December 24, 2011 at 11:19

1. Introduction.

Epsimg

Creates grey-scale Encapsulated PostScript images of matrices of numerical data
(Version 1.6 of February 21, 2004)

Written by Fredrik Jonsson

Given a matrix of floating-point numbers stored in a regular ASCII text file, this CWEB[†] program creates a grey-scale Encapsulated PostScript (EPS) image of the matrix using its elements as specification of the brightness of the corresponding pixels in the image.

I do by no means claim to have written a program that generates fully optimized Encapsulated PostScript. The output images are in many cases large, and can in many cases be considerably reduced in size, in particular for binary or few-level grayscale images, for which run-length encoding easily can be applied. (In run-length encoding a long row of identical pixels is parametrized as a loop, without the need of individual specification of each pixel.) However, for my purposes it works fine, since I often only is concerned with the evaluation of gray-scale images, generated by mathematical means and often with no a priori specification of the number of intensity levels.

Of course, there are other ways of generating Encapsulated PostScript images of sampled of simulated data, as for example using the `image()` function of MATLAB. An advantage with using a stand-alone program, however, is that it is easily incorporated in scripts for batch processing. In addition, the EPSIMG program is provided free of charge.

[†] For information on the CWEB programming language by Donald E. Knuth, as well as samples of CWEB programs, see `http://www-cs-faculty.stanford.edu/~knuth/cweb.html`. For general information on literate programming, see `http://www.literateprogramming.com`.

2. Revision history of the program.

2004-01-26 [v.1.0] <jonsson@uni-wuppertal.de>
First properly working version of the EPSIMG program. I have now for a longer time had in my mind that it would be useful to write a stand-alone program that is capable of generating Encapsulated PostScript images of data matrices, in similar to the `image()` built-in function of MATLAB. In particular, I have lately encountered some problems involving the optical analysis of diffraction patterns, and in order to visualize my generated data (without having to use MATLAB every time) I this evening started the coding in CWEB.

2004-01-27 [v.1.1] <jonsson@uni-wuppertal.de>
Continued with cleaning up the code and adding some features, such as the possibility of letting the program add a frame outlining the bounding box of the Encapsulated PostScript image, and a scaling of the x- or y-axis to leave the aspect ratio of the square pixels invariant even for non-square input matrices. Also changed the precision of the coordinates and gray scale specifications in order to get really smooth images. However, there still seem to remain some bug that causes the program to refuse to accept data files containing lines with trailing blank spaces and additional line feeds. [Coding finished at 00:45, 2004-01-28]

2004-01-28 [v.1.2] <jonsson@uni-wuppertal.de>
This morning fixed the remaining bug from yesterday, and wrote a basic example (`example1`) as a block in the `Makefile`, using AWK to generate a simple interference pattern that is visualized with the help of the EPSIMG program. Also wrote blocks that provide a proper rescaling of the image width or height whenever either the width or height is larger than their respective maximum values. Wrote an example (`example2`) in the `Makefile` that illustrates this automatic rescaling of the image.

2004-01-30 [v.1.3] <jonsson@uni-wuppertal.de>
This evening (time is now 01:55 Saturday morning) I started to sketch on a partitioning scheme for the reduction of data neccessary to save to disk. For many of my diffraction images, there are large areas that are of equal shade, and since they have considerable extent in the x- as well as y-direction, a run-length encoding (of the type used in the ancient program for generation of fractals that I wrote together with Tommy Ekola in 1996) of the Encapsulated PostScript will not fix the problem to any greater extent. Therefore, I started formulating a recursive scheme for the partitioning of data into smaller and smaller sub-blocks of the user-supplied matrix, which I for the sake of simplicity so far have assumed to be square, of size $[2^M \times 2^M]$ for some integer M. Wrote a MetaPost figure `matfig.mp` that illustrates the partition scheme.

2004-02-07 [v.1.4] <jonsson@uni-wuppertal.de>
[Athens, Greece] Noticed that when viewed using Ghostview, the figures could not be zoomed properly. This was corrected by letting the program explicitly state `%%!PS-Adobe-2.0 EPSF-1.2`, and by also explicitly stating the number of pages (that is to say, one) of the figure in the Encapsulated PostScript preamble, using `%%Pages: 1`.

2004-02-20 [v.1.5] <jonsson@uni-wuppertal.de>
[Östergarn, Gotland] Added the command-line options `--commmented_postscript` and `--uncommmented_postscript`, explicitly forcing the program either to include comments on PostScript routines directly into the generated code (default), or forcing the program to suppress these comments (giving a slightly reduced size on disk).

2004-02-21 [v.1.6] <jonsson@uni-wuppertal.de>
[Östergarn, Gotland] Wrote the final blocks of a major revision of the program, concerning the algorithm for generation of individual pixels. While the program previously explicitly stated the pixel boundaries as paths, I have now replaced this by a PostScript

routine `drawpixel` that takes a pixel bounding box given by the lower left and upper right corners $(\langle llx\rangle, \langle lly\rangle)$ and $(\langle urx\rangle, \langle ury\rangle)$ and draws and fills the pixel with a specified gray value. The syntax for this PostScript routine is (in the PostScript language) simply `drawpixel` $\langle llx\rangle$ $\langle lly\rangle$ $\langle urx\rangle$ $\langle ury\rangle$ $\langle w\rangle$, where $\langle w\rangle \in [0, 1]$ is the whiteness of the actual pixel. I did, however, keep the possibility of generating the previous, more extensive form of PostScript, and in order to be able to switch the program into either mode, the options `--compactified_pixelcode` and `--extensive_pixelcode` were added as parts of the startup syntax. When applied to the previously written example with a 64×64-sized matrix of real numbers, the size of the generated was radically reduced from 590.3 kB to 152.4 kB, hence corresponding to a reduction by 74%! However, there still remain to optimize the code, especially to write PostScript routines that takes the image matrix and automatically loops over the indices, instead of the current approach, where the bounding box of each individual pixel still is specified in the code. (The image generated by the `image()` routine of MATLAB is still considerably smaller in size than what the now optimized algorithm provides; so far the generated PostScript takes approximately 37 byte per pixel, which is far too much even for "educational purpose".) What remains now is to also include the more clever partitioning of the image for cases with many adjacent pixels of identical gray value.

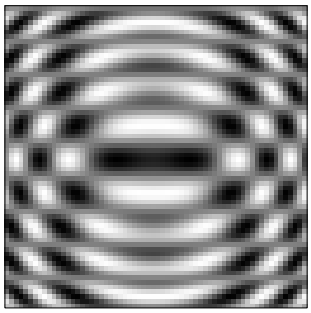

Figure R1. The example 64×64 image used in evaluating size reduction 2004-02-21.

3. Compiling the source code. The program is written in CWEB, generating ANSI-C conforming source code and documentation as TEX-source, and is to be compiled using the enclosed Makefile, leaving an executable file `epsimg`† and a PostScript file `epsimg.ps` (the document you currently are reading), which contains the full documentation of the program:

```
#
# Makefile designed for use with ctangle, cweave, gcc, and plain TeX.
#
# Copyright (C) 2004, Fredrik Jonsson <jonsson@uni-wuppertal.de>
#
CTANGLE  = ctangle
CC       = gcc
CCOPTS   = -O2 -Wall -ansi -pedantic # follow ISO C89 (ANSI) strictly
LNOPTS   = -lm
CWEAVE   = cweave
TEX      = tex
DVIPS    = dvips
DVIPSOPT = -ta4 -D1200

all:  epsimg.exe epsimg.ps

epsimg.exe:  epsimg.o # generate the executable file
        $(CC) $(CCOPTS) -o epsimg epsimg.o $(LNOPTS)

epsimg.o:  epsimg.c # generate the object file
        $(CC) $(CCOPTS) -c epsimg.c

epsimg.c:  epsimg.w # generate C code from the CWEB source
        $(CTANGLE) epsimg

epsimg.ps:  epsimg.dvi # generate the PostScript documentation
        $(DVIPS) $(DVIPSOPT) epsimg.dvi -o epsimg.ps

epsimg.dvi:  epsimg.tex # generate the device-independent documentation
        $(TEX) epsimg.tex

epsimg.tex:  epsimg.w # generate plain TeX code from the CWEB source
        $(CWEAVE) epsimg

clean:
        -rm -Rf *.c *.o *.exe *.aux *.log *.toc *.idx *.scn *.tex *.dvi
```

† On platforms running Windows NT, Windows 2000, or any other operating system by Microsoft, the executable file will instead automatically be called `epsimg.exe`.

4. Running the program. The program is entirely controlled by the command line options supplied when invoking the program, and the syntax is simply:

```
epsimg -i <infile> -o <outfile>
```

where `<infile>` is a regular text file containing the matrix of numerical data, and `<outfile>` is the name of the Encapsulated PostScript image that is to be generated. Instead of `-i` and `-o`, the switches can equivalently be specified in their longer forms `--inputfile` and `--outputfile`, respectively.

Several options may additionally be specified; to see a listing of all available options, simply invoke EPSIMG with the help switch `-h` (or, equivalently, `--help` in a longer form), as

```
epsimg -h
```

5. Compressing the size of the generated Encapsulated Postscript. In a general sense, the input to the EPSIMG program is just an arbitrary matrix of numbers, with no a priori assumption on their individual values or their ordering. In many images, however, there are large areas of equal colour (or brightness, if we stick to the fact that the EPSIMG program primarily is designed for the visualization of gray scale images), and instead of sequentially writing a large list of identical squares, of the same shading but slightly displaced with respect to one another, one may start thinking that there must be a more efficient method of saving the image to file.

One possibility is to check the structure sequentially in the order the squares are written, and in case many boxes of the same shade appear, say in the row direction of the supplied matrix, a rectangle with this shade and with a length corresponding to the number of equal squares is to be drawn instead. This, however, may be an inefficient method as well, since any directionality in the image in the column direction of the matrix will be left unnoticed. In addition, if there are large fields of equal shade, a lot of neighbouring rows should be possible to further reduce, for example by instead drawing general rectangles which no longer need to be of the same height as the basic pixels.

The question therefore arises: Would it not be possible to make a relatively simple divide-and-conquer description of the matrix, partitioning the matrix into rectangular building blocks of equal shade? In Fig. 1, on possibility of a partitioning scheme for the reduction of the data needed to save to disk is illustrated.

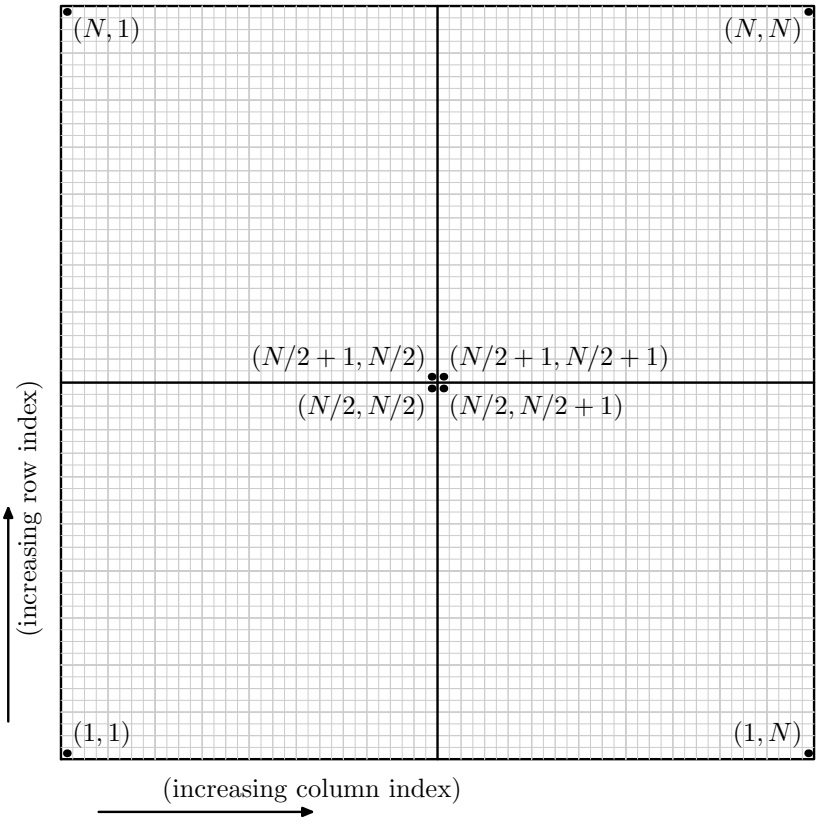

Figure 1. A possible partitioning scheme for square matrices of size $[2^M \times 2^M]$.

