(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Introduction. This program prepares data for the examples in my paper "Fonts for digital halftones." The input file (*stdin*) is assumed to be an EPS file output by Adobe Photoshop™ on a Macintosh with the binary EPS option, having a resolution of 72 pixels per inch. This file either has m rows of n columns each, or $m+n-1$ rows of $m+n-1$ columns each, or $2m$ rows of $2n$ columns each; in the second case the image has been rotated 45° clockwise. (These images were obtained by starting with a given $km \times kn$ image, optionally rotating it 45°, and then using Photoshop's Image Size operation to reduce to the desired number of pixel units. In my experiments I took $k = 8$, so that I could also use the dot diffusion method; but k need not be an integer. Larger values of k tend to make the reduced images more accurate than smaller values do.)

The output file (*stdout*) is a sequence of ASCII characters that can be placed into TEX files leading to typeset output images of size $8m \times 8n$, using fonts like those described in the paper. In the first case, we output m lines of 65-level pixel data. In the second (rotated) case, we output $2m$ lines of 33-level pixel data. In the third case, we output $2m$ lines of 17-level pixel data.

```
#define m 64 /* base number of rows */
#define n 55 /* base number of columns */
#define r 64 /* max(m,n) */
#include <stdio.h>
  float a[m+m+2][n+r]; /* darknesses: 0.0 is white, 1.0 is black */
  ⟨ Global variables 4 ⟩;
  main(argc,argv)
      int argc;
      char *argv[];
  {
    register int i,j,k,l,p;
    int levels,trash,ii,jj;
    float dampening = 1.0, brightness = 1.0;
    ⟨ Check for nonstandard dampening and brightness factors 2 ⟩;
    ⟨ Determine the type of input by looking at the bounding box 3 ⟩;
    fprintf(stderr,"Making␣%d␣lines␣of␣%d-level␣data\n",(levels < 65 ? m+m : m),levels);
    printf("\\begin%shalftone\n",levels ≡ 33 ? "alt" : "");
    ⟨ Input the graphic data 5 ⟩;
    ⟨ Translate input to output 12 ⟩;
  }
```

2. Optional command-line arguments allow the user to multiply the diffusion constants by a *dampening* factor and/or to multiply the brightness by a *brightness* factor.

```
⟨ Check for nonstandard dampening and brightness factors 2 ⟩ ≡
  if (argc > 1 ∧ sscanf(argv[1],"%g",&dampening) ≡ 1) {
    fprintf(stderr,"Using␣dampening␣factor␣%g\n",dampening);
    if (argc > 2 ∧ sscanf(argv[2],"%g",&brightness) ≡ 1)
      fprintf(stderr,"␣␣and␣brightness␣factor␣%g\n",brightness);
  }
```

This code is used in section 1.

3. Macintosh conventions indicate the end of a line by the ASCII ⟨ carriage return ⟩ character (i.e., control-M, aka `\r`), but the C library is set up to work best with newlines (i.e., control-J, aka `\n`). We aren't worried about efficiency, so we simply input one character at a time. This program assumes Macintosh conventions.

The job here is to look for the sequence `Box:` in the input, followed by 0, 0, the number of columns, and the number of rows.

```
#define panic(s)
        {
          fprintf(stderr, s); exit(−1);
        }
⟨ Determine the type of input by looking at the bounding box 3 ⟩ ≡
  k = 0;
scan:
  if (k++ > 1000) panic("Couldn't␣find␣the␣bounding␣box␣info!\n");
  if (getchar( ) ≠ 'B') goto scan;
  if (getchar( ) ≠ 'o') goto scan;
  if (getchar( ) ≠ 'x') goto scan;
  if (getchar( ) ≠ ':') goto scan;
  if (scanf("%d␣%d␣%d␣%d", &llx, &lly, &urx, &ury) ≠ 4 ∨ llx ≠ 0 ∨ lly ≠ 0)
    panic("Bad␣bounding␣box␣data!\n");
  if (urx ≡ n ∧ ury ≡ m) levels = 65;
  else if (urx ≡ n + n ∧ ury ≡ m + m) levels = 17;
  else if (urx ≡ m + n − 1 ∧ ury ≡ urx) levels = 33;
  else panic("Bounding␣box␣doesn't␣match␣the␣formats␣I␣know!\n");
```

This code is used in section 1.

4. ⟨ Global variables 4 ⟩ ≡

```
int llx, lly, urx, ury;      /* bounding box parameters */
```

See also section 8.

This code is used in section 1.

