POLYMULT 1.3
Fast Polynomial Multiplication

Anton S. Mosunov

October 29, 2016

Contents

1 Introduction

The POLYMULT library provides subroutines for the out-of-core Fast-Fourier-Transform (FFT) based polynomial multiplication using `OpenMP` for parallelization. The "out-of-core" here means that the resulting files and the intermediate computations get stored into the hard disk. The library is designed for multicore and multiprocessor environments, but can be used on regular computers as well.

The library is designed specifically for very large polynomials with non-negative integer coefficients, possibly exceeding 2^{37} in their degree. Further in this manual, we use the words "series" and "polynomial" interchangeably, as all the polynomials that are available in POLYMULT get initialized from specific infinite series.

The library also allows to invert and divide large polynomials to a certain degree. The inversion algorithm works correctly only with polynomials whose degree is of the form $2^n - 1$, $n \in \mathbb{N}$, and constant coefficient is 1. It is based on the Newton's algorithm, which computes the inverse of a non-zero polynomial $f(x)$ with $\deg f = 2^n - 1$ iteratively to degrees $3, 7, \dots, 2^{n-1} - 1, 2^n - 1$ [GG03, Algorithm 9.3]. Since the coefficients of an inverse polynomial might grow very fast, the inversion of each polynomial is performed modulo some pre-determined prime.

The division of a polynomial $f(x)$ by a non-zero polynomial $g(x)$ is performed by computing $g^{-1}(x)$ first, and then multiplying $f(x)$ by $g^{-1}(x)$.

The out-of-core FFT-based polynomial multiplication technique is due to Hart, Tornaría and Watkins [HTW10]. The detailed description of this technique can also be found in [Mos14, Section 4]. In simple terms, the problem of multiplication of two large polynomials gets reduced to several multiplications of smaller polynomials over finite fields. The multiplication of smaller polynomials (in our experiments, of degree less than 2^{25}) is performed using the subroutines implemented in the `FLINT` library.

POLYMULT is maintained by Anton S. Mosunov, University of Waterloo, and is an appendix to the Master's thesis [Mos14], written under the supervision of Michael J. Jacobson, Jr. It is highly recommended to read the thesis (Section 4 in particular) before utilizing the library, as certain notions, such as the bundling parameter or the bit size parameter, are not defined in this manual.

The manual for POLYMULT is based on the LaTeX template for the manual of FLINT 1.0: http://web.mit.edu/sage/export/tmp/flint-1.1/doc/flint-roadmap.tex. The souce . The contents of source files as well as header files are organized FLINT 2.5.2.

2 Changes since the previous version

Since the version 1.2, the following changes had been made:

- The `configure` file is now a part of POLYMULT. In contrast, the previous version contained only the Makefile and required the user to edit it manually;
- All the files are now supplied with headers, providing information on the GNU General Public License.

3 Dependencies in POLYMULT

POLYMULT depends on several libraries and specifications that need to be present on your system prior to the installation. These libraries are:

1. `FLINT`, flintlib.org. Fast library for number theory;

2. `GMP`, gmplib.org. The GNU multiple precision arithmetic library;

3. `OpenMP`, openmp.org. The OpenMP API specification for parallel programming. Since version 4.2, every GCC compiler contains the implementation of the OpenMP specification.

Before installing POLYMULT, make sure that each of those libraries is installed.

4 Building and using POLYMULT

The easiest way to use POLYMULT is to build each module separately using `make`. The `make` command creates an executable, while the `make lib` command creates a static library.

5 Reporting issues

The maintainers wish to be made aware of any bugs in the library or typos in this manual. Please send an email with your bug report to `amosunov@uwaterloo.ca`.

If possible please include details of your system, version of gcc, version of GMP and precise details of how to replicate the bug.

Note that POLYMULT needs to be linked against version 4.2.1 or later of GMP and must be compiled with gcc version 4.2 or later.

6 Files

The POLYMULT library consists of two parts:

1. The subroutines implemented in `init.c` allow to initialize specific polynomials. See Subsection 7 for the series currently available, and Subsection 10 for the generic description of the contents of this file;

2. The `mult.c` file contains the implementation of the out-of-core FFT-based multiplication technique of Hart, Tornaría and Watkins, with OpenMP used for parallelization. It also contains the `invert` routine for the out-of-core polynomial inversion, based on the Newton's algorithm. See Subsection 11 for more details;

3. The `main.c` contains the implementation of the command line program `polymult`, through which the multiplication is performed. See Subsection 13 for the instructions on how to run the program.

