karatsuba-BMM

November 18, 2025

[3]:
```
import numpy as np
```

[4]:
```
qp = np.array([
    281474976317441, 140737518764033, 140737470791681, 140737513783297,
    140737471578113, 140737513259009, 140737471971329, 140737509851137,
    140737480359937, 140737509457921, 140737481801729, 140737508671489,
    140737482981377, 140737506705409, 140737483898881, 140737504608257,
    140737484685313, 140737499496449, 140737485864961, 140737493729281,
    140737486520321, 140737490976769, 140737487306753, 140737488486401,
    281474975662081, 281474974482433, 281474966880257, 281474962554881,
    281474960326657, 281474957180929, 281474955476993, 281474952462337
], dtype=object)
```

[5]:
```
def folded_karatsuba3_barrett_modmul(a, b, q, mu=None):
    """
    Folded Karatsuba-3 Barrett Modular Multiplication
    Implements the exact algorithm from your LaTeX
    """
    if q <= 0:
        raise ValueError("Modulus q must be positive")

    # Calculate k = ceil(log2(q))
    k = q.bit_length()
    # Calculate m = ceil(k/3)
    m = (k + 2) // 3

    # Precompute mu if not provided
    if mu is None:
        mu = (1 << (2 * k)) // q  # floor(2^(2k) / q)

    # Step 1: Karatsuba-3 for a × b
    mask = (1 << m) - 1
    x0 = a & mask
    x1 = (a >> m) & mask
    x2 = a >> (2 * m)

    y0 = b & mask
    y1 = (b >> m) & mask
```

```
    y2 = b >> (2 * m)

    # Compute partial products (Karatsuba-split-three structure)
    z0 = x0 * y0
    z2 = x1 * y1
    z4 = x2 * y2
    z1 = (x0 + x1) * (y0 + y1)
    z3 = (x1 + x2) * (y1 + y2)
    z5 = (x0 + x2) * (y0 + y2)

    # Subtractions for avoiding recomputation
    z1_sub = z1 - z0 - z2
    z3_sub = z3 - z4 - z2
    z5_sub = z5 - z0 - z4 + z2

    z = z0 + (z1_sub << m) + (z5_sub << (2*m)) + (z3_sub << (3*m)) + (z4 <<␣
↪(4*m))

    # Step 2: Karatsuba-3 for m1 ×  (to compute m2)
    m1 = z >> (k - 1)

    # Split m1 and  for Karatsuba-3
    # m_mu = (max(m1.bit_length(), mu.bit_length()) + 2) // 3
    m_mu = 17
    mask_mu = (1 << m_mu) - 1

    m1_0 = m1 & mask_mu
    m1_1 = (m1 >> m_mu) & mask_mu
    m1_2 = m1 >> (2 * m_mu)

    mu_0 = mu & mask_mu
    mu_1 = (mu >> m_mu) & mask_mu
    mu_2 = mu >> (2 * m_mu)

    # Karatsuba-3 for m1 ×
    w_mu_0 = m1_0 * mu_0
    w_mu_2 = m1_1 * mu_1
    w_mu_4 = m1_2 * mu_2
    w_mu_1 = (m1_0 + m1_1) * (mu_0 + mu_1)
    w_mu_3 = (m1_1 + m1_2) * (mu_1 + mu_2)
    w_mu_5 = (m1_0 + m1_2) * (mu_0 + mu_2)

    w_mu_1_sub = w_mu_1 - w_mu_0 - w_mu_2
    w_mu_3_sub = w_mu_3 - w_mu_4 - w_mu_2
    w_mu_5_sub = w_mu_5 - w_mu_0 - w_mu_4 + w_mu_2

    # Original version of m3 calculation
```

```
    # m3 = w_mu_0 + (w_mu_1_sub << m_mu) + (w_mu_5_sub << (2*m_mu)) +␣
↪(w_mu_3_sub << (3*m_mu)) + (w_mu_4 << (4*m_mu)) >> (k + 1)

    # folded version of m3 calculation
    # Keep terms that can affect bits >= 49
    term3 = w_mu_3_sub << (3*m_mu - k - 1)  # << 2
    term4 = w_mu_4     << (4*m_mu - k - 1)  # << 19
    term2 = w_mu_5_sub >> (k + 1 - 2*m_mu)  # >> 15  (since 34 < 49)
    term1 = w_mu_1_sub >> (k + 1 - m_mu)

    m3 = term4 + term3 + term2  + term1

    # Step 3: Karatsuba-3 for m3 × q (using same structure as Step 1)
    m3_0 = m3 & mask
    m3_1 = (m3 >> m) & mask
    m3_2 = m3 >> (2 * m)

    # print(m3_0, m3_1, m3_2)

    q0 = q & mask
    q1 = (q >> m) & mask
    q2 = q >> (2 * m)

    # Karatsuba-3 for m3 × q
    w0 = m3_0 * q0
    w2 = m3_1 * q1
    w4 = m3_2 * q2
    w1 = (m3_0 + m3_1) * (q0 + q1)
    w3 = (m3_1 + m3_2) * (q1 + q2)
    w5 = (m3_0 + m3_2) * (q0 + q2)


    # t = z - m3 * q
    t = z - (w0 + (w1 - w0 - w2 << m) + (w5 - w0 - w4 + w2 << (2*m)) + (w3 - w4␣
↪- w2 << (3*m)) + (w4 << (4*m)))

    # Handle final result
    if t >= 2*q:
        t = t - 2*q
    if t >= q:
        t = t - q

    return t
```

[6]:
```
import random

def test_modular_multiplier(moduli, num_tests: int = 1000):
```

```
    """
    Test modular_multiply against Python's reference.

    Parameters:
        num_tests: number of random (a, b) pairs to test
        max_bits : maximum bit-length of a and b (e.g., 64, 128, 256)
    """


    for test_idx in range(num_tests):


        for i, p in enumerate(moduli):

            max_val = len(bin(p)[2:])

            # print(max_val)

            # Generate random a, b
            a = random.randint(0, 2**max_val-1) % p
            b = random.randint(0, 2**max_val-1) % p

            # print(f"Test {test_idx}: a = {a}, b = {b}")

            # Reference result (ground truth)
            ref = (a * b) % p

            # Your implementation
            try:
                result = folded_karatsuba3_barrett_modmul(a, b, p)
                # result = barrett_reduction(a, b, p)
                # print(f" Modulus {p}: result = {result}, reference = {ref}")
            except Exception as e:
                print(f"  Error at test {test_idx}, modulus index {i} (p={p}):␣
↪{e}")
                return False

            # Validate range
            if not (0 <= result < p):
                print(f"  Out of range at test {test_idx}, modulus {p}: got␣
↪{result}")
                return False

            # Compare
            if result != ref:
                print(f"  Mismatch at test {test_idx}, modulus {p}")
                print(f"    a = {a}")
```

```
                print(f"   b = {b}")
                print(f"   a*b mod p (ref) = {ref}")
                print(f"   your result       = {result}")
                return False

        if (test_idx + 1) % 100 == 0:
            print(f"  {test_idx + 1} tests passed...")

    print(f"  All {num_tests} tests passed!")
    return True

# ------------------ Run the test ------------------
if __name__ == "__main__":
    # You can adjust bit size (e.g., 48, 64, 96, 128)
    test_modular_multiplier(qp, num_tests=1000)
```

```
100 tests passed...
200 tests passed...
300 tests passed...
400 tests passed...
500 tests passed...
600 tests passed...
700 tests passed...
800 tests passed...
900 tests passed...
1000 tests passed...
All 1000 tests passed!
```

[ ]: