(Downloaded from https://cs.stanford.edu/~knuth/programs.html and typeset on May 28, 2023)

1. Introduction. I'm reregenerating the illustrations for my paper in the Transactions on Graphics. This program has little generality, but it could be easily modified.

```
#define m 360   /* this many rows */
#define n 250   /* this many columns */
#define lisacode 1   /* say 1 for Mona Lisa */
#define spherecode 2   /* say 2 for the sphere */
#define fscode 1   /* say 1 for Floyd-Steinberg */
#define odithcode 2   /* say 2 for ordered dither */
#define ddiffcode 3   /* say 3 for dot diffusion */
#define sdiffcode 4   /* say 4 for smooth dot diffusion */
#define ariescode 5   /* say 5 for ARIES */

#include <gb_graph.h>
#include <gb_lisa.h>
#include <math.h>
#include <time.h>
  ⟨Preprocessor definitions⟩
  time_t clokk;
  double A[m + 2][256];   /* pixel data (darknesses), bordered by zero */
  int board[10][10];
  Graph *gg;
  int kk;
  ⟨Global variables 6⟩
  ⟨Subroutines 7⟩
  main(argc, argv)
      int argc;
      char *argv[];
  {
    register int i, j, k, l, ii, jj;
    register double err;
    register Graph *g;
    register Vertex *u, *v;
    register Arc *a;
    int imagecode, sharpcode, methodcode;
    ⟨Scan the command line, give help if necessary 2⟩;
    ⟨Input the image 3⟩;
    ⟨Sharpen if requested 4⟩;
    ⟨Generate and print the base matrix, if any 5⟩;
    ⟨Compute the answer 33⟩;
    ⟨Spew out the answers 29⟩;
    ⟨Print relevant statistics 34⟩;
  }
```

2. ⟨ Scan the command line, give help if necessary 2 ⟩ ≡

```
if (argc ≠ 4 ∨ sscanf(argv[1], "%d", &imagecode) ≠ 1 ∨
        sscanf(argv[2], "%d", &sharpcode) ≠ 1 ∨
        sscanf(argv[3], "%d", &methodcode) ≠ 1) {
usage: fprintf(stderr, "Usage:␣%s␣imagecode␣sharpcode␣methodcode\n", argv[0]);
  fprintf(stderr, "␣Mona␣Lisa␣=␣%d,␣Sphere␣=␣%d\n", lisacode, spherecode);
  fprintf(stderr, "␣unretouched␣=␣0,␣edges␣enhanced␣=␣1\n");
  fprintf(stderr, "␣Floyd-Steinberg␣=␣%d,␣ordered␣dither␣=␣%d,\n", fscode, odithcode);
  fprintf(stderr, "␣dot␣diffusion␣=␣%d,␣smooth␣dot␣diffusion␣=␣%d,\n", ddiffcode, sdiffcode);
  fprintf(stderr, "␣ARIES␣=␣%d\n", ariescode);
  exit(0);
}
```

This code is used in section 1.

3. ⟨ Input the image 3 ⟩ ≡

```
if (imagecode ≡ lisacode) { Area workplace;
  register int *mtx = lisa(m, n, 255, 0, 0, 0, 0, 0, 0, workplace);
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) A[i + 1][j + 1] = pow(1.0 − (*(mtx + i * n + j) + 0.5)/256.0, 2.0);
  fprintf(stderr, "(Mona␣Lisa␣image␣loaded)\n");
}
else if (imagecode ≡ spherecode) {
  for (i = 1; i ≤ m; i++)
    for (j = 1; j ≤ n; j++) {
      register double x = (i − 120.0)/111.5, y = (j − 120.0)/111.5;
      if (x * x + y * y ≥ 1.0) A[i][j] = (1500.0 * i + j * j)/1000000.0;
      else A[i][j] = (9.0 + x − 4.0 * y − 8.0 * sqrt(1.0 − x * x − y * y))/18.0;
    }
  fprintf(stderr, "(Sphere␣image␣loaded)\n");
}
else goto usage;
```

This code is used in section 1.

