Recherche de facteurs premiers

Code du TIPE

Tristan Delcourt (15699)

2025

Sommaire

1 Point d'entrée du programme

1.1 main.c

```c
#include <stdbool.h>
#include <gmp.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "system.h"
#include "vector.h"
#include "parse_input.h"
#include "factorbase.h"
#include "list_matrix_utils.h"

// Include algorithms
// Dixon's method
#include "./dixon/dixon.h"

// The Quadratic Sieve
#include "./qsieve/qsieve.h"

// Multipolynomial Quadratic Sieve
#include "./mpqs/polynomial.h"
#include "./mpqs/mpqs.h"
#include "./mpqs/parallel_mpqs.h"


/**
 *
 * START OF ALGORITHM
 *
 */


void rebuild_mpqs(mpz_t prod, mpz_t* d, int* v, int* primes, int
   n1, system_t s){
    mpz_set_ui(prod, 1);
    mpz_t temp;
    mpz_init(temp);
    for(int i = 0; i<n1; i++){
        if(s->sol[i]){
            mpz_mul(prod, prod, d[s->perm[i]]);
        }
        mpz_ui_pow_ui(temp, primes[i], v[i]);
        mpz_mul(prod, prod, temp);
    }
    mpz_clear(temp);
}

void rebuild(mpz_t prod, int* v, int* primes, int n1){
```

```
49      /** Rebuilds the product of primes to the power of half
50       * the solution found by the gaussian solve
51
52       * EX:
53       * v = (1, 2, 3, 1)
54       * primes = [2, 3, 5, 7]
55       * prod = 2**1 * 3** 2 * 5**3 * 7**1
56       * returns prod
57       *
58      */
59
60      mpz_set_ui(prod, 1);
61      mpz_t temp;
62      mpz_init(temp);
63      for(int i = 0; i<n1; i++){
64          mpz_ui_pow_ui(temp, primes[i], v[i]);
65          mpz_mul(prod, prod, temp);
66      }
67      mpz_clear(temp);
68  }
69
70  void sum_lignes(int* sum, int** v, system_t s){
71      /** Sums the lines of vectors into 'sum' according the
           solution of the
72       * output of the system 's', such that each power is even
73       */
74      for(int i = 0; i<s->n1; i++){
75          sum[i] = 0;
76      }
77
78      for(int i = 0; i<s->n2; i++){
79          if(s->sol[i]){
80              add_vect(sum, v[s->perm[i]], s->n1);
81          }
82      }
83  }
84
85  void factor(input_t* input){
86      int piB = pi(input->bound);
87      if(!input->quiet) printf("pi(B) = %d\n", piB);
88      int* p = primes(piB, input->bound);
89
90      int pb_len;
91      int* pb;
92      switch(input->algorithm){
93          case DIXON:
94              pb = p;
95              pb_len = piB;
96              break;
97          case QSIEVE:
98              pb = prime_base(input->N, &pb_len, p, piB);
99              if(!input->quiet) printf("base reduction %f%%\n", (
```

```
                float)pb_len/piB*100);
100             free(p);
101             break;
102         case MPQS:
103             pb = prime_base(input->N, &pb_len, p, piB);
104             pb[pb_len] = -1;
105             if(!input->quiet) printf("base reduction %f%%\n", (
                    float)pb_len/piB*100);
106             free(p);
107             break;
108         case PMPQS:
109             pb = prime_base(input->N, &pb_len, p, piB);
110             pb[pb_len] = -1;
111             if(!input->quiet) printf("base reduction %f%%\n", (
                    float)pb_len/piB*100);
112             free(p);
113             break;
114     }
115     int target_nb = pb_len + input->extra;
116
117     mpz_t* z = malloc((target_nb)*sizeof(mpz_t));
118     for(int i = 0; i < target_nb; i++){
119         mpz_init(z[i]);
120     }
121
122     //Getting zis
123     int** v;
124     mpz_t* d;
125     struct timeval t1, t2;
126     gettimeofday(&t1, 0);
127     switch(input->algorithm){
128         case DIXON:
129             v = dixon(z, input->N, pb_len, pb, input->extra,
                        input->quiet);
130             break;
131         case QSIEVE:
132             v = qsieve(z, input->N, pb_len, pb, input->extra,
                        input->sieving_interval, input->quiet);
133             break;
134         case MPQS:
135             d = malloc(target_nb*sizeof(mpz_t));
136             for(int i = 0; i < target_nb; i++){
137                 mpz_init(d[i]);
138             }
139             v = mpqs(z, d, input->N, pb_len, pb, input->extra,
                        input->sieving_interval, input->delta, input->
                        quiet);
140             break;
141         case PMPQS:
142             d = malloc(target_nb*sizeof(mpz_t));
143             for(int i = 0; i < target_nb; i++){
144                 mpz_init(d[i]);
```

```
145                 }
146                 v = parallel_mpqs(z, d, input->N, pb_len, pb, input->
                        extra, input->sieving_interval, input->delta,
                        input->quiet);
147                 break;
148         }
149 
150         gettimeofday(&t2, 0);
151         long seconds = t2.tv_sec - t1.tv_sec;
152         long microseconds = t2.tv_usec - t1.tv_usec;
153         double time_spent = seconds + microseconds*1e-6;
154         if(!input->quiet) printf("Time to get zi: %fs\n", time_spent)
               ;
155 
156         mpz_t f, Z1, Z2, test1, test2;
157         mpz_inits(f, Z1, Z2, test1, test2, NULL);
158 
159         //gaussian init
160         system_t s;
161         int* sum;
162         switch(input->algorithm){
163             case DIXON:
164                 s = init_gauss(v, target_nb, pb_len);
165                 sum = malloc(pb_len*sizeof(int));
166                 break;
167             case QSIEVE:
168                 s = init_gauss(v, target_nb, pb_len);
169                 sum = malloc(pb_len*sizeof(int));
170                 break;
171             case MPQS:
172                 // for -1
173                 s = init_gauss(v, target_nb, pb_len+1);
174                 sum = malloc((pb_len+1)*sizeof(int));
175                 break;
176             case PMPQS:
177                 // for -1
178                 s = init_gauss(v, target_nb, pb_len+1);
179                 sum = malloc((pb_len+1)*sizeof(int));
180                 break;
181         }
182         if(!input->quiet) printf("2^%d solutions to iterate\n", s->n2
                - s->arb);
183 
184         bool done = false;
185         while(!done){
186             gaussian_step(s);
187 
188             prod_vect(Z1, z, target_nb, s);
189             sum_lignes(sum, v, s);
190             div_vect(sum, 2, pb_len);
191 
192             switch(input->algorithm){
```

```
193                 case DIXON:
194                     rebuild(Z2, sum, pb, pb_len);
195                     break;
196                 case QSIEVE:
197                     rebuild(Z2, sum, pb, pb_len);
198                     break;
199                 case MPQS:
200                     rebuild_mpqs(Z2, d, sum, pb, pb_len, s);
201                     break;
202                 case PMPQS:
203                     rebuild_mpqs(Z2, d, sum, pb, pb_len, s);
204                     break;
205             }
206 
207             // TEST
208             mpz_set(test1, Z1);
209             mpz_mul(test1, test1, test1);
210             mpz_set(test2, Z2);
211             mpz_mul(test2, test2, test2);
212             assert(mpz_congruent_p(test1, test2, input->N) != 0);
213             // END TEST
214 
215             mpz_sub(f, Z1, Z2);
216             mpz_gcd(f, f, input->N);
217 
218             if(mpz_cmp_ui(f, 1) != 0 && mpz_cmp(f, input->N) != 0){
219                 assert(mpz_divisible_p(input->N, f));
220                 if(!input->quiet) gmp_printf("%Zd = 0 [%Zd]\n", input
                        ->N, f);
221                 done = true;
222             }
223 
224             mpz_add(f, Z1, Z2);
225             mpz_gcd(f, f, input->N);
226 
227             if(mpz_cmp_ui(f, 1) != 0 && mpz_cmp(f, input->N) != 0){
228                 assert(mpz_divisible_p(input->N, f));
229                 if(!input->quiet) gmp_printf("%Zd = 0 [%Zd]\n", input
                        ->N, f);
230                 done = true;
231             }
232 
233             if(s->done){
234                 if(!input->quiet) fprintf(stderr, "ERROR: no solution
                        for this set of zi\n");
235                 exit(1);
236             }
237         }
238 
239         free(sum);
240         free(pb);
241         free_system(s);
```

```
242         free_ll(v, target_nb);
243         for(int i = 0; i < target_nb; i++){
244             mpz_clear(z[i]);
245         }
246         free(z);
247         switch(input->algorithm){
248             case DIXON:
249                 break;
250             case QSIEVE:
251                 break;
252             case MPQS:
253                 for(int i = 0; i < target_nb; i++) mpz_clear(d[i]);
254                 free(d);
255                 break;
256             case PMPQS:
257                 for(int i = 0; i < target_nb; i++) mpz_clear(d[i]);
258                 free(d);
259                 break;
260         }
261 
262 
263         mpz_clears(f, Z1, Z2, test1, test2, NULL);
264     }
265 
266     int main(int argc, char** argv){
267         input_t* input = parse_input(argc, argv);
268         if(input==NULL){
269             fprintf(stderr, "ERROR: Invalid input\n");
270             return 1;
271         }
272 
273         if(mpz_cmp_ui(input->N, 0) == 0){
274             fprintf(stderr, "ERROR: No input number, use -n %%number
                %%\n");
275             return 1;
276         }
277 
278         if(input->bound == -1) input->bound = 10000;
279         if(input->sieving_interval == -1) input->sieving_interval =
              100000;
280         if(input->extra == -1) input->extra = 1;
281 
282         struct timeval t1, t2;
283         gettimeofday(&t1, 0);
284         factor(input);
285         gettimeofday(&t2, 0);
286         long seconds = t2.tv_sec - t1.tv_sec;
287         long microseconds = t2.tv_usec - t1.tv_usec;
288         double time_spent = seconds + microseconds*1e-6;
289         if(!input->quiet) printf("Total time: %fs\n", time_spent);
290 
291         free_input(input);
```

```
292
293         return 0;
294     }
```

