August 21, 2012 at 17:02

1. solvediophant – Solving Diophantine Linear Systems using GMP.

Return values:
– 0: normal program flow (reduction plus exhaustive enumeration)
– 1: Input error or internal error
– 2: Solution not possible, system not solvable over the reals. This may also come from parameter -c being too small
– 3: Program has been called with parameters -? or -h
– 8: Stopped after finding a random solution in phase one ("% stopafter: 1" has been set in the problem file)
– 9: Stopped after the maximum number of solutions ("% stopafter: n" has been set in the problem file)
– 10: Stopped after reaching the maximum number of solutions ("% stoploops: n" has been set in the problem file)

2. The main program.

```
format mpz_t long
format DOUBLE double
format COEFF int
#define zlength 16000
⟨ include header files 3 ⟩;
⟨ run time measurements 19 ⟩;
int main(int argc, char *argv[])
{
  int i, j, flag;
  ⟨ variables 4 ⟩;
  ⟨ read command line parameters 6 ⟩;
  ⟨ read the system size 11 ⟩;
  ⟨ allocate the matrix 12 ⟩;
  ⟨ read the linear system 13 ⟩;
  ⟨ read upper bounds 14 ⟩;
  ⟨ search preselected variables 15 ⟩;
#if 0  /* Not longer in use. Now, if u = 0, we multiply the column by R_max instead of c */
  ⟨ delete zero-bound variables 16 ⟩;
#endif
  solfile = fopen(solfilename, "w");
  time_0 = os_ticks( );
  diophant(A, rhs, upperb, no_columns, no_rows, factor_input, norm_input, silent, iterate, iterate_no,
      bkz_beta_input, bkz_p_input, stop_after_solutions, stop_after_loops, free_RHS, original_columns,
      no_original_columns, cut_after, nboundedvars, solfile);
  time_1 = os_ticks( );
  fclose(solfile);
  ⟨ final output of the run time 23 ⟩;
  ⟨ free the memory 5 ⟩;
  return 0;
}
```

3. ⟨ include header files 3 ⟩ ≡

```
#include <signal.h>
#include <stdio.h>
#include <gmp.h>
#include <stdlib.h>
#include <string.h>
#include <sys/times.h>      /* For run time measurements */
#include <unistd.h>
#include "diophant.h"
```

This code is used in section 2.

4. The variables. Global variables are not longer used in order to avoid conflicts with the global variables of *diophant*.

⟨ variables 4 ⟩ ≡

```
  mpz_t factor_input;
  mpz_t norm_input;
  mpz_t *upperb;
  mpz_t **A, *rhs;
  int bkz_beta_input = 0;
  int bkz_p_input = 0;
  int iterate = 0;
  int iterate_no = 0;
  int silent;
  int maxruntime = 0;
  int no_rows, no_columns;
  long stop_after_solutions;
  long stop_after_loops;
  int cut_after;
  int free_RHS;
  FILE *txt;
  char *inputfile_name, *rowp;
  char zeile[zlength];
  char detectstring[100];
  int *original_columns;
  int no_original_columns;
  int res = 1;
  int nboundedvars;
  FILE *solfile;
  char solfilename[1024];
  mpz_init(factor_input);
  mpz_init(norm_input);
```

See also sections 7 and 17.

This code is used in section 2.

5. ⟨ free the memory 5 ⟩ ≡

```
mpz_clear(factor_input);
mpz_clear(norm_input);
for (j = 0; j < no_rows; j++) {
  for (i = 0; i < no_columns; i++) mpz_clear(A[j][i]);
  free(A[j]);
}
free(A);
rhs = (mpz_t *) calloc(no_rows, sizeof(mpz_t));
for (i = 0; i < no_rows; i++) mpz_clear(rhs[i]);
free(rhs);
if (upperb ≠ Λ) {
  for (i = 0; i < nboundedvars; i++) {
    mpz_clear(upperb[i]);
  }
  free(upperb);
}
```

This code is used in section 2.

6. ⟨ read command line parameters 6 ⟩ ≡

```
strcpy(solfilename, "solutions");
iterate = −1;
bkz_beta_input = bkz_p_input = −1;
mpz_set_si(factor_input, −1);
mpz_set_si(norm_input, −1);
silent = 0;
for (i = 1; i < argc; i++) {
  ⟨ analyse the options 8 ⟩;
}
⟨ test command line parameters 9 ⟩;
inputfile_name = argv[argc − 1];
⟨ start alarm 10 ⟩;
```

This code is used in section 2.

