September 21, 2017 at 09:01

1. diophant – Solving Diophantine Linear Systems. The library `diophant` enables the solution of Diophantine Linear Systems based on LLL reduction. The user has to provide an integer matrix A, a right hand side vector b, optionally a vector u of upper bounds on the solution vector x. `diophant` computes all integer solutions x, such that

$$A \cdot x = b, \text{ where } 0 \le x \le u.$$

An example is the following 3×13-system, to be solved with a 0/1-vector:

$$\begin{pmatrix} 5 & 5 & 5 & 5 & 5 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 4 & 8 & 0 & 0 & 0 & 0 & 4 & 4 & 4 & 0 & 0 \\ 2 & 1 & 4 & 3 & 2 & 1 & 1 & 1 & 2 & 5 & 2 & 3 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 12 \\ 12 \\ 12 \end{pmatrix}$$

One 0/1-solution is

$$x = (1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0)^\top.$$

2. Initial definitions. The LLL and BKZ algorithms can be controlled by the following declarations.

```
#define BLAS 1
#define USE_SSE 0
#define DEEPINSERT 1
#define DEEPINSERT_CONST 100
#define VERBOSE 1
#define GIVENS 1
#define LASTLINESFACTOR "1000000"    /* "100000000" */
#define EPSILON 0.00001    /* 0.0001 */
#define LLLCONST_LOW 0.75    /* 0.75 */
#define LLLCONST_HIGH 0.90    /* 0.99 */
#define LLLCONST_HIGHER 0.999
#define SQRT sqrt
#define DOUBLE double
#define COEFF struct coe
  format mpz_t long
  format DOUBLE double
  format COEFF int
```

⟨ include header files 5 ⟩;
⟨ definition of the lattice data structures 6 ⟩;
⟨ global variables 7 ⟩;
⟨ inline functions 10 ⟩;
⟨ basic subroutines 28 ⟩;
⟨ lattice basis reduction algorithms 47 ⟩;

3. The main routine `diophant()`:

```
long diophant(mpz_t **a_input, mpz_t *b_input, mpz_t *upperbounds_input, int no_columns, int
    no_rows, mpz_t factor_input, mpz_t norm_input, mpz_t scalelastlinefactor, int silent, int
    iterate, int iterate_no, int bkz_beta_input, int bkz_p_input, long stop_after_sol_input, long
    stop_after_loops_input, int free_RHS_input, int *org_col_input, int no_org_col_input, int cut_after, int
    nboundedvars, FILE *solfile)
{
  int i, j;
  DOUBLE lD, lDnew;
  COEFF *swap_vec;
  ⟨ initialize some globals 11 ⟩;
#if BLAS    /* goto_set_num_threads(1); */
#endif    /* In case, a time limit as been set by -time the execution is stopped and the number of
      solutions is printed */
  ⟨ set the lattice dimension 13 ⟩;
  ⟨ allocate memory 14 ⟩;
  ⟨ read the system 15 ⟩;
  ⟨ handle upper bounds 16 ⟩;
  ⟨ handle preselected columns 17 ⟩;
  ⟨ append the other parts of lattice 18 ⟩;
  ⟨ open solution file 117 ⟩;
#if 0
  printf("Before scaling\n");
  print_lattice();
#endif
  ⟨ scale lattice 19 ⟩;
#if 0
  printf("After scaling\n");
  print_lattice();
#endif
#if 1
#if 0
  print_NTL_lattice();    /* Version for the NTL output */
  return 0;
#endif
  ⟨ permute lattice columns 20 ⟩;
#if 0
  printf("After permute\n");
  print_lattice();
#endif
  shufflelattice();
  ⟨ first reduction 21 ⟩;
#if 0
  printf("After first reduction\n");
  print_lattice();
#endif
  ⟨ cut the lattice 23 ⟩;
#if 0
  printf("After cutting\n");
  print_lattice();
#endif
```

```
#if 1
    shufflelattice( );
    ⟨ second reduction 22 ⟩;
#endif
#if 0
    printf("After␣second␣reduction\n");
    print_lattice( );
#endif
#if 1
    ⟨ scale last rows 24 ⟩;
    ⟨ third reduction 26 ⟩;
    ⟨ undo scaling of last rows 25 ⟩;
#endif
#else
    read_NTL_lattice( );
#endif
#if 0
    printf("Before␣enumeration\n");      /* print_NTL_lattice( ); */
      /* Version for the NTL output */
    print_lattice( );
#endif
    ⟨ explicit enumeration 27 ⟩;
    ⟨ close solution file 118 ⟩;
    ⟨ free multiprecision memory 12 ⟩;
    return nosolutions;
  }
```

4. The header file `diophant.h`. It is needed if one wants to include *diophant*( ) in his program.

```
⟨ diophant.h 4 ⟩ ≡
#ifndef _DIOPHANT_H
#define _DIOPHANT_H
#include <gmp.h>
  extern long diophant(mpz_t **a_input, mpz_t *b_input, mpz_t *upperbounds_input, int
      no_columns, int no_rows, mpz_t factor_input, mpz_t norm_input, mpz_t scalelastlinefactor, int
      silent, int iterate, int iterate_no, int bkz_beta_input, int bkz_p_input, long stop_after_sol_input, long
      stop_after_loops_input, int free_RHS_input, int *org_col_input, int no_org_col_input, int cut_after, int
      nboundedvars, FILE *solfile);
  extern void stopProgram( );
  extern long nosolutions;
#endif
```

See also section 88.

5. The following header files are included.

⟨ include header files 5 ⟩ ≡
```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#include <gmp.h>
#include "diophant.h"
#if USE_SSE      /* Intrinsic SSE */
  # include < pmmintrin.h >
#endif
#if BLAS
#include "OpenBLASsub/common.h"
#include "OpenBLASsub/cblas.h"
#endif
```
This code is used in section 2.

6. The data structure of the lattice:

⟨ definition of the lattice data structures 6 ⟩ ≡
```
  struct coe {
    mpz_t c;
    int p;
  };
```
This code is used in section 2.

7. The global variables. We start with the variables which control the scaling of the lattice. *max_norm* resembles the λ in design problems.

⟨ global variables 7 ⟩ ≡
```
  mpz_t matrix_factor;
  mpz_t max_norm;
  mpz_t max_norm_initial;
  mpz_t max_up;
  mpz_t dummy;
  long nom, denom;
  mpz_t lastlines_factor;
  mpz_t snd_q, snd_r, snd_s;
```
See also sections 8, 9, 41, and 119.

This code is used in section 2.

8. The variables which define the lattice and its dimensions.

⟨ global variables 7 ⟩ +≡
```
int system_rows, system_columns;
int lattice_rows, lattice_columns;
COEFF **lattice;
int free_RHS;
int iszeroone;
mpz_t *upperbounds;
mpz_t upperbounds_max;
mpz_t upfac;
```

9. Other variables.

⟨ global variables 7 ⟩ +≡
```
int *original_columns;
int no_original_columns;
int cut_after_coeff;
long stop_after_solutions;
long stop_after_loops;
long nosolutions;
int iterate;
int no_iterates;
int bkz_beta, bkz_p;
int SILENT;
int nboundvars;
```

10. The access to the lattice is done via inline functions. The rounding function is defined as

```
#define ROUND(r) ceil(r - 0.5)
```

⟨ inline functions 10 ⟩ ≡
```
#define put_to(i, j, val) mpz_set(lattice[i][j + 1].c, val)
#define smult_lattice(i, j, factor) mpz_mul(lattice[i][j + 1].c, lattice[i][j + 1].c, factor)
#define get_entry(i, j) lattice[i][j + 1].c
```

This code is used in section 2.

11. Read the parameters of `diophant` and initialize some global variables.

```
⟨ initialize some globals 11 ⟩ ≡
  mpz_init_set(matrix_factor, factor_input);
  mpz_init_set(max_norm, norm_input);
  mpz_init(lastlines_factor);
  mpz_init(upfac);
  mpz_init(snd_q);
  mpz_init(snd_r);
  mpz_init(snd_s);
  if (iterate) {
    no_iterates = iterate_no;
  }
  else {
    bkz_beta = bkz_beta_input;
    bkz_p = bkz_p_input;
  }
  SILENT = silent;
  stop_after_solutions = stop_after_sol_input;
  stop_after_loops = stop_after_loops_input;
  free_RHS = free_RHS_input;
  nom = 1;
  denom = 2;
  system_rows = no_rows;
  system_columns = no_columns;
  nboundvars = nboundedvars;
```

See also section 42.

This code is used in section 3.

12.

```
⟨ free multiprecision memory 12 ⟩ ≡
  mpz_clear(matrix_factor);
  mpz_clear(max_norm);
  mpz_clear(lastlines_factor);
  mpz_clear(upfac);
  mpz_clear(max_norm_initial);
  mpz_clear(max_up);
  mpz_clear(soltest_u);
  mpz_clear(soltest_s);
  mpz_clear(soltest_upfac);
  mpz_clear(upperbounds_max);
  for (j = 0; j < lattice_columns; j++) {
    for (i = 0; i ≤ lattice_rows; i++) mpz_clear(lattice[j][i].c);
  }
  free(lattice);
  if (upperbounds ≠ Λ) {
    for (i = 0; i < system_columns; i++) mpz_clear(upperbounds[i]);
    free(upperbounds);
  }
```

This code is used in section 3.

13. The lattice has two more columns than the original system. The number of rows is the number of columns plus number of rows plus one of the original system. If the right hand side is free (e.g. for design problems) there is an additional column and an additional row.

```
⟨ set the lattice dimension 13 ⟩ ≡
  lattice_rows = system_rows + system_columns + 1;
  lattice_columns = system_columns + 2;
  if (free_RHS) {
    lattice_rows++;
    lattice_columns++;
  }
  else {
#ifndef NO_OUTPUT
    fprintf(stderr, "The␣RHS␣is␣fixed␣!\n"); fflush(stderr);
#endif
  }
  cut_after_coeff = cut_after;
```

This code is used in section 3.

14. Allocate memory for the lattice array and fill it with zero.

```
⟨ allocate memory 14 ⟩ ≡
  lattice = (COEFF **) calloc(lattice_columns, sizeof(COEFF *));
  for (j = 0; j < lattice_columns; j++) {
    lattice[j] = (COEFF *) calloc(lattice_rows + 1, sizeof(COEFF));
    for (i = 0; i ≤ lattice_rows; i++) mpz_init(lattice[j][i].c);
  }
```

This code is used in section 3.

15. The input matrix and the right hand side vector are copied row by row into the lattice.

```
⟨ read the system 15 ⟩ ≡
  for (j = 0; j < system_rows; j++) {
    for (i = 0; i < system_columns; i++) {
      mpz_mul(lattice[i][j + 1].c, a_input[j][i], matrix_factor);
    }
    mpz_mul(lattice[system_columns][j + 1].c, b_input[j], matrix_factor);
  }
```

This code is used in section 3.

16. The upper bounds are copied. If the input vector points to Λ, a 0/1 system is assumed. The variable *upperbounds_max* contains the least common multiple of the upper bounds, let's call it $u_{\max}$. Starting from $u_{\max} = 1$ it is computed with $0 \le i < n$:

$$u_{\max} \leftarrow u_{\max} \cdot \frac{u_i}{\gcd(u_{\max}, u_i)}.$$

It is needed for an appropiate scaling of the lattice, see section ⟨ scale lattice 19 ⟩. Further, this determines if we have a 0/1 system, i. e. if *upperbounds_max* = 1.

```
⟨ handle upper bounds 16 ⟩ ≡
  mpz_init_set_si(upperbounds_max, 1);
  iszeroone = 1;
  if (upperbounds_input ≡ Λ) {
#ifndef NO_OUTPUT
    printf("No␣upper␣bounds:␣0/1␣variables␣are␣assumed␣\n"); fflush(stdout);
#endif
  }
  else {
    upperbounds = (mpz_t *) calloc(system_columns, sizeof(mpz_t));
    for (i = 0; i < system_columns; i++) mpz_init_set_si(upperbounds[i], 1);
    for (i = 0; i < nboundvars      /* system_columns */
    ; i++) {
      mpz_set(upperbounds[i], upperbounds_input[i]);
      if (mpz_sgn(upperbounds[i]) ≠ 0) {
        mpz_lcm(upperbounds_max, upperbounds_max, upperbounds[i]);
      }
    }
    if (mpz_cmp_si(upperbounds_max, 1) > 0) iszeroone = 0;
#ifndef NO_OUTPUT
    fprintf(stderr, "upper␣bounds␣found.␣Max=");
    mpz_out_str(stderr, 10, upperbounds_max);
    fprintf(stderr, "\n"); fflush(stderr);
#endif
  }
```

This code is used in section 3.

17. Handle the original columns. This is used for preselected columns. If the array points to Λ, we fill the array completely with 1's. This is of course only useful in case of 0/1 problems. The non-0/1 case has still to be handled.

```
⟨ handle preselected columns 17 ⟩ ≡
  if (org_col_input ≠ Λ) no_original_columns = no_org_col_input;
  else no_original_columns = system_columns;
  original_columns = (int *) calloc(no_original_columns, sizeof(int));
  if (org_col_input ≠ Λ)
    for (i = 0; i < no_original_columns; i++) original_columns[i] = org_col_input[i];
  else {
    for (i = 0; i < no_original_columns; i++) original_columns[i] = 1;
#ifndef NO_OUTPUT
    printf("No␣preselected␣columns␣\n"); fflush(stdout);
#endif
  }
```

This code is used in section 3.

18. The matrix of unity, the last columns and rows are appended. The diagonal (unity) matrix is set to *denom* ∗ *max_norm* which is $2 \cdot \lambda$ in design problems and 2 in problems with fixed right hand side. The second last column (the column corresponding to the right hand side) is set to *nom* ∗ *max_norm* which is λ in design problems and 1 in problems with fixed right hand side.

```
⟨ append the other parts of lattice 18 ⟩ ≡
  for (j = system_rows; j < lattice_rows; j++) {
    mpz_mul_si(lattice[j − system_rows][j + 1].c, max_norm, denom);
    mpz_mul_si(lattice[lattice_columns − 2][j + 1].c, max_norm, nom);
  }
  mpz_set(lattice[system_columns + free_RHS][lattice_rows].c, max_norm);
  if (free_RHS) {
    mpz_set_si(lattice[system_columns][lattice_rows − 1].c, 1);
    mpz_set_si(lattice[system_columns + 1][lattice_rows − 1].c, 0);
  }
  mpz_set(lattice[system_columns + free_RHS][lattice_rows].c, max_norm);
  for (i = 0; i < lattice_columns − 1; i++) coeffinit(lattice[i], lattice_rows);
```

This code is used in section 3.

19. The lower parts of the lattice are multiplied by factors stemming from the upper bounds. This is only necessary if we have non-0/1 bounds. Each row in the lower part of the lattice corresponds to a variable. We multiply the diagonal entry in row j (of the lower part, therefore in the whole system it is row $j+$*system_rows*) by $\frac{u_{\max}}{u_j}$. The right hand side column entry in row j is multiplied by $u_{\max}$.

```
⟨ scale lattice 19 ⟩ ≡
  mpz_init_set(max_norm_initial, max_norm);
  mpz_init_set_si(max_up, 1);
  if (¬iszeroone) {
    for (j = 0; j < nboundvars      /* system_columns */
    ; j++) {
      if (mpz_sgn(upperbounds[j]) ≠ 0) {
        mpz_divexact(upfac, upperbounds_max, upperbounds[j]);
      }
      else {
        mpz_mul(upfac, upperbounds_max, upperbounds_max);
        mpz_mul_si(upfac, upfac, 10000);
      }
      smult_lattice(j, j + system_rows, upfac);
      smult_lattice(system_columns + free_RHS, j + system_rows, upperbounds_max);
    }
    mpz_set(max_up, upperbounds_max);
    mpz_mul(max_norm, max_norm, max_up);
    if (free_RHS) smult_lattice(system_columns, lattice_rows − 2, max_up);
    smult_lattice(system_columns + free_RHS, lattice_rows − 1, max_up);
  }
```

This code is cited in section 16.

This code is used in section 3.

20. The lattice columns are cyclically shifted: the second last column becomes the first column, the first column becomes the second, ...

```
⟨ permute lattice columns 20 ⟩ ≡
  swap_vec = lattice[lattice_columns − 2];
  for (i = lattice_columns − 2; i > 0; i−−) lattice[i] = lattice[i − 1];
  lattice[0] = swap_vec;
```

This code is used in section 3.

21. The first reduction: Pure LLL to compute the kernel.

```
⟨ first reduction 21 ⟩ ≡
  mpz_set_ui(lastlines_factor, 1);
#ifndef NO_OUTPUT
  printf("\n"); fflush(stdout);
#endif
  lll(lattice, lattice_columns − 1, lattice_rows, LLLCONST_LOW);
```

This code is used in section 3.

22. The second reduction: Pure LLL with higher quality. This is done in order to find solutions by chance.

```
⟨ second reduction 22 ⟩ ≡
  mpz_set_ui(lastlines_factor, 1);
  lll(lattice, lattice_columns − 1, lattice_rows, LLLCONST_HIGH);
#ifndef NO_OUTPUT
  printf("Second␣reduction␣successful\n"); fflush(stdout);
#endif
```

This code is used in section 3.

