

# Recherche de facteurs premiers

---

Code du TIPE

Tristan Delcourt (15699)



# Sommaire

<b>1</b>	<b>Point d'entrée du programme</b>	<b>2</b>
1.1	main.c . . . . .	2
<b>2</b>	<b>Modules utiles</b>	<b>9</b>
2.1	vector.h . . . . .	9
2.2	vector.c . . . . .	10
2.3	tonellishanks.h . . . . .	11
2.4	tonellishanks.c . . . . .	12
2.5	system.h . . . . .	16
2.6	system.c . . . . .	17
2.7	parse_input.h . . . . .	21
2.8	parse_input.c . . . . .	22
2.9	list_matrix_utils.h . . . . .	25
2.10	list_matrix_utils.c . . . . .	26
2.11	factorbase.h . . . . .	27
2.12	factorbase.c . . . . .	28
<b>3</b>	<b>Algorithme de Dixon</b>	<b>30</b>
3.1	dixon/dixon.h . . . . .	30
3.2	dixon/dixon.c . . . . .	31
<b>4</b>	<b>Crible quadratique</b>	<b>33</b>
4.1	qsieve/qsieve.h . . . . .	33
4.2	qsieve/qsieve.c . . . . .	34
<b>5</b>	<b>MPQS</b>	<b>39</b>
5.1	mpqs/common_mpqs.h . . . . .	39
5.2	mpqs/common_mpqs.c . . . . .	40
5.3	mpqs/polynomial.h . . . . .	42
5.4	mpqs/polynomial.c . . . . .	43
5.5	mpqs/mpqs.h . . . . .	46
5.6	mpqs/mpqs.c . . . . .	47
<b>6</b>	<b>MPQS parallélisé</b>	<b>51</b>
6.1	mpqs/parallel_mpqs.h . . . . .	51
6.2	mpqs/parallel_mpqs.c . . . . .	52



# 1 Point d'entrée du programme

## 1.1 main.c

```
1  #include <stdbool.h>
2  #include <gmp.h>
3  #include <sys/time.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <assert.h>
7  #include "system.h"
8  #include "vector.h"
9  #include "parse_input.h"
10 #include "factorbase.h"
11 #include "list_matrix_utils.h"
12
13 // Include algorithms
14 // Dixon's method
15 #include "../dixon/dixon.h"
16
17 // The Quadratic Sieve
18 #include "../qsieve/qsieve.h"
19
20 // Multipolynomial Quadratic Sieve
21 #include "../mpqs/polynomial.h"
22 #include "../mpqs/mpqs.h"
23 #include "../mpqs/parallel_mpqs.h"
24
25
26 /**
27  *
28  *
29  * START OF ALGORITHM
30  *
31  */
32
33
34 void rebuild_mpqs(mpz_t prod, mpz_t* d, int* v, int* primes, int
    n1, system_t s){
35     mpz_set_ui(prod, 1);
36     mpz_t temp;
37     mpz_init(temp);
38     for(int i = 0; i<n1; i++){
39         if(s->sol[i]){
40             mpz_mul(prod, prod, d[s->perm[i]]);
41         }
42         mpz_ui_pow_ui(temp, primes[i], v[i]);
43         mpz_mul(prod, prod, temp);
44     }
45     mpz_clear(temp);
46 }
47
48 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
72          * solution of the
73          * output of the system 's', such that each power is even
74          */
75      for(int i = 0; i<s->n1; i++){
76          sum[i] = 0;
77      }
78
79      for(int i = 0; i<s->n2; i++){
80          if(s->sol[i]){
81              add_vect(sum, v[s->perm[i]], s->n1);
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
221             ->N, f);
222         done = true;
223     }
224
225     mpz_add(f, Z1, Z2);
226     mpz_gcd(f, f, input->N);
227
228     if(mpz_cmp_ui(f, 1) != 0 && mpz_cmp(f, input->N) != 0){
229         assert(mpz_divisible_p(input->N, f));
230         if(!input->quiet) gmp_printf("%Zd=0[%Zd]\n", input
231             ->N, f);
232         done = true;
233     }
234
235     if(s->done){
236         if(!input->quiet) fprintf(stderr, "ERROR: no solution
237             for this set of zi\n");
238         exit(1);
239     }
240
241     free(sum);
242     free(pb);
243     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
275             %%\n");
276         return 1;
277     }
278
279     if(input->bound == -1) input->bound = 10000;
280     if(input->sieving_interval == -1) input->sieving_interval =
281         100000;
282     if(input->extra == -1) input->extra = 1;
283
284     struct timeval t1, t2;
285     gettimeofday(&t1, 0);
286     factor(input);
287     gettimeofday(&t2, 0);
288     long seconds = t2.tv_sec - t1.tv_sec;
289     long microseconds = t2.tv_usec - t1.tv_usec;
290     double time_spent = seconds + microseconds*1e-6;
291     if(!input->quiet) printf("Total time: %fs\n", time_spent);
292
293     free_input(input);