7 Macros and supported series

The POLYMULT library contains several macro definitions. The most important macros are the names of various series which POLYMULT can initialize. They are defined in `init.h` along with the declaration of the initialization routines. As of version 1.1, the supported series and their macros are:

0. `THETA3` — corresponds to the 3rd Jacobi theta series:

$$\theta_3(q) = 1 + 2\sum_{n=1}^{\infty} q^{n^2} = 1 + 2q + 2q^4 + 2q^9 + \dots$$

   This series is used for the tabulation of all class numbers $h(\Delta)$ of imaginary quadratic fields with discriminant $\Delta \not\equiv 1 \pmod 8$; it is also used for the tabulation of class numbers with $\Delta \equiv 8, 12 \pmod{16}$.

1. `THETA3_SQUARED` — corresponds to $[\theta_3(q)]^2$, which captures the number of representations of each number as a sum of 2 perfect squares:

$$[\theta_3(q)]^2 = \sum_{x,y\in\mathbb{Z}} q^{x^2+y^2} = 1 + 4q + 4q^2 + 4q^4 + 8q^5 + 8q^8 + 4q^9 + 8q^{10} + \dots$$

2. `NABLA` — corresponds to $\nabla(q)$, the series indicating every triangular number:

$$\nabla(q) = \sum_{n=0}^{\infty} q^{\frac{n(n+1)}{2}} = 1 + q + q^3 + q^6 + q^{10} + \dots$$

   This series is used for the tabulation of all class numbers $h(\Delta)$ of imaginary quadratic fields with discriminant $\Delta \equiv 5 \pmod 8$.

3. `NABLA_SQUARED` — corresponds to $[\nabla(q)]^2$, which captures the number of representations of each number as a sum of 2 triangular numbers:

$$[\nabla(q)]^2 = 1 + 2q + q^2 + 2q^3 + 2q^4 + 3q^6 + 2q^7 + 2q^9 + 2q^{10} + \dots$$

4. `DOUBLE_NABLA` — corresponds to $\nabla(q^2)$:

$$\nabla(q^2) = 1 + q^2 + q^6 + \dots$$

   This series is used for the tabulation of all class numbers $h(\Delta)$ of imaginary quadratic fields with discriminant $\Delta \equiv 8, 12 \pmod{16}$.

5. `DOUBLE_NABLA_SQUARED` — corresponds to $\left[\nabla(q^2)\right]^2$:

$$\left[\nabla(q^2)\right]^2 = 1 + 2q^2 + q^4 + 2q^6 + 2q^8 + \dots$$

6. `THETA234` — corresponds to the product $\theta_2(q) \cdot \theta_3(q) \cdot \theta_4(q)$, where θ_2 and θ_4 denote the 2nd and the 4th Jacobi theta series, respectively:

$$\theta_2(q)\theta_3(q)\theta_4(q) = \sum_{n=0}^{\infty} (-1)^n (2n+1) q^{\frac{n(n+1)}{2}} = 1 - 3q + 5q^3 - 7q^6 + 9q^{10} - \dots$$

   This series is used for the tabulation of all class numbers $h(\Delta)$ of imaginary quadratic fields with discriminant $\Delta \equiv 1 \pmod 8$. In order to perform the tabulation, this series has to be inverted modulo some fixed prime p first. See the Humbert's formula in [Mos14, Section 7.1];

7. `ALPHA` — corresponds to the alpha series $\alpha(q)$, defined as follows:

$$\alpha(q) = \sum_{n=1}^{\infty} (-1)^{n+1} n^2 \frac{q^{\frac{n(n+1)}{2}} - 1}{1 + q^n}.$$

   This series is used for the tabulation of all class numbers $h(\Delta)$ of imaginary quadratic fields with discriminant $\Delta \equiv 1 \pmod 8$. It is computed modulo some fixed prime p.

