C Quellcode

```
/*** bma.c ******************************************************/
/*                                                              */
/* Analysis of Boolean Maps (S-Boxes)  V 0.4                    */
/*                                                              */
/* Klaus Pommerening                                            */
/* Institut fuer Medizinische Biometrie, Epidemilogie und       */
/*    Informatik                                                */
/* Johannes-Gutenberg-Universitaet                              */
/* D-55101 Mainz                                                */
/* pom@imsd.uni-mainz.de                                        */
/*     21 August 2001 - last change 15 September 2001           */
/*                                                              */
/* Send bug reports and comments by e-mail.                     */
/****************************************************************/
/*                                                              */
/* Copyright by the author.                                     */
/* The use of this program code is free for private and         */
/*   educational use.                                           */
/* Usual disclaimers apply.                                     */
/****************************************************************/
/*                                                              */
/* Usage: bma <input >output                                    */
/*   [I. e. use input and output redirection]                   */
/*                                                              */
/* Input from stdin: Matrix of coefficients of a boolean map in */
/*   Algebraic Normal Form (ANF) --                             */
/*     one line per component function.                         */
/*     Let K = Galois Field with 2 elements.                    */
/*     A map f: K^n ---> K^q is given by q components,          */
/*     each component given by 2^n coefficients in ANF.         */
/*     [Input in lines instead of columns for convenience.]     */
/*                                                              */
/* Output to stdout:                                            */
/*   1. ANF -- one column per component function                */
/*   2. Truth table (one column per component function)         */
/*   3. Characteristic function as a 2^n x 2^q matrix           */
/*   4. Walsh transform of characteristic function as a 2^n x 2^q */
/*        matrix                                                */
/*   5. Linear profile                                          */
/*   6. Differential profile                                    */
/*   7. Linearity/nonlinearity measures:                        */
/*        linear potential, differential potential, nonlinearity  */
/*                                                              */
/* The bit string (b_1, ..., b_n) is identified with the integer  */
/*   b_1 2^{n-1} + ... + b_n 2^0.                               */
/****************************************************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/* Argument dimension n and image dimension q <= 8             */
/* Increase if needed. The program will allocate 5 arrays of   */
/*   length up to MAXLEN*MAXLEN = 2^{2*MAXDIM}.                */
/* MAXLEN must be 2^MAXDIM.                                    */
#define MAXDIM 8
#define MAXLEN 256
```

```
/*-------------------------------------------------------------*/
/* getInput: Read component coefficients x from stdin.         */
/*           Set dimension dim1 of argument space and          */
/*           dimension dim2 of image space.                    */
/*           (dim1 = base 2 log of input line length,          */
/*            dim2 = number of input lines.)                   */
/*   Memory for x is allocated inside.                         */
/*   Return codes:                                             */
/*     0 = Input o. K.                                         */
/*     1 = dim1 or dim2 exceeds MAXDIM.                        */
/*     2 = Input line has wrong length.                        */
/*     3 = Input contains value != 0 or 1.                     */
/*-------------------------------------------------------------*/

int getInput(unsigned ***x, unsigned *dim1, unsigned *dim2)
{
  unsigned      i, n;             /* Dimension                 */
  unsigned long m, k, j;          /* Length                    */
  char          buf[MAXLEN+2];