2 Modules utiles

2.1 vector.h

```
#pragma once
#include <gmp.h>

void mod_vect(int* v, int mod, int n1);
void add_vect(int* sum, int* op, int n1);
void div_vect(int* v, int d, int n1);
void sub_vect(int** v, int i, int j, int n1);
void prod_vect(mpz_t prod, mpz_t* z, int n1, system_t s);
```

2.2 vector.c

```c
#include <gmp.h>
#include <assert.h>
#include <stdlib.h>
#include "system.h"

void mod_vect(int* v, int mod, int n1){
    for(int i = 0; i<n1; i++){
        v[i] = abs(v[i]) % mod;
    }
}

void add_vect(int* sum, int* op, int n1){
    for(int i = 0; i<n1; i++){
        sum[i] += op[i];
    }
}


void div_vect(int* v, int d, int n1){
    for(int i = 0; i<n1; i++){
        assert(v[i]%d == 0);
        v[i] /= d;
    }
}

void sub_vect(int** v, int i, int j, int n1){
    for(int k = 0; k<n1; k++){
        v[i][k] = v[i][k] - v[j][k];
    }
}

void prod_vect(mpz_t prod, mpz_t* z, int n1, system_t s){
    mpz_set_ui(prod, 1);
    for(int i = 0; i<n1; i++){
        if(s->sol[i]){
            mpz_mul(prod, prod, z[s->perm[i]]);
        }
    }
}
```

2.3 tonellishanks.h

```
#pragma once

#include <gmp.h>

void tonelli_shanks_ui(mpz_t n, int p, int* x1, int* x2);
void tonelli_shanks_mpz(mpz_t a, mpz_t p, mpz_t x1, mpz_t x2);
```

2.4 tonellishanks.c

```
#include <stdint.h>
#include <gmp.h>
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

uint64_t modpow(uint64_t a, uint64_t b, uint64_t n) {
    uint64_t x = 1, y = a;
    while (b > 0) {
        if (b % 2 == 1) {
            x = (x * y) % n; // multiplying with base
        }
        y = (y * y) % n; // squaring the base
        b /= 2;
    }
    return x % n;
}

void tonelli_shanks_ui(mpz_t n, unsigned long int p, int* x1, int
   * x2) {
    uint64_t q = p - 1;
    uint64_t ss = 0;
    uint64_t z = 2;
    uint64_t c, r, t, m;


    while ((q & 1) == 0) {
        ss += 1;
        q >>= 1;
    }

    mpz_t temp, pj;
    mpz_init(temp);
    mpz_init_set_ui(pj, p);

    if (ss == 1) {
        //uint64_t r1 = modpow(n, (p + 1) / 4, p);
        mpz_powm_ui(temp, n, (p+1)/4, pj);
        uint64_t r1 = mpz_get_ui(temp);

        *x1 = r1;
        *x2 = p - r1;
        mpz_clears(temp, pj, NULL);
        return;
    }

    while (modpow(z, (p - 1) / 2, p) != (unsigned long int) p -
       1) { // uint_64 only there for the compiler to stop
       complaining
        z++;
```

```
48      }
49
50      c = modpow(z, q, p);
51
52      //r = modpow(n, (q + 1) / 2, p);
53      mpz_powm_ui(temp, n, (q+1)/2, pj);
54      r = mpz_get_ui(temp);
55
56      //t = modpow(n, q, p);
57      mpz_powm_ui(temp, n, q, pj);
58      t = mpz_get_ui(temp);
59
60      m = ss;
61
62      while(1){
63          uint64_t i = 0, zz = t;
64          uint64_t b = c, e;
65          if (t == 1) {
66              *x1 = r;
67              *x2 = p - r;
68              mpz_clears(temp, pj, NULL);
69              return;
70          }
71          while (zz != 1 && i < (m - 1)) {
72              zz = zz * zz % p;
73              i++;
74          }
75          e = m - i - 1;
76          while (e > 0) {
77              b = b * b % p;
78              e--;
79          }
80          r = r * b % p;
81          c = b * b % p;
82          t = t * c % p;
83          m = i;
84      }
85  }
86
87  void tonelli_shanks_mpz(mpz_t n, mpz_t p, mpz_t x1, mpz_t x2){
88      assert(mpz_legendre(n, p) == 1);
89
90      mpz_t q, z;
91      mpz_init_set(q, p);
92      mpz_sub_ui(q, q, 1);
93      int ss = 0;
94      mpz_init_set_ui(z, 2);
95
96      while(mpz_divisible_ui_p(q, 2) != 0){
97          ss += 1;
98          mpz_divexact_ui(q, q, 2);
99      }
```

```
100
101         mpz_t op1;
102         mpz_init(op1);
103
104         if (ss == 1) {
105             //uint64_t r1 = modpow(n, (p + 1) / 4, p);
106             mpz_add_ui(op1, p, 1);
107             mpz_divexact_ui(op1, op1, 4);
108             mpz_powm(op1, n, op1, p);
109
110             mpz_set(x1, op1);
111             mpz_sub(x2, p, x1);
112
113             mpz_clears(q, z, op1, NULL);
114             return;
115         }
116
117         mpz_t op2, op3;
118         mpz_inits(op2, op3, NULL);
119
120         mpz_sub_ui(op1, p, 1);
121         mpz_divexact_ui(op1, op1, 2);
122         mpz_powm(op2, z, op1, p);
123
124         mpz_sub_ui(op3, p, 1);
125         while(mpz_cmp(op2, op3) != 0){
126             mpz_add_ui(z, z, 1);
127             mpz_powm(op2, z, op1, p);
128         }
129
130         mpz_t c, r, t, m, i, zz, b, e;
131         mpz_inits(c, r, t, m, i, zz, b, e, NULL);
132         mpz_powm(c, z, q, p);
133
134         mpz_add_ui(op1, q, 1);
135         mpz_divexact_ui(op1, op1, 2);
136         mpz_powm(r, n, op1, p);
137
138         mpz_powm(t, n, q, p);
139
140         mpz_set_ui(m, ss);
141
142         while(1){
143             mpz_set_ui(i, 0);
144             mpz_set(zz, t);
145             mpz_set(b, c);
146
147             if(mpz_cmp_ui(t, 1) == 0){
148                 mpz_set(x1, r);
149                 mpz_sub(x2, p, x1);
150
151                 mpz_clears(c, r, t, m, i, zz, b, e, op1, op2, op3, q,
```

```
                    z, NULL);
152             return;
153         }
154 
155         mpz_sub_ui(op1, m, 1);
156         while(mpz_cmp_ui(zz, 1) != 0 && mpz_cmp(i, op1)<0){
157             mpz_mul(zz, zz, zz);
158             mpz_mod(zz, zz, p);
159             mpz_add_ui(i, i, 1);
160         }
161 
162         mpz_sub(e, m, i);
163         mpz_sub_ui(e, e, 1);
164         while(mpz_sgn(e)>0){
165             mpz_mul(b, b, b);
166             mpz_mod(b, b, p);
167             mpz_sub_ui(e, e, 1);
168         }
169 
170         mpz_mul(r, r, b);
171         mpz_mod(r, r, p);
172 
173         mpz_mul(c, b, b);
174         mpz_mod(c, c, p);
175 
176         mpz_mul(t, t, c);
177         mpz_mod(t, t, p);
178 
179         mpz_set(m, i);
180     }
181 
182 }
```