Another important macro is `MINPOW`, which is a positive integer. It is defined in `mult.c`, and represents the smallest power up to which the `FLINT` multiplication/inversion routines are used instead of the out-of-core FFT-based approach. For example, the default value of `MINPOW` is 25, which means that each product of polynomials not exceeding 2^{25} in their degrees will get computed using `FLINT`. In Newton's iterative algorithm for the polynomial inversion, the inversion up to $2^{\texttt{MINPOW}} - 1$ is performed using `FLINT`, while inversions up to $2^{\texttt{MINPOW}+1} - 1, 2^{\texttt{MINPOW}+2} - 1, \ldots$ get computed out-of-core. The value of `MINPOW` solely depends on the amount of RAM available on your computer. The larger `MINPOW`, the better. To estimate the largest value for your machine, try running `nmod_poly_inv` or `nmod_poly_mul` routines of `FLINT` on polynomials of degrees $2^{20}, 2^{21}, \ldots, 2^k$, where k is the power when one of those two routines crush. Set `MINPOW` $= k - 1$.

During the compilation, the following two macros may be defined through the `-D` command:

- `KEEP_FILES`. In order to save some space on the hard disk, the out-of-core polynomial multiplication routines delete intermediate files automatically. In order to keep any intermediate computations saved to hard disk, please define the `KEEP_FILES` macro. This may be useful for debugging purposes, or when the crushes on a server occur and you don't want your computations to get corrupted or lost. In this case, make sure that the prefixes of various collections of files differ from each other so that they won't get overwritten;

- `DEBUG`. Many subroutines of POLYMULT have several sub steps. Such sub steps are: initializing from or saving to files, bundling polynomials or reducing polynomials modulo a prime, restoring the coefficients using the Chinese Remainder Theorem, etc. By default, the timing and the status of those sub steps do not get printed to the standard output. If you would like to see this information, for example in order to observe at which point of the execution your program crushes, please specify the `DEBUG` macro.

8 Setup

In order to prepare the POLYMULT library for compilation, please edit the `Makefile`. In particular, specify your compiler which supports `OpenMP` in `CC`, and your compilation macros (see Subsection 7) in `SYMBS`, with each macro preceded by `-D`. In `INCS`, it is especially important to specify the path to `omp.h` file (on Macintosh its location is not obvious). If needed, change the standard paths to header files and library files in `INCS` and `LIBS`, respectively. It is not recommended to edit the parameters specified in `CFLAGS`.

The `Makefile` defines the following three commands:

- The `make` command builds an executable `polymult` which allows to multiply two polynomials. See Subsection 13 on how to use it;

- The `make lib` command builds a static library `libpolymult.a`, which incorporates two object files `init.o` and `mult.o`, produced from `init.c` and `mult.c`, respectively. See Subsection 15 on how to link this library to your program;

- The `make clean` command removes all the files which have the extension `.o`. Use this command right after running `make lib` to remove all the object files.

9 File types

The out-of-core routines produce many binary files on your hard disk. They may contain coefficients of a single polynomial either over the ring of integers, or reduced modulo several primes. The files are saved in the form `prefix0`, `prefix1`, ..., `prefixM`, where `prefix` is the name of a particular collection of files and `M` is the last index. Note that in order for the program to work correctly, the total number of files $\mathtt{M}+1$ must evenly divide the total number of coefficients stored in those files.

Let $n \in \mathbb{N}$, and consider a polynomial $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$ with $a_0, a_1, \ldots, a_{n-1} \in \mathbb{N} \cup \{0\}$. Let $p_1, p_2, \ldots, p_k$ be k distinct primes, and let m be a total number of files which evenly divides n. There are three types of collections of files that get produced:

- **Type 1**. These collections of files contain coefficients of $f(x)$ that are evenly distributed between each of m files:

`prefix0`:	a_0,	a_1,	$\ldots$	$a_{n/m-1}$
`prefix1`:	$a_{n/m}$,	$a_{n/m+1}$,	$\ldots$	$a_{2n/m-1}$
$\ldots$				
`prefix`$m-1$:	$a_{(m-1)n/m}$,	$a_{(m-1)n/m+1}$,	$\ldots$	a_{n-1}

  After the out-of-core multiplication and the restoration of coefficients, these collections of files contain the final result of the multiplication. It is also possible to convert the **Type 1** files into **Type 2** files in order to prepare the resulting polynomial for yet another out-of-core multiplication.

