WRITTEN BY MARCEL K. GOH. LAST UPDATED JULY 30, 2021 AT 21:16

1. Introduction. This literate program contains various functions for experimenting with points in the unit ball of L^p-space over $\mathbf{R}^d$. We will maintain a set of points in this space, and provide functionality for randomly generating new ones, as well as printing the current state of the point set.

2. This is the main outline of the program. We have a couple of global variables storing $p \in (0, \infty]$, the dimension d, the set *points* of points we are working with (and the maximum size *max_points* of this array). The three parameters will be supplied by the user via command-line arguments. One specifies the case $p = \infty$ by supplying a negative number for p. We represent vectors in $\mathbf{R}^d$ as C **double** arrays of length $d + 1$. While perhaps a bit wasteful, this allows us to index from 1 to d rather than from 0 to $d - 1$.

```
#include <float.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
  double p;
  double p_inv;
  int d;
  int max_points;
  int num_points;
  double **points;
  ⟨ Print points to PostScript file 12 ⟩;
  ⟨ Point set functions 3 ⟩;
  ⟨ Random variate generation 4 ⟩;
  ⟨ Committee-generating algorithm 15 ⟩;
  ⟨ Inner product computation 19 ⟩;
  ⟨ Draw loci when p = ∞ 20 ⟩;
  int main(int argc, char *argv[])
  {
    ⟨ Gather input and initialise global variables 24 ⟩;
    ⟨ Seed the pseudorandom number generator 5 ⟩;
    if (d ≡ 2 ∧ p < 0) random_locus(5, "locus.ps", 150, 30);
  }
```

3. The point set. This section contains functions on points in L^p-space. First, we provide facilities to add new points, delete all points, and to list the current point set on the console. We use the current size of the point set, stored in the variable *num_points* to help index into the *points* array. Clearing the point set is done by setting *num_points* to 0, so everything in the array at and after the index *num_points* possibly contains garbage values.

```
⟨ Point set functions 3 ⟩ ≡
  void add_point(double *point)
  {
    if (num_points ≥ max_points) {
      printf("I␣failed␣to␣add␣point␣because␣array␣is␣full.\n");
      return;
    }
    for (int i = 1; i ≤ d; ++i) points[num_points][i] = point[i];
    ++num_points;
  }
  void clear_points( )
  {
    num_points = 0;
  }
  void list_points( )
  {
    for (int i = 0; i < num_points; ++i) {
      printf("(");
      for (int j = 1; j ≤ d; ++j) {
        printf("%.2f", points[i][j]);
        if (j ≠ d) printf(",␣");
      }
      printf(")\n");
    }
  }
```

This code is used in section 2.

4. Random variate generation. The main goal is to generate a point uniformly at random from the interior of the unit ball in L^p-space over $\mathbf{R}^d$. To do this, we will require an exponential random variable with mean 1.

```
⟨ Random variate generation 4 ⟩ ≡
  ⟨ Uniform and exponential variates 6 ⟩;
  ⟨ Normal random variate 11 ⟩;
  ⟨ Gamma random variate 7 ⟩;
  ⟨ Uniform random point in unit ball 8 ⟩;
```

This code is used in section 2.

5. First, in the *main* function, we initialise a pseudorandom number generator with the current time.

```
⟨ Seed the pseudorandom number generator 5 ⟩ ≡
  time_t t;
  srand((unsigned) time(&t));
  for (int i = 0; i < 10; ++i) {
    rand();
  }
```

This code is used in section 2.

6. To generate our exponential random variable, we use von Neumann's algorithm, as described by L. Devroye on p. 126 of *Non-Uniform Random Variate Generation* (New York: Springer, 1986).

⟨ Uniform and exponential variates 6 ⟩ ≡

```
double uniform01()
{
  return ((double) rand())/RAND_MAX;
}
double exponential1()
{
  int Z = 0;
  double Y;
  int k;
  do {
    Y = uniform01();
    k = 1;
    double W = Y;
    int stop = 0;
    do {
      double U = uniform01();
      if (U > W) {
        stop = 1;
      }
      else {
        W = U;
        ++k;
      }
    } while (!stop);
    ++Z;
  } while (k % 2 == 0);
  return ((double)(Z - 1)) + Y;
}
```

This code is used in section 4.