2.5 system.h

```
#pragma once
#include <stdbool.h>

typedef struct system {
    int** m;
    int* perm;
    int* sol;
    bool done;
    int n1, n2, arb;
} system_s;

typedef system_s* system_t;

system_t init_gauss(int** v, int n1, int n2);
void gaussian_step(system_t s);
void free_system(system_t s);
```

2.6 system.c

```c
#include "system.h"
#include "vector.h"
#include "list_matrix_utils.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

void swap_lines_horz(system_t s, int i, int j){
    int* temp = s->m[i];
    s->m[i] = s->m[j];
    s->m[j] = temp;
}

void swap_lines_vert(system_t s, int i, int j){
    int temp = s->perm[i];
    s->perm[i] = s->perm[j];
    s->perm[j] = temp;

    for(int k = 0; k<s->n1; k++){
        int temp = s->m[k][i];
        s->m[k][i] = s->m[k][j];
        s->m[k][j] = temp;
    }
}

int find_index(system_t s, int from, int look){
    for(int i = from; i < s->n1; i++){
        if(s->m[i][look]){
            return i;
        }
    }
    return -1;
}

system_t transpose(int** v, int n1, int n2){
    system_t s = malloc(sizeof(system_s));

    s->m = malloc(n2*sizeof(int*));
    for(int i = 0; i<n2; i++){
        s->m[i] = malloc(n1*sizeof(int));
        for(int j = 0; j<n1; j++){
            s->m[i][j] = v[j][i];
        }
    }

    s->n1 = n2;
    s->n2 = n1;
    return s;
}

```

```
51  void triangulate(system_t s){
52      s->perm = malloc(s->n2*sizeof(int));
53      for(int i = 0; i<s->n2; i++){
54          s->perm[i] = i;
55      }
56
57      int i = 0;
58      int j = 0;
59      while(i<s->n1 && j<s->n2){
60          int k = find_index(s, i, j);
61          if(k != -1){
62              if(i != j){
63                  swap_lines_vert(s, i, j);
64              }
65
66              swap_lines_horz(s, i, k);
67
68              for(int l = i + 1; l < s->n1; l++){
69                  if(s->m[l][i] == 1){
70                      sub_vect(s->m, l, i, s->n2);
71                      mod_vect(s->m[l], 2, s->n2);
72                  }
73              }
74              i++;
75              j = i;
76          }
77          else{
78              j++;
79          }
80      }
81  }
82
83  void get_arbitary(system_t triangulated){
84      for(int i = triangulated->n1-1; i>=0; i--){
85          int j = 0;
86          while(j < triangulated->n2 && !triangulated->m[i][j]){
87              j++;
88          }
89          if(j<triangulated->n2){
90              triangulated->arb = j+1;
91              return;
92          }
93      }
94
95      fprintf(stderr, "ERROR: All vectors are zero in system\n");
96      exit(1);
97  }
98
99  void init_sol(system_t s){
100     s->sol = malloc(s->n2*sizeof(int));
101     for(int i = s->arb; i<s->n2; i++){
102         s->sol[i] = 0;
```

```
103         }
104     }
105
106     void iter_sol(system_t s){
107         int i = s->arb;
108         while(i<s->n2 && (s->sol[i] == 1)){
109             s->sol[i] = 0;
110             i++;
111         }
112         if(i >= s->n2){
113             s->done = true;
114             return;
115         }
116         s->sol[i] = 1;
117     }
118
119     system_t init_gauss(int** v, int n1, int n2){
120         //printf("Initial vectors\n");
121         //print_ll(v, n1, n2);
122
123         system_t s = transpose(v, n1, n2);
124         s->done = false;
125
126         //printf("Transposed\n");
127         //print_ll(s->m, s->n1, s->n2);
128
129         for(int i = 0; i<s->n1; i++){
130             mod_vect(s->m[i], 2, s->n2);
131         }
132
133         //printf("Modded\n");
134         //print_ll(s->m, s->n1, s->n2);
135
136         triangulate(s);
137
138         //printf("Triangulated\n");
139         //print_ll(s->m, s->n1, s->n2);
140
141         get_arbitary(s);
142         init_sol(s);
143
144         return s;
145     }
146
147     void gaussian_step(system_t s){
148         iter_sol(s);
149
150         for(int i = s->n1-1; i>=0; i--){
151             int j = 0;
152             while(j < s->n2 && !s->m[i][j]){
153                 j++;
154             }
```

```
155
156             if(j<s->n2){
157                 s->sol[j] = 0;
158
159                 for(int k = s->n2-1; k>j; k--){
160                     s->sol[j] -= s->m[i][k] * s->sol[k];
161                 }
162                 s->sol[j] = abs(s->sol[j]) % 2;
163             }
164         }
165     }
166
167     void free_system(system_t s){
168         for(int i = 0; i<s->n1; i++){
169             free(s->m[i]);
170         }
171         free(s->m);
172         free(s->sol);
173         free(s->perm);
174         free(s);
175     }
```

2.7 parse_input.h

```
#pragma once
#include <gmp.h>
#include <stdbool.h>

typedef enum {DIXON, QSIEVE, MPQS, PMPQS} TYPE;

typedef struct input_s {
    char* output_file;
    int bound, sieving_interval;
    mpz_t N;
    bool quiet;
    TYPE algorithm;
    int extra;
    int delta;
} input_t;

input_t* parse_input(int argc, char** argv);
void free_input(input_t* input);
```