23. Delete the unnecessary parts of the reduced lattice, multiply the last row(s) by an appropiate factor, such that there is only one entry in these rows after the next reduction. This is done after first lattice basis reduction. If it was not successful in computing a complete basis of the kernel we leave *diophant*( ). The function *cutlattice*( ) is defined in section ⟨ cut lattice 36 ⟩.

```
⟨ cut the lattice 23 ⟩ ≡
  if (cutlattice( )) {
#ifndef NO_OUTPUT
    printf("First␣reduction␣successful\n"); fflush(stdout);
#endif
  }
  else {
#ifndef NO_OUTPUT
    printf("First␣reduction␣not␣successful\n"); fflush(stdout);
#endif
    return 0;
  }
  for (j = 0; j < lattice_columns − 1; j++) solutiontest(j);
```

This code is used in section 3.

24. The last row or in case of a free right hand side the two last rows are multiplied by a large factor given ba the commandline option $-scalelastline\ *$. Then it easier to search with the exhaustive enumeration for nonhomogeneous solutions because the columns corresponding to the right hand side (and the additional column in case of a free right hand side) appear only once in the basis vectors. bf Attention: It only works for fixed right hand side

```
⟨ scale last rows 24 ⟩ ≡        /* mpz_set_str(lastlines_factor, LASTLINESFACTOR, 10); */
  mpz_set(lastlines_factor, scalelastlinefactor);
  for (i = 0; i < lattice_columns; i++)
    mpz_mul(lattice[i][lattice_rows].c, lattice[i][lattice_rows].c, lastlines_factor);
  if (free_RHS)
    for (i = 0; i < lattice_columns; i++)
      mpz_mul(lattice[i][lattice_rows − 1].c, lattice[i][lattice_rows − 1].c, lastlines_factor);
#if 0
  for (i = 0; i < lattice_columns; i++) {
    for (j = 0; j < 40; j++) mpz_mul_ui(lattice[i][j + 1].c, lattice[i][j + 1].c, 9);
  }
#endif
```

This code is cited in section 25.

This code is used in section 3.

25. Undo the scaling of section ⟨ scale last rows 24 ⟩ after the third reduction.

```
⟨ undo scaling of last rows 25 ⟩ ≡
  for (i = 0; i < lattice_columns; i++)
    mpz_divexact(lattice[i][lattice_rows].c, lattice[i][lattice_rows].c, lastlines_factor);
  if (free_RHS)
    for (i = 0; i < lattice_columns; i++)
      mpz_divexact(lattice[i][lattice_rows − 1].c, lattice[i][lattice_rows − 1].c, lastlines_factor);
#if 0
  for (i = 0; i < lattice_columns; i++) {
    for (j = 0; j < 40; j++) mpz_divexact_ui(lattice[i][j + 1].c, lattice[i][j + 1].c, 9);
  }
#endif
```

This code is used in section 3.

26. The third reduction is done with blockwise Korkine Zolotareff reduction. The last column of *lattice* is a zero vector.

```
⟨ third reduction 26 ⟩ ≡
#ifndef NO_OUTPUT
  printf("\n"); fflush(stdout);
#endif
  if (iterate) {
    iteratedlll(lattice, lattice_columns − 1, lattice_rows, no_iterates, LLLCONST_HIGH);
  }
  else {
    shufflelattice( );
    lDnew = bkz(lattice, lattice_columns, lattice_rows, LLLCONST_HIGHER, 40, bkz_p);
    i = 0;
    do {
      lD = lDnew;
      shufflelattice( );
      lDnew = bkz(lattice, lattice_columns, lattice_rows, LLLCONST_HIGH, bkz_beta, bkz_p);
      printf("%0.3lf␣%0.3lf␣%0.3lf\n", lD, lDnew, lD − lDnew);
      i++;
    } while (i < 1 ∧ fabs(lDnew − lD) > 0.01);
  }
#ifndef NO_OUTPUT
  printf("Third␣reduction␣successful\n"); fflush(stdout);
#endif
```

This code is used in section 3.

27. The exhaustive enumeration of all solutions.

```
⟨ explicit enumeration 27 ⟩ ≡
#ifndef NO_OUTPUT
  printf("\n"); fflush(stdout);
#endif
  nosolutions = explicit_enumeration(lattice, lattice_columns − 1, lattice_rows);
```

This code is used in section 3.

28. Subroutines for diophant.

⟨ basic subroutines 28 ⟩ ≡
⟨ stop program 135 ⟩⟨ debug print 29 ⟩;
⟨ print the lattice 30 ⟩;
⟨ shuffle lattice columns 37 ⟩;
⟨ print NTL lattice 31 ⟩;
⟨ gcd 34 ⟩;
⟨ Initialize the lattice 35 ⟩;
⟨ read lattice written by NTL 32 ⟩;
⟨ cut lattice 36 ⟩;
⟨ solutiontest 40 ⟩;

This code is used in section 2.

29. ⟨ debug print 29 ⟩ ≡

```
void debug_print(char *m, int l)
{
#ifndef NO_OUTPUT
  if (VERBOSE ≥ l) {
    printf("debug>> %s\n", m); fflush(stdout);
  }
#endif
  return;
}
```

This code is used in section 28.

30. Print the lattice for debugging purposes. The lattice is printed columnwise (i.e. in transposed form).

```
⟨ print the lattice 30 ⟩ ≡
#if 1
  void print_lattice( )
  {
    int i, j;
    for (i = 0; i < lattice_columns; i++) {
      for (j = 0; j < lattice_rows; j++) {
        mpz_out_str(Λ, 10, get_entry(i, j));
        printf("␣");
      }
      printf("\n");
    }
    printf("\n"); fflush(stdout);
    return;
  }
#else
  void print_lattice( )
  {
    int i, j;
    for (j = 0; j < lattice_rows; j++) {
      for (i = 0; i < lattice_columns; i++) {
        mpz_out_str(Λ, 10, get_entry(i, j));
        printf("␣");
      }
      printf("\n");
    }
    printf("\n"); fflush(stdout);
    return;
  }
#endif
```

This code is used in section 28.

31. Print the lattice in NTL format. The lattice is printed columnwise (i.e. in transposed form).

```
⟨ print NTL lattice 31 ⟩ ≡
  void print_NTL_lattice( )
  {
    int i, j;
#if 1
    fprintf(stderr, "%d␣%d\n", lattice_columns, lattice_rows);
    printf("%d\n", system_rows);
    printf("\n[");
    for (i = 0; i < lattice_columns − 1; i++) {      /* Don't print the last vector (only zeroes). */
      printf("[");
      for (j = 0; j < lattice_rows; j++) {
        mpz_out_str(Λ, 10, get_entry(i, j));
        printf("␣");
      }
      printf("]");
      printf("\n");
    }
    printf("]\n"); fflush(stdout);
#if 1
    printf("\n");
    printf("%d␣", lattice_columns − 2);
    mpz_out_str(Λ, 10, upperbounds_max);
    printf("\n\n[");
    for (i = 0; i < lattice_columns − 2; i++) {
      mpz_out_str(Λ, 10, upperbounds[i]);
      printf("␣");
    }
    printf("]\n"); fflush(stdout);
#endif
#endif
    return;
  }
```

This code is used in section 28.

32. Read lattice written by NTL. First line: number of lattice vectors, lattice length.
Then follows the lattice, each row is a lattice vector.
Number of bounds, Maximum bound
Vector of bounds.

```
⟨ read lattice written by NTL 32 ⟩ ≡
  void read_NTL_lattice( )
  {
    int i, j, cols, rows, nbounds;
    scanf("%d␣%d\n", &rows, &cols);
    for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++) {
        mpz_inp_str(lattice[i][j + 1].c, Λ, 10);
      }
    }
    for (j = 0; j < cols; j++) {
      mpz_set_ui(lattice[rows][j + 1].c, 0);
    }
    scanf("%d", &nbounds);
    mpz_inp_str(upperbounds_max, Λ, 10);
    for (j = 0; j < nbounds; j++) {
      mpz_inp_str(upperbounds[j], Λ, 10);
    }
    lattice_columns = rows + 1;
    lattice_rows = cols;
    for (i = 0; i < lattice_columns; i++) coeffinit(lattice[j], lattice_rows);
    return;
  }
```

This code is used in section 28.

33. Write uni-modular transformation to file.

```
⟨ write tranformation matrix 33 ⟩ ≡
  void write_transform( )
  {
    FILE *f;
    int i, j, n;
    f = fopen("sdb_transform.txt", "w");
    n = lattice_columns − 1;
    fwrite("%d\n", n);
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        mpz_out_str(f, 10, get_entry(i + system_rows, j));
        fprintf(f, "␣");
      }
      fprintf(f, "\n");
    }
    fclose(f);
  }
```

34. Compute GCD in case of single precision arithmetics. This piece of code was written by Brendan McKay.

```
⟨ gcd 34 ⟩ ≡
  long gcd(long n1, long n2)
  {
    long a, b, c;
    if (n1 > n2) {
      a = n1; b = n2;
    }
    else {
      a = n2; b = n1;
    }
    while ((c = a % b) > 0) {
      a = b; b = c;
    }
    return b;
  }
```

This code is used in section 28.

35. Construct the information of the sparse structure of the lattice.

```
⟨ Initialize the lattice 35 ⟩ ≡
  void coeffinit(COEFF *v, int z)
  {
    short r = 0;
    short i;
    for (i = z; i ≥ 0; i--) {
      v[i].p = r;
      if (mpz_sgn(v[i].c) ≠ 0) r = i;
    }
    return;
  }
```

This code is used in section 28.

36. Delete the unnecessary columns and rows of the lattice after the first reduction, i.e. only the basis of the kernel remains. Returns 0 if it is not possible to achieve a solution. Finally, the unnecessary rows are deleted and the sparse structure of the lattice is updated.

```
⟨ cut lattice 36 ⟩ ≡
  int cutlattice( )
  {
    int j, i, flag;
    ⟨ delete unnecessary columns 38 ⟩;
    ⟨ test for right hand side columns 39 ⟩;
    for (j = 0; j < lattice_columns; j++) {      /* Now the rows are deleted. */
      if (nboundvars ≡ 0) {
        for (i = system_rows; i < lattice_rows; i++) put_to(j, i − system_rows, get_entry(j, i));
      }
      else {
        for (i = system_rows; i < system_rows + nboundvars; i++)
          put_to(j, i − system_rows, get_entry(j, i));
        for (i = system_rows + system_columns; i < lattice_rows; i++)
          put_to(j, i − system_rows − system_columns + nboundvars, get_entry(j, i));
      }
    }
    lattice_rows −= system_rows;
    lattice_rows −= (system_columns − nboundvars);
    for (j = 0; j < lattice_columns; j++) coeffinit(lattice[j], lattice_rows);
    return 1;
  }
```

This code is cited in section 23.

This code is used in section 28.

37. Shuffle the columns of the lattice.

```
⟨ shuffle lattice columns 37 ⟩ ≡
  void shufflelattice( )
  {
    COEFF *tmp;
    int i, j, r;
    unsigned int s;
#if 1
    s = (unsigned)(time(0)) * getpid( );
#else
    s = 1300964772;
#endif
    fprintf(stderr, "Seed=%u\n", s);
    srand(s);
    for (j = 0; j < 100; j++) {
      for (i = lattice_columns − 2; i > 0; i−−) {
        r = rand( ) % i;
        tmp = lattice[r];
        lattice[r] = lattice[i];
        lattice[i] = tmp;
      }
    }
    return;
  }
```

This code is used in section 28.

38. Delete the columns which do not belong to the kernel.

```
⟨ delete unnecessary columns 38 ⟩ ≡
  j = 0;
  do {
    if (lattice[j][0].p > system_rows) j++;
    else {
      for (i = j + 1; i < lattice_columns; i++) lattice[i − 1] = lattice[i];
      lattice_columns−−;
    }
  } while (j < lattice_columns − 1);
```

This code is used in section 36.

39. Test if the remaining columns contain a nonzero entry in the last row. Otherwise a nonhomogenous solution is not possible.

```
⟨ test for right hand side columns 39 ⟩ ≡
  flag = 0;
  for (i = 0; i < lattice_columns; i++)
    if (mpz_sgn(get_entry(i, lattice_rows − 1)) ≠ 0) {
      flag = 1;
      break;
    }
  if (flag ≡ 0) {
#ifndef NO_OUTPUT
    printf("Nonhomogenous␣solution␣not␣possible.\n"); fflush(stdout);
#endif
    exit(2);
    return 0;      /* Just for the compiler */
  }
```

This code is used in section 36.

40. Test of column *position* for a solution during the reduction phase.

```
⟨ solutiontest 40 ⟩ ≡
  int solutiontest(int position)
  {
    int i, j;
    int low, up;
    int end;
    ⟨ test the last two rows 43 ⟩;
    ⟨ test, if column is a solution 44 ⟩;
    mpz_set_si(upfac, 1);
    mpz_divexact(soltest_s, get_entry(position, lattice_rows − 1), lastlines_factor);
    ⟨ write a solution with blanks 45 ⟩;
    ⟨ test if one solution is enough 46 ⟩;
    return 1;
  }
```

This code is used in section 28.

41.
```
⟨ global variables 7 ⟩ +≡
  mpz_t soltest_u;
  mpz_t soltest_s;
  mpz_t soltest_upfac;
```

42.
```
⟨ initialize some globals 11 ⟩ +≡
  mpz_init(soltest_u);
  mpz_init(soltest_s);
  mpz_init_set_ui(soltest_upfac, 1);
```

43. Test the last two rows.

```
⟨ test the last two rows 43 ⟩ ≡
  if (mpz_cmpabs(get_entry(position, lattice_rows − 1), max_norm) ≠ 0) return 0;
  if (mpz_sgn(get_entry(position, lattice_rows − 1 − free_RHS)) ≡ 0) return 0;
```

This code is used in section 40.

44. A final test, if the column is really a solution. We have to distinguish whether this subroutine is called during the first or the secoond reduction. If *lattice_columns* = *system_columns* + 2 + *free_RHS* is true, then no columns have been deleted so far, i. e. we are in the first reduction phase. Accordingly a start index *low* is determined.

```
⟨ test, if column is a solution 44 ⟩ ≡
  low = 0;
  up = lattice_rows − 1 − free_RHS;
  if (lattice_columns ≡ system_columns + 2 + free_RHS) {
    for (i = 0; i < system_rows; i++)
      if (mpz_sgn(get_entry(position, i)) ≠ 0) return 0;
    low = system_rows;
  }
  if (iszeroone) {
    for (i = low; i < up; i++) {
      if (mpz_cmpabs(get_entry(position, i), max_norm) ≠ 0) return 0;
    }
  }
  else {
    for (i = low; i < up; i++) {
      if (mpz_cmpabs(get_entry(position, i), max_norm) > 0) return 0;
    }
  }
```

This code is used in section 40.

45. Print a solution. The variables are separated with blanks. Only in the case that just one solution is needed, the solution is also written into the solution file.

```
⟨ write a solution with blanks 45 ⟩ ≡
  i = low;
  if (cut_after_coeff ≡ −1) {
    end = no_original_columns;
#if 0
    if (nboundvars ≠ 0) {      /* conflicts with original_columns */
      end = nboundvars;
    }
#endif
  }
  else {
    end = cut_after_coeff;
  }
  for (j = 0; j < end; j++) {
    if (original_columns[j] ≡ 0) {
      mpz_set_si(soltest_u, 0);
    }
    else {
      if (¬iszeroone) {
        if (mpz_cmp_si(upperbounds[i − low], 0) ≠ 0) {
          mpz_divexact(soltest_upfac, upperbounds_max, upperbounds[i − low]);
        }
        else {
          mpz_set(soltest_upfac, upperbounds_max);
        }
      }
      mpz_set(soltest_u, get_entry(position, i));
      mpz_sub(soltest_u, soltest_u, soltest_s);
      mpz_divexact(soltest_u, soltest_u, max_norm_initial);
      mpz_divexact(soltest_u, soltest_u, soltest_upfac);
      mpz_divexact_ui(soltest_u, soltest_u, denom);
      mpz_abs(soltest_u, soltest_u);
      i++;
    }
    mpz_out_str(Λ, 10, soltest_u);
    printf("␣");
    if (stop_after_solutions ≡ 1) {
      mpz_out_str(fp, 10, soltest_u);
      fprintf(fp, "␣");
    }
  }
  if (free_RHS) {
    mpz_divexact(soltest_u, get_entry(position, up), max_up);
    mpz_divexact(soltest_u, soltest_u, lastlines_factor);
    mpz_abs(soltest_u, soltest_u);
    printf("␣L␣=␣");
    mpz_out_str(Λ, 10, soltest_u);
  }
  printf("\n"); fflush(stdout);
  if (stop_after_solutions ≡ 1) fprintf(fp, "\n");
```

This code is used in section 40.

46. The case *stop_after_solutions* = 1: We can stop already.

```
⟨ test if one solution is enough 46 ⟩ ≡
  if (stop_after_solutions ≡ 1) {
#ifndef NO_OUTPUT
    printf("Stopped␣in␣phase␣1␣after␣finding␣a␣random␣solution\n");
#endif
    exit(8);
  }
```

This code is used in section 40.

47. Elementary lattice basis reduction. We have standard lattice basis reduction (with deep insertions) and blockwise Korkine Zolotareff reduction. The main ingredient is the subroutine *lllfp* which does lattice basis reduction.