- **Type 2**. These collections of files contain coefficients of $F(x) = A_0 + A_1x + \ldots + A_{n/B-1}x^{n/B-1}$, the *bundled* polynomial produced from $f(x)$ with some bundling parameter B which evenly divides n/m, and reduced modulo primes $p_1, p_2, \ldots, p_k$:

`prefix0`:	$A_0 \pmod{p_1}$,	$A_1 \pmod{p_1}$,	$\ldots$	$A_{n/(mB)-1} \pmod{p_1}$,
	$A_0 \pmod{p_2}$,	$A_1 \pmod{p_2}$,	$\ldots$	$A_{n/(mB)-1} \pmod{p_2}$,
			$\ldots$	
	$A_0 \pmod{p_k}$,	$A_1 \pmod{p_k}$,	$\ldots$	$A_{n/(mB)-1} \pmod{p_k}$
`prefix1`:	$A_{n/(mB)} \pmod{p_1}$,	$A_{n/(mB)+1} \pmod{p_1}$,	$\ldots$	$A_{2n/(mB)-1} \pmod{p_1}$,
	$A_{n/(mB)} \pmod{p_2}$,	$A_{n/(mB)+1} \pmod{p_2}$,	$\ldots$	$A_{2n/(mB)-1} \pmod{p_2}$,
			$\ldots$	
	$A_{n/(mB)} \pmod{p_k}$,	$A_{n/(mB)+1} \pmod{p_k}$,	$\ldots$	$A_{2n/(mB)-1} \pmod{p_k}$
$\ldots$				
`prefix`$m-1$:	$A_{(m-1)n/(mB)} \pmod{p_1}$,	$A_{(m-1)n/(mB)+1} \pmod{p_1}$,	$\ldots$	$A_{n/B-1} \pmod{p_1}$,
	$A_{(m-1)n/(mB)} \pmod{p_1}$,	$A_{(m-1)n/(mB)+1} \pmod{p_2}$,	$\ldots$	$A_{n/B-1} \pmod{p_2}$,
			$\ldots$	
	$A_{(m-1)n/(mB)} \pmod{p_k}$,	$A_{(m-1)n/(mB)+1} \pmod{p_k}$,	$\ldots$	$A_{n/B-1} \pmod{p_k}$

  Before an out-of-core multiplication is performed, both polynomials get converted into collections of this type.

- **Type 3**. These collections of files contain coefficients of a polynomial that is a result of the out-of-core multiplication of two bundled polynomials modulo $p_1, p_2, \ldots, p_k$. Though their structure is essentially the same as collections of **Type 2**, they do *not* contain coefficients of a bundled polynomial. Nevertheless, it is possible to restore the coefficients of a resulting polynomial distributed over files of **Type 1** from files of **Type 3**. This is done via the restoration algorithm described in Subsection 11.

10 Initialization

All initialization subroutines are defined in `init.c`. There are three kinds of initialization functions:

- The subroutines of the form

  ```
  void init_block_name(int * block, const ulong size, const ulong min)
  void init_block_name(int * block, const ulong size, const ulong min,
                        const int mod)
  ```

  initialize a block of coefficients of a series called `name`, starting from a coefficient `min` and up to a coefficient `min + size`. The presence of a `mod` parameter indicates that the coefficients get initialized modulo a prime `mod`. This kind of subroutines correspond to those series which admit negative coefficients or the coefficients that do not fit into the `int` type. The names of the series currently supported by POLYMULT can be found in Subsection 7, and must be written *in lower case*, in contrast to an upper case in which their macros are defined.

- The subroutines of the form

  ```
  void nmod_poly_name(nmod_poly_t poly, const ulong size)
  ```

  initialize first `size` coefficients of a polynomial called `name` of type `nmod_poly_t` (defined in `FLINT`). They initialize a polynomial *as a whole*, rather than just a block of its coefficients. These subroutines utilized solely for the polynomial inversion algorithm.

- The subroutine

  ```
  void nmod_poly_to_files(const nmod_poly_t poly, const ulong size,
                           const uint files, const char * resultname)
  ```

  copies the coefficients of a polynomial `poly` with `size` coefficients into a **Type 1** collection of `files` binary files with a prefix `resultname`. Note that `size` must be evenly divisible by `files` in order for this subroutine to work correctly.

