution(f,g):
    return (f * g) % (x^N-1)
```

1.3 invertmodprime(f(x),p)

This routine calculates an inversion of a polynomial modulo $x^N - 1$ and then modulo p with assumption that p is prime number. Returns a polynomial $f_p \in \mathbb{Z}[x]$ such as $f \cdot f_p = 1(\text{mod} p$. An exception is thrown if such polynomial $f_p \in \mathbb{Z}[x]$ does not exist.

```
def invertmodprime(f,p):
    Zx.<x> = ZZ[]
    Zq.<z> = PolynomialRing(Integers(p))
    ZQphi.<Z> = Zq.quotient(z^N-1)
    a = f % p
    a = a.subs(x=z)
    k = 0
    b = 1*z^0
    c = 0*z^0
    f = a
    g = z^N-1

    if a.gcd(g) != 1:
        raise Exception("inversion dosen't exist!")
    while True:
        while list(f)[0] == 0:
            f /= Z
            c *= Z
            k += 1
        if find_degree(list(f)) == 0:
            b = 1/list(f)[0] * b
            res = Z^(N-k) * b
            return Zx(res.lift())
        if find_degree(list(f)) < find_degree(list(g)):
            f, g = g, f
            b, c = c, b
        u = list(f)[0] * (1/list(g)[0])
        f -= u*g
        b -= u*c
```

Example:

$$f = x^6 - x^4 + x^3 + x^2 - 1, p = 3, N = 7$$

$$f_p = \text{invertmodprime(f, p)} = x^6 + 2x^5 + x^3 + x^2 + x + 1$$

Note that this is exactly the inverse mentioned in lecture

1.4 invertmodprime(f(x), q, N)

This routine calculates an inversion of a polynomial modulo $x^N - 1$ and then modulo q. Returns a polynomial $f_q \in \mathbb{Z}[x]$ such as $f \cdot f_q = 1 (\text{mod} q$. An exception is thrown if such polynomial $f_q \in \mathbb{Z}[x]$ does not exist. Example:

$$f = x^6 - x^4 + x^3 + x^2 - 1, p = 3, N = 7, q = 41$$

$$f_q = \text{invertmodprime(f, q)} = 8x^6 + 26x^5 + 31x^4 + 21x^3 + 40x^2 + 2x + 37$$

Note that this is exactly the inverse mentioned in lecture.

1.5 generate_keys(f, g)

In this section we will generate public and secret key. To generate the key pair two polynomials f and g, with degree at most $N - 1$ and with coefficients in $\{-1, 0, 1\}$ are required. They can be considered as representations of the residue classes of polynomials modulo $X^N - 1$ in R. The polynomial $\mathbf{f} \in L_f$ must satisfy the additional requirement that the inverses modulo q and modulo p (computed using the Euclidean algorithm) exist, which means that $\mathbf{f} \cdot \mathbf{f}_p = 1 \pmod p$ and $\mathbf{f} \cdot \mathbf{f}_q = 1 \pmod q$ must hold. The public key $\mathbf{h}$ is generated computing the quantity. $\mathbf{h} = p\mathbf{f}_q \cdot \mathbf{g} \pmod q$. The secret key is a pair of randomly generated polynomials $(f(x), g(x))$

```
def generate_keys(polynomial_1 = None, polynomial_2= None):
    # validate params
    if validate_params():
        while True:
            try:
                if polynomial_1 is None or polynomial_2 is None:

                    f = generate_polynomial(d+1, d)
                    g = generate_polynomial(d, d)
                else:
                    # it use your polynomials
                    f = polynomial_1
                    g = polynomial_2


                # formula: find f_q, where: f_q (*) f = 1 (mod q)
                # assuming q is a power of 2
```

```
                f_q = invertmodprime(f,q)

                # formula: find f_p, where: f_p (*) f = 1 (mod p)
                # assuming p is a prime number
                f_p = invertmodprime(f,p)
                break

            except:
                pass

        #formula: public key = F_q ~ g (mod q)
        public_key = balancedmod(p * convolution(f_q,g),q)


        secret_key = f,f_p
        return public_key,secret_key

    else:
        print("")
```

Example: In this example the parameters (N, p, q) will have the values N = 7, p = 3 and q = 41 and therefore the polynomials f and g are of degree at most 6. The system parameters (N, p, q) are known to everybody. The polynomials are randomly chosen, so suppose they are represented by
$f(x) = x^6 - x^4 + x^3 + x^2 - 1$, $g(x) = x^6 + x^4 - x^2 - x$

$$public\_key = \text{get\_keys(f, g)} = 20x^6 + 40x^5 + 2x^4 + 38x^3 + 8x^2 + 26x + 30$$

1.6 encrypt(message, public_key)

The ciphertext $\mathbf{e}$ is generated computing the quantity. $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m} \pmod{q}$.

```
def encrypt(message, public_key, r):
    return balancedmod(convolution(public_key,r) + message,q)
```

Example. Let's choose message $\mathbf{m} = -x^5+x^3+x^2-x+1$ and random polynomial $\mathbf{r} = x^6 - x^5 + x - 1$

$$\mathbf{e} = \text{encrypt(message, public\_key)} = 31x^6 + 19x^5 + 4x^4 + 2x^3 + 40x^2 + 3x + 25$$

1.7 decrypt(encrypted_message, secret_key)

Let's decrypt the message. $\mathbf{a} = \mathbf{f} \cdot \mathbf{e} = p\mathbf{r} \cdot \mathbf{g} + \mathbf{f} \cdot \mathbf{m}$ (mod q). The next step will be to calculate $\mathbf{a}$ modulo p: $\mathbf{b} = \mathbf{a} = \mathbf{f} \cdot \mathbf{m}$ modulo p. $\mathbf{c} = \mathbf{f}_p \cdot \mathbf{b} = \mathbf{f}_p \cdot \mathbf{f} \cdot \mathbf{m} = \mathbf{m}$ (mod p).

```
def decrypt(encrypted_message, secret_key):
    # private key - f; additional variable stored for decryption - f_p
    f,f_p = secret_key

    # formula: a = f ~ encrypted_message (mod q)
    # balance coefficients of a for the integers in interval [-q/2, +q/2]
    a = balancedmod(convolution(encrypted_message,f),q)

    # formula: F_p ~ a (mod p) with additional balancing as above
    return balancedmod(convolution(a,f_p),p)
```

Example.

$$\mathbf{a} = \mathbf{f} \cdot \mathbf{e} = x^6 + 10x^5 + 33x^4 + 40x^3 + 40x^2 + x + 40$$

after central lift

$$\mathbf{b} = \mathbf{a} = x^6 - x^5 - x^3 + x^2 + x + 1$$

$$\mathbf{c} = -x^5 + x^3 + x^2 - x + 1$$

We got the original message.