⟨ lattice basis reduction algorithms 47 ⟩ ≡
  ⟨ lll-subroutines 65 ⟩;
  ⟨ the underlying lattice basis reduction 48 ⟩;
  ⟨ compute $\log D$ 70 ⟩;
  ⟨ orthogonal defect 71 ⟩;
  ⟨ standard-lll 72 ⟩;
  ⟨ iterated-lll 73 ⟩;
  ⟨ bkz-reduction 74 ⟩;
  ⟨ overall exhaustive enumeration 86 ⟩;

This code is used in section 2.

48. We begin with the core *lllfp*. The multiprecision version needs additionally the matrix *bs*. This matrix contains the floating point approximation of the lattice *b*.

```
⟨ the underlying lattice basis reduction 48 ⟩ ≡
#define TWOTAUHALF 67108864.0 /* 2^{τ/2} */
  int lllfp(COEFF **b, DOUBLE **mu, DOUBLE *c, DOUBLE *N, DOUBLE **bs, int start, int
      s, int z, DOUBLE delta)
  {
    ⟨ variables for lllfp 50 ⟩;
    mpz_init(musvl);
    mpz_init(hv);
    if ((z ≤ 1) ∨ (s ≤ 1)) { /* Test for trivial cases. */
#ifndef NO_OUTPUT
      printf("Wrong␣dimensions␣in␣lllfp\n"); fflush(stdout);
#endif
      return (0);
    }
    k = (start > 1) ? start : 1;
    ⟨ first step: compute norms 49 ⟩;
    counter = 0;
    while (k < s) { /* The main loop. */
#if VERBOSE > 3
      if ((counter % 500) ≡ 0) {
#ifndef NO_OUTPUT printf("LLL:␣%d␣k:%d\n", counter, k); fflush(stdout);
#endif
      }
      counter++;
#endif
      ⟨ second step: orthogonalization of b_k 51 ⟩;
      ⟨ third step: size reduction of b_k 52 ⟩;
#if defined (DEEPINSERT)
      ⟨ fourth step: deepinsert columns 63 ⟩;
#else
      ⟨ fourth step: swap columns 64 ⟩;
#endif
    }
    mpz_clear(hv);
    mpz_clear(musvl);
    return (1);
  }
```

This code is used in section 47.

49. Step 1: We begin with stage $k = 1$. Schnorr's original algorithm runs from $k = 2$ to $k = s$. Here we have $k = 1, \ldots, s - 1$.

Initially the b_i's are rounded and then their norms, $N_i = b'_i$, are computed.

Then we do steps 2 – 4 while $k \le s$.

```
⟨ first step: compute norms 49 ⟩ ≡
  if (k < 1) k = 1;
  for (i = k − 1; i < s; ++i) {
    ss = 0.0;
    for (j = 0; j < z; ++j) {
      bs[i][j] = (DOUBLE) mpz_get_d(b[i][j + 1].c);
      ss += bs[i][j] * bs[i][j];
    }
    N[i] = SQRT(ss);
  }
```

This code is used in section 48.

50.
```
⟨ variables for lllfp 50 ⟩ ≡
  int i, j, k;
  DOUBLE ss;
  int counter;
```

See also sections 53, 55, and 62.

This code is used in section 48.

51. Step 2: computation of $\mu_{k,1}, \ldots, \mu_{k,k-1}$ and $c_k = \hat{b}_k{}^2$. This is done with Gram Schmidt Orthogonalization.

```
⟨ second step: orthogonalization of b_k 51 ⟩ ≡
  if (k ≡ 1) c[0] = N[0] * N[0];
  c[k] = N[k] * N[k];
  for (j = 0; j < k; j++) {
    ss = scalarproductfp(bs[k], bs[j], z);
    if (fabs(ss) < N[k] * N[j]/TWOTAUHALF) {
      ss = (DOUBLE) scalarproductlfp(b[k], b[j]);
    }
    for (i = 0; i < j; i++) ss −= mu[j][i] * mu[k][i] * c[i];
    if (c[j] < EPSILON) {
      fprintf(stderr, "c[%d] is very small: %lf\n", j, c[j]);
    }
    mu[k][j] = ss/c[j];
    c[k] −= ss * mu[k][j];
  }
```

This code is used in section 48.

52. Step 3: size reduction. This one of the two main parts of LLL reduction. It is some kind of rounded Gram Schmidt orthogonalization of the integer basis of the lattice.

The flag *Fr* is set true if the column has been changed, i. e. if there really has been a size reduction. The flag *Fc* is set true in ⟨ round the Gram Schmidt coefficient 54 ⟩ if the Gram-Schmidt coefficient is too large.

At the end of this section we test for rounding errors and linear dependencies. If F_c is true, then the stage is decreased by one. Otherwise the new value of b'_k is computed and we proceed to step 4.

```
⟨ third step: size reduction of b_k 52 ⟩ ≡
  Fc = Fr = 0;
  for (j = k − 1; j ≥ 0; j−−) {
    if (fabs(mu[k][j]) > 0.5) {
      ⟨ round the Gram Schmidt coefficient 54 ⟩;
      Fr = 1;
      ⟨ set b_k = b_k − ⌈μ_k, j⌋b_j 56 ⟩;
      mu[k][j] −= mus;
      solutiontest(k);
    }
  }
  ⟨ recompute N_k 60 ⟩;
  if (Fc ≡ 1) {
    k = (k − 1 > 1) ? k − 1 : 1;      /* k = max(k − 1, 1) */
  }
  else {
    ⟨ test for linear dependencies 61 ⟩;
  }
```

This code is used in section 48.

53. ⟨ variables for *lllfp* 50 ⟩ +≡

```
  int Fc, Fr;
  DOUBLE mus, cc;
  mpz_t musvl;
  mpz_t hv;
  DOUBLE *swapd;
```

54. The Gram-Schmidt coefficient is rounded. If it is too large (*Fc* is true), we have to do Schnorr's correction step: We will step back due to rounding errors.

```
⟨ round the Gram Schmidt coefficient 54 ⟩ ≡
  mus = ROUND(mu[k][j]);
  mpz_set_d(musvl, mus);
  if (fabs(mus) > TWOTAUHALF) {
    Fc = 1;
#if 0
    printf("correct␣possible␣rounding␣errors\n"); fflush(stdout);
#endif
  }     /* We have to correct possible rounding errors. */
```

This code is cited in sections 52 and 56.

This code is used in section 52.

55. ⟨ variables for *lllfp* 50 ⟩ +≡

```
  int ii, iii;
  COEFF *swapvl;
  COEFF *bb;
```

56. Replace b_k by $b_k \leftarrow b_k - \mu b_j$ and replace $\mu_{k,i}$ by $\mu_{k,i} \leftarrow \mu_{j,i}\mu$, where $\mu = \lceil \mu_{k,j} \rfloor$ as computed in section ⟨ round the Gram Schmidt coefficient 54 ⟩. This computation uses the sparse structure of the vectors.

In order to save multiplications, we treat the three cases $\lceil \mu_{k,j} \rfloor = \pm 1$ and $\lceil \mu_{k,j} \rfloor \neq \pm 1$ separately.

```
⟨ set b_k = b_k − ⌈μ_k, j⌋b_j 56 ⟩ ≡
  switch (mpz_get_si(musvl)) {
  case 1: ⟨ ⌈μ_{k,j}⌋ = 1 57 ⟩;
    break;
  case −1: ⟨ ⌈μ_{k,j}⌋ = −1 58 ⟩;
    break;
  default: ⟨ ⌈μ_{k,j}⌋ ≠ ±1 59 ⟩;
  }
```

This code is used in section 52.

57. The original loop was **for** $(i = 1;\ i \le z;\ i{+}{+})\ b[k][i].c\ {-}{=}\ b[j][i].c$;

```
⟨ ⌈μ_{k,j}⌋ = 1 57 ⟩ ≡
  i = b[j][0].p;
  while (i ≠ 0) {
    bb = &(b[k][i]);
    mpz_sub(bb→c, bb→c, b[j][i].c);
    iii = bb→p;
    if ((b[k][i − 1].p ≠ i) ∧ (mpz_sgn(bb→c) ≠ 0))
      for (ii = i − 1; (ii ≥ 0) ∧ (b[k][ii].p ≡ iii); ii−−) b[k][ii].p = i;
    else if (mpz_sgn(bb→c) ≡ 0) {
      for (ii = i − 1; (ii ≥ 0) ∧ (b[k][ii].p ≡ i); ii−−) b[k][ii].p = iii;
    }
    i = b[j][i].p;
  }
  for (i = 0; i < j; i++) mu[k][i] −= mu[j][i];
```

This code is used in section 56.

58. The original loop was **for** $(i = 1;\ i \le z;\ i{+}{+})\ b[k][i].c\ {+}{=}\ b[j][i].c$;

```
⟨ ⌈μ_{k,j}⌋ = −1 58 ⟩ ≡
  i = b[j][0].p;
  while (i ≠ 0) {
    bb = &(b[k][i]);
    mpz_add(bb→c, bb→c, b[j][i].c);
    iii = bb→p;
    if ((b[k][i − 1].p ≠ i) ∧ (mpz_sgn(bb→c) ≠ 0))
      for (ii = i − 1; (ii ≥ 0) ∧ (b[k][ii].p ≡ iii); ii−−) b[k][ii].p = i;
    else if (mpz_sgn(bb→c) ≡ 0) {
      for (ii = i − 1; (ii ≥ 0) ∧ (b[k][ii].p ≡ i); ii−−) b[k][ii].p = iii;
    }
    i = b[j][i].p;
  }
  for (i = 0; i < j; i++) mu[k][i] += mu[j][i];
```

This code is used in section 56.

59. The original loop was **for** $(i = 1;\ i \le z;\ i{+}{+})\ b[k][i].c\ {-}{=}\ b[j][i].c * musvl$;

```
⟨ ⌈μ_{k,j}⌋ ≠ ±1 59 ⟩ ≡
  i = b[j][0].p;
  while (i ≠ 0) {
    bb = &(b[k][i]);
    mpz_submul(bb→c, b[j][i].c, musvl);
    iii = bb→p;
    if ((b[k][i − 1].p ≠ i) ∧ (mpz_sgn(bb→c) ≠ 0))
      for (ii = i − 1; (ii ≥ 0) ∧ (b[k][ii].p ≡ iii); ii−−) b[k][ii].p = i;
    else if (mpz_sgn(bb→c) ≡ 0) {
      for (ii = i − 1; (ii ≥ 0) ∧ (b[k][ii].p ≡ i); ii−−) b[k][ii].p = iii;
    }
    i = b[j][i].p;
  }
#if 0
  daxpy(j, −mus, mu[k], 1, mu[j], 1);
#endif
  for (i = 0; i < j; i++) mu[k][i] −= mu[j][i] * mus;
```

This code is used in section 56.

60.

```
⟨ recompute N_k 60 ⟩ ≡
  {
    N[k] = 0.0;
    for (i = 0; i < z; i++) {
      bs[k][i] = (DOUBLE) mpz_get_d(b[k][i + 1].c);
      N[k] += bs[k][i] * bs[k][i];
    }
    N[k] = SQRT(N[k]);
  }
```

This code is used in section 52.

61. Before going to step 4 we test if b_k is linear dependent. This is the case if $N_k < \frac{1}{2}$. If we found a linear dependent vector b_k, we shift b_k to the last column of the matrix and restart *lllfp* with $s := s - 1$.

```
⟨ test for linear dependencies 61 ⟩ ≡
  if (N[k] < -EPSILON) {
#ifndef NO_OUTPUT
    fprintf(stderr, "Nk negativ! contact the author.\n"); fflush(stderr);
    printf("Nk negativ! contact the author.\n"); fflush(stdout);
#endif
    exit(1);
  }
  if (N[k] < 0.5) {
    swapvl = b[k];
    ss = N[k];
    swapd = bs[k];
    for (i = k + 1; i < s; i++) {
      b[i - 1] = b[i];
      N[i - 1] = N[i];
      bs[i - 1] = bs[i];
    }
    b[s - 1] = swapvl;
    N[s - 1] = ss;
    bs[s - 1] = swapd;
    s = s - 1;
    k = 1;
    continue;
  }
```

This code is used in section 52.

62. ⟨ variables for *lllfp* 50 ⟩ +≡

```
  int Fi;
```

63. Step 4: Swapping of columns with deep insertions. Will be used if the compiler flag `DEEPINSERT` is set. The vector b_k will be inserted left as possible. Swapping of the matrix columns is done entirely with pointers arithmetics. If no insertion is possible we step forward to $k \leftarrow k+1$, otherwise we go back to $k \leftarrow k-1$.

```
⟨ fourth step: deepinsert columns 63 ⟩ ≡
  cc = N[k] * N[k];
  j = 0;
  Fi = 0; while (j < k) {
#if 1
  if ((j > DEEPINSERT_CONST ∧ j < k − DEEPINSERT_CONST) ∨ delta * c[j] ≤ cc) {
#else
    if (delta * c[j] ≤ cc) {
#endif
      cc −= mu[k][j] * mu[k][j] * c[j];
      j++;
    }
    else {
      swapvl = b[k];
      ss = N[k];
      swapd = bs[k];
      for (i = k − 1; i ≥ j; i−−) {
        b[i + 1] = b[i];
        N[i + 1] = N[i];
        bs[i + 1] = bs[i];
      }
      b[j] = swapvl;
      N[j] = ss;
      bs[j] = swapd;
      Fi = 1;
      break;
    }
  }
  if (Fi ≡ 1) k = (j − 1 > 1) ? j − 1 : 1;      /* k = max(j − 1, 1) */
  else {
    k++;
  }
```

This code is used in section 48.

64. Standard exchange of columns (original Step 4): Either swap the vectors b_{k-1} and b_k or increment k. Swapping of the matrix columns is done entirely with swapping of pointers. If the b_k and b_{k-1} are exchanged, we step back to $k \leftarrow k-1$, otherwise we go forward to $k \leftarrow k+1$.

```
⟨ fourth step: swap columns 64 ⟩ ≡
  if (delta * c[k-1] > c[k] + mu[k][k-1] * mu[k][k-1] * c[k-1]) {
    swapvl = b[k];
    b[k] = b[k-1];
    b[k-1] = swapvl;
    ss = N[k];
    N[k] = N[k-1];
    N[k-1] = ss;
    swapd = bs[k];
    bs[k] = bs[k-1];
    bs[k-1] = swapd;
    k = (k-1 > 1) ? k-1 : 1;        /* k = max(k-1,1) */
  }
  else k++;
```

This code is used in section 48.

65. Subroutines needed for lattice basis reduction. These are scalarproducts, norms, allocation and orthogonalization.

⟨ lll-subroutines 65 ⟩ ≡
  ⟨ scalarproduct with integers 66 ⟩;
  ⟨ scalarproduct with doubles 67 ⟩;
  ⟨ memory allocation 68 ⟩;
  ⟨ free the memory 69 ⟩;

This code is used in section 47.

66. The integer scalarproduct is able to use the sparse structure of the vectors. But it strongly depends on the hardware and the compiler. This is the new variant which uses the sparse structure of the input vectors.

It depends on the compiler and the machine whether this loop is fast enough to gain speed against the plain scalarproduct evaluation. t_1 runs through the vector v, t_2 runs through the vector w.

```
⟨ scalarproduct with integers 66 ⟩ ≡
  DOUBLE scalarproductlfp(COEFF *v, COEFF *w)
  {
    DOUBLE erg;
    long t1, t2;
    COEFF *vv, *ww;
    erg = 0.0;
    t1 = v[0].p;
    t2 = w[0].p;
    if ((t1 ≡ 0) ∨ (t2 ≡ 0)) return 0;
    do {
      if (t2 > t1) {
        t1 = v[t2 − 1].p;
        if (t2 ≠ t1) {
          if (t1 ≡ 0) break;
          t2 = w[t2].p;
          if (t2 ≡ 0) break;
        }
        else goto gleich;
      }
      else if (t2 < t1) {
        t2 = w[t1 − 1].p;
        if (t2 ≠ t1) {
          if (t2 ≡ 0) break;
          t1 = v[t1].p;
          if (t1 ≡ 0) break;
        }
        else goto gleich;
      }
      else {
      gleich: vv = &(v[t1]);
        ww = &(w[t2]);
        erg += (DOUBLE) mpz_get_d(vv→c) * (DOUBLE) mpz_get_d(ww→c);
        t1 = vv→p;
        if (t1 ≡ 0) break;
        t2 = ww→p;
        if (t2 ≡ 0) break;
      }
    } while (1);
    return (erg);
  }
```

This code is used in section 65.

67. ⟨ scalarproduct with doubles 67 ⟩ ≡

```
DOUBLE scalarproductfp(DOUBLE *v, DOUBLE *w, int n)
{
#if BLAS
  return cblas_ddot(n, v, 1, w, 1);
#else
  DOUBLE r;
  int i;
  r = 0.0;
  for (i = n - 1; i ≥ 0; i--) r += v[i] * w[i];
  return r;
#endif
}
```

This code is used in section 65.

68. Allocation of memory. For the upper triangular matrix μ a rectangular matrix is allocated. So it fits for orthogonalization with Gram-Schmidt, Householder and Givens rotation.

