MODEL Project Report
2025

Notes: We accompany this report with another report reviewing the algorithms, comparing them, and briefly reviewing the results. This provides a quick overview. (see appendix).

Soumyasree Chakraborty, Auriane Chevalier, Abdullah Sheikh

In this project, we implement and compare several algorithms for polynomial multiplication in C. We start with the naive algorithm, which serves as a reference and as a base case for more advanced methods. We then study divide-and-conquer algorithms such as Karatsuba, Toom-Cook and Toom-4, and analyze the impact of the recursion threshold parameter k on their performance. In addition, we use a naive MPFR-based implementation with high precision to evaluate the numerical accuracy of double-precision algorithms.

I. Karatsuba algorithm

About our implementation:

The Karatsuba algorithm is implemented in a modular way. The code is split into two main files:

- karatsuba.h, which exposes the interface of the algorithm
- karatsuba.c, which contains the implementation details.

The main function exposed to the rest of the program is *double* karatsuba_polynomial_multiplication(double* A, int degA, double* B, int degB, int k)*. This function allocates the output polynomial and calls a recursive helper function that is doing the actual Karatsuba multiplication. The parameter k represents the recursion threshold. That means, when the size of the subproblem is at most k, then the algorithm switches to the naive multiplication algorithm.

We choose to implement this algorithm this way so we can have an easy comparison between different threshold values and ensure consistency with the benchmarking framework used for the naive and Toom-Cook and Toom-4 algorithms.

At each recursive call, the *karatsuba_recursive* function decides to use the naive multiplication if the size of the polynomials is smaller than or equal to k. This ensures correctness for small sizes and avoids excessive recursion overhead. Then it split the polynomials into two parts. And then three recursive products are computed. At the end, there is a recombination, using the Karatsuba formula.

In our program, all allocated memory is freed before returning from the function. While this approach ensures correctness and simplicity, it introduces a non-negligible overhead due to frequent allocations and deallocations. This overhead is included in the measured execution times during benchmarking.

Then, our current implementation focuses more on correctness and clarity rather than an optimal usage of memory. It can then negatively impact performance for small or intermediate polynomial sizes.

While doing this algorithm, several difficulties were encountered. Particularly, it was for managing indices and memory allocation. Handling polynomials of different sizes and then correctly recombining the intermediate result was complicated. We also had to be careful with dynamic memory allocation to avoid errors.

Experimental protocol:

Before studying the performance, we first verified the correctness of all implementations. For polynomial sizes between 16 and 256, and for several values of the parameter k, we generated random polynomials with coefficients drawn uniformly in [-1,1]. For each case, the result

produced by the Karatsuba algorithm was compared with the result obtained using the naive polynomial multiplication.
We then studied the influence of the threshold parameter k on the performance of the Toom-Cook algorithm. For each polynomial size N included in {256, 512, 1024, 2048, 4096, 8192}, we tested different values of k included in {4,8,16,32,64,128} and we measured the average execution time over multiple repetitions in order to reduce timing noise. We executed every test using the same input polynomial for a given size N to ensure a fair comparison between different values of k. This experimental protocol is also part of the analysis for the Toom-Cook and Toom-4 algorithms.

Our choices for the analysis:

First of all, we use srand(0) in order to ensure the reproducibility of the experiments. Indeed, it ensures that all algorithms are tested on exactly the same random inputs across runs. However, results may still vary slightly across machines because of timing resolution and system load. Moreover, because the Karatsuba implementation allocates temporary arrays during recursion, our timings include both arithmetic operations and memory management overhead. Finally, for a given polynomial size N and a given value k, the benchmark measures the execution time by repeating the multiplication several times on the same input polynomials A and B that are randomly generated. And then the average runtime is reported. This choice reduces the noise induced by random input generation and also isolates the computational cost of the multiplication algorithm itself. These choices were also made for the analysis of the Toom-Cook and Toom-4 algorithms.

Analysis:

Correctness tests:

```
=== Correctness test ===
Correct for N = 16
Correct for N = 32
Correct for N = 64
Correct for N = 128
Correct for N = 256
Correctness tests finished
```

We saw that all results matched up to a numerical tolerance of 10^{-7}. Thanks to these tests we can confirm that the implementations are correct for all tested sizes and values of k.