5. After we've seen the bounding box, we look for `beginimage\r`; this will be followed by the pixel data, one character per byte.

```
⟨ Input the graphic data 5 ⟩ ≡
  k = 0;
skan:
  if (k++ > 10000) panic("Couldn't␣find␣the␣pixel␣data!\n");
  if (getchar( ) ≠ 'b') goto skan;
  if (getchar( ) ≠ 'e') goto skan;
  if (getchar( ) ≠ 'g') goto skan;
  if (getchar( ) ≠ 'i') goto skan;
  if (getchar( ) ≠ 'n') goto skan;
  if (getchar( ) ≠ 'i') goto skan;
  if (getchar( ) ≠ 'm') goto skan;
  if (getchar( ) ≠ 'a') goto skan;
  if (getchar( ) ≠ 'g') goto skan;
  if (getchar( ) ≠ 'e') goto skan;
  if (getchar( ) ≠ '\r') goto skan;
  if (levels ≡ 33) ⟨ Input rotated pixel data 7 ⟩
  else ⟨ Input rectangular pixel data 6 ⟩;
  if (getchar( ) ≠ '\r') panic("Wrong␣amount␣of␣pixel␣data!\n");
```

This code is used in section 1.

6. Photoshop follows the conventions of photographers who consider 0 to be black and 1 to be white; but we follow the conventions of computer scientists who tend to regard 0 as devoid of ink (white) and 1 as full of ink (black).

We use the fact that global arrays are initially zero to assume that there are all-white rows of 0s above and below the input data in the rectangular case.

```
⟨ Input rectangular pixel data 6 ⟩ ≡
  for (i = 1; i ≤ ury; i++)
    for (j = 0; j < urx; j++) a[i][j] = 1.0 − brightness ∗ getchar( )/255.0;
```

This code is used in section 5.

7. In the rotated case, we transpose and partially shift the input so that the eventual ith row is in positions $a[i][j + \lfloor i/2 \rfloor]$ for $0 \le j < n$. This arrangement turns out to be most convenient for the output phase.

For example, suppose $m = 5$ and $n = 3$; the input is a 7×7 array that can be expressed in the form

$$\begin{pmatrix} 0 & 0 & 0 & a & A & l & 0 \\ 0 & 0 & b & B & F & J & k \\ 0 & c & C & G & K & O & S \\ d & D & H & L & P & T & j \\ e & I & M & Q & U & i & 0 \\ e & N & R & V & h & 0 & 0 \\ 0 & f & W & g & 0 & 0 & 0 \end{pmatrix}.$$

In practice the boundary values a, b, c, d, e, f, g, h, h, i, j, k, l are very small, so they are essentially "white" and of little importance ink-wise. In this step we transform the input to the configuration

$$\begin{pmatrix} l & k & 0 & 0 & 0 & 0 & 0 \\ A & J & S & 0 & 0 & 0 & 0 \\ a & F & O & j & 0 & 0 & 0 \\ 0 & B & K & T & 0 & 0 & 0 \\ 0 & b & G & P & i & 0 & 0 \\ 0 & 0 & C & L & U & 0 & 0 \\ 0 & 0 & c & H & Q & h & 0 \\ 0 & 0 & 0 & D & M & V & 0 \\ 0 & 0 & 0 & d & I & R & g \\ 0 & 0 & 0 & 0 & E & N & W \end{pmatrix}$$

and later we will output

$$\begin{pmatrix} & l & & k & & 0 \\ A & & J & & S & \\ & F & & O & & j \\ B & & K & & T & \\ & G & & P & & i \\ C & & L & & U & \\ & H & & Q & & h \\ D & & M & & V & \\ & I & & R & & g \\ E & & N & & W & \end{pmatrix}.$$

```
⟨ Input rotated pixel data 7 ⟩ ≡
  {
    for (i = 0; i < ury; i++)
      for (j = 0; j < urx; j++) {
        ii = m + i − j; jj = i + j + 1 − m;
        if (ii ≥ 0 ∧ ii < m + m ∧ jj ≥ 0 ∧ jj < n + n) a[ii][i] = 1.0 − brightness ∗ getchar( )/255.0;
        else trash = getchar( );
      }
    a[0][n − 1] = 1.0 − brightness;      /∗ restore "lost value" ∗/
  }
```

This code is used in section 5.

8. Diffusing the error. We convert the darkness values to 65, 33, or 17 levels by generalizing the Floyd–Steinberg algorithm for adaptive grayscale [*Proceedings of the Society for Information Display* **17** (1976), 75–77]. The idea is to find the best available density value, then to diffuse the error into adjacent pixels that haven't yet been processed.

Given a font with k black dots in character k for $0 \le k \le l$, we might assume that the apparent density of the kth character would be k/l. But physical properties of output devices make the actual density nonlinear. The following table is based on measurements from observations on font `ddith300` with a Canon LBP-CX laserprinter, and it should be accurate enough for practical purposes on similar machines. But in fact the measurements could not be terribly precise, because the readings were not strictly monotone, and because the amount of toner was found to vary between the top and bottom of a page. Users should make their own measurements before adapting this routine to other equipment.

```
⟨ Global variables 4 ⟩ +≡
  float d[65] = {0.000, 0.060, 0.114, 0.162, 0.205, 0.243, 0.276, 0.306, 0.332, 0.355,
      0.375, 0.393, 0.408, 0.422, 0.435, 0.446, 0.456, 0.465, 0.474, 0.482,
      0.490, 0.498, 0.505, 0.512, 0.520, 0.527, 0.535, 0.543, 0.551, 0.559,
      0.568, 0.577, 0.586, 0.596, 0.605, 0.615, 0.625, 0.635, 0.646, 0.656,
      0.667, 0.677, 0.688, 0.699, 0.710, 0.720, 0.731, 0.742, 0.753, 0.764,
      0.775, 0.787, 0.798, 0.810, 0.822, 0.835, 0.849, 0.863, 0.878, 0.894,
      0.912, 0.931, 0.952, 0.975, 1.000};
```