11 Out-of-core operations on polynomials

The following subroutines are defined in `mult.c`:

```
void init_primes(ulong * primes, const uint total_primes,
                 const ulong lowerbound)
```

Initializes a set of `total_primes` distinct primes, which immediately follow the `lowerbound`. This subroutine utilizes the `n_nextprime` function of `FLINT`, and is used in `multiply` and `divide` subroutines described below.

```
void init_files(const ulong * primes, const uint total_primes,
                const ulong limit, const uint files, const uint bundle,
                const uint bitsize, const int mod, const char * resultname,
                const void * type, const char flags)
```

This subroutine "prepares" a polynomial for the out-of-core multiplication by initializing a **Type 2** collection of `files` binary files with a prefix `resultname`. The files contain coefficients of a *bundled* polynomial, produced from the first `limit` coefficients of a series called `type`, and reduced modulo `total_primes` primes specified in `primes`. If the coefficients of a series admit negative numbers and hence need to be initialized modulo a prime, this prime can be specified by a `mod` parameter (set `mod` = 0 otherwise).

Note that the `type` parameter, instead of being a macro identifying a specific series, may be a prefix of a **Type 1** collection of `files` binary files (that is, a string of type `char *`). These files get deleted after the initialization of a polynomial, unless the `flags` parameter is set to `NO_REMOVE` macro. This macro allows to preserve the files with a prefix `type` *even if* the `KEEP_FILES` macro is undefined. The `NO_REMOVE` macro is utilized solely by the `invert` routine described below, and is not recommended for utilization.

The bundling parameter is `bundle`, and the bitsize parameter is `bitsize` (see [Mos14, Section 4.1]).

```
void ooc_multiply(const ulong * primes, const uint total_primes,
                  const ulong limit, const uint files, const uint bundle,
                  const char * resultname, const char * name1, const char * name2)
```

Performs an out-of-core multiplication of two *bundled* polynomials reduced modulo `total_primes` primes, specified in `primes`. Both polynomials are specified via **Type 2** collection of `files` binary files, and their prefixes are `name1` and `name2`. The result is saved into **Type 3** collection of `files` binary files with a prefix `resultname`. Both (non-bundled) polynomials have a degree `limit` − 1, and the `limit` parameter has to be evenly divisible by `files`. The bundling parameter is `bundle`. The resulting polynomial contains the information only on the first `limit` coefficients.

```
void square(const ulong * primes, const uint total_primes,
            const ulong limit, const uint files, const uint bundle,
            const char * resultname, const char * name1)
```

Performs an out-of-core squaring of a *bundled* polynomial reduced modulo `total_primes` primes, specified in `primes`. A polynomial with `limit` coefficients is specified via a **Type 2** collection of `files` binary files, and its prefix is `name1`. The `limit` parameter has to be evenly divisible by `files`. The result is saved into a **Type 3** collection of `files` binary files with a prefix `resultname`. The bundling parameter is `bundle`. Note that the squaring gets performed *completely*; that is, no truncation occurs, and the resulting files actually contain information on 2`limit` − 1 coefficients.

```
void restore_coeff(const ulong * primes, const uint total_primes,
                   const ulong limit, const uint files, const uint bundle,
                   const uint bitsize, const int mod, const char * resultname,
                   const char * name1, const int flags)
```

Restores `limit` coefficients from a **Type 3** collection of `files` binary files with a prefix `name1`, which contain the result of multiplication of two bundled polynomials reduced modulo `total_primes` primes, specified in `primes`. The result is saved into a **Type 1** collection of `files` binary files with a prefix `resultname`. The `limit` parameter has to be evenly divisible by `files`. The bundling parameter is `bundle` and the bitsize parameter is `bitsize`. If the restored coefficients need to be reduced modulo a prime, this prime can be specified in `mod` (set `mod` = 0 otherwise). The `flags` parameter admits two flags, namely `WITH_INVERSE` and `WITH_SQUARING`. Both of those flags are used by the `invert` routine to speedup the inversion, and not recommended for utilization.

```
void invert(const ulong * primes, const uint maxpow,
            const uint files, const uint bundle, const int mod,
            const char * resultname, const char * folder, const char type)
```