2.8 parse_input.c

```c
#include "parse_input.h"
#include <stdlib.h>
#include <string.h>
#include <gmp.h>
#include <stdbool.h>

input_t* init_input(void){
    input_t* input = malloc(sizeof(input_t));
    input->bound = -1;
    input->output_file = NULL;
    input->sieving_interval = -1;
    input->extra = -1;
    input->quiet = false;
    input->algorithm = QSIEVE;
    input->delta = 0;
    mpz_init_set_ui(input->N, 0);
    return input;
}

bool valid_int(char* str){
    int i = 0;
    char c = str[i];
    while(c != '\0'){
        if(c<48 || c>57) return false;
        c = str[++i];
    }

    return true;
}

void free_input(input_t* input){
    if(input->output_file) free(input->output_file);
    mpz_clear(input->N);
    free(input);
}

input_t* parse_input(int argc, char** argv){
    input_t* input = init_input();

    int i = 1;
    while(i<argc){
        if(strcmp(argv[i], "-b") == 0 || strcmp(argv[i], "--bound
           ") == 0){
            i++;
            if(i<argc){
                if(valid_int(argv[i])) input->bound = atoi(argv[i
                   ]);
                else return NULL;}
            else return NULL;
        }
```

```
49
50          else if(strcmp(argv[i], "-s") == 0 || strcmp(argv[i], "--
               sieving_interval") == 0){
51              i++;
52              if(i<argc){
53                  if(valid_int(argv[i])) input->sieving_interval =
                       atoi(argv[i]);
54                  else return NULL;}
55              else return NULL;
56          }
57
58          else if(strcmp(argv[i], "-e") == 0 || strcmp(argv[i], "--
               extra") == 0){
59              i++;
60              if(i<argc){
61                  if(valid_int(argv[i])) input->extra = atoi(argv[i
                       ]);
62                  else return NULL;}
63              else return NULL;
64          }
65
66          else if(strcmp(argv[i], "-n") == 0 || strcmp(argv[i], "--
               number") == 0){
67              i++;
68              if(i<argc){
69                  if(valid_int(argv[i])) mpz_set_str(input->N, argv
                       [i], 10);
70                  else return NULL;}
71              else return NULL;
72          }
73
74          else if(strcmp(argv[i], "-d") == 0 || strcmp(argv[i], "--
               delta") == 0){
75              i++;
76              if(i<argc){
77                  if(valid_int(argv[i])) input->delta = atoi(argv[i
                       ]);
78                  else return NULL;}
79              else return NULL;
80          }
81
82          else if(strcmp(argv[i], "-o") == 0){
83              i++;
84              if(i<argc) input->output_file = argv[i];
85              else return NULL;
86      }
87
88          else if(strcmp(argv[i], "-t") == 0 || strcmp(argv[i], "--
               type") == 0){
89              i++;
90              if(i<argc) {
91                  if(strcmp(argv[i], "dixon") == 0) input->
```

```
                    algorithm = DIXON;
92                  else if(strcmp(argv[i], "qsieve") == 0) input->
                    algorithm = QSIEVE;
93                  else if(strcmp(argv[i], "mpqs") == 0) input->
                    algorithm = MPQS;
94                  else if(strcmp(argv[i], "pmpqs") == 0) input->
                    algorithm = PMPQS;
95                  else return NULL;}
96              else return NULL;
97          }
98
99          else if(strcmp(argv[i], "-q") == 0 ||
100                 strcmp(argv[i], "-stfu") == 0 /*easter egg*/ ||
101                 strcmp(argv[i], "--quiet") == 0){
102             input->quiet = true;
103         }
104
105         else return NULL;
106
107         i++;
108     }
109
110     return input;
111 }
```

2.9 list_matrix_utils.h

```
#pragma once

void print_list(int* l, int n);
void print_ll(int** ll, int n1, int n2);
void free_ll(int** m, int n1);
```

2.10 list_matrix_utils.c

```c
#include <stdio.h>
#include <stdlib.h>

void print_list(int* l, int n){
    for(int i = 0; i<n; i++){
        printf("%d ", l[i]);
    }
    printf("\n");
}

void print_ll(int** ll, int n1, int n2){
    for(int i = 0; i<n1; i++){
        print_list(ll[i], n2);
    }
    printf("\n");
}

void free_ll(int** m, int n1){
    for(int i = 0; i<n1; i++){
        free(m[i]);
    }
    free(m);
}
```

2.11 factorbase.h

```
#pragma once
#include <gmp.h>

// bruh
bool is_prime(int n);

// calculates pi(n), the number of prime numbers <= n
int pi(int n);

// returns a list of piB first primes
int* primes(int piB, int B);

/** Reduces the factor base of the algorithm, refer to:
 * Quadratic sieve factorisation algorithm
 * Bc. Ondˇrej Vladyka
 * Section 2.3.1 (p.16)
*/
int* prime_base(mpz_t n, int* pb_len, int* primes, int piB);
```

2.12 factorbase.c

```c
#include <stdbool.h>
#include <gmp.h>
#include <stdlib.h>

bool is_prime(int n) {
    // Corner cases
    if (n <= 1)
        return false;
    if (n <= 3)
        return true;

    // This is checked so that we can skip
    // middle five numbers in below loop
    if (n % 2 == 0 || n % 3 == 0)
        return false;

    for (int i = 5; i * i <= n; i = i + 6)
        if (n % i == 0 || n % (i + 2) == 0)
            return false;

    return true;
}

int pi(int n) {
    int k = 0;
    for (int i = 2; i <= n; i++) {
        if (is_prime(i)) k++;
    }
    return k;
}

int* primes(int piB, int B){
    int* p = malloc(piB*sizeof(int));
    int k = 0;
    for (int i = 2; i <= B; i++) {
        if (is_prime(i)){
            p[k] = i;
            k++;
        }
    }
    return p;

}

/* Used for legendre symbol, exists in gmp already
bool euler_criterion(mpz_t n, int p){
    int e = (p-1)/2;
    mpz_t r, p1;
    mpz_init(r);
    mpz_init_set_ui(p1, p);
```

```
51      mpz_powm_ui(r, n, e, p1);
52      return(mpz_cmp_ui(r, 1) == 0);
53  }
54  */
55
56  int* prime_base(mpz_t n, int* pb_len, int* primes, int piB){
57
58      int* pb = malloc(piB*sizeof(int));
59      pb[0] = 2;
60
61      int j = 1;
62      mpz_t p1;
63      mpz_init(p1);
64      for(int i = 1; i<piB; i++){
65          mpz_set_ui(p1, primes[i]);
66          if(mpz_legendre(n, p1) == 1){
67              //printf("%d\n", primes[i]);
68              pb[j] = primes[i];
69              j++;
70          }
71      }
72      *pb_len = j;
73      pb = realloc(pb, (j+1)*sizeof(int)); // +1 used for mpqs
74
75      mpz_clear(p1);
76      return pb;
77  }
```

3 Algorithme de Dixon

3.1 dixon/dixon.h

```
1 #pragma once
2
3 int** dixon(mpz_t* z, mpz_t N, int pb_len, int* pb, int extra,
     bool tests);
```

3.2 dixon/dixon.c

```c
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

bool vectorize_dixon(mpz_t n, int* v, int pb_len, int* pb){
    /** Attemps naive factorisation to 'n' with the primes in
     * the prime base 'pb' and putting the result into 'v',
        vector of powers of
     * the primes in the prime base
     * If it succeeds, returns true, otherwise, returns false
    */
    for(int i = 0; i<pb_len; i++){
        v[i] = 0;
    }

    for(int i = 0; i<pb_len && (mpz_cmp_ui(n, 1) != 0); i++){
        while (mpz_divisible_ui_p(n, pb[i])){
            v[i]++;
            mpz_divexact_ui(n, n, pb[i]);
        }
    }

    if(mpz_cmp_ui(n, 1) == 0)
        return true;
    return false;
}

int** dixon(mpz_t* z, mpz_t N, int pb_len, int* pb, int extra,
   bool tests){
    /** Gets pb_len+extra b-smooth realtions definied at:
     * Quadratic sieve factorisation algorithm
     * Bc. Ondřej Vladyka
     * Definition 1.11 (p.5)
     */

    //ceil(sqrt(n))
    mpz_t sqrt_N;
    mpz_init(sqrt_N);
    mpz_sqrt(sqrt_N, N);
    mpz_add_ui(sqrt_N, sqrt_N, 1);

    mpz_t zi;
    mpz_t zi_cpy;
    mpz_init_set(zi, sqrt_N);
    mpz_init(zi_cpy);

    int** v = malloc((pb_len+extra)*sizeof(int*));

    for(int i = 0; i < pb_len+extra; i++){
```

```
49            bool found = false;
50            int* vi = malloc(pb_len*sizeof(int));
51
52            while(!found){
53                mpz_add_ui(zi, zi, 1);
54                mpz_mul(zi_cpy, zi, zi);
55                mpz_mod(zi_cpy, zi_cpy, N);
56
57                found = vectorize_dixon(zi_cpy, vi, pb_len, pb);
58            }
59            if(!tests){
60                printf("\r");
61                printf("%.1f%%", (float)i/(pb_len+extra-1)*100);
62                fflush(stdout);
63            }
64
65            v[i] = vi;
66            mpz_set(z[i], zi);
67        }
68        if(!tests) printf("\n");
69
70        mpz_clears(sqrt_N, zi, zi_cpy, NULL);
71
72
73        return v;
74    }
```