```

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

## 2 Modules utiles

### 2.1 vector.h

```
1 | #pragma once
2 | #include <gmp.h>
3 |
4 | void mod_vect(int* v, int mod, int n1);
5 | void add_vect(int* sum, int* op, int n1);
6 | void div_vect(int* v, int d, int n1);
7 | void sub_vect(int** v, int i, int j, int n1);
8 | void prod_vect(mpz_t prod, mpz_t* z, int n1, system_t s);
```

## 2.2 vector.c

```
1  #include <gmp.h>
2  #include <assert.h>
3  #include <stdlib.h>
4  #include "system.h"
5
6  void mod_vect(int* v, int mod, int n1){
7      for(int i = 0; i<n1; i++){
8          v[i] = abs(v[i]) % mod;
9      }
10 }
11
12 void add_vect(int* sum, int* op, int n1){
13     for(int i = 0; i<n1; i++){
14         sum[i] += op[i];
15     }
16 }
17
18
19 void div_vect(int* v, int d, int n1){
20     for(int i = 0; i<n1; i++){
21         assert(v[i]%d == 0);
22         v[i] /= d;
23     }
24 }
25
26 void sub_vect(int** v, int i, int j, int n1){
27     for(int k = 0; k<n1; k++){
28         v[i][k] = v[i][k] - v[j][k];
29     }
30 }
31
32 void prod_vect(mpz_t prod, mpz_t* z, int n1, system_t s){
33     mpz_set_ui(prod, 1);
34     for(int i = 0; i<n1; i++){
35         if(s->sol[i]){
36             mpz_mul(prod, prod, z[s->perm[i]]);
37         }
38     }
39 }
```

## 2.3 tonellishanks.h

```
1 | #pragma once
2 |
3 | #include <gmp.h>
4 |
5 | void tonelli_shanks_ui(mpz_t n, int p, int* x1, int* x2);
6 | void tonelli_shanks_mpz(mpz_t a, mpz_t p, mpz_t x1, mpz_t x2);
```

## 2.4 tonellishanks.c

```
1  #include <stdint.h>
2  #include <gmp.h>
3  #include <stdio.h>
4  #include <assert.h>
5  #include <stdlib.h>
6
7  uint64_t modpow(uint64_t a, uint64_t b, uint64_t n) {
8      uint64_t x = 1, y = a;
9      while (b > 0) {
10         if (b % 2 == 1) {
11             x = (x * y) % n; // multiplying with base
12         }
13         y = (y * y) % n; // squaring the base
14         b /= 2;
15     }
16     return x % n;
17 }
18
19 void tonelli_shanks_ui(mpz_t n, unsigned long int p, int* x1, int
    * x2) {
20     uint64_t q = p - 1;
21     uint64_t ss = 0;
22     uint64_t z = 2;
23     uint64_t c, r, t, m;
24
25
26     while ((q & 1) == 0) {
27         ss += 1;
28         q >>= 1;
29     }
30
31     mpz_t temp, pj;
32     mpz_init(temp);
33     mpz_init_set_ui(pj, p);
34
35     if (ss == 1) {
36         //uint64_t r1 = modpow(n, (p + 1) / 4, p);
37         mpz_powm_ui(temp, n, (p+1)/4, pj);
38         uint64_t r1 = mpz_get_ui(temp);
39
40         *x1 = r1;
41         *x2 = p - r1;
42         mpz_clears(temp, pj, NULL);
43         return;
44     }
45
46     while (modpow(z, (p - 1) / 2, p) != (unsigned long int) p -
        1) { // uint_64 only there for the compiler to stop
        complaining
47         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,

```

```

152         z, NULL);
153     return;
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

```
1  #pragma once
2  #include <stdbool.h>
3
4  typedef struct system {
5      int** m;
6      int* perm;
7      int* sol;
8      bool done;
9      int n1, n2, arb;
10 } system_s;
11
12 typedef system_s* system_t;
13
14 system_t init_gauss(int** v, int n1, int n2);
15 void gaussian_step(system_t s);
16 void free_system(system_t s);
```

## 2.6 system.c