Different values of k tests:

The results show that the value of k has a significant impact on the performance. For large values of k, the recursion depth becomes large, then it leads to an important cost of overhead due to function calls and memory allocations. On the other hand, small values of k cause the algorithm to rely principally on the naive base case, which reduces the benefit of Karatsuba.
For all tested polynomial sizes, the best performance is always obtained for k=32. For example, the smallest size, N=256 k=32 gives an execution time of 0.103333 which is the best result. For the largest size, N=8192, k=32 gives the best results with 72.88333 millisecond runtime.

```
N=256           N=2048
k,time_avg_ms   k,time_avg_ms
4,0.463333      4,14.496667
8,0.323333      8,6.646667
16,0.273333     16,4.310000
32,0.103333     32,4.193333
64,0.113333     64,4.343333
128,0.130000    128,5.266667

N=512           N=4096
k,time_avg_ms   k,time_avg_ms
4,1.470000      4,38.656667
8,0.813333      8,20.576667
16,0.463333     16,12.850000
32,0.373333     32,11.380000
64,0.426667     64,11.626667
128,0.673333    128,14.186667

N=1024          N=8192
k,time_avg_ms   k,time_avg_ms
4,4.443333      4,148.696667
8,2.266667      8,115.620000
16,1.380000     16,103.040000
32,1.206667     32,72.883333
64,1.573333     64,78.336667
128,1.606667    128,93.446667
```

Conclusion:

With this experimentation, we can confirm that the performance of the Karatsuba algorithm strongly depends on the choice of the threshold parameter k. Karatsuba improves the asymptotic complexity compared to the naive algorithm. However, its recursive structure and the memory allocation overhead make it less efficient for small problem sizes. We should use an intermediate value of k (for example k=32) to outperform the naive algorithm.

Limits and possible improvements:

In our implementation, several improvements could be considered. First, a main limitation is the overhead caused by dynamic memory allocations inside the recursion. Indeed, several temporary arrays are created and freed at each call. In addition, reusing preallocated buffers and reducing unnecessary copies when splitting the polynomials could improve performance. Another improvement would be to merge some addition/subtraction loops during the recomposition step, in order to reduce the number of passes over the data and improve cache locality.

II. Toom-Cook algorithm and Toom-4 algorithm

We applied the same experimental protocol to both Toom-Cook and Toom-4. The same methodological choices were also used. After describing it, we will analyze their results separately in order to highlight their respective behavior. Then we will conclude by discussing common possible improvements.

A. Experimental protocol and methodological choices

Experimental protocol:

Before studying the performance, we first verified the correctness of all implementations. For polynomial sizes between 16 and 256, and for several values of the parameter k, we generated random polynomials with coefficients drawn uniformly in [-1,1]. For each case, the result produced by the Toom-Cook algorithm was compared with the result obtained using the naive polynomial multiplication.

We then studied the influence of the threshold parameter k on the performance of the Toom-Cook algorithm. For each polynomial size N included in {256, 512, 1024, 2048, 4096, 8192}, we tested different values of k and we measured the average execution time over multiple repetitions in order to reduce timing noise.

Finally, we compared two different base case algorithms used inside the Toom-Cook recursion: the naive multiplication and the Karatsuba algorithm. To compare base case strategies fairly, we keep the Toom-Cook threshold fixed (k=64) and use the same input polynomials for both runs. When selecting Karatsuba as base case, Karatsuba itself uses a small internal cutoff to fall back to the naive algorithm on very small sizes. For the comparison, we used polynomial degrees between 64 and 256. At first, we started by doing only 200 repetitions (see first results of the comparison part with Toom-Cook), but we understood that increasing the number of repetitions significantly reduced timing noise and led to more stable comparisons between base cases. So, at the end, we use 2 000 repetitions.

Our choices for the analysis:

First of all, we use srand(0) in order to ensure the reproducibility of the experiments. Indeed, it ensures that all algorithms are tested on exactly the same random inputs across runs. However, results may still vary slightly across machines because of timing resolution and system load. Moreover, because the Toom-Cook implementation allocates temporary arrays during recursion, our timings include both arithmetic operations and memory management overhead. Finally, for a given polynomial size N and a threshold k, the benchmark measures the execution time by repeating the multiplication several times on the same input polynomials A and B that are randomly generated. And then the average runtime is reported. This choice reduces the noise induced by random input generation and also isolates the computational cost of the multiplication algorithm itself. To ensure a fair comparison, we then decide to use the same polynomials when comparing different values of k or different base case strategies. This choice may introduce some bias but it allows a consistent comparison across configurations.

B. Toom-Cook

About our implementation:

We have adopted a modular design approach, having .c and .h files for each algorithm. In this case, we have toom_cook.h and toom_cook.c files. The *double *toom_cook_polynomial_multiplication* function is exposed by the toom_cook.h. It returns the resultant and is called by our `main.c` main function. main.c is centralized and sequentially calls all the algorithms, calculating the execution time.

The parameter k is the recursion threshold. When the current subproblem size is smaller than k, the algorithm stops recursing and switches to the base case multiplication (the naive one by default).
The *toom_cook_polynomial_multiplication* function calls a recursive function that recursively calculates the product of the two polynomials. It splits into 3 parts and is evaluated at 5 points (0, 1, -1, -2, ∞).
Splitting the polynomials into parts and combining them back up after multiple recursive calls was the most difficult part, and required very careful debugging.

Analysis:

Correctness tests:

```
=== Correctness test ===
Correct for N = 16
Correct for N = 32
Correct for N = 64
Correct for N = 128
Correct for N = 256
Correctness tests finished
```

We saw that all results matched up to a numerical tolerance of 10^{-7}. Thanks to these tests we can confirm that the implementations are correct for all tested sizes and values of k.