7. We will also make use of the gamma distribution, which, for a parameter $a > 0$, has density

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}.$$

We use an algorithm of R. C. H. Cheng [*Applied Statistics* **26** (1977), 71–75], described on p. 413 of *Non-Uniform Random Variate Generation*, which works for $a \geq 1$. (Note that there is a mistake in the printed algorithm, but the corrigenda on Devroye's website contain the required amendments.)

⟨ Gamma random variate 7 ⟩ ≡

```
double gamma_dist(double a)
{
  double b = a − log(4);
  double lambda = sqrt(2 ∗ a − 1);
  double c = a + lambda;
  int accept = 0;
  double U, V, Y, X, Z, R;
  double S = 4.5 ∗ Z − (1 + log(4.5));
  do {
    U = uniform01();
    V = uniform01();
    Y = (1.0/lambda) ∗ log(V/(1 − V));
    X = a ∗ exp(Y);
    Z = U ∗ V ∗ V;
    R = b + c ∗ Y − X;
    accept = (R ≥ S);
    if (¬accept) accept = (R ≥ log(Z));
  } while (¬accept);
  return X;
}
```

This code is used in section 4.

8. To get a uniform random point from the unit ball in $L^p(\mathbf{R}^d)$, we sample d independent random variables, call them $X_1, \ldots, X_d$ from the density

$$f(x) = \frac{1}{2\Gamma(1+1/p)} e^{-|x|^p}.$$

This is called an exponential power distribution, and Devroye notes in *Non-Uniform Random Variate Generation* that if $X = VY^{1/p}$ with V uniformly distributed on $[-1,1]$ and Y is gamma-distributed with parameter $1+1/p$, then X has the density $f(x)$. We also sample an exponential random variable Y with mean 1, and then output the random vector

$$\frac{(X_1, \ldots, X_d)}{(Y + \sum_{i=1}^{d} |X_i|^p)^{1/p}}.$$

This method and extensions thereof are described by F. Barthe, O. Guedon, S. Mendelson, and A. Naor [*Annals of Probability* **33** (2005), 480–513]. The function we write returns a point by modifying an array that is passed as an argument.

```
⟨ Uniform random point in unit ball 8 ⟩ ≡
  void random_point(double *point)
  {
    double *X = malloc((d + 1) * sizeof(double));
    for (int i = 1; i ≤ d; ++i) X[i] = 0.0;
    if (p < 0) {
      ⟨ The case p = ∞ 9 ⟩;
    }
    else {
      ⟨ The case p ≠ ∞ 10 ⟩;
    }
    free(X);
  }
```

This code is used in section 4.

9. When $p = \infty$, we simply return a point uniformly from the box $[-1,1]^d$.

```
⟨ The case p = ∞ 9 ⟩ ≡
  for (int i = 1; i ≤ d; ++i) point[i] = 2 * uniform01() − 1;
```

This code is used in section 8.

10. In all the other cases, we sample using the method described above.

```
⟨ The case p ≠ ∞ 10 ⟩ ≡
  for (int i = 1; i ≤ d; ++i) {
    double V = 2 * uniform01() − 1;
    X[i] = V * pow(gamma_dist(1 + p_inv), p_inv);
  }
  double Y = exponential1();
  double pow_sum = Y;
  for (int i = 1; i ≤ d; ++i) pow_sum += pow(fabs(X[i]), p);
  double scale_factor = 1.0/(pow(pow_sum, p_inv));
  for (int i = 1; i ≤ d; ++i) point[i] = scale_factor * X[i];
```

This code is used in section 8.