6. The main program. Here follows the general outline of the main program.

For the flags that are internally used, for the settings of desired program actions, the significance of the flags are: `COMPACTIFIED_PIXELCODE` If set to a positive nonzero integer value, this flag causes the program to generate a compactified PostScript code for the definitions of the individual boxes of the image, i.e. the individual pixels.

In order to have one single and generic output stream, the `OUTSTREAM` definition provides an easy solution to switching the output from file to terminal output, depending on which options that are detected at the command line during startup of the program.

```
#include <math.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>     /* to get automatically generated timestamp in EPS header */
#include <ctype.h>     /* to access isalnum( ) */
#define VERSION_NUMBER "1.6"
#define A4_PAGE_WIDTH (594)     /* A4 page width in pt (1/72 in) */
#define A4_PAGE_HEIGHT (841)     /* A4 page height in pt (1/72 in) */
#define MAXIMUM_IMAGE_WIDTH (A4_PAGE_WIDTH − 144)     /* 1.0 inch default margin */
#define MAXIMUM_IMAGE_HEIGHT (A4_PAGE_HEIGHT − 144)     /* 1.0 inch default margin */
#define DEFAULT_IMAGE_WIDTH (0.8 * MAXIMUM_IMAGE_WIDTH)
#define DEFAULT_IMAGE_XCENTER (A4_PAGE_WIDTH/2)
#define DEFAULT_IMAGE_YCENTER (A4_PAGE_HEIGHT/2)
#define DEFAULT_LINETHICKNESS 1     /* default line thickness in pt (1/72 in) */
#define OUTSTREAM (outfile_specified ? fpout : stdout)
#define SUCCESS 0     /* Return code for successful program termination */
#define FAILURE 1     /* Return code for program termination caused by failure */
#define COMPACTIFIED_PIXELCODE 1
#define EXTENSIVE_PIXELCODE 2
  ⟨ Global variables 7 ⟩
  ⟨ Subroutines 8 ⟩
  int main(int argc, char *argv[])
  {
    ⟨ Local variables 16 ⟩
    ⟨ Parse command line 17 ⟩
    ⟨ Open files 19 ⟩
    ⟨ Load text file into image matrix 20 ⟩
    ⟨ Normalize image matrix 21 ⟩
    ⟨ Initialize parameters of Encapsulated PostScript image 22 ⟩
    ⟨ Write preamble of Encapsulated PostScript image 23 ⟩
    ⟨ Write body of Encapsulated PostScript image 24 ⟩
    ⟨ Write closing of Encapsulated PostScript image 27 ⟩
    ⟨ Deallocate image matrix 28 ⟩
    ⟨ Close files 29 ⟩
    return (SUCCESS);
  }
```

7. Declaration of global variables. The only global variables allowed in my programs are *optarg*, which is the string of characters that specified the call from the command line, and *progname*, which simply is the string containing the name of the program, as it was invoked from the command line.

⟨ Global variables 7 ⟩ ≡

```
extern char *optarg;      /* command line string */
char *progname;      /* name of the program as invoked from command line */
```

This code is used in section 6.

8. Declarations of subroutines used by the program.

⟨ Subroutines 8 ⟩ ≡
⟨ Display help message 9 ⟩
⟨ Routine for allocation of double vectors 10 ⟩
⟨ Routine for allocation of double matrices 11 ⟩
⟨ Routine for deallocation of double vectors 12 ⟩
⟨ Routine for deallocation of double matrices 13 ⟩
⟨ Routine for loading matrix data from text file 14 ⟩
⟨ Routine for unloading matrix data previously loaded from text file 15 ⟩
This code is used in section 6.

9. Routine for displaying a help message at the screen.

⟨ Display help message 9 ⟩ ≡

```
void showsomehelp(void)
{
  fprintf(stderr, "Usage: %s -i infile [options] [-o outfile]\n", progname);
  fprintf(stderr, "Options:\n");
  fprintf(stderr, "  -i, --inputfile <str>    Specifies the file where to find \
      the intensity\n""                              response for the actual property.\n");
  fprintf(stderr, "  -o, --outputfile <str>   Specifies the file where to save \
      the trans-\n""                                 mitted optical pulse shape. Whene\
      ver this\n");
  fprintf(stderr, "                                   option is not present at the comm\
      and line,\n""                                  the generated time series will be\
      written\n""                                    to standard terminal output inste\
      ad, in\n""                                     which case any set verbose mode w\
      ill be turned\n""                              off (see -v option).\n");
  fprintf(stderr, "  -s, --sequential         Toggle sequential mode. Default: \
      off.\n""                                       When generating the Encapsulated \
      PostScript,\n""                                in sequential mode, the data\n");
  fprintf(stderr, "                                   is scanned column/row-wise, with \
      an individual\n""                              pixel written for each data point\
      of the input\n""                               matrix. In this mode the program \
      will ignore\n""                                any possibilities of reducing the\
      data through\n""                               a more efficient partitioning of \
      the input\n""                                  matrix.\n");
  fprintf(stderr, "  -v, --verbose            Toggle verbose mode. If no output\
      filename was\n""                               specified at the command line, ve\
      rbose mode\n""                                 will automatically be turned off,\
      in order for\n""                               output messages not to interfere \
      with the\n""                                   generated Encapsulated PostScript\
      code.\n""                                      Default: off\n");
  fprintf(stderr, "  -h, --help               Display this help message and exit clean\n");
  fprintf(stderr, "Copyright (C) 2004 Fredrik Jonsson <jonsson@uni-wuppertal.de>\n");
}
```

This code is used in section 8.

10. The *dvector*( ) routine allocate a real-valued vector of double precision, with vector index ranging from *nl* to *nh*.

```
⟨ Routine for allocation of double vectors 10 ⟩ ≡
  double *dvector(long nl, long nh)
  {
    double *v;
    v = (double *) malloc((size_t)((nh - nl + 2) * sizeof(double)));
    if (¬v) {
      fprintf(stderr, "Error: Allocation failure in dvector()\n");
      exit(FAILURE);
    }
    return v - nl + 1;
  }
```

This code is used in section 8.