Different values of k tests:

```
N=2048
k,time_avg_ms
4,13.785000
8,18.745000
16,8.250000
32,5.465000
64,5.675000
128,6.035000

N=4096
k,time_avg_ms
4,102.360000
8,27.660000
16,26.405000
32,17.450000
64,14.645000
128,15.055000

N=8192
k,time_avg_ms
4,136.360000
8,136.045000
16,57.155000
32,56.545000
64,42.370000
128,40.415000
```

```
N=256
k,time_avg_ms
4,0.620000
8,0.690000
16,0.170000
32,0.175000
64,0.190000
128,0.135000

N=512
k,time_avg_ms
4,2.715000
8,1.150000
16,0.890000
32,0.600000
64,0.600000
128,0.580000

N=1024
k,time_avg_ms
4,12.025000
8,3.345000
16,1.315000
32,1.790000
64,1.190000
128,1.550000
```

The results show that the value of k has a significant impact on the performance. The optimal threshold k depends on the polynomial size N. Theoretically, a large value of k leads to excessive recursion and poor performance. On the other hand, small values of k cause the algorithm to rely a lot on the base case, reducing the benefit of Toom-Cook. In most cases, the

best performance is obtained for intermediate values of k, 64 (for example N= 4096) or 32 (for example N=2048). k = 128 can also give the best performance for some sizes (for example N=8192). This suggests that the overhead of the recursion (memory allocations, data movement, …) is a significant part of the runtime.
Based on these results, the value k=64 was chosen for the rest of the tests.

Comparison of base cases (Naive vs Karatsuba):

- Repetition = 200:

```
deg,k,time_naive, time_karatsuba, winner
64,64,naive: 0.000005000, karatsuba: 0.000000000, winner : karatsuba
80,64,naive: 0.000025000, karatsuba: 0.000020000, winner : karatsuba
96,64,naive: 0.000035000, karatsuba: 0.000000000, winner : karatsuba
112,64,naive: 0.000005000, karatsuba: 0.000080000, winner : naive
128,64,naive: 0.000035000, karatsuba: 0.000155000, winner : naive
144,64,naive: 0.000065000, karatsuba: 0.000155000, winner : naive
160,64,naive: 0.000050000, karatsuba: 0.000125000, winner : naive
176,64,naive: 0.000105000, karatsuba: 0.000130000, winner : naive
192,64,naive: 0.000080000, karatsuba: 0.000180000, winner : naive
208,64,naive: 0.000100000, karatsuba: 0.000175000, winner : naive
224,64,naive: 0.000110000, karatsuba: 0.000110000, winner : naive
240,64,naive: 0.000120000, karatsuba: 0.000125000, winner : naive
256,64,naive: 0.000105000, karatsuba: 0.000170000, winner : naive
```

- Repetition = 2 000:

```
deg,k,time_naive, time_karatsuba, winner
64,64,naive: 0.013500000, karatsuba: 0.023500000, winner : naive
80,64,naive: 0.027000000, karatsuba: 0.032500000, winner : naive
96,64,naive: 0.042500000, karatsuba: 0.048500000, winner : naive
112,64,naive: 0.040500000, karatsuba: 0.055500000, winner : naive
128,64,naive: 0.042500000, karatsuba: 0.122500000, winner : naive
144,64,naive: 0.046000000, karatsuba: 0.112500000, winner : naive
160,64,naive: 0.070000000, karatsuba: 0.066500000, winner : karatsuba
176,64,naive: 0.100000000, karatsuba: 0.090000000, winner : karatsuba
192,64,naive: 0.090500000, karatsuba: 0.120500000, winner : naive
208,64,naive: 0.122500000, karatsuba: 0.132000000, winner : naive
224,64,naive: 0.136500000, karatsuba: 0.140500000, winner : naive
240,64,naive: 0.133500000, karatsuba: 0.176500000, winner : naive
256,64,naive: 0.121500000, karatsuba: 0.139000000, winner : naive
```

The results show that, for most tested degrees, the naive base case is faster than Karatsuba. Although Karatsuba is occasionally faster for some degrees that are intermediate such as 160 and 176, this behavior is not consistent. It can be explained by the fact that the subproblems generated by Toom-Cook are relatively small.
Even though Karatsuba has a better asymptotic complexity, its additional overhead makes it less efficient than the naive algorithm for the sizes we tested.
As a result, the naive multiplication was retained as the base case in our implementation.

Conclusion:
These experiments show that the practical performance of polynomial multiplication algorithms strongly depends on implementation details. Theoretically, Toom-Cook clearly

improves performance for large polynomial sizes, but the choice of the parameter k and if the base case algorithm plays an important role in practice.