11. I originally needed a normal random deviate to handle the case $p = 2$, so I added this function, but it is no longer necessary since we now have a general function. I left it here just for kicks. We generate the normal using the Box-Muller transform, due to G. E. P. Box and M. E. Muller [*Annals of Mathematical Statistics* **29** (1958), 610–611]. If U_1 and U_2 are two independent random variables uniformly distributed in $[0,1]$, then

$$V_1 = \sqrt{-2\ln U_1}\cos(2\pi U_2) \qquad \text{and} \qquad V_2 = \sqrt{-2\ln U_1}\sin(2\pi U_2)$$

are independent normal random variables with mean 0 and variance 1. To make the return type of our function simpler, we simply output one of these values (so we will end up calling this function twice as often as is actually necessary).

```
⟨ Normal random variate 11 ⟩ ≡
  double normal01( )
  {
    double U1 = uniform01( );
    double U2 = uniform01( );
    double scale = sqrt(−2.0 ∗ log(U1));
    return scale ∗ cos(2 ∗ M_PI ∗ U2);
  }
```

This code is used in section 4.

12. File output. When $d = 2$, it is easy to plot our set of points graphically. Our program does this by generating a PostScript file.

```
⟨ Print points to PostScript file 12 ⟩ ≡
  void plot_single_point(FILE *file, double red, double green, double blue, double x, double y)
  {
    fprintf(file, "%f %f %f setrgbcolor %f %f dot\n", red, green, blue, x, y);
  }
  void to_postscript(const char *filename, int radius)
  {
    if (d ≠ 2) {
      printf("I cannot output PostScript unless d equals 2!\n");
      return;
    }
    FILE *file = fopen(filename, "w");
    ⟨ Preamble 13 ⟩;
    ⟨ Plot the points array 14 ⟩;
    fprintf(file, "showpage\n");
    fclose(file);
  }
```

This code is used in section 2.

13. We first add a bare-bones preamble to the file to draw the axes and declare various PostScript functions. The axes are drawn with center at (*radius*, *radius*)

```
⟨ Preamble 13 ⟩ ≡
  char *preamble = "%!PS\n/dot { 1.5 0 360 arc closepath fill } def\n/square { \
      /r exch def /y exch def /x exch def\nnewpath x r sub y r sub moveto 0 r 2 mul rl\
      ineto\nr 2 mul 0 rlineto 0 r 2 mul neg rlineto\nr 2 mul neg 0 rlineto closepath \
      fill} def\n0.5 setlinewidth\n";
  fprintf(file, "%s", preamble);
  fprintf(file, "%d %d translate\n", radius, radius);
  fprintf(file, "newpath 0 %d moveto 0 %d lineto ", −radius, radius);
  fprintf(file, "%d 0 moveto %d 0 lineto stroke\n", −radius, radius);
```

This code is used in sections 12 and 23.

14. Next, we plot each of the elements of the *points* array, with a colour gradient to indicate the relative orderings of points. The first point in the array is drawn in *colour1*, the final point is drawn in *colour2*, and points in between have their colours interpolated accordingly.

```
⟨ Plot the points array 14 ⟩ ≡
  double colour1[] = {1.0, 0.0, 0.0};    /* red */
  double colour2[] = {0.0, 0.0, 1.0};    /* blue */
  for (int i = 0; i < num_points; ++i) {
    double t = ((double) i)/((double)(num_points − 1));
    double r = (1.0 − t) * colour1[0] + t * colour2[0];
    double g = (1.0 − t) * colour1[1] + t * colour2[1];
    double b = (1.0 − t) * colour1[2] + t * colour2[2];
    plot_single_point(file, r, g, b, points[i][1] * radius, points[i][2] * radius);
  }
```

This code is used in section 12.

15. Growing a committee by consensus voting. Consider the point set as representing a group of people (each person is a vector of real numbers each describing a different trait). The committee grows in discrete time steps. At each step, two candidates are presented to the committee, and each committee member votes for the candidate closest to itself in L^p-distance. A candidate can only win by *consensus*; that is, if any two committee members vote for different candidates, then the election is inconclusive and no new member is added.

```
⟨ Committee-generating algorithm 15 ⟩ ≡
  ⟨ Distance computation 16 ⟩;
  ⟨ Consensus algorithm 17 ⟩;
```

This code is used in section 2.