7. ⟨ variables 4 ⟩ +≡

```
char suffix[1024];
```

8. ⟨ analyse the options 8 ⟩ ≡

```
  if (strcmp(argv[i], "-silent") ≡ 0) {
    silent = 1;
#ifndef NO_OUTPUT
    fprintf(stderr, "No output of solutions, just counting.\n");
#endif
  }
  else if (strncmp(argv[i], "-iterate", 8) ≡ 0) {
    strcpy(suffix, argv[i] + 8);
    iterate_no = atoi(suffix);
    iterate = 1;
  }
  else if (strncmp(argv[i], "-bkz", 4) ≡ 0) {
    iterate = 0;
  }
  else if (strncmp(argv[i], "-beta", 5) ≡ 0) {
    strcpy(suffix, argv[i] + 5);
    bkz_beta_input = atoi(suffix);
  }
  else if (strncmp(argv[i], "-p", 2) ≡ 0) {
    strcpy(suffix, argv[i] + 2);
    bkz_p_input = atoi(suffix);
  }
  else if (strncmp(argv[i], "-time", 5) ≡ 0) {
    strcpy(suffix, argv[i] + 5);
    maxruntime = atoi(suffix);
  }
  else if (strncmp(argv[i], "-c", 2) ≡ 0) {
    strcpy(suffix, argv[i] + 2);
#if 1
    mpz_set_str(factor_input, suffix, 10);      /* Regular version */
#else
    mpz_ui_pow_ui(factor_input, 10, atoi(suffix));      /* Version for the NTL output */
#endif
  }
  else if (strncmp(argv[i], "-maxnorm", 8) ≡ 0) {
    strcpy(suffix, argv[i] + 8);
    mpz_set_str(norm_input, suffix, 10);
  }
  else if (strncmp(argv[i], "-i", 2) ≡ 0) {
    strcpy(suffix, argv[i] + 2);
  }
  else if (strncmp(argv[i], "-o", 2) ≡ 0) {
    strcpy(solfilename, argv[i] + 2);
  }
  else if (strcmp(argv[i], "-?") ≡ 0 ∨ strcmp(argv[i], "-h") ≡ 0) {
#ifndef NO_OUTPUT
    fprintf(stderr, "\nsolvediophant ---␣multiple precision version ---␣\n");
    fprintf(stderr, "solvediophant");
    fprintf(stderr, "␣-iterate*|(-bkz␣-beta*␣-p*)␣[-c*]␣[-maxnorm*]␣[-time*]␣[-silent]␣[-o*]");
    fprintf(stderr, "␣inputfile\n\n");
#endif
```

```
    exit(3);
  }
```

This code is used in section 6.

9. ⟨ test command line parameters 9 ⟩ ≡

```
  if (argc < 2 ∨ strncmp(argv[argc − 1], "-", 1) ≡ 0) {
#ifndef NO_OUTPUT
    fprintf(stderr, "The␣last␣parameter␣on␣the␣command␣line\n");
    fprintf(stderr, "has␣to␣be␣the␣input␣file␣name.\n");
#endif
    exit(1);
  }
  if (iterate ≡ −1) {
#ifndef NO_OUTPUT
    fprintf(stderr, "No␣reduction␣was␣chosen.\n");
    fprintf(stderr, "It␣is␣set␣to␣iterate=1.\n");
#endif
    iterate = 1;
    iterate_no = 1;
  }
  if (iterate ≡ 0 ∧ (bkz_beta_input ≡ −1 ∨ bkz_p_input ≡ −1)) {
#ifndef NO_OUTPUT
    fprintf(stderr, "You␣have␣chosen␣bkz␣reduction.␣You␣also␣have␣to␣specify␣the␣parameters");
    fprintf(stderr, "␣-beta*␣-p*\n");
#endif
    exit(1);
  }
  if (mpz_cmp_si(factor_input, 0) ≤ 0) {
#ifndef NO_OUTPUT
    fprintf(stderr, "You␣did␣not␣supply␣the␣options␣-c*.␣");
    fprintf(stderr, "It␣is␣set␣to␣10000000000000.\n");
#endif
    mpz_set_str(factor_input, "10000000000000", 10);
  }
  if (mpz_cmp_si(norm_input, 0) ≤ 0) {
#ifndef NO_OUTPUT
    fprintf(stderr, "You␣did␣not␣supply␣the␣options␣-maxnorm*.␣");
    fprintf(stderr, "It␣is␣set␣to␣1.\n");
#endif
    mpz_set_si(norm_input, 1);
  }
```

This code is used in section 6.

10. With alarm a maximal run time can be given.

⟨ start alarm 10 ⟩ ≡

```
  if (maxruntime > 0) {
    signal(SIGALRM, stopProgram);
    alarm(maxruntime);
  }
```

This code is used in section 6.