9. In the main loop, we will want to find the best approximation to $a[i][j]$ from among the available densities $d[0]$, $d[p]$, $d[2p]$, $d[3p]$, ..., where p is 1, 2, or 4. A straightforward modification of binary search works well for this purpose:

```
⟨ Find l so that d[l] is as close as possible to a[i][j] 9 ⟩ ≡
  if (a[i][j] ≤ 0.0) l = 0;
  else if (a[i][j] ≥ 1.0) l = 64;
  else { register int lo_l = 0, hi_l = 64;
    while (hi_l − lo_l > p) { register int mid_l = (lo_l + hi_l) ≫ 1;
        /* hi_l − lo_l is halved each time, so mid_l is a multiple of p */
      if (a[i][j] ≥ d[mid_l]) lo_l = mid_l;
      else hi_l = mid_l;
    }
    if (a[i][j] − d[lo_l] ≤ d[hi_l] − a[i][j]) l = lo_l;
    else l = hi_l;
  }
```

This code is used in sections 10 and 11.

10. The rectangular case is simplest, so we consider it first. Our strategy will be to go down each column, starting at the left, and to disperse the error to the four unprocessed neighbors.

```
#define alpha 0.4375 /* 7/16, error diffusion to S neighbor */
#define beta 0.1875 /* 3/16, error diffusion to NE neighbor */
#define gamma 0.3125 /* 5/16, error diffusion to E neighbor */
#define delta 0.0625 /* 1/16, error diffusion to SE neighbor */
```

⟨ Process $a[i][j]$ in the rectangular case 10 ⟩ ≡

```
{ register float err;
  if (i ≡ 0 ∨ i > ury) l = 0; /* must use white outside the output region */
  else ⟨ Find l so that d[l] is as close as possible to a[i][j] 9 ⟩;
  err = a[i][j] − d[l];
  a[i][j] = (float)(l/p); /* henceforth a[i][j] is a level not a density */
  if (i ≤ ury) a[i + 1][j] += alpha ∗ dampening ∗ err;
  if (j < urx − 1) {
    if (i > 0) a[i − 1][j + 1] += beta ∗ dampening ∗ err;
    a[i][j + 1] += gamma ∗ dampening ∗ err;
    if (i ≤ ury) a[i + 1][j + 1] += delta ∗ dampening ∗ err;
  }
}
```

This code is used in section 12.

11. The rotated case is essentially the same, but the unprocessed neighbors of $a[i][j]$ are now $a[i + 1][j]$, $a[i][j + 1]$, $a[i + 1][j + 1]$, and $a[i + 2][j + 1]$. (For example, the eight neighbors of K in the matrices of section 7 are B, F, J, O, T, P, L, G.)

Some of the computation in this step is redundant because the values are known to be zero.

⟨ Process $a[i][j]$ in the rotated case 11 ⟩ ≡

```
{ register float err;
  if ((i ≫ 1) ≤ j − n ∨ (i ≫ 1) > j) l = 0; /* must use white outside the output region */
  else ⟨ Find l so that d[l] is as close as possible to a[i][j] 9 ⟩;
  err = a[i][j] − d[l];
  a[i][j] = (float)(l/p); /* henceforth a[i][j] is a level not a density */
  if (i < m + m − 1) a[i + 1][j] += alpha ∗ dampening ∗ err;
  if (j < m + n − 2) {
    a[i][j + 1] += beta ∗ dampening ∗ err;
    if (i < m + m − 1) a[i + 1][j + 1] += gamma ∗ dampening ∗ err;
    if (i < m + m − 2) a[i + 2][j + 1] += delta ∗ dampening ∗ err;
  }
}
```

This code is used in section 12.

12. Finally we are ready to put everything together.

```
⟨ Translate input to output 12 ⟩ ≡
  p = 64/(levels − 1);
  if (p ≠ 2) {
    for (j = 0; j < urx; j++)
      for (i = 0; i ≤ ury + 1; i++) ⟨ Process a[i][j] in the rectangular case 10 ⟩;
    for (i = 1; i ≤ ury; i++) {
      for (j = 0; j < urx; j++) printf("%c", (p ≡ 1 ? '0' : ((i + j) & 1) ? 'a' : 'A') + (int) a[i][j]);
      printf(".\n");
    }
  }
  else {
    for (j = 0; j < m + n − 1; j++)
      for (i = 0; i < m + m; i++) ⟨ Process a[i][j] in the rotated case 11 ⟩;
    for (i = 0; i < m + m; i++) {
      for (j = 0; j < n; j++) printf("%c", '0' + (int) a[i][j + (i ≫ 1)]);
      printf(".\n");
    }
  }
  printf("\\endhalftone\n");
```

This code is used in section 1.

13. Index.

HALFTONE