⟨ memory allocation 68 ⟩ ≡

```
int lllalloc(DOUBLE ***mu, DOUBLE **c, DOUBLE **N, DOUBLE ***bs, int s, int z)
{
  int i, m;
#if USE_SSE
  int zeven;
#endif
  if ((z < 1) ∨ (s < 1)) return 0;
  (*c) = (DOUBLE *) calloc(s, sizeof(DOUBLE));
  (*N) = (DOUBLE *) calloc(s, sizeof(DOUBLE));
  (*mu) = (DOUBLE **) calloc(s, sizeof(DOUBLE *));
  for (i = 0; i < s; i++) (*mu)[i] = (DOUBLE *) calloc(z, sizeof(DOUBLE));
  m = (z > s) ? z : s;
  (*bs) = (DOUBLE **) calloc(m, sizeof(DOUBLE *));
#if USE_SSE
  zeven = (m % 8 ≠ 0) ? (m/8 + 1) * 8 : m;
  for (i = 0; i < m; i++) (*bs)[i] = (DOUBLE *) calloc(zeven, sizeof(DOUBLE));
#else
  for (i = 0; i < m; i++) (*bs)[i] = (DOUBLE *) calloc(z, sizeof(DOUBLE));
#endif
  return 1;
}
```

This code is used in section 65.

69. ⟨ free the memory 69 ⟩ ≡

```
int lllfree(DOUBLE **mu, DOUBLE *c, DOUBLE *N, DOUBLE **bs, int s)
{
  int i;
  for (i = 0; i < s; ++i) free(bs[i]);
  free(bs);
  for (i = 0; i < s; ++i) free(mu[i]);
  free(mu);
  free(N);
  free(c);
  return 1;
}
```

This code is used in section 65.

70. Compute $\log D$ for a lattice L, see the LLL paper.

⟨ compute $\log D$ 70 ⟩ ≡

```
double logD(COEFF **lattice, DOUBLE *c, int s, int z)
{
  double d = 0.0;
  int i;
  for (i = 0; i < s; i++) {
    d += log(c[i]) * (s - i);
  }
  d *= 0.5;
  return d;
}
```

This code is used in section 47.

71. The logarithm of the orthogonality defect of a lattice is computed: $d = \frac{\prod_{i=0}^{s-1} \|b_i\|}{\det(L)}$. For $\det(L)$ we use $\prod_{i=0}^{s-1} \|\hat{b}_i\|$.

⟨ orthogonal defect 71 ⟩ ≡

```
double orthogonal_defect(COEFF **lattice, DOUBLE *c, int s, int z)
{
  double defect = 0.0;
#if 0
  int i;
  for (i = 0; i < s; i++) defect += log((double) normfp(lattice[i])) - log((double) c[i]);
#endif
  defect /= 2.0;
  return defect;
}
```

This code is used in section 47.

72. Standard LLL reduction

```
⟨ standard-lll 72 ⟩ ≡
  void lll(COEFF **b, int s, int z, DOUBLE quality)
  {
    DOUBLE **mu;
    DOUBLE *c;
    DOUBLE *N;
    DOUBLE **bs;
    int r;
    lllalloc(&mu, &c, &N, &bs, s, z);
    r = lllfp(b, mu, c, N, bs, 1, s, z, quality);
    lllfree(mu, c, N, bs, s);
    return;
  }
```

This code is used in section 47.

73. Iterated LLL reduction. Standard LLL-reduction is applied to the lattice. Then the columns are sorted in descending (or ascending) order and the process is iterated. The Euclidean length of the lattice vectors is stored in N.

```
⟨ iterated-lll 73 ⟩ ≡
  DOUBLE iteratedlll(COEFF **b, int s, int z, int no_iterates, DOUBLE quality)
  {
    DOUBLE **mu;
    DOUBLE *c;
    DOUBLE *N;
    DOUBLE **bs;
    int r, l, i, j, runs;
    COEFF *swapvl;
    DOUBLE lD;
    lllalloc(&mu, &c, &N, &bs, s, z);
    r = lllfp(b, mu, c, N, bs, 1, s, z, quality);
    lD = logD(b, c, s, z);
#ifndef NO_OUTPUT
    printf("␣␣␣log(D)=␣%f\n", lD); fflush(stdout);
#endif
    for (runs = 1; runs < no_iterates; runs++) {
      for (j = s − 1; j > 0; j−−) {
        for (l = j − 1; l ≥ 0; l−−) {
                /*if (N[l] < N[j]) {*/      /* < sorts 'in descending order.' */
          if (N[l] > N[j]) {      /* > sorts 'in ascending order.' */
            swapvl = b[l];
            for (i = l + 1; i ≤ j; i++) b[i − 1] = b[i];
            b[j] = swapvl;
          }
        }
      }
      r = lllfp(b, mu, c, N, bs, 1, s, z, quality);
      lD = logD(b, c, s, z);
#ifndef NO_OUTPUT
      printf("%d:␣log(D)=␣%f\n", runs, lD); fflush(stdout);
#endif
    }
    lllfree(mu, c, N, bs, s);
    return lD;
  }
```

This code is used in section 47.

74. Blockwise Korkine Zolotareff reduction.

⟨ bkz-reduction 74 ⟩ ≡
  ⟨ compute gamma function 85 ⟩;
  ⟨ exhaustive enumeration of a block 82 ⟩;
  ⟨ bkz-algorithm 75 ⟩;

This code is used in section 47.

75. The control algorithm of blockwise Korkine Zolotarev Reduction. There are 2 parameters β and δ with $\frac{1}{4} < \delta < 1$ (same as in LLL) and $2 < \beta < m$ (the blocksize). The parameter s is the number of columns of the lattice, where the last column is the zero vector. Therefore the last vector of the mathematical lattice is $last \leftarrow s - 2$. *bkz* uses the subroutines *lllfp* and *enumerate*. p controls the pruning in the pruned Gauss enumeration. *start_block* is cyclically shifted through $0, 1, \ldots, s-2$. *zaehler* counts the number of positions j that satisfy $\delta \hat{b}_j{}^2 \le \lambda_1(\pi_j(L(b_j, \ldots, b_k)))$. *zaehler* is reset to 0 if the inequality does not hold for j.

Step 1: μ and c have to be allocated with $s+1$ columns.

```
⟨ bkz-algorithm 75 ⟩ ≡
  DOUBLE bkz(COEFF **b, int s, int z, DOUBLE delta, int beta, int p)
  {
    ⟨ bkz variables 76 ⟩;
    last = s − 2;      /* last points to the last nonzero vector of the lattice. */
    if (last < 1) {
#ifndef NO_OUTPUT
      printf("BKZ: the number of basis vectors is too small.\n");
      printf("Probably the number of rows is less or equal");
      printf(" to number of columns in the original system\n");
      printf("Maybe you have to increase c0 (the first parameter)!\n");
#endif
      mpz_clear(hv);
      return 0.0;
    }
    u = (long *) calloc(s, sizeof(long));
    for (i = 0; i < s; i++) u[i] = 0;
    lllalloc(&mu, &c, &N, &bs, s, z);
    lllfp(b, mu, c, N, bs, 1, s, z, delta);
    start_block = zaehler = −1;
    while (zaehler < last) {
      start_block++;
      if (start_block ≡ last) start_block = 0;
      end_block = (start_block + beta − 1 < last) ? start_block + beta − 1 : last;
        /* end_block := min(start_block + β − 1, last) */
#if 0
      printf("start_block=%d, end_block=%d\n", start_block, end_block);
#endif
      new_cj = enumerate(mu, c, u, s, start_block, end_block, p);      /* The exhaustive enumeration. */
      h = (end_block + 1 < last) ? end_block + 1 : last;      /* h := min(end_block + 1, last) */
      if (delta * c[start_block] > new_cj) {
        ⟨ successful enumeration 77 ⟩;
        zaehler = −1;
      }
      else {
        if (h > 0) {
          lllfp(b, mu, c, N, bs, h − 2, h + 1, z, delta);
            /* For some unkown reason we have to use h − 2 as start. */
        }
        zaehler++;
      }
    }    /* end of while */
    lD = logD(b, c, s − 1, z);
#ifndef NO_OUTPUT
    printf("bkz: log(D)= %f\n", lD); fflush(stdout);
```

```
#endif
    lllfree(mu, c, N, bs, s);
    free(u);
    mpz_clear(hv);
    return lD;
  }
```

This code is used in section 74.

76. ⟨ bkz variables 76 ⟩ ≡

```
DOUBLE **mu, *c, *N;
DOUBLE **bs;
static mpz_t hv;
int zaehler;
int h, i, last;
int start_block, end_block;
long *u;
DOUBLE new_cj;
DOUBLE lD;
mpz_init(hv);
```

See also section 78.

This code is used in section 75.

77. Now we found a new vector whose corresponding Gram Schmidt vector is shorter than c_j by at least a factor δ. Write the new linear combination at position $j-1$ after shifting all vectors behind $j-1$ one position to the right.

If $N_{h-1} < 0.5$ the vectors in b were linear dependent. Since at most one of the vectors $b_j, \ldots, b_h$ was linear dependent and is now zero at position h, swap the vectors back. Otherwise the vectors weren't linear dependent. This is the case if a multiple of b_{s-1} was found in *enumerate*.

⟨ successful enumeration 77 ⟩ ≡

```
#if defined (ORIGINAL_SCHNORR_EUCHNER)
  ⟨ old integration of the new vector, part 1 80 ⟩;
#else
  ⟨ build new basis 79 ⟩;
#endif
  lllfp(b, mu, c, N, bs, start_block - 1, h + 1, z, delta);
  if (N[h] < -EPSILON) {
#ifndef NO_OUTPUT
    fprintf(stderr, "NN negativ\n"); fflush(stderr);
    printf("NN negativ\n"); fflush(stdout);#endif
    exit(1);
  }
#if defined (ORIGINAL_SCHNORR_EUCHNER)
  ⟨ old integration of the new vector, part 2 81 ⟩;
#endif
```

This code is used in section 75.

78. ⟨ bkz variables 76 ⟩ +≡

```
int g, ui, q, j;
COEFF *swapvl;
```

79. The new vector $b_a = \sum_{i=a}^{e} u_i b_i$ is computed, where $a = start\_block$ and $e = end\_block$. The other old vectors $b_{a+1}, \ldots, b_e$ are adapted, in order to keep the whole block linear independent. The algorithm is described in the doctoral thesis of H. Ritter.

```
⟨ build new basis 79 ⟩ ≡
  for (j = 1; j ≤ z; j++) mpz_set_si(b[last + 1][j].c, 0);
  for (i = start_block; i ≤ end_block; i++) {
    if (u[i] ≠ 0)
      for (j = 1; j ≤ z; j++) {
        if (u[i] > 0) {
          mpz_addmul_ui(b[last + 1][j].c, b[i][j].c, u[i]);
        }
        else {
          mpz_submul_ui(b[last + 1][j].c, b[i][j].c, −u[i]);
        }
      }
  }
  g = end_block;
  while (u[g] ≡ 0) g−−;
  i = g − 1;
  while (labs(u[g]) > 1) {
    while (u[i] ≡ 0) i−−;
    q = (int) ROUND((1.0 ∗ u[g])/u[i]);
    ui = u[i];
    u[i] = u[g] − q ∗ u[i];
    u[g] = ui;
    for (j = 1; j ≤ z; j++) {
      mpz_set(hv, b[g][j].c);
      mpz_mul_si(b[g][j].c, b[g][j].c, (long) q);
      mpz_add(b[g][j].c, b[g][j].c, b[i][j].c);
      mpz_set(b[i][j].c, hv);
    }
    coeffinit(b[g], z);
    coeffinit(b[i], z);
  }
  swapvl = b[g];
  for (i = g; i > start_block; i−−) b[i] = b[i − 1];
  b[start_block] = b[last + 1];
  coeffinit(b[start_block], z);
  b[last + 1] = swapvl;
  for (j = 1; j ≤ z; j++) mpz_set_si(b[last + 1][j].c, 0);
  coeffinit(b[last + 1], z);
```

This code is used in section 77.

80. This is the original way to include the new vector: just write it to the beginning of the block and reduce the new system. Then one vector is reduced to zero. This vector is deleted.

```
⟨ old integration of the new vector, part 1 80 ⟩ ≡
  for (l = 1; l ≤ z; l++) mpz_set_si(b[last + 1][l].c, 0);
  for (i = start_block; i ≤ end_block; i++)
    for (l = 1; l ≤ z; l++) {
      mpz_addmul_si(b[last + 1][l].c, b[i][l].c, ui);
    }
  coeffinit(b[last + 1], z);
  solutiontest(last + 1);
  swapvl = b[last + 1];
  for (i = last; i ≥ start_block; i−−) b[i + 1] = b[i];
  b[start_block] = swapvl;
```

This code is used in section 77.

81. ⟨ old integration of the new vector, part 2 81 ⟩ ≡

```
  if (N[h] < 0.5) {      /* After reduction this vector should be zero */
    swapvl = b[h];
    for (i = h + 1; i ≤ last + 1; i++) b[i − 1] = b[i];
    b[last + 1] = swapvl;
  }
  else {
#ifndef NO_OUTPUT
    printf("Not␣linear␣dependent;␣%f\n", (double)(N[h − 1])); fflush(stdout);
#endif
    exit(1);
  }
```

This code is used in section 77.

82. Pruned Gauss-Enumeration. Enumerate in depth first search. Pruned Version without goto's as it is described in H. H. Hörner's diploma thesis. Blocks with blocksize lower than `SCHNITT` are completely enumerated, otherwise the enumeration is pruned. 2^{-p} is the probability that lattice points are lost.

```
⟨ exhaustive enumeration of a block 82 ⟩ ≡
  DOUBLE enumerate(DOUBLE **mu, DOUBLE *c, long *u, int s,
          int start_block, int end_block, int p)
  {
    DOUBLE cd, dum;
    DOUBLE *y, *cs, *eta;
    DOUBLE **sigma;
    int *r;
    long *us, *delta, *d, *v;
    int t, i, t_up, len;
    double alpha;
    int tmax;
    static DOUBLE pi = 3.14159265358979323846264338 3;
    static DOUBLE e = 2.71828182845904523536028747135266249775724709 3;
    int SCHNITT = 40;
    if (c[start_block] ≤ EPSILON) {
#ifndef NO_OUTPUT
      fprintf(stderr, "Hier␣ist␣was␣faul!␣start_block=%d␣%f\n", start_block, (double) c[start_block]);
      fflush(stderr);
      printf("Hier␣ist␣was␣faul!␣start_block=%d␣%f\n", start_block, (double) c[start_block]);
      fflush(stdout);
#endif
      exit(1);
    }
    us = (long *) calloc(s + 1, sizeof(long));
    cs = (DOUBLE *) calloc(s + 1, sizeof(DOUBLE));
    y = (DOUBLE *) calloc(s + 1, sizeof(DOUBLE));
    delta = (long *) calloc(s + 1, sizeof(long));
    d = (long *) calloc(s + 1, sizeof(long));
    eta = (DOUBLE *) calloc(s + 1, sizeof(DOUBLE));
    v = (long *) calloc(s + 1, sizeof(long));
    sigma = (DOUBLE **) calloc(s, sizeof(DOUBLE *));
    r = (int *) calloc(s + 1, sizeof(int));
    for (i = 0; i < s; i++) {
      sigma[i] = (DOUBLE *) calloc(s, sizeof(DOUBLE));
      r[i] = i - 1;
    }
    len = (end_block + 1 - start_block);
    for (i = start_block; i ≤ end_block + 1; i++) {
      cs[i] = y[i] = 0.0;
      u[i] = us[i] = v[i] = delta[i] = 0;
      d[i] = 1;
    }
    us[start_block] = u[start_block] = 1;
    cd = c[start_block];
    t = tmax = start_block;      /* Now we start from t = start_block instead of t = end_block. */
    ⟨ precompute η 84 ⟩;
    while (t ≤ end_block) {
```

```
    ⟨ the block search loop 83 ⟩;
  }
  free(us);
  free(cs);
  free(y);
  free(delta);
  free(d);
  free(eta);
  free(v);
  for (i = s − 1; i ≥ 0; i−−) {
    free(sigma[i]);
  }
  free(sigma);
  free(r);
  return (cd);
}
```

This code is used in section 74.

83. The search loop.

```
⟨ the block search loop 83 ⟩ ≡
  { dum = us[t] + y[t];
  cs[t] = cs[t + 1] + dum * dum * c[t];
#if 0
  if (cs[t] < cd − eta[t]) {      /* alpha = (t > t2) ? 0.7 * cd : cd; */
#else
    if (len ≤ SCHNITT) {
      alpha = 1.0;
    }
    else {
      alpha = sqrt(1.20 * (end_block + 1 − t)/len);
      if (alpha ≥ 1.0) alpha = 1.0;
    }
    alpha *= cd;      /* alpha = (2 * t > len) ? 0.8 * cd : cd; */
    if (cs[t] < alpha − EPSILON) {
#endif
      if (t > start_block) {
        t−−;
        if (r[t + 1] > r[t]) r[t] = r[t + 1];
        delta[t] = 0;
        for (i = r[t + 1]; i > t; i−−) sigma[i][t] = sigma[i + 1][t] + us[i] * mu[i][t];
#if 0
        dum = 0.0;
        for (i = t + 1; i ≤ tmax; i++) dum += us[i] * mu[i][t];
        if (fabs(dum − sigma[t + 1][t]) > 0.001) {
          printf("1diff: %0.6lf %0.6lf %0.8lf\n", dum, sigma[t + 1][t], dum − sigma[t + 1][t]);
        }
#endif
        y[t] = sigma[t + 1][t];      /* dum; */
        us[t] = v[t] = (long)(ROUND(−y[t]));
        d[t] = (v[t] > −y[t]) ? −1 : 1;
      }
      else {
#if 0
        printf("success %0.4lf %0.4lf %0.8lf\n", cd, cs[start_block], cd − cs[start_block]);
#endif
        cd = cs[start_block];
        for (i = start_block; i ≤ end_block; i++) u[i] = us[i];
        goto nextstep;
      }
    }
    else {
      t++;
      r[t] = t;
    nextstep:
      if (tmax < t) tmax = t;
      if (t < tmax) delta[t] = −delta[t];
      if (delta[t] * d[t] ≥ 0) delta[t] += d[t];
      us[t] = v[t] + delta[t];
    }
  }
```

This code is used in section 82.