4. ⟨ Sharpen if requested 4 ⟩ ≡

```
if (sharpcode ≡ 1) {
  for (i = 1; i ≤ m; i++)
    for (j = 1; j ≤ n; j++) A[i − 1][j − 1] = 9 * A[i][j] −
          (A[i − 1][j − 1] + A[i − 1][j] + A[i − 1][j + 1] + A[i][j − 1] +
          A[i][j + 1] + A[i + 1][j − 1] + A[i + 1][j] + A[i + 1][j + 1]);
  for (i = m; i > 0; i−−)
    for (j = n; j > 0; j−−)
      A[i][j] = (A[i − 1][j − 1] ≤ 0.0 ? 0.0 : A[i − 1][j − 1] ≥ 1.0 ? 1.0 : A[i − 1][j − 1]);
  for (i = 0; i < m; i++) A[i][0] = 0.0;
  for (j = 1; j < n; j++) A[0][j] = 0.0;
  fprintf(stderr, "(with␣enhanced␣edges)\n");
}
else if (sharpcode ≡ 0) fprintf(stderr, "(no␣sharpening)\n");
else goto usage;
```

This code is used in section 1.

5. ⟨ Generate and print the base matrix, if any 5 ⟩ ≡

```
switch (methodcode) {
case fscode: fprintf(stderr, "(using␣Floyd-Steinberg␣error␣diffusion)\n"); goto done;
case odithcode: fprintf(stderr, "(using␣ordered␣dithering)\n");
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
      for (k = 0; k < 4; k++) {
        ii = 4 * di[k] + 2 * di[j] + di[i] + 2;
        jj = 4 * dj[k] + 2 * dj[j] + dj[i] + 2;
        kk = 16 * i + 4 * j + k;
        board[8 - (jj & 7)][1 + (ii & 7)] = kk;
      }
  goto finishit;
case ddiffcode: fprintf(stderr, "(using␣dot␣diffusion)\n"); break;
case sdiffcode: fprintf(stderr, "(using␣smooth␣dot␣diffusion)\n"); break;
case ariescode: fprintf(stderr, "(using␣ARIES)\n"); break;
default: goto usage;
}
⟨ Set up the board for dot diffusion 9 ⟩;
finishit:
for (i = 1; i ≤ 8; i++) board[i][0] = board[i][8], board[i][9] = board[i][1];
for (j = 0; j ≤ 9; j++) board[0][j] = board[8][j], board[9][j] = board[1][j];
if (methodcode ≥ ddiffcode) ⟨ Install the vertices and arcs of the control graph 11 ⟩;
⟨ Print the board 10 ⟩;
done:
```

This code is used in section 1.

6. ⟨ Global variables 6 ⟩ ≡

```
int di[4] = {0, 1, 0, 1};
int dj[4] = {0, 1, 1, 0};
```

See also sections 8, 14, 16, 19, and 26.

This code is used in section 1.

7. ⟨ Subroutines 7 ⟩ ≡

```
void store(i, j)
    int i, j;
{
  Vertex *v;
  if (i < 1) i += 8; else if (i > 8) i -= 8;
  if (j < 1) j += 8; else if (j > 8) j -= 8;
  board[i][j] = kk;
  v = gg->vertices + kk;
  sprintf(name_buffer, "%d", kk);
  v->name = gb_save_string(name_buffer);
  v->row = i; v->col = j;
  kk++;
}
void store_eight(i, j)
    int i, j;
{
  store(i, j); store(i - 4, j + 4); store(1 - j, i - 4); store(5 - j, i);
  store(j, 5 - i); store(4 + j, 1 - i); store(5 - i, 5 - j); store(1 - i, 1 - j);
}
```

See also section 25.

This code is used in section 1.

8. ⟨ Global variables 6 ⟩ +≡

```
char name_buffer[] = "99";
```

9.

```
#define row u.I
#define col v.I
#define weight w.I
#define del_i a.I
#define del_j b.I
```

⟨ Set up the board for dot diffusion 9 ⟩ ≡

```
kk = 0;
gg = g = gb_new_graph(64);
store_eight(7, 2); store_eight(8, 3); store_eight(8, 2); store_eight(8, 1);
store_eight(1, 4); store_eight(1, 3); store_eight(1, 2); store_eight(2, 3);
```

This code is used in section 5.

10. ⟨ Print the board 10 ⟩ ≡

```
for (i = 1; i <= 8; i++) {
  for (j = 1; j <= 8; j++) fprintf(stderr, "␣%2d", board[i][j]);
  fprintf(stderr, "\n");
}
```

This code is used in section 5.