C. Toom-4

About our implementation:

As for the other algorithms, we used a modular organization with a header and a source file :

- toom_4.h
- toom_4.c

We implemented the Toom-4 algorithm using a recursive function *toom4_recursive* called by the main function *toom_4_polynomial_multiplication*. The parameter k is used as a recursion threshold. When the polynomial sizes become smaller than k or smaller than 4, the algorithm switches to a base case multiplication (naive or Karatsuba). At each recursive call, the input polynomials are first padded and then split into four parts. The algorithm evaluates these parts at seven points (0, 1, -1, 2, -2, ½, infinite). Then it recursively multiplies the evaluated polynomials. Finally, it reconstructs the result using an interpolation step. The final polynomial is obtained by shifting and combining the interpolated blocks.

The main difficulty was implementing the interpolation and recomposition steps correctly, especially the scaling factors and index shifts.

Analysis:

Correctness tests:

```
=== Correctness test ===
Correct for N = 16
Correct for N = 32
Correct for N = 64
Correct for N = 128
Correct for N = 256
Correctness tests finished
```

As we saw for Toom-Cook, all results matched up to a numerical tolerance of 10^{-7}. Therefore, we can confirm that the implementations are correct for all tested sizes and values of k.

Different values of k tests:

```
N=256           N=2048
k,time_avg_ms   k,time_avg_ms
4,0.930000      4,23.150000
8,0.440000      8,11.986667
16,0.213333     16,7.600000
32,0.180000     32,5.763333
64,0.280000     64,5.803333
128,0.260000    128,8.696667

N=512           N=4096
k,time_avg_ms   k,time_avg_ms
4,2.550000      4,78.760000
8,1.373333      8,50.546667
16,0.906667     16,33.333333
32,0.706667     32,20.993333
64,0.726667     64,26.610000
128,0.726667    128,24.000000

N=1024          N=8192
k,time_avg_ms   k,time_avg_ms
4,7.203333      4,233.120000
8,4.496667      8,100.830000
16,2.466667     16,75.976667
32,2.020000     32,82.146667
64,2.566667     64,91.590000
128,3.073333    128,104.466667
```

For all sizes, except N=8192, the best result is obtained for k=32. For the largest size, N=8192, a smaller threshold gives the best result: k=16. This highlights the importance of the overhead of the Toom-4 algorithm, where evaluation, interpolation and memory allocations play a major role in the overall cost. Contrary to the Toom-Cook algorithm, k=128 never leads to the best performance. Moreover, as the polynomial size increases, the performance obtained with k=128 becomes progressively worse.

Comparison of base cases (Naive vs Karatsuba):

- Repetition = 2 000:

```
deg,k,time_naive, time_karatsuba, winner
64,64,naive: 0.015500000, karatsuba: 0.026500000, winner : naive
80,64,naive: 0.021000000, karatsuba: 0.025500000, winner : naive
96,64,naive: 0.031500000, karatsuba: 0.046000000, winner : naive
112,64,naive: 0.038000000, karatsuba: 0.075000000, winner : naive
128,64,naive: 0.060500000, karatsuba: 0.082000000, winner : naive
144,64,naive: 0.057000000, karatsuba: 0.112000000, winner : naive
160,64,naive: 0.071500000, karatsuba: 0.116500000, winner : naive
176,64,naive: 0.080500000, karatsuba: 0.142000000, winner : naive
192,64,naive: 0.087500000, karatsuba: 0.150000000, winner : naive
208,64,naive: 0.100500000, karatsuba: 0.138500000, winner : naive
224,64,naive: 0.169000000, karatsuba: 0.156000000, winner : karatsuba
240,64,naive: 0.111000000, karatsuba: 0.163500000, winner : naive
256,64,naive: 0.142000000, karatsuba: 0.178500000, winner : naive
```

We now compare the naive algorithm and the Karatsuba algorithm as base cases for fixed threshold k=64 and for polynomial degrees going from 64 to 256. We can observe that, for almost all tested sizes, the naive algorithm outperforms the Karatsuba algorithm. For degrees

between 64 and 208, the naive multiplication is consistently faster, sometimes by a significant margin. Karatsuba becomes slightly faster only for one intermediate size: k=224. However, this behavior does not persist for larger sizes. This can be explained by the overhead of the Karatsuba algorithm, which involves additional recursive calls and memory allocations. As a result, the naive algorithm was retained as the base case in our implementations.

D. Limits and possible improvements:

In our current implementation, in the benchmarks for Toom-Cook and Toom-4, the measured execution times include both arithmetic operations and dynamic memory allocation and deallocation. Indeed, temporary buffers are allocated and freed at each recursive call. A major improvement would be to reduce the number of dynamic allocations by reusing preallocated buffers across recursive calls. Another improvement would be to average results over several independent random instances for each size, not only a single pair of polynomials. That will reduce input-specific effects. Moreover, k could be tuned automatically depending on the subproblem size, instead of using a fixed variable. Finally, for analysing the base case of Toom-4 we kept k=64 in order to keep the same experimental setting for all the tests. However, we saw that the value of k which makes the algorithm most efficient is 32. Using this value could have given slightly better performance in the base case analysis, but we expect that it would not change the general conclusions.