```
1  #include "system.h"
2  #include "vector.h"
3  #include "list_matrix_utils.h"
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <stdbool.h>
7
8  void swap_lines_horz(system_t s, int i, int j){
9      int* temp = s->m[i];
10     s->m[i] = s->m[j];
11     s->m[j] = temp;
12 }
13
14 void swap_lines_vert(system_t s, int i, int j){
15     int temp = s->perm[i];
16     s->perm[i] = s->perm[j];
17     s->perm[j] = temp;
18
19     for(int k = 0; k<s->n1; k++){
20         int temp = s->m[k][i];
21         s->m[k][i] = s->m[k][j];
22         s->m[k][j] = temp;
23     }
24 }
25
26 int find_index(system_t s, int from, int look){
27     for(int i = from; i < s->n1; i++){
28         if(s->m[i][look]){
29             return i;
30         }
31     }
32     return -1;
33 }
34
35 system_t transpose(int** v, int n1, int n2){
36     system_t s = malloc(sizeof(system_s));
37
38     s->m = malloc(n2*sizeof(int*));
39     for(int i = 0; i<n2; i++){
40         s->m[i] = malloc(n1*sizeof(int));
41         for(int j = 0; j<n1; j++){
42             s->m[i][j] = v[j][i];
43         }
44     }
45
46     s->n1 = n2;
47     s->n2 = n1;
48     return s;
49 }
50
```

```

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_arbitrary(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_arbitrary(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

```
1  #pragma once
2  #include <gmp.h>
3  #include <stdbool.h>
4
5  typedef enum {DIXON, QSIEVE, MPQS, PMPQS} TYPE;
6
7  typedef struct input_s {
8      char* output_file;
9      int bound, sieving_interval;
10     mpz_t N;
11     bool quiet;
12     TYPE algorithm;
13     int extra;
14     int delta;
15 } input_t;
16
17 input_t* parse_input(int argc, char** argv);
18 void free_input(input_t* input);
```



## 2.8 parse\_input.c

```
1  #include "parse_input.h"
2  #include <stdlib.h>
3  #include <string.h>
4  #include <gmp.h>
5  #include <stdbool.h>
6
7  input_t* init_input(void){
8      input_t* input = malloc(sizeof(input_t));
9      input->bound = -1;
10     input->output_file = NULL;
11     input->sieving_interval = -1;
12     input->extra = -1;
13     input->quiet = false;
14     input->algorithm = QSIEVE;
15     input->delta = 0;
16     mpz_init_set_ui(input->N, 0);
17     return input;
18 }
19
20 bool valid_int(char* str){
21     int i = 0;
22     char c = str[i];
23     while(c != '\0'){
24         if(c<48 || c>57) return false;
25         c = str[++i];
26     }
27
28     return true;
29 }
30
31 void free_input(input_t* input){
32     if(input->output_file) free(input->output_file);
33     mpz_clear(input->N);
34     free(input);
35 }
36
37 input_t* parse_input(int argc, char** argv){
38     input_t* input = init_input();
39
40     int i = 1;
41     while(i<argc){
42         if(strcmp(argv[i], "-b") == 0 || strcmp(argv[i], "--bound") == 0){
43             i++;
44             if(i<argc){
45                 if(valid_int(argv[i])) input->bound = atoi(argv[i]);
46                 else return NULL;}
47             else return NULL;
48         }
```

```

49
50     else if(strcmp(argv[i], "-s") == 0 || strcmp(argv[i], "--
51         sieving_interval") == 0){
52         i++;
53         if(i<argc){
54             if(valid_int(argv[i])) input->sieving_interval =
55                 atoi(argv[i]);
56             else return NULL;}
57         else return NULL;
58     }
59
60     else if(strcmp(argv[i], "-e") == 0 || strcmp(argv[i], "--
61         extra") == 0){
62         i++;
63         if(i<argc){
64             if(valid_int(argv[i])) input->extra = atoi(argv[i
65             ]);
66             else return NULL;}
67         else return NULL;
68     }
69
70     else if(strcmp(argv[i], "-n") == 0 || strcmp(argv[i], "--
71         number") == 0){
72         i++;
73         if(i<argc){
74             if(valid_int(argv[i])) mpz_set_str(input->N, argv
75             [i], 10);
76             else return NULL;}
77         else return NULL;
78     }
79
80     else if(strcmp(argv[i], "-d") == 0 || strcmp(argv[i], "--
81         delta") == 0){
82         i++;
83         if(i<argc){
84             if(valid_int(argv[i])) input->delta = atoi(argv[i
85             ]);
86             else return NULL;}
87         else return NULL;
88     }
89
90     else if(strcmp(argv[i], "-o") == 0){
91         i++;
92         if(i<argc) input->output_file = argv[i];
93         else return NULL;
94     }
95
96     else if(strcmp(argv[i], "-t") == 0 || strcmp(argv[i], "--
97         type") == 0){
98         i++;
99         if(i<argc) {
100             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