84. Precomputation of η_t. If we know c, an upper bound of the square of the norm of a shortest vector, and we arrived in our enumeration at level t and a vector whose square norm equals $\tilde{c}_t$, then the probability that we this branch of the enumeration tree yields a shorter vector than the bound c is $\frac{\mathrm{vol}\, S(\sqrt{c-\tilde{c}_t},z)}{\det L}$. We compute η_t such that

$$\frac{\mathrm{vol} S(\sqrt{\eta},z)}{\det L} = 2^{-p}.$$

Then, in our enumeration we stop if

$$\tilde{c}_t \geq c - \eta_t.$$

The volume of the z-dimensional ball with radius $\sqrt{\eta}$ is equal to

$$\mathrm{vol}\, S(\sqrt{\eta},z) = \frac{(\pi\cdot\eta)^{z/2}}{\Gamma(z/2+1)}$$

The dimension z is $t-1$, one less than the enumeration level, and we have $t = i - \mathit{start\_block}$. H.H. Hörner computes η by an approximation with Stirlings's formular. Here it is computed with the Euler-MacLaurin-formular, see Abramowitz, Stegun.

Update 19.1.2010: now we try Hörners approximation:

$$\eta_i = \frac{i-j}{2\pi e}\left(\pi(i-j)p^2\sum_{h=j}^{i} c_h\right)^{1/(i-j)}$$

```
⟨ precompute η 84 ⟩ ≡
  eta[start_block] = 0.0;
  if (end_block − start_block ≤ SCHNITT) {
    for (i = start_block + 1; i ≤ end_block; i++) eta[i] = 0.0;
  }
  else {
    dum = log(c[start_block]);      /* This is my version to cokkmpute the parameters of the Gaussian
        volume heuristics. The Hoerner version seems to be better. */
#if 0
    for (i = start_block + 1; i ≤ end_block; i++) {
      t_up = i − start_block;
      eta[i] = exp((log_gamma(t_up/2.0 + 1) − p ∗ log(2.0)) ∗ 2.0/t_up + dum/t_up)/pi;
      if (i < end_block) dum += log(c[i]);
#if 0
      if (i ≡ start_block + 1) printf("eta:␣\n");
      printf("%0.2lf␣", eta[i]);
      if (i ≡ end_block) printf("\n");
#endif
    }
#else      /* Hoerners version of the Gaussian volume heuristics. */
    dum = log(c[start_block]);
    for (i = start_block + 1; i ≤ end_block; i++) {
      t_up = i − start_block;
      eta[i] = 0.5 ∗ t_up ∗ exp((log(pi ∗ t_up) − 2.0 ∗ p ∗ log(2.0) + dum)/t_up)/(pi ∗ e);
      if (i < end_block) dum += log(c[i]);
#if 0
      if (i ≡ start_block + 1) printf("eta:␣");
      printf("%0.2lf␣", eta[i]);
      if (i ≡ end_block) {
        printf("\n");
```

```
      fflush(stdout);
    }
#endif
  }
#endif
}
```

This code is used in section 82.

85. ⟨ compute gamma function 85 ⟩ ≡

```
DOUBLE laurin(DOUBLE x)
{
  static DOUBLE K1 = 0.91819385332046727417803297364056 20;
  static DOUBLE K2 = 0.08333333333333333333333333333333333333;
  static DOUBLE K3 = 0.00277777777777777777777777777777777777;
  static DOUBLE K4 = 0.00079365079365079365079365079365079365;
  static DOUBLE K5 = 0.00059523809523809523809523809523809523;
  static DOUBLE K6 = 0.00084175084175084175084175084175084175;
  static DOUBLE K7 = 0.00191752691752691752691752691752691752;
  static DOUBLE K8 = 0.00641025641025641025641025641025641025;
  static DOUBLE K9 = 0.02955065295100212097167968 75;
  static DOUBLE K10 = 0.17968122661113739013671875;
  static DOUBLE K11 = 1.39243221282958984375;
  DOUBLE y;
  y = 1.0/(x * x);
  y = (x - 0.5) * log(x) - x + K1 + (1.0/x) * (K2 - y * (K3 - y * (K4 - y * (K5 - y * (K6 - y * (K7 - y * (K8 -
      y * (K9 - y * (K10 - y * K11)))))))));
  return y;
}
DOUBLE log_gamma(DOUBLE x)
{
  DOUBLE y;
  int i, n;
  static int MM = 13;
  if (x ≤ 0.0) return -1.0;
  if (x > 100000000.0) {
    y = x * (log(x) - 1.0);
  }
  else {
    if (x ≥ MM) {
      y = laurin(x);
    }
    else {
      n = MM - (int)(floor(x));
      y = x - floor(x) + MM;
      y = laurin(y);
      for (i = 0; i < n; i++) y -= log(y + i);
    }
  }
  return y;
}
```

This code is used in section 74.

86. Exhaustive enumeration. The algorithm of H. Ritter.

```
#define FINCKEPOHST 1
#define EIGENBOUND 0
⟨ overall exhaustive enumeration 86 ⟩ ≡
  ⟨ globals for enumeration 87 ⟩;
  ⟨ some vector computations 120 ⟩;
  ⟨ orthogonalization 121 ⟩;
  ⟨ matrix inversion for Fincke-Pohst 124 ⟩;
  ⟨ pruning subroutines and output 125 ⟩;
  ⟨ output polyhedra 133 ⟩;
  ⟨ output LP 134 ⟩;
  DOUBLE explicit_enumeration(COEFF **lattice, int columns, int rows)
  {
    ⟨ local variables for explicit_enumeration( ) 89 ⟩;    /* Vector to collect enumeration statistics */
    long nlow[1000];
    for (i = 0; i < 1000; i++) nlow[i] = 0;
    ⟨ test the size of the basis 90 ⟩;
    ⟨ allocate the memory for enumeration 91 ⟩;
    ⟨ allocate the memory for Eigen bound 93 ⟩;
    ⟨ initialize arrays 95 ⟩;
    ⟨ count nonzero entries in the last rows(s) 98 ⟩;
#if 0
    ⟨ sort lattice columns 96 ⟩;
#endif
    ⟨ set the simple pruning bounds 99 ⟩;
    ⟨ orthogonalize the basis 101 ⟩;
#if 0
    basis2poly( );
#endif
#if FINCKEPOHST
    ⟨ determine Fincke-Pohst bounds 102 ⟩;
#endif    /* Remove trailing unnecessary columns. That means, columns whose corresponding
         Finke-Pohst bounds are equal to 0 can be removed. This is important for the Selfdual Bent
         Functions Problems */
#if 1
    for (i = columns − 1; i ≥ 0; i−−) {
      if (fipo[i] < 0.9) {
        printf("DEL\n");
        columns−−;
      }
      else {
        break;
      }
    }
#endif    /* print_lattice( ); */
#if 0
    ⟨ orthogonalize the basis 101 ⟩;
    ⟨ determine Fincke-Pohst bounds 102 ⟩;
#endif
#if EIGENBOUND
    ⟨ initialize Eigen bounds 103 ⟩;
```

```
#endif
#if 0
    basis2LP(fipo_l, fipo_u);
#endif
    ⟨ initialize first-nonzero arrays 106 ⟩;
    ⟨ initialize second-nonzero arrays 107 ⟩;
    ⟨ more initialization 109 ⟩;
    ⟨ the loop of the exhaustive enumeration 111 ⟩;
    ⟨ final output 115 ⟩;
    ⟨ free allocated memory for enumeration 116 ⟩;
    return 1;
  }
```

This code is used in section 47.

87. ⟨ globals for enumeration 87 ⟩ ≡

```
#if 0
  static FILE *fp;
#endif
```

See also sections 92, 94, 100, and 108.

This code is used in section 86.

88.

```
⟨ diophant.h 4 ⟩ +≡
  struct constraint {
    double val[2];
    int parent;
    int isSet;
  } CONSTRAINT;
```

89. ⟨ local variables for *explicit_enumeration*( ) 89 ⟩ ≡ /* __attribute((aligned(16))) */

```
  int level, level_max;
  int i, j, l;
  long loops;
  DOUBLE *y, *cs, *us;
  long *delta, *d, *eta;
#if 0
  mpz_t *v;
#else
  long *v;
#endif
  int *first_nonzero, *first_nonzero_in_column, *firstp;
  int *snd_nonzero, *snd_nonzero_in_column, *sndp;
  struct constraint *cons;
  DOUBLE *N, **mu, *c, **w, **bd, **mu_trans;
  DOUBLE Fd, Fq, Fqeps;
  DOUBLE *dum;
  DOUBLE tmp;
  COEFF *swap_vec;
#if USE_SSE
  int rowseven;
#endif
  int isSideStep = 0;
  DOUBLE stepWidth = 0.0;
  DOUBLE olddum;
#if defined (FINCKEPOHST)
  DOUBLE *fipo;
#endif
```

See also section 104.

This code is used in section 86.

90. It is tested, if the remaining columns build a basis of the kernel.

⟨ test the size of the basis 90 ⟩ ≡

```
#ifndef NO_OUTPUT
  printf("Dimension of solution space (k): %d compared to s-z+2: %d\n", columns,
      system_columns − system_rows + 1 + free_RHS);
  fflush(stdout);
#endif
  if (columns < system_columns − system_rows + 1 + free_RHS) {
#ifndef NO_OUTPUT
    fprintf(stderr, "LLL didn't succeed in computing a basis of the kernel.\n");
    fprintf(stderr, "Please increase c0 (the first parameter)!\n");
    printf("LLL didn't succeed in computing a basis of the kernel.\n");
    printf("Please increase c0 (the first parameter)!\n");
#endif
    return 0;
  }
```

This code is used in section 86.

91. The memory is allocated. If multiprecision arithmetics is used *bd* has already been allocated in *lllalloc*.

```
⟨ allocate the memory for enumeration 91 ⟩ ≡
  lllalloc(&mu, &c, &N, &bd, columns, rows);
  us = (DOUBLE *) calloc(columns + 1, sizeof(DOUBLE));
  cs = (DOUBLE *) calloc(columns + 1, sizeof(DOUBLE));
  y = (DOUBLE *) calloc(columns + 1, sizeof(DOUBLE));
  delta = (long *) calloc(columns + 1, sizeof(long));
  d = (long *) calloc(columns + 1, sizeof(long));
  first_nonzero = (int *) calloc(rows, sizeof(int));
  first_nonzero_in_column = (int *) calloc(columns + rows + 1, sizeof(int));
  if (first_nonzero_in_column ≡ Λ) return (0);
  firstp = (int *) calloc(columns + 1, sizeof(int));
  snd_nonzero = (int *) calloc(rows, sizeof(int));
  snd_nonzero_in_column = (int *) calloc(columns + rows + 1, sizeof(int));
  if (snd_nonzero_in_column ≡ Λ) return (0);
  sndp = (int *) calloc(columns + 1, sizeof(int));
  cons = (struct constraint *) calloc(columns, sizeof(struct constraint));
  for (i = 0; i < columns; ++i) {
    cons[i].isSet = 0;
  }
  eta = (long *) calloc(columns + 1, sizeof(long));
  v = (long *) calloc(columns + 1, sizeof(long));
  w = (DOUBLE **) calloc(columns + 1, sizeof(DOUBLE *));
#if USE_SSE
  rowseven = (rows % 8 ≠ 0) ? (rows/8 + 1) * 8 : rows;
  for (i = 0; i ≤ columns; i++) w[i] = (DOUBLE *) calloc(rowseven, sizeof(DOUBLE));
#else
  for (i = 0; i ≤ columns; i++) w[i] = (DOUBLE *) calloc(rows, sizeof(DOUBLE));
#endif
  mu_trans = (DOUBLE **) calloc(columns + 1, sizeof(DOUBLE *));
  for (i = 0; i ≤ columns; i++) mu_trans[i] = (DOUBLE *) calloc(columns + 1, sizeof(DOUBLE));
  dum = (DOUBLE *) calloc(columns + 1, sizeof(DOUBLE));
#if FINCKEPOHST
  fipo = (DOUBLE *) calloc(columns + 1, sizeof(DOUBLE));
  muinv = (DOUBLE **) calloc(columns, sizeof(DOUBLE *));
  for (i = 0; i < columns; ++i) muinv[i] = (DOUBLE *) calloc(rows, sizeof(DOUBLE));
  fipo_LB = (DOUBLE **) calloc(columns + 1, sizeof(DOUBLE *));
  fipo_UB = (DOUBLE **) calloc(columns + 1, sizeof(DOUBLE *));
  for (i = 0; i ≤ columns; ++i) {
    fipo_LB[i] = (DOUBLE *) calloc(columns + 1, sizeof(DOUBLE));
    fipo_UB[i] = (DOUBLE *) calloc(columns + 1, sizeof(DOUBLE));
  }
#endif
```

This code is used in section 86.

92. Additional variables for Fincke-Pohst bounds.

```
⟨ globals for enumeration 87 ⟩ +≡
#if FINCKEPOHST
  DOUBLE **muinv;
  DOUBLE **fipo_UB, **fipo_LB;
#endif    /* mpz_t *upb, *lowb; */
  long fipo_success;
```

93. The memory for R and $Rinv$ and the additional arrays for EIGENBOUND is allocated.

```
⟨ allocate the memory for Eigen bound 93 ⟩ ≡
#if EIGENBOUND
  Rinv = (DOUBLE **) calloc(columns, sizeof(DOUBLE *));
  for (i = 0; i < columns; ++i) Rinv[i] = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
  R = (DOUBLE **) calloc(columns, sizeof(DOUBLE *));
  for (i = 0; i < columns; ++i) R[i] = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
  eig_f = (DOUBLE **) calloc(columns, sizeof(DOUBLE *));
  for (i = 0; i < columns; ++i) eig_f[i] = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
  eig_RinvR = (DOUBLE **) calloc(columns, sizeof(DOUBLE *));
  for (i = 0; i < columns; ++i) eig_RinvR[i] = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
  eig_min = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
  eig_bound = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
  Rinvi = (DOUBLE **) calloc(columns, sizeof(DOUBLE *));
  for (i = 0; i < columns; ++i) Rinvi[i] = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
  bnd_up = (DOUBLE **) calloc(columns, sizeof(DOUBLE *));
  for (i = 0; i < columns; ++i) bnd_up[i] = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
  bnd_lo = (DOUBLE **) calloc(columns, sizeof(DOUBLE *));
  for (i = 0; i < columns; ++i) bnd_lo[i] = (DOUBLE *) calloc(columns, sizeof(DOUBLE));
#endif
```

This code is used in section 86.

94. Additional variables for "Eigen bounds".

```
⟨ globals for enumeration 87 ⟩ +≡
#if EIGENBOUND
  DOUBLE **Rinv;
  DOUBLE **R;
  DOUBLE **eig_f;
  DOUBLE **eig_RinvR;
  DOUBLE eig_Rs;
  DOUBLE *eig_min;
  DOUBLE *eig_bound;
  long eig_cut;
  DOUBLE **Rinvi;
  DOUBLE **bnd_up;
  DOUBLE **bnd_lo;
#endif
```

95. The starting positions of the arrays are set.

```
⟨ initialize arrays 95 ⟩ ≡
  for (i = 0; i ≤ columns; i++) {
    cs[i] = y[i] = us[i] = 0.0;
    delta[i] = 0;
#if 0
    mpz_set_si(v[i], 0);
#else
    v[i] = 0;
#endif
    eta[i] = d[i] = 1;
    for (l = 0; l < rows; l++) w[i][l] = 0.0;
  }
```

This code is used in section 86.

96. Do simple sorting along the last row such that the columns with a nonzero entry in the last row are at the end.

```
⟨ sort lattice columns 96 ⟩ ≡
  for (j = columns − 1; j > 0; j−−) {
    for (l = j − 1; l ≥ 0; l−−) {
      if (mpz_cmpabs(get_entry(l, rows − 1), get_entry(j, rows − 1)) > 0) {
        swap_vec = lattice[l];
        for (i = l + 1; i ≤ j; i++) lattice[i − 1] = lattice[i];
        lattice[j] = swap_vec;
      }
    }
  }
```

This code is used in section 86.

97. Do simple sorting along Fincke-Pohst bounds.

```
⟨ sort by Fincke-Pohst bounds 97 ⟩ ≡
  for (j = columns − 3; j > 0; j−−) {
    for (l = j − 1; l ≥ 0; l−−) {
#if 1
      if (fipo[l] < fipo[j]) {      /* < means: sort in descending order. */
#endif
        swap_vec = lattice[l];
        for (i = l + 1; i ≤ j; i++) lattice[i − 1] = lattice[i];
        lattice[j] = swap_vec;
      }
    }
  }
```

98. ⟨ count nonzero entries in the last rows(s) 98 ⟩ ≡

```
if (free_RHS) {
  i = 0;
  for (j = columns − 1; j ≥ 0; j−−)
    if (mpz_sgn(get_entry(j, rows − 2)) ≠ 0) i++;
#ifndef NO_OUTPUT
  printf("Number␣of␣nonzero␣entries␣in␣the␣second␣last␣row:␣%d\n", i);
  fflush(stdout);
#endif
}
i = 0;
for (j = columns − 1; j ≥ 0; j−−)
  if (mpz_sgn(get_entry(j, rows − 1)) ≠ 0) i++;
#ifndef NO_OUTPUT
printf("Number␣of␣nonzero␣entries␣in␣the␣last␣row:␣%d\n", i);
fflush(stdout);
#endif
```

This code is used in section 86.