16. First we need to compute distances between two points $x = (x_1, \ldots, x_d)$ and $y = (y_1, \ldots, y_d)$ in $L^p(\mathbf{R}^d)$. When $p \neq \infty$, this is given by the formula

$$\|x - y\|_p = \Bigl(\sum_{i=1}^{d} |x_i - y_i|^p\Bigr)^{1/p},$$

and when $p = \infty$, we simply take the maximum of the coordinate-wise distances:

$$\|x - y\|_\infty = \max_{1 \le i \le d} |x_i - y_i|$$

```
⟨ Distance computation 16 ⟩ ≡
  double p_dist(double *x, double *y)
  {
    if (p < 0) {
      double max = DBL_MIN;
      for (int i = 1; i ≤ d; ++i) {
        double diff = fabs(x[i] − y[i]);
        if (diff > max) max = diff;
      }
      return max;
    }
    else {
      double sum = 0;
      for (int i = 1; i ≤ d; ++i) sum += pow(fabs(x[i] − y[i]), p);
      return pow(sum, 1.0/p);
    }
  }
```

This code is used in section 15.

17. Now we present the algorithm for growing the committee, which takes a parameter indicating how many rounds of voting should be undertaken. We start by clearing the point set and initialising the committee with a point chosen uniformly at random from the unit ball (this counts as the first round of voting). In each round, we compute distances from each point to the two candidates and keep track of whether each candidate has votes. If at any point, both candidates have votes, we can continue to the next election, since that round is inconclusive. The return value is the size of the committee after all rounds have elapsed.

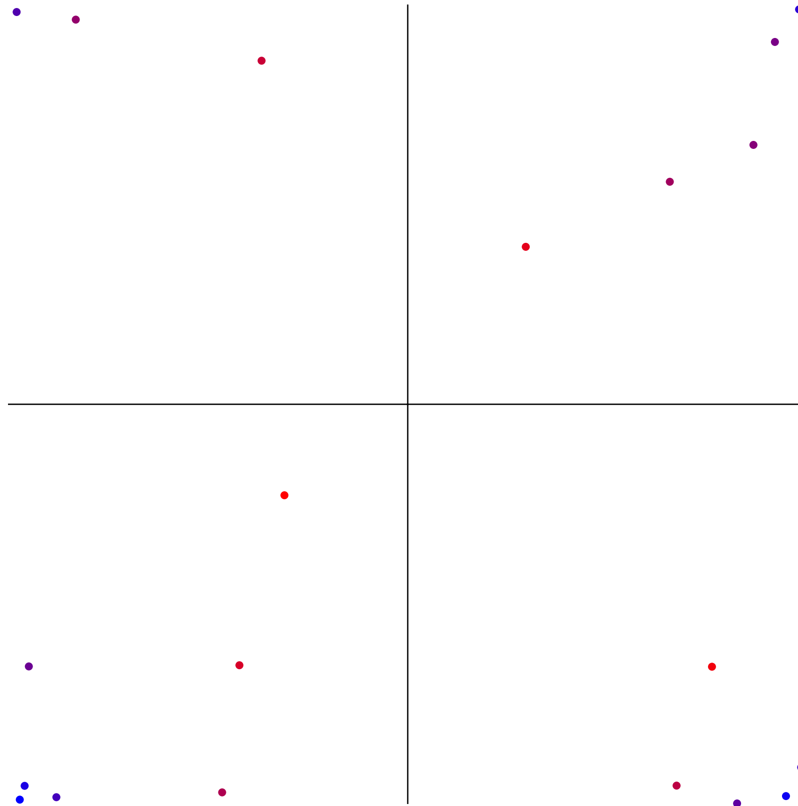
⟨ Consensus algorithm 17 ⟩ ≡

```
int consensus(int max_t, double(*dist)(double *, double *))
{
  clear_points();
  double *cand1 = malloc((d + 1) * sizeof(double));
  double *cand2 = malloc((d + 1) * sizeof(double));
  random_point(cand1);
  add_point(cand1);
  for (int t = 0; t < max_t − 1; ++t) {
    random_point(cand1);
    random_point(cand2);
    int voted1 = 0;
    int voted2 = 0;
    for (int i = 0; i < num_points; ++i) {
      double dist1 = dist(points[i], cand1);
      double dist2 = dist(points[i], cand2);
      if (dist1 > dist2) {
        voted1 = 1;
      }
      else {
        voted2 = 1;
      }
      if (voted1 ∧ voted2) break;
    }
    if (voted1 ∧ ¬voted2) add_point(cand1);
    if (voted2 ∧ ¬voted1) add_point(cand2);
  }
  free(cand1);
  free(cand2);
  return num_points;
}
```