```
1 | #pragma once
2 |
3 | void print_list(int* l, int n);
4 | void print_ll(int** ll, int n1, int n2);
5 | void free_ll(int** m, int n1);
```

## 2.10 list\_matrix\_utils.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void print_list(int* l, int n){
5      for(int i = 0; i<n; i++){
6          printf("%d_", l[i]);
7      }
8      printf("\n");
9  }
10
11 void print_ll(int** ll, int n1, int n2){
12     for(int i = 0; i<n1; i++){
13         print_list(ll[i], n2);
14     }
15     printf("\n");
16 }
17
18 void free_ll(int** m, int n1){
19     for(int i = 0; i<n1; i++){
20         free(m[i]);
21     }
22     free(m);
23 }
```

## 2.11 factorbase.h

```
1  #pragma once
2  #include <gmp.h>
3
4  // bruh
5  bool is_prime(int n);
6
7  // calculates pi(n), the number of prime numbers <= n
8  int pi(int n);
9
10 // returns a list of piB first primes
11 int* primes(int piB, int B);
12
13 /** Reduces the factor base of the algorithm, refer to:
14  * Quadratic sieve factorisation algorithm
15  * Bc. Ondrej Vladyka
16  * Section 2.3.1 (p.16)
17  */
18 int* prime_base(mpz_t n, int* pb_len, int* primes, int piB);
```

## 2.12 factorbase.c

```
1  #include <stdbool.h>
2  #include <gmp.h>
3  #include <stdlib.h>
4
5  bool is_prime(int n) {
6      // Corner cases
7      if (n <= 1)
8          return false;
9      if (n <= 3)
10         return true;
11
12     // This is checked so that we can skip
13     // middle five numbers in below loop
14     if (n % 2 == 0 || n % 3 == 0)
15         return false;
16
17     for (int i = 5; i * i <= n; i = i + 6)
18         if (n % i == 0 || n % (i + 2) == 0)
19             return false;
20
21     return true;
22 }
23
24 int pi(int n) {
25     int k = 0;
26     for (int i = 2; i <= n; i++) {
27         if (is_prime(i)) k++;
28     }
29     return k;
30 }
31
32 int* primes(int piB, int B){
33     int* p = malloc(piB*sizeof(int));
34     int k = 0;
35     for (int i = 2; i <= B; i++) {
36         if (is_prime(i)){
37             p[k] = i;
38             k++;
39         }
40     }
41     return p;
42 }
43
44
45 /* Used for legendre symbol, exists in gmp already
46 bool euler_criterion(mpz_t n, int p){
47     int e = (p-1)/2;
48     mpz_t r, p1;
49     mpz_init(r);
50     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

```
1  #include <gmp.h>
2  #include <stdbool.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  bool vectorize_dixon(mpz_t n, int* v, int pb_len, int* pb){
7      /** Attempts naive factorisation to 'n' with the primes in
8       * the prime base 'pb' and putting the result into 'v',
9       * vector of powers of
10      * the primes in the prime base
11      * If it succeeds, returns true, otherwise, returns false
12      */
13      for(int i = 0; i < pb_len; i++){
14          v[i] = 0;
15      }
16
17      for(int i = 0; i < pb_len && (mpz_cmp_ui(n, 1) != 0); i++){
18          while (mpz_divisible_ui_p(n, pb[i])){
19              v[i]++;
20              mpz_divexact_ui(n, n, pb[i]);
21          }
22      }
23
24      if(mpz_cmp_ui(n, 1) == 0)
25          return true;
26      return false;
27 }
28
29 int** dixon(mpz_t* z, mpz_t N, int pb_len, int* pb, int extra,
30 bool tests){
31     /** Gets pb_len+extra b-smooth realtions definied at:
32     * Quadratic sieve factorisation algorithm
33     * Bc. Ondřej Vladyka
34     * Definition 1.11 (p.5)
35     */
36
37     //ceil(sqrt(n))
38     mpz_t sqrt_N;
39     mpz_init(sqrt_N);
40     mpz_sqrt(sqrt_N, N);
41     mpz_add_ui(sqrt_N, sqrt_N, 1);
42
43     mpz_t zi;
44     mpz_t zi_cpy;
45     mpz_init_set(zi, sqrt_N);
46     mpz_init(zi_cpy);
47
48     int** v = malloc((pb_len+extra)*sizeof(int*));
49
50     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

```
1 | #pragma once
2 | #include <gmp.h>
3 | #include <stdbool.h>
4 |
5 | bool vectorize_qsieve(mpz_t n, int* v, int pb_len, int* pb);
6 | int** qsieve(mpz_t* z, mpz_t N, int pb_len, int* pb, int extra,
   | int s, bool tests);
```