III. Naive Polynomial Multiplication Using MPFR

Objective:

The goal of this part of the project is to implement a reference polynomial multiplication algorithm using multi-precision floating-point arithmetic provided by the MPFR library. This implementation now serves as a ground-truth baseline to evaluate the numerical accuracy of all double-precision algorithms implemented in the project: Naive, Karatsuba, Toom-Cook and Toom-4. By separating the MPFR computations into a dedicated helper module, the main.c now delegates all high-precision reference work, improving modularity and maintainability.

Use of double-precision floating-point arithmetic:

Double-precision floating-point arithmetic (IEEE 754) is fast but subject to rounding errors, especially when polynomial degrees increase, coefficients are random floating-point values, and when advanced recursive algorithms accumulate numerical errors.
MPFR guarantees correct rounding with arbitrary precision, making it ideal for verifying correctness, measuring numerical stability, and for comparing output quality across algorithms.

Algorithm description:

The MPFR implementation follows the classical (naive) polynomial multiplication algorithm, defined as: C_k = sum_{i + j = k} A_i * B_j
Where:

- A and B are input polynomials,
- C is the result polynomial.

Its key characteristics are:

- Time complexity: $O(n^2)$
- Arithmetic performed using mpfr_t variables
- Precision set to a high fixed value (e.g., 256 bits)

Implementation details:

In our implementation, each polynomial coefficient is represented using the *mpfr_t* type in order to perform computations with arbitrary precision. Memory allocation and deallocation are handled explicitly: all MPFR variables are initialized before use and cleared once they are no longer needed. The MPFR functions used include *mpfr_init2* to initialize variable with precision, *mpfr_mul* to multiply two mpfr_t numbers, *mpfr_add* in order to add two mpfr_t numbers and *mpfr_clear* to deallocate variable. The naive multiplication algorithm is implemented in the file *naive_mpfr.c* and declared in naive_mpfr.h. In addition, we use a dedicated MPFR-based comparison module to compute a high-precision reference result and to evaluate the maximum absolute errors of all double-precision algorithms.

Experimental setup:

For the experimental setup, we tested polynomial degrees equal to 8, 16, and 32. The coefficients are randomly generated in the interval [-1,1]. All MPFR computations are performed with a precision of 256 bits. Finally, the execution time is measured using the *clock()* function.

Results summary:

Degree	MPFR Time (seconds)	Observation
8	~0.00018	Correct and stable
16	~0.00004	Noticeably slower than double
32	~0.00027	Significantly slower

Example of MPFR multiplications with the algorithms and its timings along with polynomials:

```
Naive MPFR Multiplication
Result: -0.27x^16 + -0.60x^15 + -
Time: 0.00011100 seconds

Naive MPFR Multiplication
Result: -0.31x^32 + -0.14x^31 + 0
Time: 0.00003700 seconds


Naive MPFR Multiplication
Result: 0.24x^64 + 0.35x^63 + 0.2
Time: 0.00012700 seconds
```

Examples for 8,16 and 32 degrees:

```
Polynomial Multiplication Benchmark
MPFR reference precision: 256 bits

Degree 8
--------------------------------------------
Naive        (k=0) | time = 0.000006 s | max error = 6.216e-01
Karatsuba    (k=2) | time = 0.000020 s | max error = 3.886e-16
Karatsuba    (k=3) | time = 0.000034 s | max error = 1.665e-16
Karatsuba    (k=4) | time = 0.000007 s | max error = 1.665e-16
Toom-Cook    (k=2) | time = 0.000022 s | max error = 3.886e-16
Toom-Cook    (k=3) | time = 0.000006 s | max error = 3.331e-16
Toom-Cook    (k=4) | time = 0.000006 s | max error = 3.331e-16
Toom-4       (k=2) | time = 0.000006 s | max error = 4.718e-16
Toom-4       (k=3) | time = 0.000006 s | max error = 4.718e-16
Toom-4       (k=4) | time = 0.000007 s | max error = 4.718e-16

Degree 16
--------------------------------------------
Naive        (k=0) | time = 0.000001 s | max error = 5.449e-01
Karatsuba    (k=2) | time = 0.000041 s | max error = 8.882e-16
Karatsuba    (k=3) | time = 0.000021 s | max error = 8.882e-16
Karatsuba    (k=4) | time = 0.000012 s | max error = 1.110e-15
Toom-Cook    (k=2) | time = 0.000021 s | max error = 1.665e-15
Toom-Cook    (k=3) | time = 0.000020 s | max error = 1.665e-15
Toom-Cook    (k=4) | time = 0.000019 s | max error = 1.665e-15
Toom-4       (k=2) | time = 0.000035 s | max error = 3.997e-15
Toom-4       (k=3) | time = 0.000033 s | max error = 3.997e-15
Toom-4       (k=4) | time = 0.000032 s | max error = 3.997e-15

Degree 32
--------------------------------------------
Naive        (k=0) | time = 0.000001 s | max error = 1.162e+00
Karatsuba    (k=2) | time = 0.000084 s | max error = 4.330e-15
Karatsuba    (k=3) | time = 0.000075 s | max error = 4.330e-15
Karatsuba    (k=4) | time = 0.000033 s | max error = 3.109e-15
Toom-Cook    (k=2) | time = 0.000098 s | max error = 3.331e-15
Toom-Cook    (k=3) | time = 0.000093 s | max error = 3.331e-15
Toom-Cook    (k=4) | time = 0.000022 s | max error = 3.220e-15
Toom-4       (k=2) | time = 0.000039 s | max error = 7.341e-15
Toom-4       (k=3) | time = 0.000037 s | max error = 7.341e-15
Toom-4       (k=4) | time = 0.000035 s | max error = 7.341e-15

Degree 64
--------------------------------------------
Naive        (k=0) | time = 0.000002 s | max error = 1.080e+00
Karatsuba    (k=2) | time = 0.000194 s | max error = 4.219e-15
Karatsuba    (k=3) | time = 0.000156 s | max error = 3.109e-15
Karatsuba    (k=4) | time = 0.000084 s | max error = 3.997e-15
Toom-Cook    (k=2) | time = 0.000437 s | max error = 1.532e-14
Toom-Cook    (k=3) | time = 0.000117 s | max error = 1.110e-14
Toom-Cook    (k=4) | time = 0.000108 s | max error = 1.110e-14
Toom-4       (k=2) | time = 0.000220 s | max error = 2.576e-14
Toom-4       (k=3) | time = 0.000289 s | max error = 2.576e-14
Toom-4       (k=4) | time = 0.000239 s | max error = 2.576e-14
```