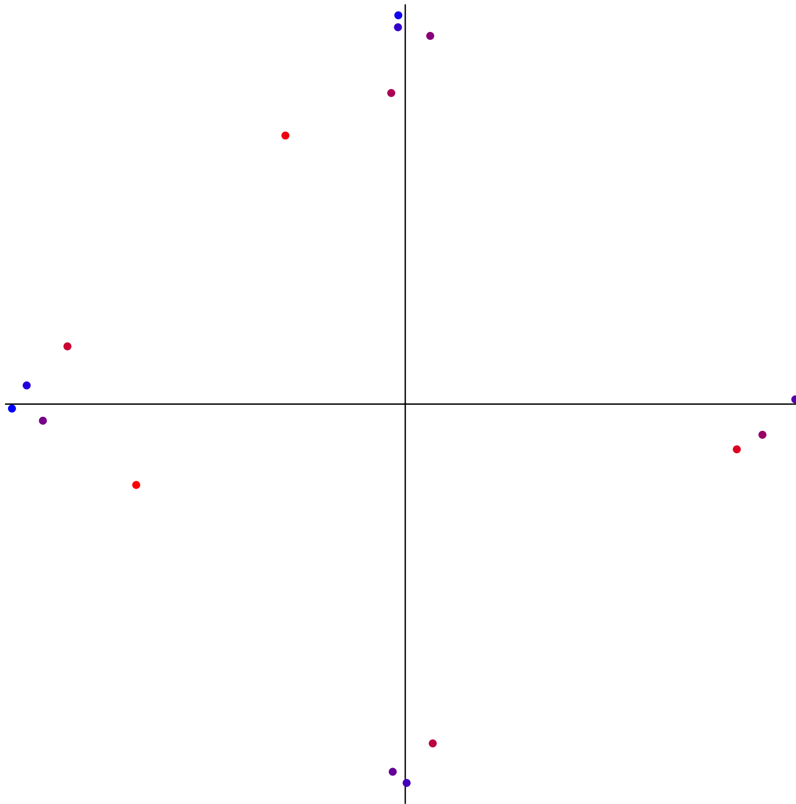
This code is used in section 15.

18. Illustrations and examples. Here are some examples of the output of the committee-generating algorithm. In each case, $1\,000,000$ rounds of voting were conducted. In each figure, redder points were added earlier and bluer points were added later.