11. ⟨Install the vertices and arcs of the control graph 11⟩ ≡

```
if (methodcode ≡ ddiffcode) {      /* dot diffusion, two dots per 8 × 8 cell */
  for (v = g→vertices; v < g→vertices + 64; v++) {
    i = v→row;
    j = v→col;
    v→weight = 0;
    for (ii = i − 1; ii ≤ i + 1; ii++)
      for (jj = j − 1; jj ≤ j + 1; jj++) {
        u = g→vertices + board[ii][jj];
        if (u > v) {
          gb_new_arc(v, u, 0);
          v→arcs→del_i = ii − i;
          v→arcs→del_j = jj − j;
          v→weight += 3 − (ii − i) ∗ (ii − i) − (jj − j) ∗ (jj − j);
        }
      }
  }
}
else {      /* each vertex has a neighborhood covering 32 classes */
  for (v = g→vertices; v < g→vertices + 64; v++) {
    i = v→row;
    j = v→col;
    for (jj = j − 3; jj ≤ j + 3; jj++) { register int del = (jj < j ? j − jj : jj − j);
      for (ii = i − 3 + del; ii ≤ i + 4 − del; ii++) {
        u = g→vertices + board[ii & 7][jj & 7];
        if (u > v) {
          gb_new_arc(v, u, 0);
          v→arcs→del_i = ii − i;
          v→arcs→del_j = jj − j;
        }
      }
    }
  }
  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++) board[i][j] ≫= 1;
}
```

This code is used in section 5.

12. Error diffusion. The Floyd-Steinberg algorithm uses a threshold of 0.5 at each pixel and distributes the error to the four unprocessed neighbors.

```
#define alpha 0.4375  /* 7/16, error diffusion to E neighbor */
#define beta 0.1875  /* 3/16, error diffusion to SW neighbor */
#define gamma 0.3125  /* 5/16, error diffusion to S neighbor */
#define delta 0.0625  /* 1/16, error diffusion to SE neighbor */
#define check(i, j)
        {
          if (A[i][j] < lo_A) lo_A = A[i][j];
          if (A[i][j] > hi_A) hi_A = A[i][j];
        }
⟨Do Floyd-Steinberg 12⟩ ≡
  for (i = 1; i ≤ m; i++)
    for (j = 1; j ≤ n; j++) {
      err = A[i][j];
      if (err ≥ .5) err −= 1.0;
      A[i][j] −= err;  /* now it's 0 or 1 */
      A[i][j + 1] += err ∗ alpha; check(i, j + 1);
      A[i + 1][j − 1] += err ∗ beta; check(i + 1, j − 1);
      A[i + 1][j] += err ∗ gamma; check(i + 1, j);
      A[i + 1][j + 1] += err ∗ delta; check(i + 1, j + 1);
    }
```

This code is used in section 33.

13. ⟨Print boundary leakage and extreme values 13⟩ ≡

```
  if (methodcode ≠ sdiffcode) {
    for (i = 0; i ≤ m + 1; i++) edge_accum += fabs(A[i][0]) + fabs(A[i][n + 1]);
    for (j = 1; j ≤ n; j++) edge_accum += fabs(A[0][j]) + fabs(A[m + 1][j]);
  }
  fprintf(stderr, "Total␣leakage␣at␣boundaries:␣%.20g\n", edge_accum);
  fprintf(stderr, "Data␣remained␣between␣%.20g␣and␣%.20g\n", lo_A, hi_A);
```

This code is used in section 34.

14. ⟨Global variables 6⟩ +≡

```
  double edge_accum;
  double lo_A = 100000.0, hi_A = −100000.0;  /* record-breaking values */
```

15. Ordered dithering. The ordered dither algorithm uses a threshold based on the pixel's place in the grid.

⟨ Do ordered dither 15 ⟩ ≡

```
for (i = 1; i ≤ m; i++)
  for (j = 1; j ≤ n; j++) {
    k = board[i & 7][j & 7];
    err = A[i][j];
    if (err ≥ (k + 0.5)/64.0) err −= 1.0;
    A[i][j] −= err;   /* now it's 0 or 1 */
    accum += fabs(err);   /* accumulate undiffused error */
    block_err[(i − 1) ≫ 3][(j − 1) ≫ 3] += err;   /* accumulate error in 8 × 8 block */
  }
```

This code is used in section 33.

16. ⟨ Global variables 6 ⟩ +≡

```
double accum;
double block_err[(m + 7) ≫ 3][(n + 7) ≫ 3];
int bad_blocks;
```

17. ⟨ Print accumulated lossage 17 ⟩ ≡

```
fprintf(stderr, "Total␣undiffused␣error:␣%.20g\n", accum);
for (i = 0, accum = 0.0; i < m; i += 8)
  for (j = 0; j < n; j += 8) {
    if (fabs(block_err[i ≫ 3][j ≫ 3]) > 1.0) bad_blocks++;
    accum += fabs(block_err[i ≫ 3][j ≫ 3]);
  }
fprintf(stderr, "Total␣block␣error:␣%.20g␣(%d␣bad)\n", accum, bad_blocks);
```

This code is used in section 34.