## 4.2 qsieve/qsieve.c

```
1  #include <gmp.h>
2  #include <stdbool.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <assert.h>
6  #include <math.h>
7
8  #include "../system.h"
9  #include "../tonellishanks.h"
10
11 bool vectorize_qsieve(mpz_t n, int* v, int pb_len, int* pb){
12     /** Attempts naive factorisation to 'n' with the primes in
13     * the prime base 'pb' and putting the result into 'v',
14     * vector of powers of
15     * the primes in the prime base
16     * If it succeeds, returns true, otherwise, returns false
17     */
18     for(int i = 0; i < pb_len; i++){
19         v[i] = 0;
20     }
21
22     for(int i = 0; i < pb_len && (mpz_cmp_ui(n, 1) != 0); i++){
23         while (mpz_divisible_ui_p(n, pb[i])){
24             v[i]++;
25             mpz_divexact_ui(n, n, pb[i]);
26         }
27     }
28
29     if(mpz_cmp_ui(n, 1) == 0)
30         return true;
31     return false;
32 }
33
34 float* prime_logs(int* pb, int pb_len){
35     float* plogs = malloc(pb_len*sizeof(float));
36
37     for(int i = 0; i < pb_len; i++){
38         plogs[i] = log2(pb[i]);
39     }
40
41     return plogs;
42 }
43
44 int calculate_threshhold(mpz_t N, mpz_t sqrt_N, int s, int
45     loop_number, int* pb, int pb_len){
46
47     mpz_t qstart;
48     mpz_init_set_ui(qstart, s);
49     mpz_mul_ui(qstart, qstart, loop_number);
50     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,
58             int s, bool quiet){
59     /** Gets pb_len+extra zis that are b-smooth, definied at:
60     * Quadratic sieve factorisation algorithm
61     * Bc. Ondřej Vladyka
62     * Definition 1.11 (p.5)
63     */
64
65     //ceil(sqrt(n))
66     mpz_t sqrt_N;
67     mpz_init(sqrt_N);
68     mpz_sqrt(sqrt_N, N);
69     mpz_add_ui(sqrt_N, sqrt_N, 1);
70
71     mpz_t zi;
72     mpz_init_set(zi, sqrt_N);
73     mpz_t qx;
74     mpz_init(qx);
75
76     int** v = malloc((pb_len+extra)*sizeof(int*));
77     for(int i = 0; i<pb_len+extra; i++){
78         v[i] = malloc(pb_len*sizeof(int*));
79     }
80     float* sinterval = malloc(s*sizeof(float));
81     float* plogs = prime_logs(pb, pb_len);
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  $(\text{sqrt}(N) + x)^2$ 
107         // = n [p]
108         mpz_set_ui(temp, x1[i]);
109         mpz_sub(temp, temp, sqrt_N);
110         mpz_mod_ui(temp, temp, pb[i]);
111
112         x1[i] = mpz_get_ui(temp);
113
114         mpz_set_ui(temp, x2[i]);
115         mpz_sub(temp, temp, sqrt_N);
116         mpz_mod_ui(temp, temp, pb[i]);
117
118         x2[i] = mpz_get_ui(temp);
119     }
120     mpz_clear(temp);
121
122     int loop_number = 0;
123     int relations_found = 0;
124     int tries = 0;
125     while(relations_found < pb_len + extra){
126
127         for(int i = 0; i<s; i++){
128             sinterval[i] = 0;
129         }
130
131         // sieve for 2
132         while(x1[0]<s){
133             sinterval[x1[0]] += plogs[0];
134             x1[0] += pb[0];
135         }
136         x1[0] = x1[0] - s;
137
138         // sieve other primes
139         for(int i = 1; i < pb_len; i++){
140
141             while(x1[i]<s){
142                 sinterval[x1[i]] += plogs[i];
143                 x1[i] += pb[i];
144             }
145
146             while(x2[i]<s){
147
148                 sinterval[x2[i]] += plogs[i];
149                 x2[i] += pb[i];
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,
157         pb, pb_len);
158     //printf("t = %d\n", t);
159
160     bool found;
161     for(int i = 0; i<s && relations_found < pb_len + extra; i
162         ++){
163         if(sinterval[i] > t){
164             tries++;
165
166             // zi = sqrt(n) + x where x = s*loopnumber + i
167             mpz_set_ui(zi, s);
168             mpz_mul_ui(zi, zi, loop_number);
169             mpz_add_ui(zi, zi, i);
170             mpz_add(zi, zi, sqrt_N);
171
172             // qx = zi**2 - N
173             mpz_mul(qx, zi, zi);
174             mpz_sub(qx, qx, N);
175
176             found = vectorize_qsieve(qx, v[relations_found],
177                 pb_len, pb);
178
179             if(found){
180                 mpz_set(z[relations_found], zi);
181                 relations_found++;
182                 found = false;
183                 if(!quiet){
184                     printf("\r");
185                     printf("%.1f%%\u25a0\u25a0%.1f%%", (float)
186                         relations_found/(pb_len+extra)*100, (
187                             float)relations_found/tries*100);
188                     fflush(stdout);
189                 }
190             }
191         }
192     }
193     loop_number++;
194 }
195
196 if(!quiet) printf("\n");
197
198 mpz_clears(sqrt_N, zi, qx, NULL);
199 free(x1);
200 free(x2);
201 free(sinterval);
202 free(plogs);