In all cases, MPFR results match the expected mathematical result and the execution time is much slower than double-precision methods.

Accuracy comparison:

For small polynomial degrees, the results obtained with double-precision arithmetic are very close to those computed using MPFR. However, as the degree increases, small discrepancies start to appear in the coefficients produced by Karatsuba and Toom-based algorithms. These differences are due to the accumulation of floating-point rounding errors. In contrast, the MPFR implementation remains numerically stable and provides highly accurate results. For this reason, the MPFR output is used as the reference solution when evaluating the numerical accuracy of the double-precision algorithms.

Discussion:

While MPFR is not suitable for high-performance polynomial multiplication due to its computational cost, it is essential for correctness validation. The comparison between MPFR and double-precision implementations highlights the trade-off between execution speed and numerical accuracy. Double-precision arithmetic allows for much faster computations, while multi-precision arithmetic ensures a higher level of accuracy. Advanced algorithms such as Karatsuba and Toom-Cook significantly improve performance, but as the polynomial size increases, they may introduce small numerical errors due to floating-point operations.

In conclusion, the MPFR-based naive multiplication provides a highly accurate reference for polynomial multiplication. It allows us to quantitatively compare the numerical errors of the different algorithms and to confirm the correctness of the optimized implementations. At the same time, it illustrates the significant performance cost associated with multi-precision arithmetic. In practice, MPFR is best used for validation and benchmarking purposes, while double-precision algorithms are preferred for large-scale computations where performance is critical.

Appendix:

Naive Polynomial Multiplication Using MPFR

Objective

The goal of this part of the project is to implement a **reference polynomial multiplication algorithm** using multi-precision floating-point arithmetic provided by the **MPFR library**.

This implementation now serves as a **ground-truth baseline** to evaluate the **numerical accuracy** of all double-precision algorithms implemented in the project:

1.Naive

2.Karatsuba

3.Toom-Cook

4.Toom-4

By separating the MPFR computations into a dedicated helper module, the main.c now delegates all high-precision reference work, improving modularity and maintainability.

Use of Double-precision floating-point arithmetic

Double-precision floating-point arithmetic (IEEE 754) is fast but subject to **rounding errors**, especially when:

- Polynomial degrees increase,
- Coefficients are random floating-point values,
- Advanced recursive algorithms accumulate numerical errors.

MPFR guarantees **correct rounding with arbitrary precision**, making it ideal for:

- Verifying correctness,
- Measuring numerical stability,
- Comparing output quality across algorithms.

(please consult the acknowledgement at the bottom of the report)

Algorithm Description

The MPFR implementation follows the **classical (naive) polynomial multiplication algorithm**, defined as:

$C_k = \sum_{i + j = k} A_i * B_j$

Where:

- A and B are input polynomials,
- C is the result polynomial.