4 Crible quadratique

4.1 qsieve/qsieve.h

```
#pragma once
#include <gmp.h>
#include <stdbool.h>

bool vectorize_qsieve(mpz_t n, int* v, int pb_len, int* pb);
int** qsieve(mpz_t* z, mpz_t N, int pb_len, int* pb, int extra,
   int s, bool tests);
```

4.2 qsieve/qsieve.c

```c
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

#include "../system.h"
#include "../tonellishanks.h"

bool vectorize_qsieve(mpz_t n, int* v, int pb_len, int* pb){
    /** Attemps naive factorisation to 'n' with the primes in
     * the prime base 'pb' and putting the result into 'v',
        vector of powers of
     * the primes in the prime base
     * If it succeeds, returns true, otherwise, returns false
    */
    for(int i = 0; i<pb_len; i++){
        v[i] = 0;
    }

    for(int i = 0; i<pb_len && (mpz_cmp_ui(n, 1) != 0); i++){
        while (mpz_divisible_ui_p(n, pb[i])){
            v[i]++;
            mpz_divexact_ui(n, n, pb[i]);
        }
    }

    if(mpz_cmp_ui(n, 1) == 0)
        return true;
    return false;
}

float* prime_logs(int* pb, int pb_len){
    float* plogs = malloc(pb_len*sizeof(float));

    for(int i = 0; i<pb_len; i++){
        plogs[i] = log2(pb[i]);
    }

    return plogs;
}

int calculate_threshhold(mpz_t N, mpz_t sqrt_N, int s, int
   loop_number, int* pb, int pb_len){

    mpz_t qstart;
    mpz_init_set_ui(qstart, s);
    mpz_mul_ui(qstart, qstart, loop_number);
    mpz_add(qstart, qstart, sqrt_N);
```

```
49      mpz_mul(qstart, qstart, qstart);
50      mpz_sub(qstart, qstart, N);
51
52      int t = mpz_sizeinbase(qstart, 2) - (int) log2(pb[pb_len-1]);
53      mpz_clear(qstart);
54      return t;
55  }
56
57  int** qsieve(mpz_t* z, mpz_t N, int pb_len, int* pb, int extra,
        int s, bool quiet){
58      /** Gets pb_len+extra zis that are b-smooth, definied at:
59       * Quadratic sieve factorisation algorithm
60       * Bc. Ondřej Vladyka
61       * Definition 1.11 (p.5)
62       */
63
64      //ceil(sqrt(n))
65      mpz_t sqrt_N;
66      mpz_init(sqrt_N);
67      mpz_sqrt(sqrt_N, N);
68      mpz_add_ui(sqrt_N, sqrt_N, 1);
69
70      mpz_t zi;
71      mpz_init_set(zi, sqrt_N);
72      mpz_t qx;
73      mpz_init(qx);
74
75      int** v = malloc((pb_len+extra)*sizeof(int*));
76      for(int i = 0; i<pb_len+extra; i++){
77          v[i] = malloc(pb_len*sizeof(int*));
78      }
79      float* sinterval = malloc(s*sizeof(float));
80      float* plogs = prime_logs(pb, pb_len);
81
82
83      // TESTS
84      mpz_t temp;
85      mpz_init(temp);
86      // END TESTS
87
88
89      int* x1 = malloc(pb_len*sizeof(int));
90      int* x2 = malloc(pb_len*sizeof(int));
91
92      // find solution for 2
93      mpz_set(temp, sqrt_N);
94      mpz_mul(temp, temp, temp);
95      mpz_sub(temp, temp, N);
96      x1[0] = 0;
97      if(mpz_divisible_ui_p(temp, 2) == 0) x1[0] = 1;
98
99      int sol1, sol2;
```

```
100         for(int i = 1; i < pb_len; i++){
101 
102                 tonelli_shanks_ui(N, pb[i], &sol1, &sol2);
103                 x1[i] = sol1;
104                 x2[i] = sol2;
105 
106                 // change solution from x^2 = n [p] to (sqrt(N) + x)^2
                       = n [p]
107                 mpz_set_ui(temp, x1[i]);
108                 mpz_sub(temp, temp, sqrt_N);
109                 mpz_mod_ui(temp, temp, pb[i]);
110 
111                 x1[i] = mpz_get_ui(temp);
112 
113                 mpz_set_ui(temp, x2[i]);
114                 mpz_sub(temp, temp, sqrt_N);
115                 mpz_mod_ui(temp, temp, pb[i]);
116 
117                 x2[i] = mpz_get_ui(temp);
118         }
119         mpz_clear(temp);
120 
121         int loop_number = 0;
122         int relations_found = 0;
123         int tries = 0;
124         while(relations_found < pb_len + extra){
125 
126             for(int i = 0; i<s; i++){
127                 sinterval[i] = 0;
128             }
129 
130             // sieve for 2
131             while(x1[0]<s){
132                 sinterval[x1[0]] += plogs[0];
133                 x1[0] += pb[0];
134             }
135             x1[0] = x1[0] - s;
136 
137             // sieve other primes
138             for(int i = 1; i < pb_len; i++){
139 
140                 while(x1[i]<s){
141                     sinterval[x1[i]] += plogs[i];
142                     x1[i] += pb[i];
143                 }
144 
145                 while(x2[i]<s){
146 
147                     sinterval[x2[i]] += plogs[i];
148                     x2[i] += pb[i];
149                 }
150 
```

```
151             //next interval
152             x1[i] = x1[i] - s;
153             x2[i] = x2[i] - s;
154         }
155 
156         int t = calculate_threshhold(N, sqrt_N, s, loop_number,
                pb, pb_len);
157         //printf("t = %d\n", t);
158 
159         bool found;
160         for(int i = 0; i<s && relations_found < pb_len + extra; i
                ++){
161             if(sinterval[i] > t){
162                 tries++;
163 
164                 // zi = sqrt(n) + x where x = s*loopnumber + i
165                 mpz_set_ui(zi, s);
166                 mpz_mul_ui(zi, zi, loop_number);
167                 mpz_add_ui(zi, zi, i);
168                 mpz_add(zi, zi, sqrt_N);
169 
170                 // qx = zi**2 - N
171                 mpz_mul(qx, zi, zi);
172                 mpz_sub(qx, qx, N);
173 
174                 found = vectorize_qsieve(qx, v[relations_found],
                        pb_len, pb);
175 
176                 if(found){
177                     mpz_set(z[relations_found], zi);
178                     relations_found++;
179                     found = false;
180                     if(!quiet){
181                         printf("\r");
182                         printf("%.1f%% | %.1f%%", (float)
                                relations_found/(pb_len+extra)*100, (
                                float)relations_found/tries*100);
183                         fflush(stdout);
184                     }
185                 }
186             }
187         }
188         loop_number++;
189     }
190 
191     if(!quiet) printf("\n");
192 
193     mpz_clears(sqrt_N, zi, qx, NULL);
194     free(x1);
195     free(x2);
196     free(sinterval);
197     free(plogs);
```

```
198
199         return v;
200     }
```