```



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

## 5 MPQS

### 5.1 mpqs/common\_mpqs.h

```
1 | #pragma once
2 | #include <gmp.h>
3 | #include <stdbool.h>
4 |
5 | int calculate_threshold_mpqs(mpz_t sqrt_N, int s, int* pb, int
   | pb_len, int delta);
6 | float* prime_logs_mpqs(int* pb, int pb_len);
7 | bool vectorize_mpqs(mpz_t n, int* v, int pb_len, int* pb);
8 | bool already_added(mpz_t zi, mpz_t* z, int relations_found);
```

## 5.2 mpqs/common\_mpqs.c

```
1  #include <gmp.h>
2  #include <stdbool.h>
3  #include <math.h>
4  #include <stdlib.h>
5  #include <stdio.h>
6
7  int calculate_threshold_mpqs(mpz_t sqrt_N, int s, int* pb, int
   pb_len, int delta){
8
9      mpz_t qstart;
10     mpz_init_set_ui(qstart, s);
11     mpz_mul(qstart, qstart, sqrt_N);
12
13     int t = mpz_sizeinbase(qstart, 2) - (int) log2(pb[pb_len-1])
        - delta;
14     mpz_clear(qstart);
15     return t;
16 }
17
18 float* prime_logs_mpqs(int* pb, int pb_len){
19     float* plogs = malloc(pb_len*sizeof(float));
20
21     for(int i = 0; i<pb_len; i++){
22         plogs[i] = log2(pb[i]);
23     }
24
25     return plogs;
26 }
27
28 bool vectorize_mpqs(mpz_t n, int* v, int pb_len, int* pb){
29     /** Attempts naive factorisation to 'n' with the primes in
30      * the prime base 'pb' and putting the result into 'v',
31      * vector of powers of
32      * the primes in the prime base
33      * If it succeeds, returns true, otherwise, returns false
34      */
35     for(int i = 0; i<pb_len; i++){
36         v[i] = 0;
37     }
38     if(mpz_sgn(n)<0){
39         v[pb_len] = 1;
40         mpz_neg(n, n);
41     }
42     else{
43         v[pb_len] = 0;
44     }
45
46     for(int i = 0; i<pb_len && (mpz_cmp_ui(n, 1) != 0); i++){
47         while (mpz_divisible_ui_p(n, pb[i])){
48             v[i]++;
49         }
50     }
51 }
```

```

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
17     // everytime
18     mpz_t op1, op2, op3;
19 };
20
21 typedef struct poly_s* poly_t;
22
23 void get_next_poly(poly_t p);
24 poly_t init_poly(mpz_t N, int M);
25 void calc_poly(poly_t p, mpz_t x);
26 poly_t copy_poly(poly_t p);
27 void free_poly(poly_t p);
```

## 5.4 mpqs/polynomial.c

```
1  #include "polynomial.h"
2  #include <gmp.h>
3  #include <stdlib.h>
4  #include <assert.h>
5  #include <stdio.h>
6
7  #include "../tonellishanks.h"
8
9  void calc_coefficients(poly_t p){
10     mpz_mul(p->a, p->d, p->d);
11
12     mpz_t x1, x2;
13     mpz_inits(x1, x2, NULL);
14     tonelli_shanks_mpz(p->N, p->d, x1, x2);
15
16     // getting ready for congruence solve for raising solution
17     mpz_mul_ui(p->op1, x1, 2);
18
19     mpz_mul(p->op2, x1, x1);
20     mpz_sub(p->op2, p->op2, p->N);
21     mpz_divexact(p->op2, p->op2, p->d);
22     mpz_neg(p->op2, p->op2);
23     mpz_mod(p->op2, p->op2, p->d);
24
25     mpz_t g, n, m;
26     mpz_inits(g, n, m, NULL);
27     mpz_gcdext(g, n, m, p->d, p->op1);
28     assert(mpz_cmp_ui(g, 1) == 0);
29     mpz_mul(p->op1, p->op2, m); // t
30     mpz_clears(g, n, m, NULL);
31
32     mpz_set(p->b, p->d);
33     mpz_mul(p->b, p->b, p->op1);
34     mpz_add(p->b, p->b, x1);
35
36     mpz_mul(p->op1, p->b, p->b);
37     assert(mpz_congruent_p(p->op1, p->N, p->a) != 0);
38
39     mpz_sub(p->c, p->op1, p->N);
40     mpz_divexact(p->c, p->c, p->a);
41
42     mpz_clears(x1, x2, NULL);
43 }
44
45 void get_next_poly(poly_t p){
46     mpz_nextprime(p->d, p->d);
47     while(mpz_legendre(p->N, p->d) != 1){
48         mpz_nextprime(p->d, p->d);
49     }
50     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