11. The *dmatrix*( ) routine allocate a real-valued matrix of double precision, with row index ranging from *nrl* to *nrh*, and column index ranging from *ncl* to *nch*.

```
⟨ Routine for allocation of double matrices 11 ⟩ ≡
  double **dmatrix(long nrl, long nrh, long ncl, long nch)
  {
    long i, nrow = nrh - nrl + 1, ncol = nch - ncl + 1;
    double **m;
    m = (double **) malloc((size_t)((nrow + 1) * sizeof(double *)));
    if (¬m) {
      fprintf(stderr, "%s: Allocation failure 1 in dmatrix() routine!\n", progname);
      exit(FAILURE);
    }
    m += 1;
    m -= nrl;
    m[nrl] = (double *) malloc((size_t)((nrow * ncol + 1) * sizeof(double)));
    if (¬m[nrl]) {
      fprintf(stderr, "%s: Allocation failure 2 in dmatrix() routine!\n", progname);
      exit(FAILURE);
    }
    m[nrl] += 1;
    m[nrl] -= ncl;
    for (i = nrl + 1; i ≤ nrh; i++) m[i] = m[i - 1] + ncol;
    return m;
  }
```

This code is used in section 8.

12. The *free_dvector*( ) routine release the memory occupied by the real-valued vector $v[nl \,..\, nh]$.

```
⟨ Routine for deallocation of double vectors 12 ⟩ ≡
  void free_dvector(double *v, long nl, long nh)
  {
    free((char *)(v + nl - 1));
  }
```

This code is used in section 8.

13. The *free_dmatrix*( ) routine release the memory occupied by the real-valued matrix $m[nrl \,..\, nrh][ncl \,..\, nch]$.

⟨ Routine for deallocation of double matrices 13 ⟩ ≡

```
void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch)
{
  free((char *)(m[nrl] + ncl - 1));
  free((char *)(m + nrl - 1));
}
```

This code is used in section 8.

14. The *load_matrix*( ) routine takes as input a character string *inputfilename*, specifying a regular text file of ASCII data stored as a matrix, and returns a pointer m to a matrix of double precision, containing the numerical values as appearing if the text file. The routine also automatically scans the input matrix size, and returns the number of rows in nr and the number of columns in nc. The number of columns is determined as the number of elements in the first row of data of the supplied text file. All subsequent rows are assumed to contain exactly the same number of elements; if this is not the case, an error message will be displayed on standard terminal output. The minimum and maximum elements found in the matrix are returned in the variables *min* and *max*, respectively.

Example of usage:

```
double **imagematrix,min,max;
long int nr,nc;
imagematrix=load_matrix("image.dat",nr,nc,min,max);
fprintf(stdout,"Detected %ld rows and %ld columns of data.n",nr,nc);
fprintf(stdout,"Minimum element:  %fn",min);
fprintf(stdout,"Maximum element:  %fn",max);
```

⟨ Routine for loading matrix data from text file 14 ⟩ ≡

```
char validchar(char ch)
{
  return (isalnum(ch) ∨ (ch ≡ '+') ∨ (ch ≡ '-') ∨ (ch ≡ '.'));
}
double **load_matrix(char inputfilename[], long *nr, long *nc, double *min, double *max)
{
  FILE *fpin = Λ;
  char tmpch;
  long j, k, nrt, nct;
  double tmpd, **m, tmin, tmax;
  if ((fpin = fopen(inputfilename, "r")) ≡ Λ) {
    fprintf(stderr, "%s: Could not open file %s for reading!\n", progname, inputfilename);
    exit(FAILURE);
  }
  fseek(fpin, 0L, SEEK_SET);
  fscanf(fpin, "%lf", &tmpd);
  tmin = tmpd;      /* initialize memory for minimum element */
  tmax = tmpd;      /* initialize memory for maximum element */
  fseek(fpin, 0L, SEEK_SET);
  nct = 0;      /* initialize column counter */
  while ((tmpch = getc(fpin)) ≠ '\n') {      /* determine column size nc */
    ungetc(tmpch, fpin);
    while ((tmpch = getc(fpin)) ≡ ' ') ;      /* get rid of any leading blanks */
    ungetc(tmpch, fpin);
    while (validchar(tmpch = getc(fpin))) ;      /* scan pass field for valid char */
    ungetc(tmpch, fpin);
    nct++;
    while ((tmpch = getc(fpin)) ≡ ' ') ;      /* get read of any trailing blanks */
    ungetc(tmpch, fpin);
  }
  fseek(fpin, 0L, SEEK_SET);
  nrt = 0;      /* initialize row counter */
  while ((tmpch = getc(fpin)) ≠ EOF) {      /* determine row size nr */
    ungetc(tmpch, fpin);
```

```
    for (k = 1; k ≤ nct; k++) fscanf(fpin, "%lf", &tmpd);
    nrt++;
    tmpch = getc(fpin);
    while ((tmpch ≡ ' ') ∨ (tmpch ≡ '\n')) tmpch = getc(fpin);
    if (tmpch ≠ EOF) ungetc(tmpch, fpin);
  }
  m = dmatrix(1, nrt, 1, nct);
  fseek(fpin, 0L, SEEK_SET);
  for (j = 1; j ≤ nrt; j++) {      /* for all rows, ... */
    for (k = 1; k ≤ nct; k++) {      /* and for all columns, ... */
      fscanf(fpin, "%lf", &tmpd);
      m[j][k] = tmpd;
      if (tmpd < tmin) tmin = tmpd;
      else if (tmpd > tmax) tmax = tmpd;
    }
  }
  fclose(fpin);
  *nr = nrt;
  *nc = nct;
  *min = tmin;
  *max = tmax;
  return m;
}
```

This code is used in section 8.

15. The *unload_matrix*( ) routine simply releases memory previously allocated by the *load_matrix*( ) routine.

⟨ Routine for unloading matrix data previously loaded from text file 15 ⟩ ≡

```
void unload_matrix(double **m, long nr, long nc)
{
  free_dmatrix(m, 1, nr, 1, nc);      /* yes, it is this simple... */
}
```

This code is used in section 8.