  char          *bufptr;

  i = 0;
  n = 0;
                                   /* 257 = MAXLENGTH + 1        */
  while ((scanf("%257s", buf) != EOF) & (i <= MAXDIM)) {
    if (i == MAXDIM) {
    fprintf(stderr, "bma.getInput: Too many lines.\n");
    return 1;
    }
    m = strlen(buf);
    if (m > MAXLEN) {
      fprintf(stderr, "bma.getInput: Line too long.\n");
      return 2;
      }
    if (i == 0) {                  /* First line                 */
      while (m > 1) {              /* Calculate dim1 and 2^dim1  */
        k = m >> 1;
        if ((k << 1) != m) {       /* m not a power of 2 */
          fprintf(stderr, "bma.getInput: Line length not a power of 2.\n");
          return 2;
          }
        n++;
        m = k;
        }                           /* Now n = dim1              */
        *dim1 = n;
        (*x) = (unsigned **) malloc(MAXDIM * sizeof(unsigned *));
        k = 1 << n;                 /* Line length 2^n           */
      }
    else {                          /* Follow-up lines           */
```

```
      if (m != k) {
        fprintf(stderr, "bma.getInput: Line lengths differ.\n");
        return 2;
        }
      } /* Now buf contains a line of good length ----------------*/

    (*x)[i] = (unsigned *) malloc(k * sizeof(unsigned));
    bufptr = buf;                    /* Cursor pointer into buf   */
    for (j = 0; j < k; j++) {
      sscanf(bufptr, "%1u", &((*x)[i][j]));
      if ((*x)[i][j] > 1) {
        fprintf(stderr, "bma.getInput: Input not in range.\n");
        return 3;
        }
      bufptr++;
      }
    i++;
    } /* end while -----------------------------------------------*/

  *dim2 = i;
  return 0;
}
```

```
/*--------------------------------------------------------------------*/
/* rev: Recursive Evaluation of a Boolean function f                  */
/*   Input: Array x of coefficients of Algebraic NF                   */
/*   Output: Truth table y of f                                       */
/* There is no error handling. The input is assumed to be correct.*/
/*--------------------------------------------------------------------*/
void rev(unsigned *x, unsigned *y, unsigned dim)
{
  unsigned      z[MAXLEN];
  unsigned long m, k, mi;
  unsigned      n, i;

  n = dim;
  m = 1 << n;                            /* length of array       */
  for (k = 0; k < m; k++) y[k] = x[k];
  mi = 1;
  for (i = 0; i < n; i++) {
    for (k = 0; k < m; k++)
      if ((k >> i) % 2) z[k] = y[k-mi] ^ y[k];
        else z[k] = y[k];
    for (k = 0; k < m; k++) y[k] = z[k];
    mi *= 2;
    }
  return;
}
```

```
/*------------------------------------------------------------------*/
/* wt: Walsh transform of an integer valued function phi on K^dim */
/*   Input: Array of values of phi                                  */
/*   Output: Array of values of Walsh transform                     */
/* There is no error handling. The input is assumed to be correct.*/
/*------------------------------------------------------------------*/
void wt(long *x, long *y, unsigned dim)
{
  long          z[MAXLEN*MAXLEN];
  unsigned long m, k, mi;
  unsigned      n, i;

  n = dim;
  m = 1 << n;                 /* Length of truth table              */
  for (k = 0; k < m; k++) y[k] = x[k];
  mi = 1;
  for (i = 0; i < n; i++) {
    for (k = 0; k < m; k++)
      if ((k >> i) % 2) z[k] = y[k-mi] - y[k];
        else z[k] = y[k] + y[k+mi];
    for (k = 0; k < m; k++) y[k] = z[k];
    mi *= 2;
    }
  return;
}
```

```
/*-----------------------------------------------------------------*/
/* printBin: Print the w lowest bits of the number k to stdout.   */
/* There is no error handling. The input is assumed to be correct.*/
/*-----------------------------------------------------------------*/

void printBin(unsigned w, unsigned long k)
{
  unsigned      i, b[MAXDIM];
  unsigned long p, x;
  p = 1;
  for (i = 0; i < w; i++) {
    x = k&p;
    if (x) b[w-i] = 1;
      else b[w-i] = 0;
    p = p*2;
    }
  for (i = 1; i <= w; i++) printf("%1d", b[i]);
}
```

```
/*-------------------------------------------------------------------*/
/* reduce: Divide the input vector x of length lt by the highest  */
/*   possible power 2^r of 2.                                      */
/* Return value: The exponent r.                                   */
/* There is no error handling. The input is assumed to be correct.*/
/*-------------------------------------------------------------------*/

int reduce(long *x, unsigned long lt)
{
  unsigned long i = 0, j;