```
1  #pragma once
2
3  #include <gmp.h>
4  #include <stdbool.h>
5
6  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

```
1  #include <gmp.h>
2  #include <stdbool.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <assert.h>
6  #include <math.h>
7  #include <time.h>
8
9  #include "polynomial.h"
10 #include "common_mpqs.h"
11 #include "../system.h"
12 #include "../tonellishanks.h"
13
14 int** mpqs(mpz_t* z, mpz_t* d, mpz_t N, int pb_len, int* pb, int
    extra, int s, int delta, bool quiet){
15     /** Gets pb_len+extra zis that are b-smooth, definied at:
16     * Quadratic sieve factorisation algorithm
17     * Bc. Ondrej Vladyka
18     * Definition 1.11 (p.5)
19     */
20
21     //ceil(sqrt(n))
22     mpz_t sqrt_N;
23     mpz_init(sqrt_N);
24     mpz_sqrt(sqrt_N, N);
25     mpz_add_ui(sqrt_N, sqrt_N, 1);
26
27     mpz_t x;
28     mpz_init(x);
29     poly_t Q = init_poly(N, s);
30
31     int** v = malloc((pb_len+extra)*sizeof(int*));
32     for(int i = 0; i<pb_len+extra; i++){
33         v[i] = malloc((pb_len+1)*sizeof(int*)); // +1 for -1
34     }
35     float* sinterval = malloc(2*s*sizeof(float));
36     float* plogs = prime_logs_mpqs(pb, pb_len);
37     int t = calculate_threshhold_mpqs(sqrt_N, s, pb, pb_len,
        delta);
38
39
40     // TESTS
41     mpz_t temp;
42     mpz_init(temp);
43     // END TESTS
44
45
46     int* r = malloc(pb_len*sizeof(int));
47     int* x1 = malloc(pb_len*sizeof(int));
48     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
77                     the current implementation of MPQS\n");
78                 exit(1);
79             }
80
81             mpz_set_ui(temp, r[i]);
82             mpz_sub(temp, temp, Q->b);
83             mpz_mul(temp, temp, m);
84             mpz_mod(temp, temp, pi);
85
86             x1[i] = mpz_get_ui(temp);
87
88             //calc_poly(Q, temp);
89             //assert(mpz_divisible_ui_p(Q->qx, pb[i]) != 0);
90
91             mpz_set_ui(temp, pb[i]);
92             mpz_sub_ui(temp, temp, r[i]);
93             mpz_sub(temp, temp, Q->b);
94             mpz_mul(temp, temp, m);
95             mpz_mod(temp, temp, pi);
96
97             x2[i] = mpz_get_ui(temp);
98
99             //calc_poly(Q, temp);
100            //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;
147     i++){
148     if(sinterval[i] > t){
149         tries++;
150         mpz_set_si(x, -s);
151         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[
155                 relations_found], pb_len, pb);
156             if(found){
157                 mpz_set(z[relations_found], Q->zi);
158                 mpz_set(d[relations_found], Q->d);
159                 relations_found++;
160                 update_time = true;
161                 found = false;
162                 if(!quiet){
163                     printf("\r");
164                     printf("%.1f%%|_%.1f%%", (float)
165                         relations_found/(pb_len+extra)
166                         *100, (float)relations_found/tries
167                         *100);
168                     fflush(stdout);
169                 }
170             }
171         }
172     }
173 }
174
175 if(update_time && !quiet) printf("_(~%.0fs_left)_"
176     , (double)(clock() - start)/CLOCKS_PER_SEC/
177     relations_found*((pb_len+extra - relations_found)));
178 get_next_poly(Q);
179 }
180
181 if(!quiet) printf("\n");
182 mpz_clears(sqrt_N, temp, g, m, n, pi, x, NULL);
183 free(x1);
184 free(x2);
185 free(r);
186 free(sinterval);
187 free(plogs);
188 free_poly(Q);
189
190 return v;
191 }