16. Declaration of local variables of the *main* program.

⟨ Local variables 16 ⟩ ≡

```
double **imagematrix, min, max, dx, dy, llx, lly, urx, ury;
double imagewidth, imageheight, imagexcenter, imageycenter;
double linethickness = DEFAULT_LINETHICKNESS;
time_t now = time(Λ);
long int j, k, nr, nc;
int no_arg, bbllx, bblly, bburx, bbury;
FILE *fpout = Λ;
char inputfilename[256] = "", outputfilename[256] = "";
short verbose = 0, write_floatform = 0, write_frame = 1;
short infile_specified = 0, outfile_specified = 0, parse_data_sequentially = 1;
short write_title = 0;
short pixel_generation_mode = COMPACTIFIED_PIXELCODE;
short comments_in_postscript = 1;
```

This code is used in section 6.

17. Parsing command line options. All input parameters are passed to the program through command line options and arguments to the program. The syntax of command line options is listed whenever the program is invoked without any options, or if the `--help` option is specified at startup.

```
⟨ Parse command line 17 ⟩ ≡
  {
    progname = argv[0];
    no_arg = argc;
    while (−− argc) {
      if (¬strcmp(argv[no_arg − argc], "-o") ∨ ¬strcmp(argv[no_arg − argc], "--outputfile")) {
        −− argc;
        strcpy(outputfilename, argv[no_arg − argc]);
        outfile_specified = 1;
      }
      else if (¬strcmp(argv[no_arg − argc], "-i") ∨ ¬strcmp(argv[no_arg − argc], "--inputfile")) {
        −− argc;
        strcpy(inputfilename, argv[no_arg − argc]);
        infile_specified = 1;
      }
      else if ((¬strcmp(argv[no_arg − argc], "-f")) ∨ (¬strcmp(argv[no_arg − argc], "--floatform"))) {
        write_floatform = (write_floatform ? 0 : 1);
        if (verbose) fprintf(stdout, "%s: Using floating number output.\n", progname);
      }
      else if ((¬strcmp(argv[no_arg − argc], "-r")) ∨ (¬strcmp(argv[no_arg − argc], "--writeframe"))) {
        write_frame = (write_frame ? 0 : 1);
      }
      else if (¬strcmp(argv[no_arg − argc], "--commmented_postscript")) {
        comments_in_postscript = 1;
      }
      else if (¬strcmp(argv[no_arg − argc], "--uncommmented_postscript")) {
        comments_in_postscript = 0;
      }
      else if (¬strcmp(argv[no_arg − argc], "--compactified_pixelcode")) {
        pixel_generation_mode = COMPACTIFIED_PIXELCODE;
      }
      else if (¬strcmp(argv[no_arg − argc], "--extensive_pixelcode")) {
        pixel_generation_mode = EXTENSIVE_PIXELCODE;
      }
      else if (¬strcmp(argv[no_arg − argc], "-v") ∨ ¬strcmp(argv[no_arg − argc], "--verbose")) {
        verbose = (verbose ? 0 : 1);
      }
      else if (¬strcmp(argv[no_arg − argc], "-s") ∨ ¬strcmp(argv[no_arg − argc], "--sequential")) {
        parse_data_sequentially = (parse_data_sequentially ? 0 : 1);
      }
      else {
        fprintf(stderr, "%s: Unknown option '%s'.\n", progname, argv[no_arg − argc]);
        exit(FAILURE);
      }
    }
    if (¬outfile_specified) verbose = 0;      /* terminal output EPS should be clean */
  }
```

This code is used in section 6.

18. Opening and closing files for data output.

19. Open files for reading and writing.

```
⟨ Open files 19 ⟩ ≡
  {
    if (outfile_specified) {
      if ((fpout = fopen(outputfilename, "w")) ≡ Λ) {
        fprintf(stderr, "%s: Could not open file %s for writing!\n", progname, outputfilename);
        exit(FAILURE);
      }
      fseek(fpout, 0L, SEEK_SET);
    }
    else {
      if (verbose)
        fprintf(stdout, "%s: No output file specified. (Writing to stdout).\n", progname);
    }
  }
```

This code is used in section 6.

20. Loading the text file into memory. In this first step, the specified input text file is opened, and is loaded into memory allocated by the *dmatrix* routine. The memory area is accessed via the pointer $**imagematrix$, which is the basic variable used later on by the blocks that write the Encapsulated PostScript image to file. After the data is loaded, the input file is closed.

```
⟨ Load text file into image matrix 20 ⟩ ≡
  {
    if (infile_specified) {
      if (verbose) fprintf(stderr, "%s: Loading data from file %s.\n", progname, inputfilename);
      imagematrix = load_matrix(inputfilename, &nr, &nc, &min, &max);
      if (verbose) {
        fprintf(stdout, "%s: Detected %ld rows and %ld columns of data in file '%s'.\n",
            progname, nr, nc, inputfilename);
        fprintf(stdout, "%s: Maximum element in '%s': %f\n", progname, inputfilename, max);
        fprintf(stdout, "%s: Minimum element in '%s': %f\n", progname, inputfilename, min);
      }
    }
    else {
      fprintf(stderr, "%s: Error: Specify an input filename.\n", progname);
      showsomehelp( );
      exit(FAILURE);
    }
  }
```

This code is used in section 6.

21. Normalize the image matrix. In order to write a properly scaled Encapsulated PostScript image to file, the loaded data need to be normalized, so that the elements arenumerical values between 0 and 1.

```
⟨ Normalize image matrix 21 ⟩ ≡
  {
    if (verbose) fprintf(stdout, "%s: Normalizing image matrix.\n", progname);
    for (j = 1; j ≤ nr; j++) {      /* for all rows, ... */
      for (k = 1; k ≤ nc; k++) {      /* and for all columns, ... */
        imagematrix[j][k] = imagematrix[j][k] − min;
        imagematrix[j][k] = imagematrix[j][k]/(max − min);
      }
    }
  }
```

This code is used in section 6.