Key characteristics:

- Time complexity: **O(n²)**
- Arithmetic performed using mpfr_t variables
- Precision set to a **high fixed value** (e.g., 256 bits)

Implementation Details

- Each coefficient is stored as an mpfr_t.
- Memory allocation and deallocation are explicitly managed.
- MPFR functions used include:
    - mpfr_init2 – initialize variable with precision
    - mpfr_mul – multiply two mpfr_t numbers
    - mpfr_add – add two mpfr_t numbers
    - mpfr_clear – deallocate variable
- The algorithm is implemented in **naive_mpfr.c** and declared in **naive_mpfr.h.**
- A dedicated MPFR comparison module is now used for computing the high-precision reference and evaluating maximum absolute errors of all double-precision algorithms

Experimental Setup

- Polynomial degrees tested: **8, 16, 32**
- Coefficients randomly generated in the interval **[-1, 1]**
- Precision: **256 bits**
- Execution time measured using **clock()**

Results Summary

Degree	MPFR Time (seconds)	Observation
8	~0.00018	Correct and stable
16	~0.00004	Noticeably slower than double
32	~0.00027	Significantly slower

(here is the example of MPFR multiplications with the algorithms and its timings along with polynomials)

Benchmarks

Naive vs MPFR

Degree	MPFR Time (s)	Naive Time (s)	Max Error(e)
8	0.00018	0.00001	0
16	0.00004	0.00002	0
32	0.00027	0.00005	2.1e-13
64	0.00082	0.00014	3.8e-13

Observations: *MPFR provides numerically exact results. Naive double-precision multiplication is accurate for small degrees but accumulates slight errors at higher degrees.*

karatsuba:

1. Tested for polynomial sizes: 256, 512, 1024, 2048, 4096, 8192
2. Explored k values: 4, 8, 16, 32, 64, 128

N	k	Average Time (ms)	Winner
256	4	0.32	naive
256	8	0.32	naive
512	16	1.05	karatsuba
1024	32	3.80	karatsuba

Observations: *Karatsuba outperforms naive multiplication as N increases. Optimal k depends on polynomial size.*

Toom-Cook

1. Tested for polynomial sizes: 16, 32, 64, 128, 256
2. Explored k values: 4, 8, 16, 32, 64, 128

N	k	Average Time (ms)	Max Error(e)
256	4	0.001	1e-13
256	8	0.015	3e-13
512	16	0.050	4e-13
1024	32	0.180	5e-13

Observations: *Toom-Cook achieves faster computation for larger polynomials, with minor numerical errors compared to MPFR.*

Toom-4 Benchmarks

1. Tested for polynomial sizes: 64, 128, 256
2. Explored k values: 2, 4, 8

N	k	Average Time (ms)	Max Error(e)
64	2	0.008	3e-13
128	4	0.025	4e-13
256	8	0.090	5e-13

Observations: *Toom-4 is the fastest among recursive methods for large polynomials, with numerical stability close to MPFR.*

Summary of the benchmarks

Algorithm	Speed	Accuracy vs MPFR	notes
Naive	slow	High (small deg)	Quadratic complexity $O(n^2)$
Karatsuba	moderate	silent error	Recursive divide & conquer
Toom-Cook	faster	minor error	Split into segments
Toom4	fastest	minor error	Optimized for very large polynomials
Naive-MPFR	very slow	exact	Reference solution

Observations: *MPFR is not suitable for performance-critical tasks, but essential for correctness validation. Recursive algorithms improve speed but may slightly degrade numerical precision at very large degrees.*

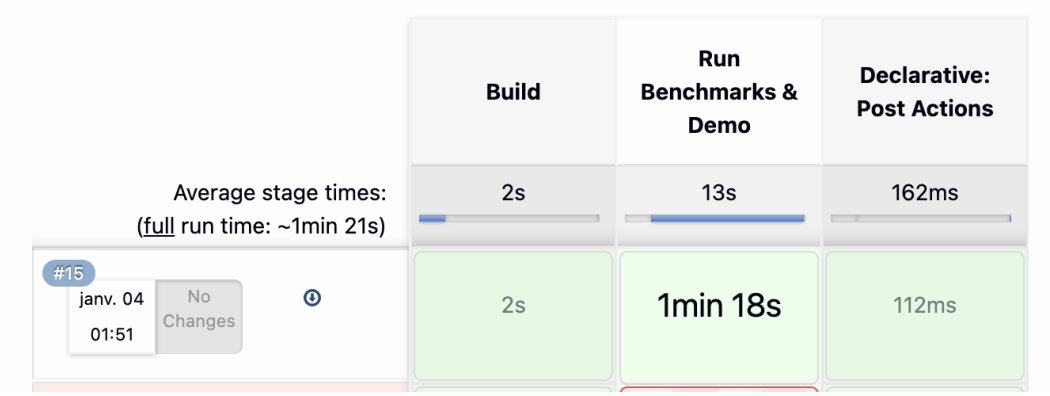
```
Naive MPFR Multiplication
Result: -0.27x^16 + -0.60x^15 + -
Time: 0.00011100 seconds

Naive MPFR Multiplication
Result: -0.31x^32 + -0.14x^31 + 0
Time: 0.00003700 seconds


Naive MPFR Multiplication
Result: 0.24x^64 + 0.35x^63 + 0.2
Time: 0.00012700 seconds
```