18. Dot diffusion. The dot diffusion algorithm uses a fixed threshold of 0.5 and distributes errors to higher-class neighbor pixels, except at baron positions.

⟨ Do dot diffusion 18 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + 64; v++)
  for (i = v→row; i ≤ m; i += 8)
    for (j = v→col; j ≤ n; j += 8) {
      err = A[i][j];
      if (err ≥ .5) err −= 1.0;
      A[i][j] −= err;   /* now it's 0 or 1 */
      if (v→arcs) ⟨ Distribute the error to near neighbors 20 ⟩
      else {   /* baron */
        accum += fabs(err);
        barons++;
        if (fabs(err) > 0.5) bad_barons++;
        if (err < lo_err) lo_err = err;
        if (err > hi_err) hi_err = err;
      }
    }
```

This code is used in section 33.

19. ⟨ Global variables 6 ⟩ +≡

```
int barons;   /* how many barons are there? */
int bad_barons;   /* how many of them eat more than 0.5 error? */
double lo_err = 100000.0, hi_err = −100000.0;   /* record-breaking errors */
```

20. ⟨ Distribute the error to near neighbors 20 ⟩ ≡

```
for (a = v→arcs; a; a = a→next) {
  ii = i + a→del_i; jj = j + a→del_j;
  A[ii][jj] += err * (double)(3 − a→del_i * a→del_i − a→del_j * a→del_j)/(double) v→weight;
  check(ii, jj);
}
```

This code is used in section 18.

21. Smooth dot diffusion is similar, but it uses a class-based threshold and considers a larger neighborhood of size 32.

⟨ Do smooth dot diffusion 21 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + 64; v++)
  for (i = v→row; i ≤ m; i += 8)
    for (j = v→col; j ≤ n; j += 8) {
      k = (v − g→vertices) ≫ 1;    /* class number */
      err = A[i][j];
      if (err ≥ .5/(double)(32 − k)) err −= 1.0;
      A[i][j] −= err;    /* now it's 0 or 1 */
      if (v→arcs) ⟨ Distribute the error to dot neighbors 22 ⟩
      else {    /* baron */
        accum += fabs(err);
        barons++;
        if (fabs(err) > 0.5) bad_barons++;
        if (err < lo_err) lo_err = err;
        if (err > hi_err) hi_err = err;
      }
    }
```

This code is used in section 33.

22. This pixel has $31 - k$ neighbors of higher classes; each shares equally in the distribution.

⟨ Distribute the error to dot neighbors 22 ⟩ ≡

```
for (a = v→arcs; a; a = a→next) {
  ii = i + a→del_i; jj = j + a→del_j;
  if (ii > 0 ∧ ii ≤ m ∧ jj > 0 ∧ jj ≤ n) {
    A[ii][jj] += err/(double)(31 − k); check(ii, jj);
  }
  else edge_accum += fabs(err);    /* error leaks out the boundary */
}
```

This code is used in section 21.

23. ⟨ Print baronial lossage 23 ⟩ ≡

```
fprintf(stderr, "Total␣undiffused␣error␣%.20g␣at␣%d␣barons\n", accum, barons);
fprintf(stderr, "␣␣(%d␣bad,␣min␣%.20g,␣max␣%.20g)\n", bad_barons, lo_err, hi_err);
```

This code is used in section 34.

24. Alias-Reducing Image-Enhancing Screening. The ARIES method works with 32-pixel dots and dithers them but adjusts the threshold by considering the average intensity in the dot.

⟨ Do ARIES 24 ⟩ ≡

```
for (i = −1; i ≤ m + 3; i += 4)
  for (j = (i & 4) ? 2 : −2; j ≤ n + 3; j += 8) { double s = 0.5;
    ll = 0;    /* number of cells in current dot */
    for (jj = j − 3; jj ≤ j + 3; jj++) { register int del = (jj < j ? j − jj : jj − j);
      for (ii = i − 3 + del; ii ≤ i + 4 − del; ii++)
        if (ii > 0 ∧ ii ≤ m ∧ jj > 0 ∧ jj ≤ n) s += A[ii][jj], rank(ii, jj);
    }
    ⟨ Blacken the top ⌊s⌋ pixels of the dot 27 ⟩;
  }
```

This code is used in section 33.