11. Open the input file and read the size of the Diophantine linear system and some other control parameters.

```
⟨ read the system size 11 ⟩ ≡
  txt = fopen(inputfile_name, "r");
  if (txt ≡ Λ) {
#ifndef NO_OUTPUT
    printf("Could␣not␣open␣file␣'%s'!\n", inputfile_name);
    fflush(stdout);
#endif
    exit(1);
  }
  flag = 0;
  free_RHS = 0;
  stop_after_loops = 0;
  stop_after_solutions = 0;
  cut_after = −1;
  do {
    fgets(zeile, zlength, txt);
    if (strstr(zeile, "%␣stopafter") ≠ Λ) {
      sscanf(zeile, "%%␣stopafter␣%ld", &stop_after_solutions);
    }
    if (strstr(zeile, "%␣stoploops") ≠ Λ) {
      sscanf(zeile, "%%␣stoploops␣%ld", &stop_after_loops);
    }
    if (strstr(zeile, "%␣cutafter") ≠ Λ) {
      sscanf(zeile, "%%␣cutafter␣%d", &cut_after);
    }
    if (strstr(zeile, "%␣FREERHS") ≠ Λ) {
      free_RHS = 1;
    }
  } while (zeile[0] ≡ '%');
  sscanf(zeile, "%d%d%d", &no_rows, &no_columns, &flag);
```

This code is used in section 2.

12. ⟨ allocate the matrix 12 ⟩ ≡

```
  A = (mpz_t **) calloc(no_rows, sizeof(mpz_t *));
  for (j = 0; j < no_rows; j++) {
    A[j] = (mpz_t *) calloc(no_columns, sizeof(mpz_t));
    for (i = 0; i < no_columns; i++) mpz_init(A[j][i]);
  }
  rhs = (mpz_t *) calloc(no_rows, sizeof(mpz_t));
  for (i = 0; i < no_rows; i++) mpz_init(rhs[i]);
```

This code is used in section 2.

13. ⟨ read the linear system 13 ⟩ ≡

```
for (j = 0; j < no_rows; j++) {
  for (i = 0; i < no_columns; i++) {
    mpz_inp_str(A[j][i], txt, 10);
    if (res ≡ 0) {
      ⟨ incorrect input file 18 ⟩;
    }
  }
  res = mpz_inp_str(rhs[j], txt, 10);
  if (res ≡ 0) {
    ⟨ incorrect input file 18 ⟩;
  }
}
```

This code is used in section 2.

14. After searching for upper bounds the input file is closed and opened again.

```
⟨ read upper bounds 14 ⟩ ≡
  upperb = Λ;
  fclose(txt);
  txt = fopen(inputfile_name, "r");
  if (txt ≡ Λ) {
#ifndef NO_OUTPUT
    printf("Could␣not␣open␣file␣%s␣!\n", inputfile_name);
    fflush(stdout);
#endif
    exit(1);
  }
  zeile[0] = '\0';
  sprintf(detectstring, "BOUNDS");
  do {
    rowp = fgets(zeile, zlength, txt);
  } while ((rowp ≠ Λ) ∧ (strstr(zeile, detectstring) ≡ Λ));
  if (rowp ≡ Λ) {
    upperb = Λ;
#ifndef NO_OUTPUT
    printf("No␣%s␣\n", detectstring); fflush(stdout);
#endif
    nboundedvars = no_columns;
  }
  else {
    nboundedvars = no_columns;
    sscanf(zeile, "BOUNDS␣%d", &nboundedvars);
    if (nboundedvars > 0) {
#ifndef NO_OUTPUT
      fprintf(stderr, "Nr.␣bounded␣variables=%d\n", nboundedvars);
#endif
    }
    else {
      nboundedvars = 0;
    }
    upperb = (mpz_t *) calloc(no_columns, sizeof(mpz_t));
    for (i = 0; i < nboundedvars; i++) {
      mpz_init(upperb[i]);
      mpz_inp_str(upperb[i], txt, 10);
    }
  }
  fclose(txt);
  txt = fopen(inputfile_name, "r");
  if (txt ≡ Λ) {
#ifndef NO_OUTPUT
    printf("Could␣not␣open␣file␣%s␣!\n", inputfile_name);
    fflush(stdout);
#endif
    exit(1);
  }
```

This code is used in section 2.

15. Search for preselected variables and close the input file.

```
⟨ search preselected variables 15 ⟩ ≡
  sprintf(detectstring, "SELECTEDCOLUMNS");
  do {
    rowp = fgets(zeile, zlength, txt);
  } while ((rowp ≠ Λ) ∧ (strstr(zeile, detectstring) ≡ Λ));
  if (rowp ≠ Λ) {
#ifndef NO_OUTPUT
    printf("SELECTEDCOLUMNS␣detected\n"); fflush(stdout);
#endif
    res = fscanf(txt, "%d", &(no_original_columns));
    if (res ≡ (long) Λ ∨ res ≡ (long) EOF) {
      ⟨ incorrect input file 18 ⟩;
    }
  }
  else no_original_columns = no_columns;
  original_columns = (int *) calloc(no_original_columns, sizeof(int));
  if (rowp ≠ Λ) {
    for (i = 0; i < no_original_columns; i++) {
      res = fscanf(txt, "%d", &(original_columns[i]));
      if (res ≡ (long) Λ ∨ res ≡ (long) EOF) {
        ⟨ incorrect input file 18 ⟩;
      }
    }
  }
  else {
    for (i = 0; i < no_original_columns; i++) original_columns[i] = 1;
  }
  fclose(txt);
```

This code is used in section 2.