99. Set the simple bounds for pruning. Fq is the maximum norm of a solution, Fd is the square of the euclidean norm of a solution, EPSILON is a security buffer to avoid pruning due to rounding errors.

⟨ set the simple pruning bounds 99 ⟩ ≡

```
Fq = (DOUBLE) mpz_get_d(max_norm);
Fd = (rows ∗ Fq ∗ Fq) ∗ (1.0 + EPSILON);
Fqeps = (1.0 + EPSILON) ∗ Fq;      /∗ Used in prune() ∗/
#ifndef NO_OUTPUT
#if VERBOSE > 0
printf("Fq:␣%f\n", (double) Fq);
printf("Fd:␣%f\n", (double) Fd);
#endif
#endif
```

This code is used in section 86.

100. ⟨ globals for enumeration 87 ⟩ +≡

```
DOUBLE dum1, dum2;
```

101.

⟨ orthogonalize the basis 101 ⟩ ≡

```
#if GIVENS
givens(lattice, columns, rows, mu, bd, c, N, Fq);
#else
gramschmidt(lattice, columns, rows, mu, bd, c, N, Fq);
#endif      /∗ compute mu^⊤, the transpose of mu. ∗/
for (i = 0; i < columns; i++)
  for (j = 0; j < columns; j++) mu_trans[j][i] = mu[i][j];
```

This code is used in section 86.

102. Compute the Fincke-Pohst bounds. See for example H. Cohen, "A course in computational number theory", page 104. If we have the quadratic form $Q(x) = x^t R^t R x$, and r_i are the rows of the matrix R and r'_i are the rows of R^{-1}. Then from $R^{-1}Rx = x$ we obtain $x_i = r'_i Rx$. With Cauchy-Schwarz this gives

$$x_i^2 \le \|r'_i\|^2 (x^t R^t R x) \le \|r'_i\|^2 C$$

where C is an upper bound for the solution vector. We have orthogonalized B such that $B = \tilde{B}\mu$. And

$$\tilde{B}^{-1} = \begin{pmatrix} 1/c_1 & 0 & \dots & 0 \\ 0 & 1/c_2 & \dots & 0 \\ & & \dots & \\ 0 & & \dots & 1/c_n \end{pmatrix} \cdot \tilde{B}^\top$$

Therefore, the above $\|r'_i\|^2$ is in our case

$$r'_{ij} = \sum_l muinv[i][l] \cdot bd[l][j]/c[l].$$

Then we also compute an upper bound with the ℓ_1 norm and we take the lower of the two bounds.

$$|x_i| \le \|r'_i\|_1 \|Rx\|_\infty$$

```
⟨ determine Fincke-Pohst bounds 102 ⟩ ≡
  fipo_success = 0;
  inverse(mu, muinv, columns);
#ifndef NO_OUTPUT
#if VERBOSE > -1
  printf("\nFincke-Pohst␣bounds:\n"); fflush(stdout);
#endif
#endif
  for (i = 0; i < columns; i++) {      /* Symmetric Fincke-Pohst */
    fipo[i] = 0.0;
    dum1 = 0.0;
    for (j = 0; j < rows; j++) {
      tmp = 0.0;
      for (l = i; l < columns; l++) tmp += muinv[i][l] * bd[l][j]/c[l];
      fipo[i] += tmp * tmp;
      dum1 += fabs(tmp);
    }
    fipo[i] = SQRT(fipo[i] * Fd);
    dum1 = fabs(dum1 * Fq);
    if (dum1 < fipo[i]) fipo[i] = dum1;
    fipo[i] *= (1.0 + EPSILON);
    fipo_LB[columns][i] = -fipo[i];
    fipo_UB[columns][i] = fipo[i];
#ifndef NO_OUTPUT
      /* printf("%03d:␣%0.1lf\t%0.1lf\n", i, fipo_LB[columns][i], fipo_UB[columns][i]); */
#endif
#ifndef NO_OUTPUT
#if VERBOSE > -1
    printf("%0.3lf␣", fipo[i]);
#endif
#endif
```

```
  }
#ifndef NO_OUTPUT
#if VERBOSE > -1
  printf("\n\n"); fflush(stdout);
#endif
#endif
```

This code is used in section 86.

103. Initialize Eigen bounds.

```
⟨ initialize Eigen bounds 103 ⟩ ≡
#if EIGENBOUND
  for (i = 0; i < columns; i++) {
    for (j = 0; j < columns; j++) {
      R[i][j] = mu[j][i] * SQRT(c[i]);
      Rinv[i][j] = muinv[i][j]/SQRT(c[j]);
    }
  }
#if 0
  printf("\nCheck\n");
  for (i = 0; i < columns; i++) {
    for (j = 0; j < columns; j++) {
      for (l = 0, eig_s = 0.0; l < columns; l++) eig_s += Rinv[i][l] * R[l][j];
      printf("%0.3lf\t", eig_s);
    }
    printf("\n");
  }
  printf("\n");
#endif
  for (k = 0; k < columns; k++) {
    for (i = 0; i < k; i++) {
      eig_s = 0.0;
      for (j = 0; j < k; j++) {
        eig_s += Rinv[i][j] * R[j][k];
      }
      eig_RinvR[k][i] = eig_s;
    }
  }
  for (k = 0; k < columns; k++) {
    for (i = 0; i < k; i++) {
      Rinvi[k][i] = 0.0;
      for (j = i; j < k; j++) {
        Rinvi[k][i] += Rinv[i][j] * Rinv[i][j];
      }
    }
  }
#if 0
  printf("Jacobi: \n");
  for (k = 0; k < columns; k++) {
    eig_s = Jacobi(R, k);
    eig_min[k] = eig_s;
    printf("%0.3lf ", eig_s);
    fflush(stdout);
  }
  printf("\n");
  printf("End Jacobi \n");
  eig_cut = 0;
#endif
#endif
```

This code is used in section 86.

104. ⟨ local variables for *explicit_enumeration*( ) 89 ⟩ +≡

```
#if EIGENBOUND
  int k;
  DOUBLE eig_s, eig_term2;
#endif
```

105. ⟨ compute new Eigen bound 105 ⟩ ≡

```
#if 1
  if (level ≡ columns − 1) {
    printf("Oben\n");
    eig_s = 0.0;
    for (i = 0; i < level; i++) {
      eig_f[level][i] = eig_RinvR[level][i] ∗ us[level];
      dum1 = fabs(eig_f[level][i]);
      dum1 −= round(dum1);
      eig_s += dum1 ∗ dum1;
    }
    eig_bound[level] = eig_s ∗ eig_min[level];
    for (i = 0; i < level; i++) {
      bnd_up[level − 1][i] = fipo[i];
      bnd_lo[level − 1][i] = −fipo[i];
    }
  }
  else {
    eig_term2 = 0.0;
    for (k = level + 1; k < columns; k++) {
      eig_term2 += R[level][k] ∗ us[k];
    }
    eig_s = 0.0;
    for (i = 0; i < level; i++) {
      eig_f[level][i] = eig_f[level + 1][i] − Rinv[i][level] ∗ eig_term2 + eig_RinvR[level][i] ∗ us[level];
    }
    for (i = 0; i < level; i++) {
      dum1 = SQRT(Rinvi[level][i] ∗ (Fd − cs[level]));
      bnd_up[level − 1][i] = bnd_up[level][i];
      if (bnd_up[level − 1][i] > dum1 − eig_f[level][i]) bnd_up[level − 1][i] = dum1 − eig_f[level][i];
      bnd_lo[level − 1][i] = bnd_lo[level][i];
      if (bnd_lo[level − 1][i] < −dum1 − eig_f[level][i]) bnd_lo[level − 1][i] = −dum1 − eig_f[level][i];
        /∗ printf("(%0.11f, %0.11f) ", bnd_lo[level − 1][i], bnd_up[level − 1][i]); ∗/
      if (bnd_lo[level − 1][i] > bnd_up[level − 1][i]) {
        printf("CUT\n");
      }
    }
  }
#else
  if (level ≡ 0) {
    eig_bound[level] = 0.0;
  }
  else {
    for (i = 0; i < level; i++) {
      eig_s = 0.0;
      for (k = level; k < columns; k++) {
        eig_s += R[i][k] ∗ us[k];
      }
      eig_f[0][i] = eig_s;
    }
    eig_s = 0.0;
    for (i = 0; i < level; i++) {
```

```
      eig_f[level][i] = 0.0;
      for (k = 0; k < level; k++) {
        eig_f[level][i] += Rinv[i][k] * eig_f[0][k];
      }
      dum1 = fabs(eig_f[level][i]);
      dum1 -= round(dum1);
      eig_s += dum1 * dum1;
    }
    if (fabs(eig_s * eig_min[level] - eig_bound[level]) > 0.000001) {
      printf("WRONG: %0.5lf %0.5lf\n", eig_s * eig_min[level], eig_bound[level]);
    }
    eig_bound[level] = eig_s * eig_min[level];
  }
#endif
```

This code is used in section 113.

106. First, find the index of the first non-zero entry in each row. Then, detect in each column which rows have its first non-zero entry in this column.

```
⟨ initialize first-nonzero arrays 106 ⟩ ≡
  for (l = 0; l < rows; l++) {
    for (i = 0; i < columns; i++)
      if (mpz_sgn(get_entry(i, l)) ≠ 0) {
        first_nonzero[l] = i;
        break;
      }
  }
  printf("First non-zero entries:\n");
  j = 0;
  for (l = 0; l < columns; l++) {
    firstp[l] = j;
    first_nonzero_in_column[j] = 0;
    j++;
    for (i = 0; i < rows; i++) {
      if (first_nonzero[i] ≡ l) {
        first_nonzero_in_column[j] = i;
        first_nonzero_in_column[firstp[l]]++;
        j++;
      }
    }
    printf("%d ", first_nonzero_in_column[firstp[l]]);
  }
  printf(": %d\n", rows);
  firstp[columns] = j;
  first_nonzero_in_column[j] = 0;
```

This code is used in section 86.

107. Find the index of the second non-zero entry in each row. Then, detect in each column which rows have its second non-zero entry in this column.

```
⟨ initialize second-nonzero arrays 107 ⟩ ≡
  for (l = 0; l < rows; l++) {
    for (i = first_nonzero[l] + 1; i < columns; i++)
      if (mpz_sgn(get_entry(i, l)) ≠ 0) {
        snd_nonzero[l] = i;
        break;
      }
  }
  printf("Second␣non-zero␣entries:\n");
  j = 0;
  for (l = 0; l < columns; l++) {
    sndp[l] = j;
    snd_nonzero_in_column[j] = 0;
    j++;
    for (i = 0; i < rows; i++) {
      if (snd_nonzero[i] ≡ l) {
        snd_nonzero_in_column[j] = i;
        snd_nonzero_in_column[sndp[l]]++;
        j++;
      }
    }
    printf("%d␣", snd_nonzero_in_column[sndp[l]]);
  }
  printf(":␣%d\n\n", rows);
  sndp[columns] = j;
  snd_nonzero_in_column[j] = 0;
#if 0     /* Display gaps between first non-zero entry and second non-zero entry. */
  for (l = 0; l < rows; l++) {
    printf("%d␣", snd_nonzero[l] - first_nonzero[l]);
  }
  printf("\n");
#endif
```

This code is used in section 86.

108.

```
⟨ globals for enumeration 87 ⟩ +≡
  long only_zeros_no, only_zeros_success, hoelder_no, hoelder_success;
  long cs_success;
  long N_success;
  long N2_success;
  long N3_success;
```

109. ⟨ more initialization 109 ⟩ ≡

```
level = first_nonzero[rows − 1];
if (level < 0) level = 0;
level_max = level;
us[level] = 1;
#if 0
mpz_set_si(v[level], 1);
#else
v[level] = 1;
#endif
only_zeros_no = only_zeros_success = 0;
hoelder_no = hoelder_success = 0;
cs_success = nosolutions = loops = 0;
N_success = 0;
N2_success = 0;
N3_success = 0;
```

This code is used in section 86.

110. Increase the loop counter and test if we already can stop after collecting enough solutions.

⟨ increase loop counter 110 ⟩ ≡

```
loops++;
if ((stop_after_loops > 0) ∧ (stop_after_loops ≤ loops)) goto afterloop;
#ifndef NO_OUTPUT
#if VERBOSE > −1
if (loops % 100000000 ≡ 0) {      /* 10000000 */
  printf("%ld␣loops,␣solutions:␣%ld␣", loops, nosolutions);
#if FINCKEPOHST
  printf("fipo:␣%ld␣", fipo_success);
#endif
#if EIGENBOUND
  printf("eig_bound:␣%ld␣", eig_cut);
#endif
  printf("pruneN:␣%ld␣", N2_success);
  printf("pruneN2:␣%ld␣", N3_success);
  printf("\n");
#if 0     /* Write statistics about enumeration levels */
  for (i = 0; i < level_max; i++) {
    printf("%03d:␣%ld\n", i, nlow[i]);
  }
#endif      /* printf("us:␣"); for (i = columns − 1; i ≥ 0; i−−) { printf("(%d,%d)", i, (int) us[i]); }
        printf("\n"); */
  fflush(stdout);
}
#endif
#endif
```

This code is used in section 111.

111. The search loop with various pruning tests.

```
⟨ the loop of the exhaustive enumeration 111 ⟩ ≡
  do { ⟨ increase loop counter 110 ⟩;
  ⟨ compute new cs 112 ⟩;
  if ((cs[level] < Fd)      /* ∧ (¬prune0(fabs(dum[level]), N[level])) */
  ) {
#if FINCKEPOHST
#if 1
    if (fabs(us[level]) > fipo[level]) {
#else
      if (level ≠ columns − 1 ∧ (us[level] > fipo_UB[columns][level] ∨ us[level] < fipo_LB[columns][level]))
      {
#endif
        fipo_success++;
        goto side_step;
      }
#endif
#if EIGENBOUND
      if (level < columns − 1 ∧ (us[level] < bnd_lo[level][level] ∨ us[level] > bnd_up[level][level])) {
        printf("%d:\t%0.3lf␣%0.3lf␣%0.3lf␣->␣cut\n", level, bnd_lo[level][level], us[level],
            bnd_up[level][level]);
        goto side_step;
      }
#endif
      if (isSideStep) {
        compute_w2(w, bd, stepWidth, level, rows);
      }
      else {
        compute_w(w, bd, dum[level], level, rows);
      }
      if (level > 0) {
        ⟨ not at a leave 113 ⟩;
      }
      else {
        ⟨ at level = 0 114 ⟩;
      }
    }
    else {
      cs_success++;
    step_back:    /* Up: we go to level + 1. */
      nlow[level]++;
      level++;
      if (level_max < level) level_max = level;
    side_step:    /* Side step: the next value in the same level is chosen. */
      if (eta[level] ≡ 0) {
        delta[level] *= −1;
        if (delta[level] * d[level] ≥ 0) delta[level] += d[level];
      }
      else {
        delta[level] += (delta[level] * d[level] ≥ 0) ? d[level] : −d[level];
      }
#if 0
```

```
        us[level] = mpz_get_d(v[level]) + delta[level];
#else
        us[level] = v[level] + delta[level];
#endif
        isSideStep = 1;
      }
    }
    while (level < columns) ;
    afterloop:
```

This code is used in section 86.

112. Compute the square of the euclidean length of the projection.

```
⟨ compute new cs 112 ⟩ ≡
    olddum = dum[level];
    dum[level] = us[level] + y[level];
    cs[level] = cs[level + 1] + dum[level] * dum[level] * c[level];
    if (isSideStep) {
      stepWidth = dum[level] − olddum;
    }
```

This code is used in section 111.

113. We are not at a leave. We test, if we can prune the enumeration, otherwise we decrease the *level*.

```
⟨ not at a leave 113 ⟩ ≡
  i = prune_only_zeros(w, level, rows, Fq, first_nonzero_in_column, firstp, bd, y, us, columns);
  if (i < 0) {
    goto step_back;
  }
  else if (i > 0) {
    goto side_step;
  }
  i = prune_snd_nonzero(columns, rows, level, Fq, first_nonzero, snd_nonzero_in_column, sndp, us, cons);
  if (i > 0) {
    goto side_step;
  }
#if 0
  printf("%d␣", level);
#endif
  if (prune(w[level], cs[level], rows, Fqeps)) {
    if (eta[level] ≡ 1) {
      goto step_back;
    }
    eta[level] = 1;
    delta[level] *= −1;
    if (delta[level] * d[level] ≥ 0) delta[level] += d[level];
#if 0
    us[level] = mpz_get_d(v[level]) + delta[level];
#else
    us[level] = v[level] + delta[level];
#endif
  }
  else {
#if 0
    if (pruneN(w, cs, level, rows, columns, Fq)) {
      goto side_step;
    }
#endif
#if EIGENBOUND
    if (2 * level > 0.0 * columns) {
      ⟨ compute new Eigen bound 105 ⟩;
#if 0
      if (cs[level] + eig_bound[level] > Fd) {
        printf("%d:\t%0.3lf␣%0.3lf␣%0.3lf␣%0.1lf␣->␣cut\n", level, cs[level], Fd,
            Fd − eig_bound[level], us[level]);
      }
      else {
        printf("%d:\t%0.3lf␣%0.3lf␣%0.3lf␣%0.1lf␣\n", level, cs[level], Fd, Fd − eig_bound[level],
            us[level]);
      }
      if (cs[level] + eig_bound[level] > Fd) {
        eig_cut++;
        goto side_step;
      }
#endif
```

```
    }
#endif
    level−−;
#if 0
    fipo_LB[columns][level] = −fipo[level];
    fipo_UB[columns][level] = fipo[level];
#endif
    delta[level] = eta[level] = 0;
    y[level] = compute_y(mu_trans, us, level, level_max);
#if 0
    mpz_set_d(v[level], ROUND(−dum));
    us[level] = mpz_get_si(v[level]);
#else
    us[level] = v[level] = ROUND(−y[level]);
#endif
    d[level] = (v[level] > −y[level]) ? −1 : 1;
    isSideStep = 0;
  }
```

This code is used in section 111.