22. Initialize the parameters to be used for the Encapsulated PostScript image. The parameters to be set prior to the calculation of positioning of the individual pixels of the image are the corner coordinates for the bounding box. The x-height and y-width of the image are generally scaled such that the aspect ratio of the image is left invariant under scaling of any of the coordinate axes.

By default, the program will use the width as reference for scaling the height of the image, to give an aspect ratio (height/width) that leaves the individual pixels as squares. If, however, the program finds that the calculated image height exceed the maximum allowed, then the height will be fixed to its maximum value, instead scaling the width of the image (to still give an equal aspect ratio).

The values here used for the maximum extents of the picture are based on that for an A4 paper, the limiting bounding box is between the lower left corner at $(0, 0)$ pt and upper right corner at $(594, 841)$ pt.

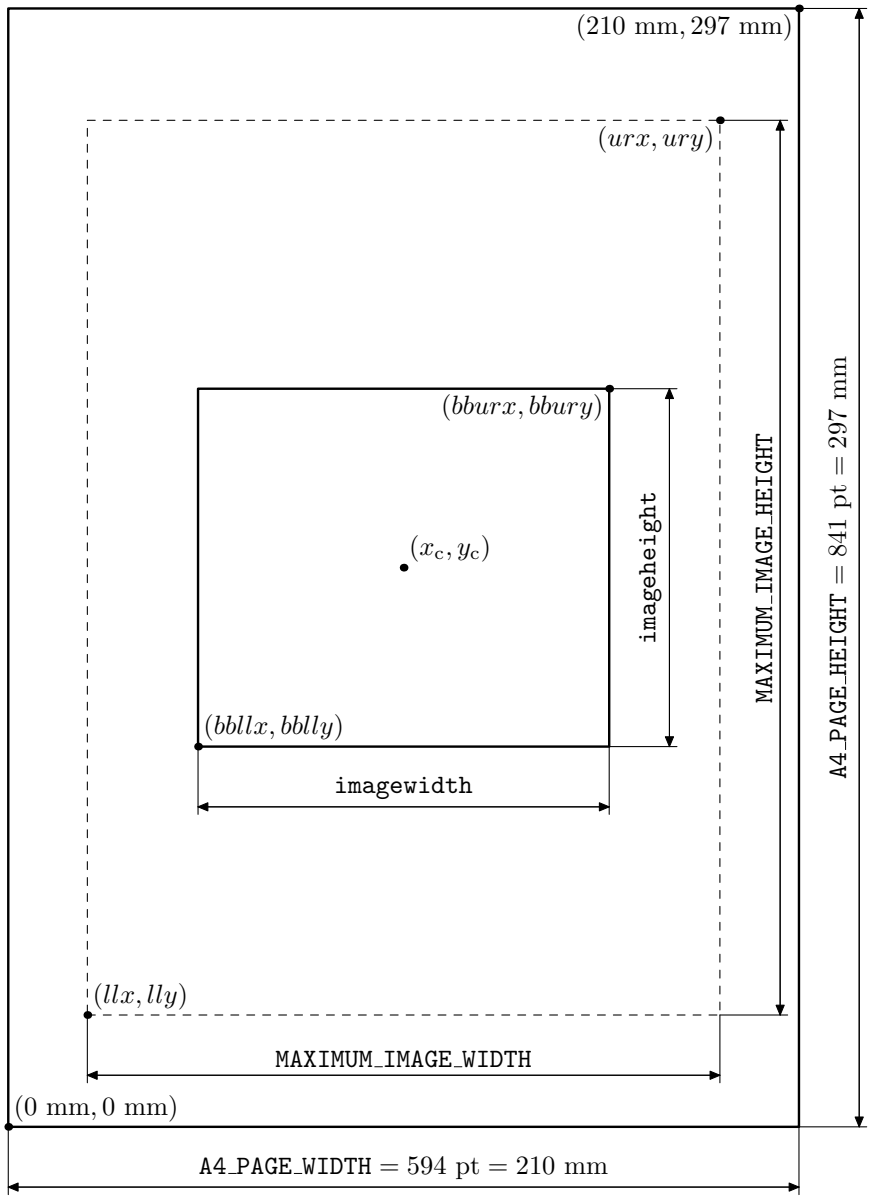

Figure 2. The page layout and definitions as used for the initialization of the Encapsulated PostScript image.

⟨ Initialize parameters of Encapsulated PostScript image 22 ⟩ ≡

{

```
  imagewidth = ((double)(DEFAULT_IMAGE_WIDTH));
  imageheight = (((double) nr)/((double) nc)) * ((double)(DEFAULT_IMAGE_WIDTH));
  imagexcenter = DEFAULT_IMAGE_XCENTER;
  imageycenter = DEFAULT_IMAGE_YCENTER;
  if (imageheight > MAXIMUM_IMAGE_HEIGHT) {
    if (verbose) {
      fprintf(stdout, "%s: Warning. I found that the height of ", progname);
      fprintf(stdout, "the image exceeds its maximum\n");
      fprintf(stdout, "%s: value of %d pt.\n", progname, ((int) MAXIMUM_IMAGE_HEIGHT));
      fprintf(stdout, "%s: Will now instead scale the width of the image.\n", progname);
    }
    imageheight = MAXIMUM_IMAGE_HEIGHT;
    imagewidth = (((double) nc)/((double) nr)) * imageheight;
  }
  else {
    if (verbose) {
      fprintf(stdout, "%s: Image height automatically scaled to ", progname);
      fprintf(stdout, "width (to give equal aspect ratio).\n");
    }
  }
  bbllx = imagexcenter - imagewidth/2.0;
  bblly = imageycenter - imageheight/2.0;
  bburx = imagexcenter + imagewidth/2.0;
  bbury = imageycenter + imageheight/2.0;
}
```

This code is used in section 6.

23. Write the leading blocks of Encapsulated PostScript code. If the flag *pixel_generation_mode* is set to `COMPACTIFIEDPIXELCODE`, then an additional routine for the generation of the individual pixels will be added just after the comments in the preamble; otherwise, the generated code will be self-contained in the sense that the individual pixels are defined as free-standing drawing statements in the code. Notice that the type of output stream (terminal output or file pointer, depending on the options present at the command line at startup of the program) is determined by the current definition provided by `OUTSTREAM`. Notice that the string returned by the *ctime*( ) routine ends with a linefeeed.