```

## 6 MPQS parallélisé

### 6.1 mpqs/parallel\_mpqs.h

```
1  #pragma once
2  #include <gmp.h>
3  #include "polynomial.h"
4  #include <sys/time.h>
5  #include <stdint.h>
6
7  struct sieve_arg_s {
8      // used for sieving
9      int* pb;
10     int pb_len;
11     int extra;
12     int* r;
13     float* plogs;
14     int s;
15     int t;
16     int* relations_found;
17     int** v;
18     bool quiet;
19     mpz_t* z;
20     mpz_t* d;
21     poly_t Qinit;
22
23     // used to print progress and predicted time left
24     struct timeval begin;
25     uint_fast64_t* tries;
26
27     // used to constantly have a certain number of threads
28     // running
29     int thread_id;
30     bool* threads_running;
31 };
32
33 typedef struct sieve_arg_s sieve_arg_t;
34
35 bool already_added(mpz_t zi, mpz_t* z, int relations_found);
36 void* sieve_100_polys (void* args);
37 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

```
1  #include <gmp.h>
2  #include <stdbool.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <assert.h>
6  #include <math.h>
7  #include <time.h>
8  #include <pthread.h>
9  #include <sys/time.h>
10
11 #include "polynomial.h"
12 #include "common_mpqs.h"
13 #include "parallel_mpqs.h"
14 #include "../system.h"
15 #include "../tonellishanks.h"
16
17 pthread_mutex_t mutex;
18
19
20 void* sieve_100_polys (void* args){
21     sieve_arg_t* arg = (sieve_arg_t*) args;
22
23     poly_t Q = copy_poly(arg->Qinit);
24
25     mpz_t temp, g, m, n, pi, x;
26     mpz_inits(temp, g, m, n, pi, x, NULL);
27     float* sinterval = malloc(2*arg->s*sizeof(float));
28     int* x1 = malloc(arg->pb_len*sizeof(int));
29     int* x2 = malloc(arg->pb_len*sizeof(int));
30
31     for(int i = 0; i<100 && *(arg->relations_found) < arg->pb_len
32         + arg->extra; i++){
33         get_next_poly(Q);
34
35         //get sol for 2
36         mpz_set_ui(temp, 0);
37         calc_poly(Q, temp);
38         x1[0] = 0;
39         if(mpz_divisible_ui_p(Q->qx, 2) == 0) x1[0] = 1;
40
41         //get sol for others
42         for(int i = 1; i<arg->pb_len; i++){
43             mpz_set_ui(pi, arg->pb[i]);
44             mpz_gcdext(g, m, n, Q->a, pi);
45             if(mpz_cmp_ui(g, 1) != 0){
46                 fprintf(stderr, "ERROR: Number is too small for
47                     the current implementation of MPQS\n");
48                 exit(1);
49             }
50         }
51     }
```

```

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) <
114     arg->pb_len + arg->extra; i++){
115     if(sinterval[i] > arg->t){
116         *(arg->tries) += 1;
117         mpz_set_si(x, -arg->s);
118         mpz_add_ui(x, x, i);
119         calc_poly(Q, x);
120
121         if(!already_added(Q->zi, arg->z, *(arg->
122             relations_found))){
123             found = vectorize_mpqs(Q->qx, arg->v[*(arg->
124                 relations_found)], arg->pb_len, arg->pb);
125             if(found){
126                 mpz_set(arg->z[*(arg->relations_found)],
127                     Q->zi);
128                 mpz_set(arg->d[*(arg->relations_found)],
129                     Q->d);
130                 *(arg->relations_found) += 1;
131                 found = false;
132                 update_time = true;
133                 if(!arg->quiet){
134                     printf("\r");
135                     printf("%.1f%%\u00a0|%.1f%%", (float)(*(
136                         arg->relations_found))/(arg->
137                         pb_len+arg->extra)*100, (float)(*(
138                         arg->relations_found))/(*(arg->
139                         tries))*100);
140                     fflush(stdout);
141                 }
142             }
143         }
144     }
145 }
146
147 struct timeval current;
148 gettimeofday(&current, 0);
149 long seconds = current.tv_sec - arg->begin.tv_sec;
150 long microseconds = current.tv_usec - arg->begin.tv_usec;
151 double elapsed = seconds + microseconds*1e-6;
152 if(update_time && !arg->quiet) printf("\u00a0(~%.0fs\u00a0left)\u00a0\u00a0\u00a0\u00a0

```

```

        "      ", 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. Ondrej 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,
222                     args+i);
223             }
224             for(int i = 0; i<100; i++){
225                 get_next_poly(Q);
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;

```