114. We arrived at $level = 0$. We test if we really got a solution and print it.

```
⟨ at level = 0 114 ⟩ ≡
  if (exacttest(w[0], rows, Fq) ≡ 1) {
    print_solution(w[level], rows, Fq, us, columns);
#if 0
    printf("us:␣");
    for (i = columns − 1; i ≥ 0; i−−) {
      printf("%d␣", (int) us[i]);
    }
    printf("after␣%ld␣loops\n", loops);
#endif
    if ((stop_after_solutions > 0) ∧ (stop_after_solutions ≤ nosolutions)) goto afterloop;
  }
  goto side_step;
```

This code is used in section 111.

115. Additional output at the end of the enumeration.

```
⟨ final output 115 ⟩ ≡
#ifndef NO_OUTPUT
  printf("Prune_cs: %ld\n", cs_success);
  printf("Prune_only_zeros: %ld of %ld\n", only_zeros_success, only_zeros_no);
  printf("Prune_hoelder: %ld of %ld\n", hoelder_success, hoelder_no);
  printf("Prune_N: %ld\n", N_success);
  printf("Prune_N2: %ld\n", N2_success);
  printf("Prune_N3: %ld\n", N3_success);
#if FINCKEPOHST
  printf("Fincke-Pohst: %ld\n", fipo_success);
#endif
#if EIGENBOUND
  printf("Eigen bound: %ld\n", eig_cut);
#endif
  printf("Loops: %ld\n", loops);
#endif
  if ((stop_after_solutions ≤ nosolutions ∧ stop_after_solutions > 0) ∨ (stop_after_loops ≤
      loops ∧ stop_after_loops > 0)) {
#ifndef NO_OUTPUT
    printf("Stopped after number of solutions: %ld\n", nosolutions);
#endif
    ⟨ close solution file 118 ⟩;
    if ((stop_after_loops ≤ loops ∧ stop_after_loops > 0)) {
      exit(10);
    }
    else {
      exit(9);
    }
  }
  else {
#ifndef NO_OUTPUT
    printf("Total number of solutions: %ld\n", nosolutions);
#endif
  }
#ifndef NO_OUTPUT
  printf("\n"); fflush(stdout);
#endif
```

This code is used in section 86.

116. ⟨ free allocated memory for enumeration 116 ⟩ ≡

```
free(us);
free(cs);
free(y);
free(delta);
free(d);
free(first_nonzero);
free(first_nonzero_in_column);
free(firstp);
free(snd_nonzero);
free(snd_nonzero_in_column);
free(sndp);
free(cons);
free(eta);
free(v);
for (l = 0; l ≤ columns; l++) free(w[l]);
free(w);
free(original_columns);
#if FINCKEPOHST
free(fipo);
for (l = 0; l < columns; l++) free(muinv[l]);
free(muinv);
#endif
#if EIGENBOUND
for (l = 0; l < columns; l++) free(Rinv[l]);
free(Rinv);
for (l = 0; l < columns; l++) free(R[l]);
free(R);
for (l = 0; l < columns; l++) free(eig_f[l]);
free(eig_f);
for (l = 0; l < columns; l++) free(eig_RinvR[l]);
free(eig_RinvR);
free(eig_min);
free(eig_bound);
for (l = 0; l < columns; l++) free(Rinvi[l]);
free(Rinvi);
for (l = 0; l < columns; l++) free(bnd_up[l]);
free(bnd_up);
for (l = 0; l < columns; l++) free(bnd_lo[l]);
free(bnd_lo);
#endif
lllfree(mu, c, N, bd, columns);
for (l = 0; l < columns; l++) free(mu_trans[l]);
free(mu_trans);
```

This code is used in section 86.

117. ⟨ open solution file 117 ⟩ ≡

```
fp = solfile;
if (SILENT) fprintf(fp, "SILENT\n");
fflush(fp);
```

This code is used in section 3.

118. ⟨ close solution file 118 ⟩ ≡

```
if (SILENT) fprintf(fp, "%ld␣solutions\n", nosolutions);
fflush(fp);
```

This code is used in sections 3, 115, and 135.

119. ⟨ global variables 7 ⟩ +≡

```
static FILE *fp;
```

120.

```
⟨ some vector computations 120 ⟩ ≡
  DOUBLE compute_y(DOUBLE **mu_trans, DOUBLE *us, int level, int level_max)
  {
#if BLAS
    return cblas_ddot(level_max − level, &(us[level + 1]), 1, &(mu_trans[level][level + 1]), 1);
#else
    int i;
    DOUBLE dum;
    i = level_max;
    dum = 0.0;
    while (i ≥ level + 1) {
      dum += mu_trans[level][i] * us[i];
      i−−;
    }
    return dum;
#endif
  }
  void compute_w2(DOUBLE **w, DOUBLE **bd, DOUBLE alpha, int level, int rows){ #
      if BLAS cblas_daxpy(rows, alpha, bd[level], 1, w[level], 1);
      #
      else int i;
        for (i = 0; i < rows; ++i) {
          w[level][i] += alpha * bd[level][i];
        }
      # endif return; } void compute_w(DOUBLE **w, DOUBLE **bd, DOUBLE alpha, int
            level, int rows){
#if USE_SSE
          __m128d a, x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4;
          int l;
          a = _mm_loaddup_pd(&alpha);
          for (l = 0; l < rows; l += 8) {      /* w[level][l] = w[level + 1][l] + alpha * bd[level][l]; */
            x1 = _mm_load_pd(&(bd[level][l]));
            y1 = _mm_load_pd(&(w[level + 1][l]));
            x2 = _mm_load_pd(&(bd[level][l + 2]));
            y2 = _mm_load_pd(&(w[level + 1][l + 2]));
            x3 = _mm_load_pd(&(bd[level][l + 4]));
            y3 = _mm_load_pd(&(w[level + 1][l + 4]));
            x4 = _mm_load_pd(&(bd[level][l + 6]));
            y4 = _mm_load_pd(&(w[level + 1][l + 6]));
            x1 = _mm_mul_pd(x1, a);
            z1 = _mm_add_pd(x1, y1);
            _mm_storeu_pd((double *) &(w[level][l]), z1);
            x2 = _mm_mul_pd(x2, a);
            z2 = _mm_add_pd(x2, y2);
            _mm_storeu_pd((double *) &(w[level][l + 2]), z2);
            x3 = _mm_mul_pd(x3, a);
            z3 = _mm_add_pd(x3, y3);
            _mm_storeu_pd((double *) &(w[level][l + 4]), z3);
            x4 = _mm_mul_pd(x4, a);
            z4 = _mm_add_pd(x4, y4);
```

```
    _mm_storeu_pd((double *) &(w[level][l + 6]), z4);
  }
  return;
#else
  #
  if BLAS cblas_dcopy(rows, w[level + 1], 1, w[level], 1);
  cblas_daxpy(rows, alpha, bd[level], 1, w[level], 1); #
  else int l;
    l = rows - 1;
  while (l ≥ 0) {
    w[level][l] = w[level + 1][l] + alpha * bd[level][l];
    l--;
  }
  # endif return;
#endif
  }
```

This code is used in section 86.

121. The algorithms of Gram-Schmidt and Givens are used for orthogonalization.

```
⟨ orthogonalization 121 ⟩ ≡
  ⟨ Gram-Schmidt algorithm 122 ⟩;
  ⟨ Givens algorithm 123 ⟩;
```

This code is used in section 86.

122. Orthogonalization with Gram-Schmidt.

```
⟨ Gram-Schmidt algorithm 122 ⟩ ≡
  void gramschmidt(COEFF **lattice, int columns, int rows, DOUBLE **mu, DOUBLE
          **bd, DOUBLE *c, DOUBLE *N, DOUBLE Fq)
  {
    int i, l, j;
    DOUBLE dum;
    for (i = 0; i < columns; i++) {
      for (l = 0; l < rows; l++) bd[i][l] = (DOUBLE) mpz_get_d(get_entry(i, l));
      N[i] = 0.0;
      for (j = 0; j < i; j++) {
        dum = 0.0;
        for (l = 0; l < rows; l++) dum += (DOUBLE) mpz_get_d(get_entry(i, l)) * bd[j][l];
        mu[i][j] = dum/c[j];
        for (l = 0; l < rows; l++) bd[i][l] -= mu[i][j] * bd[j][l];
      }
      c[i] = scalarproductfp(bd[i], bd[i], rows);
      for (l = 0; l < rows; l++) N[i] += fabs(bd[i][l]);
      N[i] /= c[i];
      N[i] *= Fq;
#ifndef NO_OUTPUT
#if VERBOSE > 0
      printf("%lf␣", (double) c[i]);
#endif
#endif      /* N[i] *= (1.0 + EPSILON); */
    }
#ifndef NO_OUTPUT
#if VERBOSE > 0
    printf("\n\n"); fflush(stdout);
#endif
#endif
    return;
  }
```

This code is used in section 121.

123. Orthogonalization with Givens rotation.

```
⟨ Givens algorithm 123 ⟩ ≡
  void givens(COEFF **lattice, int columns, int rows, DOUBLE **mu, DOUBLE **bd, DOUBLE
        *c, DOUBLE *N, DOUBLE Fq)
  {
    int i, l, j;
    int mm;
    DOUBLE d1, d2;
    DOUBLE gc, gs;
    DOUBLE t;      /* The matrix b is copied to mu. bd is set to a z × z unity matrix. */
    for (i = 0; i < columns; i++) {
      for (l = 0; l < rows; l++) {
        mu[i][l] = (DOUBLE) mpz_get_d(get_entry(i, l));
      }
    }
    for (i = 0; i < rows; i++) {
      for (l = 0; l < rows; l++) bd[i][l] = 0.0;
      bd[i][i] = 1.0;
    }
    for (j = 1; j < rows; j++) {      /* The Givens rotation */
      mm = (j < columns) ? j : columns;
      for (i = 0; i < mm; i++) {
        if (mu[i][j] ≡ 0.0) {      /* Nothing has to be done */
          gc = 1.0;
          gs = 0.0;
        }
        else {      /* Stable computation of the rotation coefficients. */
          if (fabs(mu[i][j]) ≥ fabs(mu[i][i])) {
            t = mu[i][i]/mu[i][j];
            gs = 1.0/SQRT(1.0 + t * t);
            gc = gs * t;
          }
          else {
            t = mu[i][j]/mu[i][i];
            gc = 1.0/SQRT(1.0 + t * t);
            gs = gc * t;
          }      /* Rotation of mu */
          for (l = i; l < columns; l++) {
            d1 = mu[l][i];
            d2 = mu[l][j];
            mu[l][i] = gc * d1 + gs * d2;
            mu[l][j] = −gs * d1 + gc * d2;
          }      /* Rotation of the matrix Q^t */
          for (l = 0; l < rows; l++) {
            d1 = bd[i][l];
            d2 = bd[j][l];
            bd[i][l] = gc * d1 + gs * d2;
            bd[j][l] = −gs * d1 + gc * d2;
          }
        }
      }
    }      /* Finally some scaling has to be done, since Q is a orthonormal matrix */
```

```
    for (i = 0; i < columns; i++) {
      c[i] = mu[i][i] * mu[i][i];
      N[i] = 0.0;
      for (j = 0; j < rows; j++) {
        bd[i][j] *= mu[i][i];
        N[i] += fabs(bd[i][j]);
      }
      N[i] /= c[i];
      N[i] *= Fq;      /* N[i] *= 1.0 + EPSILON; */
      for (j = columns - 1; j ≥ i; j--) mu[j][i] /= mu[i][i];
#ifndef NO_OUTPUT
#if VERBOSE > -1
      printf("%6.3f␣", (double) c[i]);
      if (i > 0 ∧ i % 15 ≡ 0) printf("\n");
#endif
#endif
    }
#ifndef NO_OUTPUT
#if VERBOSE > -1
    printf("\n\n"); fflush(stdout);
#endif
#endif
    return;
  }
```

This code is used in section 121.

124. The upper triangular matrix μ is inverted.

```
⟨ matrix inversion for Fincke-Pohst 124 ⟩ ≡
#if FINCKEPOHST
  void inverse(DOUBLE **mu, DOUBLE **muinv, int columns)
  {
    int i, j, k;
    DOUBLE sum;
    for (j = 0; j < columns; j++)
      for (i = j; i ≥ 0; i--) {
        sum = 0.0;
        for (k = i + 1; k < columns; k++) sum += mu[k][i] * muinv[k][j];
        if (i ≡ j) muinv[i][j] = 1.0 - sum;
        else muinv[i][j] = -sum;
      }
    return;
  }
#endif
```

This code is used in section 86.

125. There are several pruning methods.

```
⟨ pruning subroutines and output 125 ⟩ ≡
  ⟨ exact test 126 ⟩;
  ⟨ simple bound 127 ⟩;
  ⟨ pruning with Hölder 128 ⟩;
  ⟨ prune zeros in row 129 ⟩;
  ⟨ prune second nonzero in row 130 ⟩;
  ⟨ print a solution 132 ⟩;
  ⟨ Jacobi method 131 ⟩;
```

This code is used in section 86.

126. Exact test of all entries on $level = 0$.

```
⟨ exact test 126 ⟩ ≡
  int exacttest(DOUBLE *v, int rows, DOUBLE Fq)
  {
    int i;
    i = rows − 1;
    do {
      if (fabs(v[i]) > Fq + EPSILON) {
        return 0;
      }
      i−−;
    } while (i ≥ 0);
    return 1;
  }
```

This code is used in section 125.

127. Additional pruning. It seems to be faster, not to use this test.

```
⟨ simple bound 127 ⟩ ≡
  int prune0(DOUBLE li, DOUBLE re)
  {
    if (li > re * (1 + EPSILON)) {
      N_success++;
      return 1;
    }
    else {
      return 0;
    }
  }
```

This code is used in section 125.

128. Pruning according to Hölders inequality.

```
⟨ pruning with Hölder 128 ⟩ ≡
  int prune(DOUBLE *w, DOUBLE cs, int rows, DOUBLE Fqeps)
  {     /* hoelder_no++; */
#if BLAS
    if (cs < Fqeps * cblas_dasum(rows, w, 1)) return 0;
#else
    DOUBLE reseite;
    int i;
    reseite = 0.0;      /* -cs/Fqeps; */      /* *(1 - eps) */
    i = rows - 1;
    do {
      reseite += fabs(w[i]);
      i--;
    } while (i ≥ 0);
    if (cs < Fqeps * reseite) return 0;
#endif      /* hoelder_success++; */
    return 1;
  }
  int pruneN(DOUBLE **w, DOUBLE *cs, int t, int rows, int cols, DOUBLE Fq)
  {
    int i, t_up;
    DOUBLE sum;
    DOUBLE r;
    if (t ≥ cols - 2) return 0;
    if (t < cols/2 + 10) return 0;
    t_up = t + 1;      /* cols - 1; */
    r = 0.4;
    sum = 0.0;
    for (i = 0; i < rows; i++) {
      sum += fabs(w[t][i] - r * w[t_up][i]);
    }
    sum *= Fq * (1.000001);
    sum -= fabs(cs[t] - r * cs[t_up]);
    if (sum < 0.0) {
      N2_success++;
#if 0
      printf("PRUNEN␣%d␣%d␣%0.3lf\n", t, t_up, r);
#endif
      return 1;
    }
    t_up = t + 2;      /* cols - 1; */
    r = 0.4;
    sum = 0.0;
    for (i = 0; i < rows; i++) {
      sum += fabs(w[t][i] - r * w[t_up][i]);
    }
    sum *= Fq * (1.000001);
    sum -= fabs(cs[t] - r * cs[t_up]);
    if (sum < 0.0) {
      N2_success++;
```

```
#if 0
      printf("PRUNEN␣%d␣%d␣%0.3lf\n", t, t_up, r);
#endif
      return 1;
    }
    t_up = cols − 1;
    r = 0.2;
    sum = 0.0;
    for (i = 0; i < rows; i++) {
      sum += fabs(w[t][i] − r ∗ w[t_up][i]);
    }
    sum ∗= Fq ∗ (1.000001);
    sum −= fabs(cs[t] − r ∗ cs[t_up]);
    if (sum < 0.0) {
      N2_success++;
#if 0
      printf("PRUNEN␣%d␣%d␣%0.3lf\n", t, t_up, r);
#endif
      return 1;
    }
    if (0 ∧ t > cols − 20) {
      r = 0.2;
      sum = 0.0;
      for (i = 0; i < rows; i++) {
        sum += fabs(w[t][i] − r ∗ w[t_up][i]);
      }
      sum ∗= Fq ∗ (1.000001);
      sum −= fabs(cs[t] − r ∗ cs[t_up]);
      if (sum < 0.0) {
        N3_success++;
#if 0
        printf("PRUNEN␣%d␣%d␣%0.3lf\n", t, t_up, r);
#endif
        return 1;
      }
      r = 0.3;
      sum = 0.0;
      for (i = 0; i < rows; i++) {
        sum += fabs(w[t][i] − r ∗ w[t_up][i]);
      }
      sum ∗= Fq ∗ (1.000001);
      sum −= fabs(cs[t] − r ∗ cs[t_up]);
      if (sum < 0.0) {
        N3_success++;
#if 0
        printf("PRUNEN␣%d␣%d␣%0.3lf\n", t, t_up, r);
#endif
        return 1;
      }
    }
    return 0;
  }
```

This code is used in section 125.