  long          z;
  unsigned      r = 0;

  while (x[i] == 0) i++;
  if (i == lt) return 0;          /* All components of x were 0. */
  /* Now x[i] != 0 -----------------------------------------------*/

  z = x[i];                       /* Calculate start value for r */
  while (!(z%2)) {                /* z is even                   */
    z = z >> 1;
    r++;
    }
  /* Now r = max exponent with 2^r | x[i] ------------------------*/

  if (r == 0) return 0;           /* No sense in continuing      */
  for (j = i+1; j < lt; j++) {
    z = x[j];
    if (((z>>r)<<r) != z) {       /* Else r unchanged            */
      r = 0;                      /* Calculate new value for r   */
      while (!(z%2)) {
        z = z >> 1;
        r++;
        }
      if (r == 0) return 0;
      }
    }
  /* Now r = max exponent with 2^r | x[i], ..., x[lt-1] ----------*/
  /*   and r > 0. ------------------------------------------------*/

  for (j = 0; j < lt; j++) x[j] = x[j] >> r;
  return r;
}
```

```
/*--------------------------------------------------------------*/
/* max: Calculate the maximum entry of the input vector x        */
/*   between indices start and end-1, that is,                   */
/*     max(x[start], x[end-1]).                                  */
/* There is almost no error handling.                            */
/*   If end <= start, the function returns 0.                    */
/*   Otherwise the input is assumed to be correct.               */
/*--------------------------------------------------------------*/

long max(long *x, unsigned long start, unsigned long end)
{
  long          mm;
  unsigned long i;

  if (end <= start) return 0;

  mm = x[start];
  for (i = start+1; i < end; i++)
    if (x[i] > mm) mm = x[i];
  return mm;
}
```

```
/*-------------------------------------------------------------*/
/* prtTabl0: Print a 2^n x q matrix.                           */
/* There is no error handling. The input is assumed to be correct.*/
/*-------------------------------------------------------------*/

void prtTabl0(unsigned **x, unsigned dim1, unsigned dim2)
{
  unsigned long length1, length2, k, l;
  unsigned      i;
  length1 = 1 << dim1;
  length2 = dim2;

  for (i = 0; i <= dim1; i++) printf(" ");   /* Table header ...  */
  for (l = 1; l <= length2; l++) printf("%2d", l);
  printf("\n");                               /* ...               */
  for (i = 0; i <= dim1; i++) printf(" ");   /* ...               */
  for (l = 0; l < length2; l++) printf("--");
  printf("\n");                               /* ...               */

  for (k = 0; k < length1; k++) {          /* Print length1 lines: */
    printBin(dim1,k);                         /*   line header ... */
    printf("|");                              /*   ...             */
    for (l = 0; l < dim2; l++) printf(" %d", x[l][k]);
    printf("\n");
    }
}
```

```
/*--------------------------------------------------------------------*/
/* prtTabl1: Print a 2^n x 2^q matrix.                                */
/* There is no error handling. The input is assumed to be correct.*/
/*--------------------------------------------------------------------*/

void prtTabl1(long *x, unsigned dim1, unsigned dim2)
{
  unsigned long length1, length2, k, l;
  unsigned      i;
  length1 = 1 << dim1;
  length2 = 1 << dim2;

  for (i = 0; i <= dim1; i++) printf(" ");   /* Table header ...  */
  for (l = 0; l < length2; l++) {
    printf(" ");                             /* ...               */
    printBin(dim2,l);                        /* ...               */
    }
  printf("\n");                              /* ...               */
  for (i = 0; i <= dim1; i++) printf(" ");   /* ...               */
  for (l = 0; l < length2*dim2+length2; l++) printf("-");
  printf("\n");                              /* ...               */
  for (k = 0; k < length1; k++) {         /* Print length1 lines: */
    printBin(dim1,k);                        /*   line header ... */
    printf("|");                             /*   ...             */
    for (l = 0; l < length2; l++) {
      if (dim2 > 2) for (i = 0; i < dim2-2; i++) printf(" ");
      if (dim2 == 1) printf("%2d", x[(k<<dim2)+l]);
      else printf("%3d", x[(k<<dim2)+l]);
      }
    printf("\n");
    }
}
```

```
/*****************************************************************/
int main()
{
  unsigned      **a = NULL;        /* Array of coefficients       */
  unsigned      **f = NULL;        /* Array of values (truth tbl) */