5 MPQS

5.1 mpqs/common_mpqs.h

```
#pragma once
#include <gmp.h>
#include <stdbool.h>

int calculate_threshhold_mpqs(mpz_t sqrt_N, int s, int* pb, int
    pb_len, int delta);
float* prime_logs_mpqs(int* pb, int pb_len);
bool vectorize_mpqs(mpz_t n, int* v, int pb_len, int* pb);
bool already_added(mpz_t zi, mpz_t* z, int relations_found);
```

5.2 mpqs/common_mpqs.c

```c
#include <gmp.h>
#include <stdbool.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

int calculate_threshhold_mpqs(mpz_t sqrt_N, int s, int* pb, int pb_len, int delta){

    mpz_t qstart;
    mpz_init_set_ui(qstart, s);
    mpz_mul(qstart, qstart, sqrt_N);

    int t = mpz_sizeinbase(qstart, 2) - (int) log2(pb[pb_len-1]) - delta;
    mpz_clear(qstart);
    return t;
}

float* prime_logs_mpqs(int* pb, int pb_len){
    float* plogs = malloc(pb_len*sizeof(float));

    for(int i = 0; i<pb_len; i++){
        plogs[i] = log2(pb[i]);
    }

    return plogs;
}

bool vectorize_mpqs(mpz_t n, int* v, int pb_len, int* pb){
    /** Attemps naive factorisation to 'n' with the primes in
     * the prime base 'pb' and putting the result into 'v', vector of powers of
     * the primes in the prime base
     * If it succeeds, returns true, otherwise, returns false
    */
    for(int i = 0; i<pb_len; i++){
        v[i] = 0;
    }
    if(mpz_sgn(n)<0){
        v[pb_len] = 1;
        mpz_neg(n, n);
    }
    else{
        v[pb_len] = 0;
    }

    for(int i = 0; i<pb_len && (mpz_cmp_ui(n, 1) != 0); i++){
        while (mpz_divisible_ui_p(n, pb[i])){
            v[i]++;
```

```
48                  mpz_divexact_ui(n, n, pb[i]);
49              }
50          }
51
52          if(mpz_cmp_ui(n, 1) == 0)
53              return true;
54          return false;
55      }
56
57      bool already_added(mpz_t zi, mpz_t* z, int relations_found){
58          for(int i = 0; i<relations_found; i++){
59              if(mpz_cmp(zi, z[i]) == 0){
60                  return true;
61              }
62          }
63          return false;
64      }
```

5.3 mpqs/polynomial.h

```
1  #pragma once
2  #include <gmp.h>
3  #include <stdbool.h>
4
5  struct poly_s {
6      mpz_t d;
7      mpz_t N;
8
9      mpz_t a;
10     mpz_t b;
11     mpz_t c;
12
13     mpz_t zi;
14     mpz_t qx;
15
16     // used to make operations without declaring and freeing
           everytime
17     mpz_t op1, op2, op3;
18 };
19
20 typedef struct poly_s* poly_t;
21
22 void get_next_poly(poly_t p);
23 poly_t init_poly(mpz_t N, int M);
24 void calc_poly(poly_t p, mpz_t x);
25 poly_t copy_poly(poly_t p);
26 void free_poly(poly_t p);
```

5.4 mpqs/polynomial.c

```c
#include "polynomial.h"
#include <gmp.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

#include "../tonellishanks.h"

void calc_coefficients(poly_t p){
    mpz_mul(p->a, p->d, p->d);

    mpz_t x1, x2;
    mpz_inits(x1, x2, NULL);
    tonelli_shanks_mpz(p->N, p->d, x1, x2);

    // getting ready for congruence solve for raising solution
    mpz_mul_ui(p->op1, x1, 2);

    mpz_mul(p->op2, x1, x1);
    mpz_sub(p->op2, p->op2, p->N);
    mpz_divexact(p->op2, p->op2, p->d);
    mpz_neg(p->op2, p->op2);
    mpz_mod(p->op2, p->op2, p->d);

    mpz_t g, n, m;
    mpz_inits(g, n, m, NULL);
    mpz_gcdext(g, n, m, p->d, p->op1);
    assert(mpz_cmp_ui(g, 1) == 0);
    mpz_mul(p->op1, p->op2, m); // t
    mpz_clears(g, n, m, NULL);

    mpz_set(p->b, p->d);
    mpz_mul(p->b, p->b, p->op1);
    mpz_add(p->b, p->b, x1);

    mpz_mul(p->op1, p->b, p->b);
    assert(mpz_congruent_p(p->op1, p->N, p->a) != 0);

    mpz_sub(p->c, p->op1, p->N);
    mpz_divexact(p->c, p->c, p->a);

    mpz_clears(x1, x2, NULL);
}

void get_next_poly(poly_t p){
    mpz_nextprime(p->d, p->d);
    while(mpz_legendre(p->N, p->d) != 1){
        mpz_nextprime(p->d, p->d);
    }
    calc_coefficients(p);
```

```
51  }
52
53  poly_t init_poly(mpz_t N, int M){
54      poly_t p = malloc(sizeof(struct poly_s));
55
56      mpz_inits(p->d, p->N, p->a, p->b, p->c, p->op1, p->op2, p->
            op3, p->zi, p->qx, NULL);
57      mpz_set(p->N, N);
58
59      // choose value of d according to 2.4.2
60      // sqrt( (sqrt(2N))/M )
61      mpz_mul_ui(p->op1, N, 2);
62      mpz_sqrt(p->op1, p->op1);
63      mpz_div_ui(p->op1, p->op1, M);
64      mpz_sqrt(p->op1, p->op1);
65      mpz_prevprime(p->d, p->op1);
66
67      // get next prime such that (n/p) = 1
68      while(mpz_legendre(N, p->d) != 1){
69          mpz_nextprime(p->d, p->d);
70      }
71
72      calc_coefficients(p);
73      return p;
74  }
75
76  void calc_poly(poly_t p, mpz_t x){
77      mpz_mul(p->zi, p->a, x);
78      mpz_add(p->zi, p->zi, p->b);
79
80      mpz_mul(p->qx, x, x);
81      mpz_mul(p->qx, p->qx, p->a);
82
83      mpz_mul(p->op1, p->b, x);
84      mpz_mul_ui(p->op1, p->op1, 2);
85      mpz_add(p->qx, p->qx, p->op1);
86
87      mpz_add(p->qx, p->qx, p->c);
88
89  }
90
91  void free_poly(poly_t p){
92      mpz_clears(p->d, p->N, p->a, p->b, p->c, p->op1, p->op2, p->
            op3, p->zi, p->qx, NULL);
93      free(p);
94  }
95
96  poly_t copy_poly(poly_t p){
97      poly_t cpy = malloc(sizeof(struct poly_s));
98
99      mpz_inits(cpy->d, cpy->N, cpy->a, cpy->b, cpy->c, cpy->op1,
            cpy->op2, cpy->op3, cpy->zi, cpy->qx, NULL);
```

```
100
101         mpz_set(cpy->d, p->d);
102         mpz_set(cpy->N, p->N);
103
104         mpz_set(cpy->a, p->a);
105         mpz_set(cpy->b, p->b);
106         mpz_set(cpy->c, p->c);
107
108         return cpy;
109     }
```

5.5 mpqs/mpqs.h

```
#pragma once

#include <gmp.h>
#include <stdbool.h>

int** mpqs(mpz_t* z, mpz_t* d, mpz_t N, int pb_len, int* pb, int
   extra, int s, int delta, bool quiet);
```