129. Prune if there remain only zeros in a not finished row. The function computes two bounds u_1 and u_2 which fulfill

$$|w + (u_{1,2} + y)\hat{b}| \le F_q$$

If in case of a zero/one problem both u_i's are non-integer values we can not reach integer values by addition of other integers. That means, we may abandon enumeration at that level and step back, i.e. return -1.

If one of the coordinates which has only zeroes to its left violates the bounds we make a side step. That is we return 1.

```
⟨ prune zeros in row 129 ⟩ ≡
  int prune_only_zeros(DOUBLE **w, int level, int rows, DOUBLE Fq, int *first_nonzero_in_column, int
          *firstp, DOUBLE **bd, DOUBLE *y, DOUBLE *us, int columns)
  {
    int i;
    int f;
    DOUBLE u1, u2, swp;
    only_zeros_no++;
    for (i = 0; i < first_nonzero_in_column[firstp[level]]; i++) {
      f = first_nonzero_in_column[firstp[level] + 1 + i];
      u1 = (Fq − w[level + 1][f])/bd[level][f] − y[level];
      u2 = (−Fq − w[level + 1][f])/bd[level][f] − y[level];
#if 0
      if (u2 < u1) {
        swp = u1;
        u1 = u2;
        u2 = swp;
      }
      fipo_LB[columns][level] = u1 − EPSILON;
      fipo_UB[columns][level] = u2 + EPSILON;
#endif
      if (iszeroone) {
        if (fabs(u1 − round(u1)) > EPSILON ∧ fabs(u2 − round(u2)) > EPSILON) {
          only_zeros_success++;
          return −1;
        }
        if (fabs(fabs(w[level][f]) − Fq) > EPSILON) {
          only_zeros_success++;
          return 1;
        }
#if 0
        if (fabs(u1 − us[level]) > EPSILON ∧ fabs(u2 − us[level]) > EPSILON) {
          return 1;
        }
#endif
      }
      else {      /* Not zero-one */      /* Here we have to be very conservative */
        if (u2−u1 ≤ 1.0+EPSILON∧fabs(w[level][f]) < UINT32_MAX∧fabs(w[level][f]−round(w[level][f])) >
              0.001) {
          only_zeros_success++;
          return −1;
        }
        if (fabs(w[level][f]) > Fq ∗ (1 + EPSILON)) {
          return 1;
```

```
        }
#if 0
        if (us[level] < u1 - EPSILON ∨ us[level] > u2 + EPSILON) {
          return 1;
        }
#endif
      }
#if 0
      if (iszeroone) {
        if (fabs(fabs(w[level][f]) − Fq) > EPSILON) {      /* only_zeros_success++; */
          return 1;
        }
      }
      else {
        if (fabs(w[level][f]) > Fq * (1 + EPSILON)) {
          only_zeros_success++;
          return 1;
        }
      }
#endif
    }
    return 0;
  }
```

This code is used in section 125.

130.

```
⟨ prune second nonzero in row 130 ⟩ ≡
  int prune_snd_nonzero(int columns, int rows, int level, DOUBLE Fq, int *first_nonzero, int
          *snd_nonzero_in_column, int *sndp, DOUBLE *us, struct constraint *cons)
  {
    int i, k;
    int ro;
    int f1;
    return 0;
    for (i = 0; i < snd_nonzero_in_column[sndp[level]]; i++) {
      ro = snd_nonzero_in_column[sndp[level] + 1 + i];
      f1 = first_nonzero[ro];
      if (level − f1 ≤ 5) {
        continue;
      }
      mpz_set_si(soltest_s, 0);
      for (k = level; k < columns; k++) {
        if (ROUND(us[k]) > 0) {
          mpz_addmul_ui(soltest_s, get_entry(k, ro), ROUND(us[k]));
        }
        else {
          mpz_submul_ui(soltest_s, get_entry(k, ro), −ROUND(us[k]));
        }
      }
      mpz_sub(snd_s, max_norm, soltest_s);
      mpz_fdiv_qr(snd_q, snd_r, snd_s, get_entry(f1, ro));
      if (mpz_sgn(snd_r) ≠ 0) {
        printf("Contradiction\n");
        return 1;
      }
#if 0
      if (cons[f1].isSet ≡ 0) {
        cons[f1].isSet = 1;
        cons[f1].val[0] = u1;
        cons[f1].val[1] = u2;
      }
#endif
    }
    return 0;
  }
```

This code is used in section 125.

131. Jacobi method to compute minimum Eigen value of the symmetric matrix $R^t opR$.

```
⟨ Jacobi method 131 ⟩ ≡
  DOUBLE Jacobi(DOUBLE **Ain, int n)
  {
    int i, j, k, nloops;
    DOUBLE aa, si, co, tt, eps;
    DOUBLE sum, ssum, amax, amin;
    DOUBLE **V;
    DOUBLE **A;
    sum = 0.0;
    eps = 0.000001;
    nloops = 0;
    V = (DOUBLE **) calloc(n, sizeof(DOUBLE *));
    for (i = 0; i < n; ++i) V[i] = (DOUBLE *) calloc(n, sizeof(DOUBLE));
    A = (DOUBLE **) calloc(n, sizeof(DOUBLE *));
    for (i = 0; i < n; ++i) A[i] = (DOUBLE *) calloc(n, sizeof(DOUBLE));
        /* Initialization. Set initial Eigenvectors. Compute A = Ain^top · Ain */
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        V[i][j] = 0.0;
        A[i][j] = 0.0;
        for (k = 0; k < n; k++) {
          A[i][j] += Ain[k][i] * Ain[k][j];
        }       /* A[i][j] = Ain[i][j]; */
        sum += fabs(A[i][j]);
      }
      V[i][i] = 1.0;
    }      /* Trivial problems */
    if (n ≡ 1) {
      return A[0][0];
    }
    if (sum ≤ 0.0) {
      return 0.0;
    }
    sum /= (n * n);       /* Reduce matrix to diagonal */
    do {
      ssum = 0.0;
      amax = 0.0;
      for (j = 1; j < n; j++) {
        for (i = 0; i < j; i++) {       /* Check if A[i][j] is to be reduced */
          aa = fabs(A[i][j]);
          if (aa > amax) {
            amax = aa;
          }
          ssum += aa;
          if (aa ≥ eps) {       /* calculate rotation angle */
            aa = atan2(2.0 * A[i][j], A[i][i] - A[j][j]) * 0.5;
            si = sin(aa);
            co = cos(aa);       /* Modify i and j columns */
            for (k = 0; k < n; k++) {
              tt = A[k][i];
              A[k][i] = co * tt + si * A[k][j];
```

```
            A[k][j] = -si * tt + co * A[k][j];
            tt = V[k][i];
            V[k][i] = co * tt + si * V[k][j];
            V[k][j] = -si * tt + co * V[k][j];
          }     /* Modify diagonal terms */
          A[i][i] = co * A[i][i] + si * A[j][i];
          A[j][j] = -si * A[i][j] + co * A[j][j];
          A[i][j] = 0.0;     /* Make A matrix symmetrical */
          for (k = 0; k < n; k++) {
            A[i][k] = A[k][i];
            A[j][k] = A[k][j];
          }     /* A[i][j] made zero by rotation */
        }
      }
    }
    nloops++;
  } while (fabs(ssum)/sum > eps ∧ nloops < 100000);
  amin = A[0][0];
  for (i = 1; i < n; i++)
    if (A[i][i] < amin) amin = A[i][i];
  for (i = 0; i < n; i++) free(A[i]);
  free(A);
  for (i = 0; i < n; i++) free(V[i]);
  free(V);
  return amin;
}
```

This code is used in section 125.

132. Output of solutions. There are difficulties if there are integers bigger than 64 bit integers, even in the multiprecision version.

```
⟨ print a solution 132 ⟩ ≡
  int print_solution(DOUBLE *w, int rows, DOUBLE Fq, DOUBLE *us, int columns){ int i, j, k;
      int upper;
      int end;      /* Test again, if the vector is really a solution */
#if 1
      if (fabs(fabs(w[rows − 1]) − Fq) > 0.5 * Fq * EPSILON) {
#else      /* Wrong, because it allows w[rows − 1] ≡ 0.0. Why did I ever use this? */
      if (fabs(w[rows − 1]) > Fq * (1 + EPSILON)) {
#endif
        return 0;
      }
      upper = rows − 1 − free_RHS;
#if 0
      if (free_RHS ∧ fabs(fabs(w[upper]) − Fq) > 0.5 * Fq * EPSILON) {
#else
        if (free_RHS ∧ fabs(w[upper]) > Fq * (1 + EPSILON)) {
#endif
          return 0;
        }
        if (¬SILENT) {
          mpz_set_si(soltest_upfac, 1);
#if 0
          mpz_set_d(soltest_s, ROUND(w[rows − 1]));
#else
          mpz_set_si(soltest_s, 0);
          for (k = 0; k < columns; k++) {
            if (ROUND(us[k]) > 0) {
              mpz_addmul_ui(soltest_s, get_entry(k, rows − 1), ROUND(us[k]));
            }
            else {
              mpz_submul_ui(soltest_s, get_entry(k, rows − 1), −ROUND(us[k]));
            }
          }
#endif
          i = 0;
          if (cut_after_coeff ≡ −1) {
            end = no_original_columns;
#if 0
            if (nboundvars ≠ 0) {      /* Conflicts with original_columns */
              end = nboundvars;
            }
#endif
          }
          else {
            end = cut_after_coeff;
          }
          for (j = 0; j < end; j++) {
            if (original_columns[j] ≡ 0) {
              mpz_set_si(soltest_u, 0);
            }
```

```
      else {
        if (¬iszeroone) {
          if (mpz_cmp_si(upperbounds[i], 0) ≠ 0) {
            mpz_divexact(soltest_upfac, upperbounds_max, upperbounds[i]);
          }
          else {
            mpz_set(soltest_upfac, upperbounds_max);
          }
        }
        mpz_set_si(soltest_u, 0);
        for (k = 0; k < columns; k++) {
          if (ROUND(us[k]) > 0) {
            mpz_addmul_ui(soltest_u, get_entry(k, i), ROUND(us[k]));
          }
          else {
            mpz_submul_ui(soltest_u, get_entry(k, i), −ROUND(us[k]));
          }
        }
        mpz_sub(soltest_u, soltest_u, soltest_s);
        mpz_divexact(soltest_u, soltest_u, max_norm_initial);
        mpz_divexact(soltest_u, soltest_u, soltest_upfac);
        mpz_divexact_ui(soltest_u, soltest_u, denom);
        mpz_abs(soltest_u, soltest_u);
        if (¬iszeroone ∧ (mpz_cmp_si(soltest_u, 0) < 0 ∨ mpz_cmp(soltest_u, upperbounds[i]) > 0)) {
            /* upperbounds not defined for 0/1 problems */
          fprintf(stderr, "␣rounding␣error␣->␣this␣is␣not␣a␣solution!\n");
          return 0;
        }
        i++;
      }
      mpz_out_str(Λ, 10, soltest_u);
      fflush(stdout);
      mpz_out_str(fp, 10, soltest_u);      /* Meanwhile, all solution vectors are written with
          separating blanks. */      /* if (¬iszeroone) { } */
      printf("␣");
      fprintf(fp, "␣");
    }
    if (free_RHS) {
      mpz_set_d(soltest_u, ROUND(w[i]));
      mpz_divexact(soltest_u, soltest_u, max_up);
      mpz_abs(soltest_u, soltest_u);
      printf("␣L␣=␣");
      mpz_out_str(Λ, 10, soltest_u);
    }
    printf("\n"); fflush(stdout);
    fprintf(fp, "\n"); fflush(fp);
  }
  nosolutions++;
#ifndef NO_OUTPUT
  if (nosolutions % 10000 ≡ 0) {
    printf("%ld\n", nosolutions); fflush(stdout);
  }
```

```
#endif
        return 1;
    }
```

This code is used in section 125.

133. Output of the polyhedra. We write the basis vectors as input for the polyhedra programs cdd and lrs.

```
⟨ output polyhedra 133 ⟩ ≡
  void basis2poly( )
  {
    return;
  }
```

This code is used in section 86.

134. Output of the LP-relaxation. We write the linear combination of the basis vectors as LP.

```
⟨ output LP 134 ⟩ ≡
  void basis2LP(double *low, double *up)
  {
    return;
  }
```

This code is used in section 86.

135. Stop the execution of the program and write out the number of solutions.

```
⟨ stop program 135 ⟩ ≡
  void stopProgram( )
  {
#ifndef NO_OUTPUT
    printf("Stopped after SIGALRM, number of solutions: %ld\n", nosolutions);
#endif
    ⟨ close solution file 118 ⟩;
    exit(11);
  }
```

This code is used in section 28.

136. Index.

⟨ initialize some globals 11, 42 ⟩ Used in section 3.
⟨ inline functions 10 ⟩ Used in section 2.
⟨ iterated-lll 73 ⟩ Used in section 47.
⟨ lattice basis reduction algorithms 47 ⟩ Used in section 2.
⟨ lll-subroutines 65 ⟩ Used in section 47.
⟨ local variables for *explicit_enumeration*( ) 89, 104 ⟩ Used in section 86.
⟨ matrix inversion for Fincke-Pohst 124 ⟩ Used in section 86.
⟨ memory allocation 68 ⟩ Used in section 65.
⟨ more initialization 109 ⟩ Used in section 86.
⟨ not at a leave 113 ⟩ Used in section 111.
⟨ old integration of the new vector, part 1 80 ⟩ Used in section 77.
⟨ old integration of the new vector, part 2 81 ⟩ Used in section 77.
⟨ open solution file 117 ⟩ Used in section 3.
⟨ orthogonal defect 71 ⟩ Used in section 47.
⟨ orthogonalization 121 ⟩ Used in section 86.
⟨ orthogonalize the basis 101 ⟩ Used in section 86.
⟨ output LP 134 ⟩ Used in section 86.
⟨ output polyhedra 133 ⟩ Used in section 86.
⟨ overall exhaustive enumeration 86 ⟩ Used in section 47.
⟨ permute lattice columns 20 ⟩ Used in section 3.
⟨ precompute η 84 ⟩ Used in section 82.
⟨ print NTL lattice 31 ⟩ Used in section 28.
⟨ print a solution 132 ⟩ Used in section 125.
⟨ print the lattice 30 ⟩ Used in section 28.
⟨ prune second nonzero in row 130 ⟩ Used in section 125.
⟨ prune zeros in row 129 ⟩ Used in section 125.
⟨ pruning subroutines and output 125 ⟩ Used in section 86.
⟨ pruning with Hölder 128 ⟩ Used in section 125.
⟨ read lattice written by NTL 32 ⟩ Used in section 28.
⟨ read the system 15 ⟩ Used in section 3.
⟨ recompute N_k 60 ⟩ Used in section 52.
⟨ round the Gram Schmidt coefficient 54 ⟩ Cited in sections 52 and 56. Used in section 52.
⟨ scalarproduct with doubles 67 ⟩ Used in section 65.
⟨ scalarproduct with integers 66 ⟩ Used in section 65.
⟨ scale last rows 24 ⟩ Cited in section 25. Used in section 3.
⟨ scale lattice 19 ⟩ Cited in section 16. Used in section 3.
⟨ second reduction 22 ⟩ Used in section 3.
⟨ second step: orthogonalization of b_k 51 ⟩ Used in section 48.
⟨ set $b_k = b_k - \lceil \mu_k, j \rfloor b_j$ 56 ⟩ Used in section 52.
⟨ set the lattice dimension 13 ⟩ Used in section 3.
⟨ set the simple pruning bounds 99 ⟩ Used in section 86.
⟨ shuffle lattice columns 37 ⟩ Used in section 28.
⟨ simple bound 127 ⟩ Used in section 125.
⟨ solutiontest 40 ⟩ Used in section 28.
⟨ some vector computations 120 ⟩ Used in section 86.
⟨ sort by Fincke-Pohst bounds 97 ⟩
⟨ sort lattice columns 96 ⟩ Used in section 86.
⟨ standard-lll 72 ⟩ Used in section 47.
⟨ stop program 135 ⟩ Used in section 28.
⟨ successful enumeration 77 ⟩ Used in section 75.
⟨ test for linear dependencies 61 ⟩ Used in section 52.
⟨ test for right hand side columns 39 ⟩ Used in section 36.

DIOPHANT