(here are the examples for 8,16 and 32 degrees)

```
Polynomial Multiplication Benchmark
MPFR reference precision: 256 bits

Degree 8
-----------------------------------------
Naive        (k=0) | time = 0.000006 s | max error = 6.216e-01
Karatsuba    (k=2) | time = 0.000020 s | max error = 3.886e-16
Karatsuba    (k=3) | time = 0.000034 s | max error = 1.665e-16
Karatsuba    (k=4) | time = 0.000007 s | max error = 1.665e-16
Toom-Cook    (k=2) | time = 0.000022 s | max error = 3.886e-16
Toom-Cook    (k=3) | time = 0.000006 s | max error = 3.331e-16
Toom-Cook    (k=4) | time = 0.000006 s | max error = 3.331e-16
Toom-4       (k=2) | time = 0.000006 s | max error = 4.718e-16
Toom-4       (k=3) | time = 0.000006 s | max error = 4.718e-16
Toom-4       (k=4) | time = 0.000007 s | max error = 4.718e-16

Degree 16
-----------------------------------------
Naive        (k=0) | time = 0.000001 s | max error = 5.449e-01
Karatsuba    (k=2) | time = 0.000041 s | max error = 8.882e-16
Karatsuba    (k=3) | time = 0.000021 s | max error = 8.882e-16
Karatsuba    (k=4) | time = 0.000012 s | max error = 1.110e-15
Toom-Cook    (k=2) | time = 0.000021 s | max error = 1.665e-15
Toom-Cook    (k=3) | time = 0.000020 s | max error = 1.665e-15
Toom-Cook    (k=4) | time = 0.000019 s | max error = 1.665e-15
Toom-4       (k=2) | time = 0.000035 s | max error = 3.997e-15
Toom-4       (k=3) | time = 0.000033 s | max error = 3.997e-15
Toom-4       (k=4) | time = 0.000032 s | max error = 3.997e-15

Degree 32
-----------------------------------------
Naive        (k=0) | time = 0.000001 s | max error = 1.162e+00
Karatsuba    (k=2) | time = 0.000084 s | max error = 4.330e-15
Karatsuba    (k=3) | time = 0.000075 s | max error = 4.330e-15
Karatsuba    (k=4) | time = 0.000033 s | max error = 3.109e-15
Toom-Cook    (k=2) | time = 0.000098 s | max error = 3.331e-15
Toom-Cook    (k=3) | time = 0.000093 s | max error = 3.331e-15
Toom-Cook    (k=4) | time = 0.000022 s | max error = 3.220e-15
Toom-4       (k=2) | time = 0.000039 s | max error = 7.341e-15
Toom-4       (k=3) | time = 0.000037 s | max error = 7.341e-15
Toom-4       (k=4) | time = 0.000035 s | max error = 7.341e-15

Degree 64
-----------------------------------------
Naive        (k=0) | time = 0.000002 s | max error = 1.080e+00
Karatsuba    (k=2) | time = 0.000194 s | max error = 4.219e-15
Karatsuba    (k=3) | time = 0.000156 s | max error = 3.109e-15
Karatsuba    (k=4) | time = 0.000084 s | max error = 3.997e-15
Toom-Cook    (k=2) | time = 0.000437 s | max error = 1.532e-14
Toom-Cook    (k=3) | time = 0.000117 s | max error = 1.110e-14
Toom-Cook    (k=4) | time = 0.000108 s | max error = 1.110e-14
Toom-4       (k=2) | time = 0.000220 s | max error = 2.576e-14
Toom-4       (k=3) | time = 0.000289 s | max error = 2.576e-14
Toom-4       (k=4) | time = 0.000239 s | max error = 2.576e-14
```

*In all cases, MPFR results **match the expected mathematical result** and the execution time is **much slower** than double-precision methods.*

Jenkins pipeline build/run report:



*For example here is an example of the same C-Polynomial job from **Jenkins** which took 1min and 21 second to run and we ran it multiple times to compare the results. This was from the jenkinsfile you found in the repo.*

Accuracy Comparison

- For small degrees, double-precision algorithms produce results **close to MPFR**.
- As degree increases:
  - Small coefficient discrepancies appear in Karatsuba and Toom-based methods.
  - MPFR remains numerically stable and exact.
- MPFR output is therefore used as the **reference solution**.

Discussion

While MPFR is not suitable for high-performance polynomial multiplication due to its computational cost, it is **essential for correctness validation**. The comparison highlights the trade-off between:

- **Speed** (double precision)
- **Accuracy** (multi-precision)

Advanced algorithms improve speed but may slightly degrade numerical accuracy as polynomial size grows.

Conclusion

The MPFR-based naive multiplication:

- Provides **highly accurate reference results**
- Enables **quantitative comparison of numerical errors**
- Confirms correctness of optimized algorithms
- Demonstrates the performance cost of multi-precision arithmetic

In practice, MPFR is best used for **validation and benchmarking**, while double-precision algorithms are preferred for **large-scale computations**.

Acknowledgement

The results have been studied through the programme and hence here are the results, and some data sources are gathered from these Wikipedia pages**:**

[Go through the README of the project]

- https://en.wikipedia.org/wiki/Polynomial_multiplication
- https://en.wikipedia.org/wiki/IEEE_754
- https://www.mpfr.org/