16.

```
⟨ delete zero-bound variables 16 ⟩ ≡
  if (upperb ≠ Λ) {
    for (i = nboundedvars − 1; i ≥ 1; i−−) {
      if (mpz_cmp_si(upperb[i], 0) ≡ 0) {
        for (j = i + 1; j < no_columns      /* nboundedvars */
        ; j++) {
          for (k = 0; k < no_rows; k++) {
            mpz_set(A[k][j − 1], A[k][j]);
          }
          mpz_set(upperb[j − 1], upperb[j]);
        }
        if (i < nboundedvars) nboundedvars−−;
        no_columns−−;
        k = l = 0;
        while (k < i) {
          if (original_columns[l] ≡ 1) {
            k++;
          }
          l++;
        }
        original_columns[l] = 0;
      }
    }
#ifndef NO_OUTPUT
    fprintf(stderr, "cols=%d\n", nboundedvars);
#endif
  }
```

This code is used in section 2.

17. ⟨ variables 4 ⟩ +≡

```
  ;
#if 0
  int k, l;
#endif
```

18. Incorrect or incomplete input file. We stop immediately.

```
⟨ incorrect input file 18 ⟩ ≡
#ifndef NO_OUTPUT
  fprintf(stderr, "Incomplete␣input␣file␣->␣exit\n");
  fflush(stderr);
#endif
  exit(1);
```

This code is used in sections 13 and 15.

19. Global variables and subroutines to measure the run time of the algorithm. The time is measured by calling *os_ticks*( ) before and after running *diophant*( ). *print_delta_time*( ) prints the run time. The other subroutines are just for formatting purposes.

```
⟨ run time measurements 19 ⟩ ≡
  int user_time, time_0, time_1;
  char timestring[256];
  ⟨ system calls 20 ⟩;
  ⟨ convert ticks to seconds 21 ⟩;
  ⟨ give time string 22 ⟩;
```

This code is used in section 2.

20. These are the system calls. *os_ticks*( ) gives the system ticks. *os_ticks_per_second*( ) gives the system dependent relation ticks vs. seconds.

```
⟨ system calls 20 ⟩ ≡
  int os_ticks( )
  {
    struct tms tms_buffer;
    if (−1 ≡ times(&tms_buffer)) return (−1);
    return (tms_buffer.tms_utime);
  }
  int os_ticks_per_second( )
  {
    int clk_tck = 1;
    clk_tck = sysconf(_SC_CLK_TCK);
    return (clk_tck);
  }
```

This code is used in section 19.

21. *tps* contains the system dependent number "ticks per second." The number of ticks are converted into seconds, minutes and hours.

```
⟨ convert ticks to seconds 21 ⟩ ≡
  int os_ticks_to_hms_tps(int ticks, int tps, int *h, int *m, int *s)
  {
    int l1;
    l1 = ticks/tps;      /* l1 is set to overall the number of seconds. */
    *s = l1 % 60;      /* number of seconds */
    l1 −= *s;
    l1 /= 60;
    *m = l1 % 60;      /* number of minutes */
    l1 −= *m;
    l1 /= 60;      /* number of hours */
    *h = l1;
    return (1);
  }
  int os_ticks_to_hms(int ticks, int *h, int *m, int *s)
  {
    os_ticks_to_hms_tps(ticks, os_ticks_per_second( ), h, m, s);
    return (1);
  }
```

This code is used in section 19.

22. ⟨ give time string 22 ⟩ ≡

```
void print_delta_time_tps(int l, int tps, char *str)
{
  int h, m, s;
  os_ticks_to_hms_tps(l, tps, &h, &m, &s);
  sprintf(str, "%d:%02d:%02d", h, m, s);
}
void print_delta_time(int l, char *str)
{
  print_delta_time_tps(l, os_ticks_per_second( ), str);
}
```

This code is used in section 19.

23. ⟨ final output of the run time 23 ⟩ ≡

```
user_time = time_1 - time_0;
timestring[0] = 0;
print_delta_time(user_time, timestring);
#ifndef NO_OUTPUT
fprintf(stderr, "total enumeration time: %s\n", timestring); fflush(stdout);
#endif
```

This code is used in section 2.

24. Index.

SOLVEDIOPHANT