Inverts the polynomial of type `type` to degree $2^{\texttt{maxpow}}$ via Newton's algorithm using out-of-core subroutines described above. The result is saved into **Type 1** collection of `files` binary files with a prefix `resultname` in a folder `folder`. Note that `files` must evenly divide $2^{\texttt{maxpow}}$. If the inversion need to be performed modulo a prime, this prime can be specified in `mod` (set `mod` = 0 otherwise). The bundling parameter is `bundle`. The primes used for the out-of-core multiplication are specified in `primes`. Make sure that enough primes get provided, as the total number of primes used for the computation gets recomputed on every iteration.

```
void multiply(const ulong limit, const uint files, const uint bundle,
              const uint bound, const char * resultname, const char * folder,
              const char type1, const char type2)
```

Performs the out-of-core multiplication of two polynomials of types `type1` and `type2`. The result is saved into **Type 1** collection of `files` binary files with a prefix `resultname` in a folder `folder`. The number of coefficients of both polynomials is `limit`, and the upper bound on the coefficients of the resulting polynomial is `bound`. The bundling parameter is `bundle`.

```
void divide(const ulong limit, const uint files, const uint bundle,
            const uint bound, const char * resultname, const char * folder,
            const char type1, const char type2)
```

Performs the inversion of a polynomial of type `type2`, followed by an out-of-core multiplication by a polynomial of type `type1`. The result is saved into **Type 1** collection of `files` binary files with a prefix `resultname` in a folder `folder`. The number of coefficients of both polynomials is `limit`, and the upper bound on the coefficients of the resulting polynomial is `bound`. The bundling parameter is `bundle`.

12 Defining your own polynomials

If you would like to multiply polynomials that are not present in POLYMULT, feel free to define them by following the process described below. We let `K` be the total number of series defined (so the last series has index `K` − 1).

1. Let `name` be the name of your series. In the first part of `init.h`, define the macro `NAME` as follows:

   ```
   #define NAME K
   ```

2. In the first part of `init.h`, change the definition of the macro `IS_FILE` as follows:

   ```
   #define IS_FILE(X) (((ulong) X) > NAME)
   ```

3. In the first part of `init.h`, right after the declaration of your macro, define the subroutine `init_block_name` either in the form

   ```
   void init_block_name(int * block, const ulong size, const ulong min)
   ```

   or in the form

   ```
   void init_block_name(int * block, const ulong size, const ulong min,
                        const int mod)
   ```

   The latter case is utilized when your series admits negative values or coefficients that exceed the `int` type in bit size. In order for the program to work correctly, such a series needs to be considered modulo some fixed prime `mod`.

4. (Optional) If the first coefficient of your polynomial is one and you intend to invert it, in the second part of `init.h` define the subroutine

   ```
   nmod_poly_name(nmod_poly_t poly, const ulong size)
   ```

5. Implement the subroutines `init_block_name` (and possibly `nmod_poly_name`) in `init.c`;
6. In the file `mult.c`, include your polynomial macro in the list occurring in `init_files` in one of the following two ways, depending on whether the `mod` parameter is required by the `init_block_name` routine:

   ```
   else if (type == (void *) NAME)
   {
   init_block_name(INIT_PARAMS);
   }
   ```

   ```
   else if (type == (void *) NAME)
   {
   init_block_name(INIT_PARAMS, mod);
   }
   ```

7. (Optional) If the polynomial admits the inversion, include your polynomial macro in the list occurring in `invert`:

   ```
   else if (type == NAME)
   {
           nmod_poly_name(poly, limit);
   }
   ```

8. In `main.c`, add the description of your polynomial in the list of `printf` statements:

   ```
   printf(K: name\n);
   ```

9. Update the information in the documentation.

Alternatively, please send a request to `amosunov@uwaterloo.ca` to implement the series you have in mind, and it will appear in the next release. Make sure to include the description of your series (the most preferred description is the actual implementation), and before that verify that its block of coefficients can be initialized in a reasonable time.

13 Utilizing the executable

The `polymult` executable admits 9 parameters. If the total number of parameters differs from 9, the following helping prompt comes out:

```
Format: ./polymult [multiply/divide] [poly1] [poly2] [limit] [files] [bundle]
                   [bound] [resultname] [folder]
Types of poly1/poly2:
0: theta3             = 1 + 2q + 2q^4 + 2q^9 + ...
1: theta3 squared
2: nabla              = 1 + q + q^3 + q^6 + q^10 + ...
3: nabla squared
4: double nabla       = 1 + q^2 + q^6 + q^12 + ...
5: double nabla squared
6: theta2*theta3*theta4 = 1 - 3q + 5q^3 - 7q^6 + 9q^10 - ...
7: alpha series
```