5.6 mpqs/mpqs.c

```c
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <time.h>

#include "polynomial.h"
#include "common_mpqs.h"
#include "../system.h"
#include "../tonellishanks.h"

int** mpqs(mpz_t* z, mpz_t* d, mpz_t N, int pb_len, int* pb, int extra, int s, int delta, bool quiet){
    /** Gets pb_len+extra zis that are b-smooth, definied at:
     * Quadratic sieve factorisation algorithm
     * Bc. Ondřej Vladyka
     * Definition 1.11 (p.5)
     */

    //ceil(sqrt(n))
    mpz_t sqrt_N;
    mpz_init(sqrt_N);
    mpz_sqrt(sqrt_N, N);
    mpz_add_ui(sqrt_N, sqrt_N, 1);

    mpz_t x;
    mpz_init(x);
    poly_t Q = init_poly(N, s);

    int** v = malloc((pb_len+extra)*sizeof(int*));
    for(int i = 0; i<pb_len+extra; i++){
        v[i] = malloc((pb_len+1)*sizeof(int*)); // +1 for -1
    }
    float* sinterval = malloc(2*s*sizeof(float));
    float* plogs = prime_logs_mpqs(pb, pb_len);
    int t = calculate_threshhold_mpqs(sqrt_N, s, pb, pb_len, delta);


    // TESTS
    mpz_t temp;
    mpz_init(temp);
    // END TESTS


    int* r = malloc(pb_len*sizeof(int));
    int* x1 = malloc(pb_len*sizeof(int));
    int* x2 = malloc(pb_len*sizeof(int));
```

```
49
50      int sol1, sol2;
51      for(int i = 1; i < pb_len; i++){
52          tonelli_shanks_ui(N, pb[i], &sol1, &sol2);
53          r[i] = sol1;
54      }
55
56      mpz_t g, m, n, pi;
57      mpz_inits(g, m, n, pi, NULL);
58
59      int relations_found = 0;
60      clock_t start;
61      start = clock();
62      int tries = 0;
63      while(relations_found < pb_len + extra){
64
65          // for 2
66          mpz_set_ui(temp, 0);
67          calc_poly(Q, temp);
68          x1[0] = 0;
69          if(mpz_divisible_ui_p(Q->qx, 2) == 0) x1[0] = 1;
70
71          //others
72          for(int i = 1; i<pb_len; i++){
73              mpz_set_ui(pi, pb[i]);
74              mpz_gcdext(g, m, n, Q->a, pi);
75              if(mpz_cmp_ui(g, 1) != 0){
76                  fprintf(stderr, "ERROR: Number is too small for
                        the current implementation of MPQS\n");
77                  exit(1);
78              }
79
80              mpz_set_ui(temp, r[i]);
81              mpz_sub(temp, temp, Q->b);
82              mpz_mul(temp, temp, m);
83              mpz_mod(temp, temp, pi);
84
85              x1[i] = mpz_get_ui(temp);
86
87              //calc_poly(Q, temp);
88              //assert(mpz_divisible_ui_p(Q->qx, pb[i]) != 0);
89
90              mpz_set_ui(temp, pb[i]);
91              mpz_sub_ui(temp, temp, r[i]);
92              mpz_sub(temp, temp, Q->b);
93              mpz_mul(temp, temp, m);
94              mpz_mod(temp, temp, pi);
95
96              x2[i] = mpz_get_ui(temp);
97
98              //calc_poly(Q, temp);
99              //assert(mpz_divisible_ui_p(Q->qx, pb[i]) != 0);
```

```
100
101
102             //realign sieving interval to [-s, s]
103             int k = (x1[i] + s)/pb[i];
104             x1[i] -= k * pb[i];
105             x1[i] += s;
106
107             k = (x2[i] + s)/pb[i];
108             x2[i] -= k * pb[i];
109             x2[i] += s;
110
111             //mpz_set_si(temp, -s);
112             //mpz_add_ui(temp, temp, x1[i]);
113             //calc_poly(Q, temp);
114             //assert(mpz_divisible_ui_p(Q->qx, pb[i]) != 0);
115         }
116
117         for(int i = 0; i<2*s; i++){
118             sinterval[i] = 0;
119         }
120
121         /*
122         // sieve for 2
123         while(x1[0]<2*s){
124             sinterval[x1[0]] += plogs[0];
125             x1[0] += pb[0];
126         }
127         */
128
129         // sieve other primes
130         for(int i = 30; i < pb_len; i++){
131
132             while(x1[i]<2*s){
133                 sinterval[x1[i]] += plogs[i];
134                 x1[i] += pb[i];
135             }
136
137             while(x2[i]<2*s){
138                 sinterval[x2[i]] += plogs[i];
139                 x2[i] += pb[i];
140             }
141         }
142
143
144         bool found;
145         bool update_time = false;
146         for(int i = 0; i<2*s && relations_found < pb_len + extra;
                i++){
147             if(sinterval[i] > t){
148                 tries++;
149                 mpz_set_si(x, -s);
150                 mpz_add_ui(x, x, i);
```

```
151                     calc_poly(Q, x);
152
153                     if(!already_added(Q->zi, z, relations_found)){
154                         found = vectorize_mpqs(Q->qx, v[
                               relations_found], pb_len, pb);
155                         if(found){
156                             mpz_set(z[relations_found], Q->zi);
157                             mpz_set(d[relations_found], Q->d);
158                             relations_found++;
159                             update_time = true;
160                             found = false;
161                             if(!quiet){
162                                 printf("\r");
163                                 printf("%.1f%% | %.1f%%", (float)
                                       relations_found/(pb_len+extra)
                                       *100, (float)relations_found/tries
                                       *100);
164                                 fflush(stdout);
165                             }
166                         }
167                     }
168                 }
169             }
170
171             if(update_time && !quiet) printf(" (~%.0fs left)        "
                    , (double)(clock() - start)/CLOCKS_PER_SEC/
                   relations_found*((pb_len+extra - relations_found)));
172             get_next_poly(Q);
173         }
174
175         if(!quiet) printf("\n");
176         mpz_clears(sqrt_N, temp, g, m, n, pi, x, NULL);
177         free(x1);
178         free(x2);
179         free(r);
180         free(sinterval);
181         free(plogs);
182         free_poly(Q);
183
184         return v;
185     }
```

6 MPQS parallélisé

6.1 mpqs/parallel_mpqs.h

```
#pragma once
#include <gmp.h>
#include "polynomial.h"
#include <sys/time.h>
#include <stdint.h>

struct sieve_arg_s {
    // used for sieveing
    int* pb;
    int pb_len;
    int extra;
    int* r;
    float* plogs;
    int s;
    int t;
    int* relations_found;
    int** v;
    bool quiet;
    mpz_t* z;
    mpz_t* d;
    poly_t Qinit;

    // used to print progress and predicted time left
    struct timeval begin;
    uint_fast64_t* tries;

    // used to constantly have a certain number of threads
       running
    int thread_id;
    bool* threads_running;
};
typedef struct sieve_arg_s sieve_arg_t;

bool already_added(mpz_t zi, mpz_t* z, int relations_found);
void* sieve_100_polys (void* args);
int** parallel_mpqs(mpz_t* z, mpz_t* d, mpz_t N, int pb_len, int*
    pb, int extra, int s, int delta, bool quiet);
```

6.2 mpqs/parallel_mpqs.c

```
#include <gmp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <time.h>
#include <pthread.h>
#include <sys/time.h>

#include "polynomial.h"
#include "common_mpqs.h"
#include "parallel_mpqs.h"
#include "../system.h"
#include "../tonellishanks.h"

pthread_mutex_t mutex;


void* sieve_100_polys (void* args){
    sieve_arg_t* arg = (sieve_arg_t*) args;

    poly_t Q = copy_poly(arg->Qinit);

    mpz_t temp, g, m, n, pi, x;
    mpz_inits(temp, g, m, n, pi, x, NULL);
    float* sinterval = malloc(2*arg->s*sizeof(float));
    int* x1 = malloc(arg->pb_len*sizeof(int));
    int* x2 = malloc(arg->pb_len*sizeof(int));

    for(int i = 0; i<100 && *(arg->relations_found) < arg->pb_len
        + arg->extra; i++){
        get_next_poly(Q);

        //get sol for 2
        mpz_set_ui(temp, 0);
        calc_poly(Q, temp);
        x1[0] = 0;
        if(mpz_divisible_ui_p(Q->qx, 2) == 0) x1[0] = 1;

        //get sol for others
        for(int i = 1; i<arg->pb_len; i++){
            mpz_set_ui(pi, arg->pb[i]);
            mpz_gcdext(g, m, n, Q->a, pi);
            if(mpz_cmp_ui(g, 1) != 0){
                fprintf(stderr, "ERROR: Number is too small for
                    the current implementation of MPQS\n");
                exit(1);
            }