The blocks dealing with the definition of the PostScript routine for a more "compactified" output code clearly deserves some more detailed description. The syntax for drawing an individual pixel, determined by the bounding box given by its lower left and upper right corners $(x_{\rm ll}, y_{\rm ll})$ and $(x_{\rm ur}, y_{\rm ur})$ is

`/drawpixel` ⟨*llx*⟩ ⟨*lly*⟩ ⟨*urx*⟩ ⟨*ury*⟩ ⟨*w*⟩,

where $llx = x_{\rm ll}$, $lly = y_{\rm ll}$, $urx = x_{\rm ur}$, and $ury = y_{\rm ur}$. This definition is illustrated in Fig. 3 below. In this description of the syntax, $w \in [0, 1]$ is the whiteness value of the pixel, with 0 corresponding to black, and 1 corresponding to white.

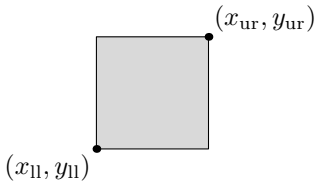

Figure 3. Illustration of the definition of a pixel in terms of its lower left and upper right corners.

In the PostScript routine `/drawpixel`, the following commands of the PostScript languange are used for manipulation of the stack:

`dup` Duplicates the bottom element in the stack, and then pushes it into the stack. This operation is similar to the ENTER as used in reverse polish notation employed in, for example, Hewlett–Packard calculators.

`exch` Interchanges the two bottom-most elements in the stack. This operation is identical to SWAP.

⟨*m*⟩ ⟨*n*⟩ `roll` Rolls down the m bottom-most elements of the stack n times, that is to say, applying cyclic permutation n times on the first m elements. In analogy with the ROLLD operation of reverse polish notation of Hewlett–Packard calculators, this is identical to executing the operation "m ROLLD" exactly n times. Notice that ⟨*m*⟩ ⟨*m*⟩ `roll` always just gives the identity operation on the stack for arbitrary m (of course provided that m is not greater than the number of elements that currently are present in the stack).

⟨ Write preamble of Encapsulated PostScript image 23 ⟩ ≡

```
{
  fprintf(OUTSTREAM, "%%!PS-Adobe-2.0 EPSF-1.2\n");
  fprintf(OUTSTREAM, "%%%%BoundingBox: %d %d %d %d\n", bbllx, bblly, bburx, bbury);
  fprintf(OUTSTREAM, "%%%%Creator: epsimg %s", VERSION_NUMBER);
  fprintf(OUTSTREAM, " Copyright (C) 2004 Fredrik Jonsson\n");
  if (outfile_specified) fprintf(OUTSTREAM, "%%%%Title: %s\n", outputfilename);
  else fprintf(OUTSTREAM, "%%%%Title: (image written to stdout)\n");
  fprintf(OUTSTREAM, "%%%%CreationDate: %s", ctime(&now));
  fprintf(OUTSTREAM, "%%%%Pages: 1\n");
  fprintf(OUTSTREAM, "%%%%EndProlog\n");
  fprintf(OUTSTREAM, "%%%%Pages: 1\n");
  fprintf(OUTSTREAM, "%%%%Page: 1 1\n");
```

```
    if (pixel_generation_mode ≡ COMPACTIFIED_PIXELCODE) {
      if (comments_in_postscript) {
        fprintf(OUTSTREAM, "%%\n");
        fprintf(OUTSTREAM, "%% Routine for duplicating the bottom-most pair");
        fprintf(OUTSTREAM, " of elements in the stack.");
        fprintf(OUTSTREAM, "%%\n");
      }
      fprintf(OUTSTREAM, "/dupc {dup 3 2 roll dup 4 1 roll exch} bind def");
      if (comments_in_postscript) {
        fprintf(OUTSTREAM, "%%\n");
        fprintf(OUTSTREAM, "%% Routine for calculating the lower right corner");
        fprintf(OUTSTREAM, " coordinates of the pixel.\n");
        fprintf(OUTSTREAM, "%% The syntax is simply '<llx> <lly> <urx> <ury> ");
        fprintf(OUTSTREAM, " lrc', where (<llx>,<lly>)\n");
        fprintf(OUTSTREAM, "%% and (<urx>,<ury>) are the");
        fprintf(OUTSTREAM, " lower left and upper right corner coordinates\n");
        fprintf(OUTSTREAM, "%% of the pixel. The resulting (<lrx>,<lry>) pair");
        fprintf(OUTSTREAM, " are after the calculation\n%% pushed onto the");
        fprintf(OUTSTREAM, " stack, preserving the previously present stack");
        fprintf(OUTSTREAM, " at above\n%% levels.\n");
        fprintf(OUTSTREAM, "%%\n");
      }
      fprintf(OUTSTREAM, "/lrc {4 1 roll dup 5 2 roll dup 5 -1 roll exch");
      fprintf(OUTSTREAM, " 4 2 roll 6 2 roll} bind def\n");
      fprintf(OUTSTREAM, "/ulc {4 3 roll dup 5 2 roll dup 6 -1 roll exch}");
      fprintf(OUTSTREAM, " bind def\n");
      if (comments_in_postscript) {
        fprintf(OUTSTREAM, "%%\n");
        fprintf(OUTSTREAM, "%% Routine for drawing individual pixels\n");
        fprintf(OUTSTREAM, "%%\n");
      }
      fprintf(OUTSTREAM, "/pixelstack {lrc 6 2 roll ulc 4 2 roll 8 4 roll");
      fprintf(OUTSTREAM, " dupc 10 2 roll} bind def\n");
      fprintf(OUTSTREAM, "/drawpixel {setgray pixelstack newpath moveto lineto\n");
      fprintf(OUTSTREAM, " lineto lineto lineto closepath fill} bind def\n");
      if (comments_in_postscript) {
        fprintf(OUTSTREAM, "%%\n");
        fprintf(OUTSTREAM, "%% The dp routine is short-hand for drawpixel\n");
        fprintf(OUTSTREAM, "%%\n");
      }
      fprintf(OUTSTREAM, "/dp {drawpixel} bind def\n");
    }
  }
```

This code is used in section 6.