The parameters provided are:

- `[multiply/divide]`: write either `multiply` or `divide` as your first parameter to specify which action would you like to perform;
- `[poly1]`: type of the first polynomial (an integer from 0 to 7, see the prompt above);
- `[poly2]`: type of the second polynomial (an integer from 0 to 7, see the prompt above);
- `[limit]`: degree to which polynomials get multiplied or divided;
- `[files]`: number of files in a single collection to which the coefficients of a resulting polynomial, as well as intermediate computations, will be saved;
- `[bundle]`: bundling parameter;
- `[bound]`: upper bound on the coefficients of a resulting polynomial;
- `[resultname]`: prefix of files where the result get saved;
- `[folder]`: folder.

See Subsection 14 for some examples on how to run the program.

14 Examples

Example 1. The following command tabulates all Hurwitz class numbers $H(\Delta)$ of imaginary quadratic fields to $|\Delta| < 2^{40}$, where $|\Delta| \equiv 8 \pmod{16}$. In particular, it multiplies $\theta_3(q)$ by $\nabla^2(q^2)$ to degree $2^{36} = 68719476736$ out-of-core, with the bundling parameter $2^{11} = 2048$ and the upper bound on the coefficients of $\theta_3(q)\nabla^2(q^2)$ given by 2316050. The coefficients of $\theta_3(q)\nabla^2(q^2)$ get saved into $2^{12} = 4096$ files: `/home/h8mod16.0`, `/home/h8mod16.1`, ..., `/home/h8mod16.4095`.

```
./polymult multiply 0 5 68719476736 4096 2048 2316050 h8mod16. /home
```

Example 2. The following command tabulates all Hurwitz class numbers $H(\Delta)$ (multiplied by 2) of imaginary quadratic fields to $|\Delta| < 2^{40}$, where $|\Delta| \equiv 4 \pmod{16}$. In particular, it multiplies $\theta_3^2(q)$ by $\nabla(q^2)$ to degree $2^{36} = 68719476736$ out-of-core, with the bundling parameter $2^{11} = 2048$ and the upper bound on the coefficients of $\theta_3^2(q)\nabla(q^2)$ given by 10189617. The coefficients of $\theta_3^2(q)\nabla(q^2)$ get saved into $2^{12} = 4096$ files: `/home/h4mod16.0`, `/home/h4mod16.1`, ..., `/home/h4mod16.4095`.

```
./polymult multiply 1 4 68719476736 4096 2048 10189617 h4mod16. /home
```

Example 3. The following command tabulates all Hurwitz class numbers $H(\Delta)$ (multiplied by 3) of imaginary quadratic fields to $|\Delta| < 2^{40}$, where $|\Delta| \equiv 3 \pmod{8}$. In particular, it multiplies $\nabla(q)$ by $\nabla^2(q)$ to degree $2^{37} = 137438953472$ out-of-core, with the bundling parameter $2^{12} = 4096$ and the upper bound on the coefficients of $\nabla^3(q)$ given by 29180730. The coefficients of $\nabla^3(q)$ get saved into $2^{12} = 4096$ files: `/home/h3mod8.0`, `/home/h3mod8.1`, ..., `/home/h3mod8.4095`.

```
./polymult multiply 2 3 137438953472 4096 4096 29180730 h3mod8. /home
```

15 Utilizing the library

In order to utilize the library, place the files `init.h` and `mult.h` into your include path, and copy the library `libpolymult.a` into your library path. By writing

```
#include <mult.h>
```

among other inclusions in your file you will gain access to all the subroutines defined in `mult.h`. Same applies to `init.h`.

When compiling, link the library to your program by writing `-lpolymult`. Don't forget to link all the other libraries that POLYMULT depends on (see Subsection 3 for the complete list).

References

[GG03] J. von zur Gathen, J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, 2nd edition, 2003.

[HTW10] W. B. Hart, G. Tornaría, M. Watkins, *Congruent number theta coefficients to* 10^{12}, Algorithmic Number Theory — ANTS-IX (Nancy, France), Lecture Notes in Computer Science 6197, Springer-Verlag, Berlin, pp. 186 – 200, 2010.

[Mos14] A. S. Mosunov, *Unconditional Class Group Tabulation to* 2^{40}, Master's thesis, University of Calgary, Calgary, Alberta, 2014.