```

```
49                  mpz_set_ui(temp, arg->r[i]);
50                  mpz_sub(temp, temp, Q->b);
51                  mpz_mul(temp, temp, m);
52                  mpz_mod(temp, temp, pi);
53
54                  x1[i] = mpz_get_ui(temp);
55
56                  //calc_poly(Q, temp);
57                  //assert(mpz_divisible_ui_p(Q->qx, arg->pb[i]) != 0);
58
59                  mpz_set_ui(temp, arg->pb[i]);
60                  mpz_sub_ui(temp, temp, arg->r[i]);
61                  mpz_sub(temp, temp, Q->b);
62                  mpz_mul(temp, temp, m);
63                  mpz_mod(temp, temp, pi);
64
65                  x2[i] = mpz_get_ui(temp);
66
67                  //calc_poly(Q, temp);
68                  //assert(mpz_divisible_ui_p(Q->qx, arg->pb[i]) != 0);
69
70                  //realign sieving interval to [-s, s]
71                  int k = (x1[i] + arg->s)/arg->pb[i];
72                  x1[i] -= k * arg->pb[i];
73                  x1[i] += arg->s;
74
75                  k = (x2[i] + arg->s)/arg->pb[i];
76                  x2[i] -= k * arg->pb[i];
77                  x2[i] += arg->s;
78
79                  //mpz_set_si(temp, -arg->s);
80                  //mpz_add_ui(temp, temp, x1[i]);
81                  //calc_poly(Q, temp);
82                  //assert(mpz_divisible_ui_p(Q->qx, arg->pb[i]) != 0);
83              }
84
85              //reset sieveing_interval
86              for(int i = 0; i<2*arg->s; i++){
87                  sinterval[i] = 0;
88              }
89
90              /*
91              // sieve for 2
92              while(x1[0]<2*arg->s){
93                  sinterval[x1[0]] += arg->plogs[0];
94                  x1[0] += arg->pb[0];
95              }
96              */
97
98              // sieve other primes
99              for(int i = 30; i < arg->pb_len; i++){
100                 while(x1[i]<2*arg->s){
```

```
101                     sinterval[x1[i]] += arg->plogs[i];
102                     x1[i] += arg->pb[i];
103                 }
104                 while(x2[i]<2*arg->s){
105                     sinterval[x2[i]] += arg->plogs[i];
106                     x2[i] += arg->pb[i];
107                 }
108             }
109 
110             bool found;
111             bool update_time = false;
112             pthread_mutex_lock(&mutex);
113             for(int i = 0; i<2*arg->s && *(arg->relations_found) <
                    arg->pb_len + arg->extra; i++){
114                 if(sinterval[i] > arg->t){
115                     *(arg->tries) += 1;
116                     mpz_set_si(x, -arg->s);
117                     mpz_add_ui(x, x, i);
118                     calc_poly(Q, x);
119 
120                     if(!already_added(Q->zi, arg->z, *(arg->
                            relations_found))){
121                         found = vectorize_mpqs(Q->qx, arg->v[*(arg->
                                relations_found)], arg->pb_len, arg->pb);
122                         if(found){
123                             mpz_set(arg->z[*(arg->relations_found)],
                                    Q->zi);
124                             mpz_set(arg->d[*(arg->relations_found)],
                                    Q->d);
125                             *(arg->relations_found) += 1;
126                             found = false;
127                             update_time = true;
128                             if(!arg->quiet){
129                                 printf("\r");
130                                 printf("%.1f%% | %.1f%%", (float)(*(
                                        arg->relations_found))/(arg->
                                        pb_len+arg->extra)*100, (float)(*(
                                        arg->relations_found))/(*(arg->
                                        tries))*100);
131                                 fflush(stdout);
132                             }
133                         }
134                     }
135                 }
136             }
137 
138             struct timeval current;
139             gettimeofday(&current, 0);
140             long seconds = current.tv_sec - arg->begin.tv_sec;
141             long microseconds = current.tv_usec - arg->begin.tv_usec;
142             double elapsed = seconds + microseconds*1e-6;
143             if(update_time && !arg->quiet) printf(" (~%.0fs left)    
```

```
            ␣␣␣␣" , elapsed/(*arg->relations_found)*(arg->pb_len+
            arg->extra - (*arg->relations_found)));
144         pthread_mutex_unlock(&mutex);
145     }
146
147     mpz_clears(temp, g, m, n, pi, x, NULL);
148     free(x1);
149     free(x2);
150     free(sinterval);
151     free_poly(Q);
152
153     arg->threads_running[arg->thread_id] = false;
154     return NULL;
155 }
156
157 int** parallel_mpqs(mpz_t* z, mpz_t* d, mpz_t N, int pb_len, int*
        pb, int extra, int s, int delta, bool quiet){
158     /** Gets pb_len+extra zis that are b-smooth, definied at:
159      * Quadratic sieve factorisation algorithm
160      * Bc. Ondˇrej Vladyka
161      * Definition 1.11 (p.5)
162      */
163
164     //ceil(sqrt(n))
165     mpz_t sqrt_N;
166     mpz_init(sqrt_N);
167     mpz_sqrt(sqrt_N, N);
168     mpz_add_ui(sqrt_N, sqrt_N, 1);
169
170     poly_t Q = init_poly(N, s);
171
172     int** v = malloc((pb_len+extra)*sizeof(int*));
173     for(int i = 0; i<pb_len+extra; i++){
174         v[i] = malloc((pb_len+1)*sizeof(int*)); // +1 for -1
175     }
176     float* plogs = prime_logs_mpqs(pb, pb_len);
177
178
179     int* r = malloc(pb_len*sizeof(int));
180     int sol1, sol2;
181     for(int i = 1; i < pb_len; i++){
182         tonelli_shanks_ui(N, pb[i], &sol1, &sol2);
183         r[i] = sol1;
184     }
185     int t = calculate_threshhold_mpqs(sqrt_N, s, pb, pb_len,
          delta);
186
187     sieve_arg_t* args = malloc(8*sizeof(sieve_arg_t));
188     pthread_t* threads = malloc(8*sizeof(pthread_t));
189     bool* threads_running = malloc(8*sizeof(bool));
190     for(int i = 0; i<8; i++){
191         threads_running[i] = false;
```

```
192         }
193
194         int relations_found = 0;
195         uint_fast64_t tries = 0;
196         struct timeval begin;
197         gettimeofday(&begin, 0);
198         while(relations_found < pb_len + extra){
199             for(int i = 0; i<8; i++){
200                 if(!threads_running[i]){
201                     args[i] = (sieve_arg_t) {
202                         pb,
203                         pb_len,
204                         extra,
205                         r,
206                         plogs,
207                         s,
208                         t,
209                         &relations_found,
210                         v,
211                         quiet,
212                         z,
213                         d,
214                         Q,
215                         begin,
216                         &tries,
217                         i,
218                         threads_running
219                     };
220                     threads_running[i] = true;
221                     pthread_create(threads+i, NULL, sieve_100_polys,
                            args+i);
222                 }
223                 for(int i = 0; i<100; i++){
224                     get_next_poly(Q);
225                 }
226             }
227         }
228         if(!quiet) printf("\n");
229
230         for(int i = 0; i<8; i++){
231             pthread_join(threads[i], NULL);
232         }
233
234         free(threads);
235         free(args);
236         free(r);
237         free(plogs);
238         free(threads_running);
239         free_poly(Q);
240         mpz_clear(sqrt_N);
241
242         return v;
```

```
243 |     }
```