24. Write the body of Encapsulated PostScript code.

```
⟨ Write body of Encapsulated PostScript image 24 ⟩ ≡
  {
    if (parse_data_sequentially) {
      ⟨ Write body of sequential Encapsulated PostScript image 25 ⟩
    }
    else {
      ⟨ Write body of partitioned Encapsulated PostScript image 26 ⟩
    }
  }
```

This code is used in section 6.

25. Write sequential body of Encapsulated PostScript code. The `moveto` sets the current starting point of each pixel. The path of the boundary of each pixel is traversed in counter-clockwise direction, starting in the lower left corner of each pixel. Here (llx, lly) give the (x, y)-coordinates of the lower left corner of the pixel, while (urx, ury) give the (x, y)-coordinates of the upper right corner.

```
⟨ Write body of sequential Encapsulated PostScript image 25 ⟩ ≡
  {
    dx = ((double)(bburx − bbllx))/((double) nc);
    dy = ((double)(bbury − bblly))/((double) nr);
    for (j = 1; j ≤ nr; j++) {
      lly = ((double) bblly) + ((double)(j − 1)) ∗ dy;
      ury = lly + dy ∗ (1.0 + 8.0 · 10^−2);
      ;
      for (k = 1; k ≤ nc; k++) {
        llx = bbllx + ((double)(k − 1) ∗ dx);
        urx = llx + dx ∗ (1.0 + 8.0 · 10^−2);
        if (pixel_generation_mode ≡ COMPACTIFIED_PIXELCODE) {
          fprintf(OUTSTREAM, "%1.2f␣%1.2f␣%1.2f␣%1.2f␣%1.3f␣dp\n", llx, lly, urx, ury,
              imagematrix[j][k]);
        }
        else {
          fprintf(OUTSTREAM, "%1.3f␣setgray\n", imagematrix[j][k]);
          fprintf(OUTSTREAM, "newpath␣%1.2f␣%1.2f␣moveto\n", llx, lly);
          fprintf(OUTSTREAM, "␣%1.2f␣%1.2f␣lineto", urx, lly);
          fprintf(OUTSTREAM, "␣%1.2f␣%1.2f␣lineto\n", urx, ury);
          fprintf(OUTSTREAM, "␣%1.2f␣%1.2f␣lineto", llx, ury);
          fprintf(OUTSTREAM, "␣%1.2f␣%1.2f␣lineto␣closepath␣fill\n", llx, lly);
        }
      }
    }
    if (0 ≡ 1) {      /∗ for debugging purposes only ∗/
      for (j = 1; j ≤ nr; j++) {
        for (k = 1; k ≤ nc; k++) {
          fprintf(stdout, "%2.4f␣␣", imagematrix[j][k]);
        }
        fprintf(stdout, "\n");
      }
    }
  }
```

This code is used in section 24.

26. Write partitioned body of Encapsulated PostScript code.

```
⟨ Write body of partitioned Encapsulated PostScript image 26 ⟩ ≡
  {
    fprintf(stdout, "Not␣yet␣finished␣with␣non-sequential␣partitioning␣of␣data\n");
    exit(−1);
  }
```

This code is used in section 24.

27. Write the blocks ending the Encapsulated PostScript code.

```
⟨ Write closing of Encapsulated PostScript image 27 ⟩ ≡
  {
    if (write_frame) {      /* write frame corresponding to bounding box */
      fprintf(OUTSTREAM, "0␣setgray␣0␣%1.2f␣dtransform␣truncate␣", linethickness);
      fprintf(OUTSTREAM, "idtransform␣setlinewidth␣pop\n");
      fprintf(OUTSTREAM, "␣[]␣0␣setdash␣1␣setlinejoin␣10␣setmiterlimit\n");
      fprintf(OUTSTREAM, "newpath␣%d␣%d␣moveto\n", bbllx, bblly);
      fprintf(OUTSTREAM, "␣%d␣%d␣lineto", bburx, bblly);
      fprintf(OUTSTREAM, "␣%d␣%d␣lineto\n", bburx, bbury);
      fprintf(OUTSTREAM, "␣%d␣%d␣lineto", bbllx, bbury);
      fprintf(OUTSTREAM, "␣%d␣%d␣lineto␣closepath␣stroke\n", bbllx, bblly);
    }
    if (write_title) {
      fprintf(stderr, "Still␣to␣be␣finished!!\n");
      exit(−1);
      fprintf(OUTSTREAM, "%%IncludeResource:␣font␣Helvetica\n");
      fprintf(OUTSTREAM, "/Helvetica␣/WindowsLatin1Encoding␣120␣FMSR\n");
      fprintf(OUTSTREAM, "2345␣2372␣moveto\n");
      fprintf(OUTSTREAM, "(Intensity␣distribution␣in␣observation␣plane)␣s\n");
      fprintf(OUTSTREAM, "504␣2372␣moveto␣-90␣rotate\n");
      fprintf(OUTSTREAM, "(y␣[)␣s\n");
      fprintf(OUTSTREAM, "90␣rotate\n");
      fprintf(OUTSTREAM, "%%IncludeResource:␣font␣Symbol\n");
      fprintf(OUTSTREAM, "/Symbol␣/WindowsLatin1Encoding␣120␣FMSR\n");
      fprintf(OUTSTREAM, "504␣2540␣moveto␣-90␣rotate\n");
      fprintf(OUTSTREAM, "(m)␣s\n");
      fprintf(OUTSTREAM, "90␣rotate\n");
      fprintf(OUTSTREAM, "504␣2372␣moveto␣-90␣rotate\n");
      fprintf(OUTSTREAM, "(])␣s\n");      /* AND SO ON, IN THIS STYLE .... */
    }
    fprintf(OUTSTREAM, "showpage\n");
    fprintf(OUTSTREAM, "%%%%EOF\n");
  }
```

This code is used in section 6.

28. Deallocate memory occupied by the image matrix.

```
⟨ Deallocate image matrix 28 ⟩ ≡
  {
    unload_matrix(imagematrix, nr, nc);
  }
```

This code is used in section 6.

29. Close all open files.

```
⟨ Close files 29 ⟩ ≡
  {
    fclose(fpout);
  }
```

This code is used in section 6.

30. Index.

EPSIMG