    /* Note that a and f have their components arranged in rows - */
    /*   that's convenient for input and for calling rev on       */
    /*   the component functions.                                  */
  long          *g = NULL;         /* Characteristic function     */
  long          *c = NULL;         /* Walsh spectrum              */
  long          *lp = NULL;        /* Linear profile              */
  long          *dp = NULL;        /* Differential profile        */
  unsigned      n, q, i, j, n2;
  unsigned long m, p, k, l, y;
  int           rc;
  long          ll, dd, nl, maxnl;
  double        la;

/* Step 1: Algebraic Normal Form ---------------------------------*/

  rc = getInput(&a, &n, &q);
  if (rc) exit(1);
  printf("Argument Dimension = %d\n", n);
  m = 1 << n;                     /* Argument space has m elements */
  printf("Argument space has %d elements.\n", m);
  printf("Image Dimension = %d\n", q);
  p = 1 << q;                     /* Image space has p elements    */
  printf("Image space has %d elements.\n", p);

  printf("\n");
  printf("1. Algebraic Normal Form:\n");
  printf("[Columns = Image components]\n\n");
  prtTabl0(a, n, q);

/* Step 2: Truth Table -------------------------------------------*/

  f = (unsigned **) malloc(q * sizeof(unsigned *));
  for (i = 0; i < q; i++) {
    f[i] = (unsigned *) malloc(m * sizeof(unsigned));
    rev(a[i], f[i], n);           /* calculate truth table         */
    }
  printf("\n");
  printf("2. Truth Table:\n");
  printf("[Columns = Image components]\n\n");
  prtTabl0(f, n, q);
```

```
/* Step 3: Characteristic Function -------------------------------*/

  g = (long *) malloc(m * p * sizeof(long));
  for (k = 0; k < m; k++) {                 /* One line of values */
    for (l = 0; l < p; l++)
      g[(k<<q)+l] = 0;                      /* Set default value  */
    y = 0;                         /* Calculate value of map at k */
    for (i = 0; i < q; i++) y = y + (f[q-1-i][k] << i);
    g[(k<<q)+y] = 1;              /* Here charact. function is 1. */
    }
  printf("\n");
  printf("3. Characteristic Function:\n");
  printf("\n");
  prtTabl1(g, n, q);

/* Step 4: Walsh Spectrum ----------------------------------------*/

  c = (long *) malloc(m * p * sizeof(long));
  wt(g, c, n+q);                 /* calculate Walsh transform     */
  printf("\n");
  printf("4. Walsh Spectrum:\n");
  printf("\n");
  prtTabl1(c, n, q);

/* Step 5: Linear Profile ----------------------------------------*/

  lp = (long *) malloc(m * p * sizeof(long));
  for (k = 0; k < m*p; k++) lp[k] = c[k]*c[k];
  rc = reduce(lp, m*p);
  printf("\n");
  printf("5. Linear Profile:\n");
  printf("[To normalize divide by %d]\n", lp[0]);
  prtTabl1(lp, n, q);

/* Step 6: Differential Profile ------------------------------------*/

  dp = (long *) malloc(m * p * sizeof(long));
  wt(lp, dp, n+q);
  rc = reduce(dp, m*p);
  printf("\n");
  printf("6. Differential Profile:\n");
  printf("[To normalize divide by %d]\n", dp[0]);
  prtTabl1(dp, n, q);
```

```
/* Step 7: Linearity/nonlinearity measures -----------------------*/

  ll = max(lp, 1, m*p);     /* Numerator of linear potential       */
                            /* Denominator is lp[0].               */
  dd = max(dp, 1, m*p);     /* Numerator of differential potential */
                            /* Denominator is dp[0].               */
  la = (double) ll;
  la = la/lp[0];
  la = 1 - sqrt(la);
  la = la * (m >> 1);
  nl = floor(la + 0.5);     /* Nonlinearity                        */
  maxnl = m >> 1;
  if (n <= 1) {
    maxnl = 0;
    }
  else if (!(n%2)) {            /* n is even                       */
    n2 = (n >> 1) - 1;
    maxnl = maxnl - (1 << n2);
    }
  else {                        /* n is odd >=3                    */
    la = 1 << (n-2);
    la = sqrt(la);
    maxnl = floor(maxnl - la);
    }

  printf("\n");
  printf("7. Linearity/nonlinearity measures:\n");
  printf("\n");
  printf("Linear potential: %d/%d\n", ll, lp[0]);
  printf("  [Higher values mean more linearity.]\n");
  printf("  [Theoretical minimum = 1/%d | maximum = 1]\n", m);
  printf("Differential potential: %d/%d\n", dd, dp[0]);
  printf("  [Higher values mean more linearity.]\n");
  printf("  [Theoretical minimum = 1/%d | maximum = 1]\n", p);
  printf("Nonlinearity: %d\n", nl);
  printf("  [Lower values mean more linearity.]\n");
  printf("  [Theoretical minimum = 0 | maximum = %d]\n", maxnl);

/* Finish *******************************************************/
  exit(0);
}
```