25. The ranking procedure sorts the entries by the key $a_{ij} - k/32$, where k is the class number of cell (i, j).

⟨ Subroutines 7 ⟩ +≡

```
rank(i, j)
    int i, j;
{
  register double key = A[i][j] − board[i & 7][j & 7]/32.0;
  register int l;
  for (l = ll; l > 0; l−−)
    if (key ≥ val[l − 1]) break;
    else inxi[l] = inxi[l − 1], inxj[l] = inxj[l − 1], val[l] = val[l − 1];
  inxi[l] = i; inxj[l] = j; val[l] = key; ll++;
}
```

26. ⟨ Global variables 6 ⟩ +≡

```
int ll;    /* the number of items in the ranking table */
int inxi[32], inxj[32];    /* indices of the ranked pixels */
double val[32];    /* keys of the ranked pixels */
```

27. I have to admit that I rather like this implementation of ARIES!

⟨ Blacken the top ⌊s⌋ pixels of the dot 27 ⟩ ≡

```
if (ll) { barons++; accum += fabs(s − 0.5 − (int) s); }
while (ll > 0) {
  ll−−; s −= 1.0;
  ii = inxi[ll]; jj = inxj[ll];
  err = A[ii][jj];
  if (s ≥ 0.0) err −= 1.0;
  A[ii][jj] −= err;    /* now it's 0 or 1 */
}
```

This code is used in section 24.

28. ⟨ Print ARIES lossage 28 ⟩ ≡

```
fprintf(stderr, "Total␣lossage␣%.20g␣in␣%d␣dots\n", accum, barons);
```

This code is used in section 34.

29. Encapsulated PostScript. When all has been done (but all has not necessarily been said), we output the matrix as a PostScript file with resolution 72 pixels per inch.

⟨ Spew out the answers 29 ⟩ ≡
  ⟨ Output the header of the EPS file 30 ⟩;
  ⟨ Output the image 31 ⟩;
  ⟨ Output the trailer of the EPS file 32 ⟩;

This code is used in section 1.

30. ⟨ Output the header of the EPS file 30 ⟩ ≡

```
printf("%%!PS\n");
printf("%%%%BoundingBox:␣0␣0␣%d␣%d\n", n, m);
printf("%%%%Creator:␣togpap\n");
clokk = time(0);
printf("%%%%CreationDate:␣%s", ctime(&clokk));
printf("%%%%Pages:␣1\n");
printf("%%%%EndProlog\n");
printf("%%%%Page:␣1␣1\n");
printf("/picstr␣%d␣string␣def\n", (n + 7) ≫ 3);
printf("%d␣%d␣scale\n", n, m);
printf("%d␣%d␣true␣[%d␣0␣0␣-%d␣0␣%d]\n", n, m, n, m, m);
printf("␣{currentfile␣picstr␣readhexstring␣pop}␣imagemask\n");
```

This code is used in section 29.

31. ⟨ Output the image 31 ⟩ ≡

```
for (i = 1; i ≤ m; i++) {
  for (j = 1; j ≤ n; j += 8) {
    for (k = 0, l = 0; k < 8; k++) l = l + l + (A[i][j + k] ? 1 : 0);
    printf("%02x", l);
  }
  printf("\n");
}
```

This code is used in section 29.

32. ⟨ Output the trailer of the EPS file 32 ⟩ ≡

```
printf("%%%%EOF\n");
```

This code is used in section 29.

33. Synthesis. And now to put the pieces together:

⟨ Compute the answer 33 ⟩ ≡
  switch (*methodcode*) {
  case *fscode*: ⟨ Do Floyd-Steinberg 12 ⟩; **break**;
  case *odithcode*: ⟨ Do ordered dither 15 ⟩; **break**;
  case *ddiffcode*: ⟨ Do dot diffusion 18 ⟩; **break**;
  case *sdiffcode*: ⟨ Do smooth dot diffusion 21 ⟩; **break**;
  case *ariescode*: ⟨ Do ARIES 24 ⟩; **break**;
  }

This code is used in section 1.

34. ⟨ Print relevant statistics 34 ⟩ ≡
  switch (*methodcode*) {
  case *odithcode*: ⟨ Print accumulated lossage 17 ⟩; **break**;
  case *ariescode*: ⟨ Print ARIES lossage 28 ⟩; **break**;
  case *ddiffcode*: **case** *sdiffcode*: ⟨ Print baronial lossage 23 ⟩;
  case *fscode*: ⟨ Print boundary leakage and extreme values 13 ⟩; **break**;
  }

This code is used in section 1.

35. Index.

TOGPAP