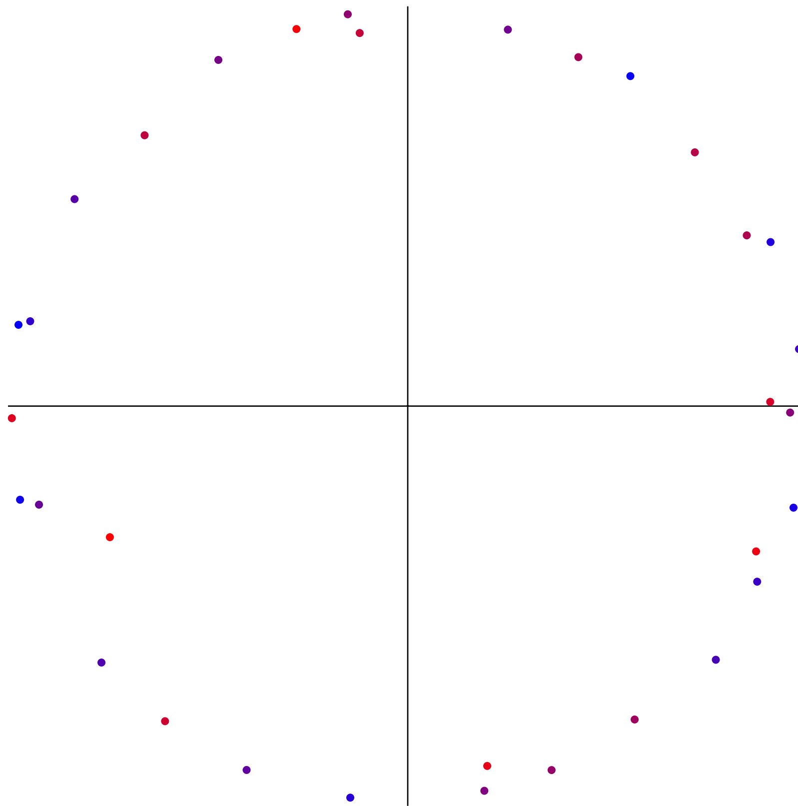
For $p = \infty$, a committee of 20 members was formed:



For $p = 1$, a committee of 16 members was formed:



And for $p = 2$, a committee of 33 members was formed:



These committee sizes seemed representative of typical behaviour, in that running the program multiple times with the same parameters did not produce wildly different values in any of the cases.

19. Orthogonal committees. We now restrict ourselves to the case $p = 2$, when $L^p(\mathbf{R}^d)$ is an inner-product space. We can perform the same experiment, but instead of measuring distance with the L^p-norm, we can measure distance between two vectors as the absolute value of their inner product.

```
⟨ Inner product computation 19 ⟩ ≡
  double inner_product(double *x, double *y)
  {
    if (p ≠ 2) {
      printf("I can only take inner products when p = 2!\n");
      return 0.0;
    }
    double sum = 0.0;
    for (int i = 1; i ≤ d; ++i) sum += x[i] * y[i];
    return sum;
  }
  double abs_inner_product(double *x, double *y)
  {
    return fabs(inner_product(x, y));
  }
  double one_minus_inner_product(double *x, double *y)
  {
    return 1.0 − inner_product(x, y);
  }
```

This code is used in section 2.

20. Locus of points further to committee than candidate. In this section, we consider the case when $p = \infty$ and $d = 2$. Given a committee G of points and a candidate c, we will draw the locus of all points c' such that the committee G will reach a consensus and elect c over c'.

```
⟨ Draw loci when p = ∞ 20 ⟩ ≡
  ⟨ Draw squares in PostScript 21 ⟩;
  ⟨ Locus computation 22 ⟩;
  ⟨ Generate a drawing with random points 23 ⟩;
```

This code is used in section 2.

21. First, we supply a facility to draw L^∞ balls, which look like squares, in PostScript.

```
⟨ Draw squares in PostScript 21 ⟩ ≡
  void draw_square(FILE *file, double red, double green, double blue, double x, double y, double r)
  {
    fprintf(file, "%f␣%f␣%f␣setrgbcolor␣%f␣%f␣%f␣square\n", red, green, blue, x, y, r);
  }
```

This code is used in section 20.

22. Next, given a set of points $G = \{g_1, \ldots, g_n\}$ representing a committee as well as a candidate point c, the locus we seek is

$$[0,1]^2 \setminus \bigcup_{i=1}^{n} B_\infty\big(g_i, d_\infty(g_i, c)\big),$$

where $d_\infty(x,y) = \|x - y\|_\infty$ and $B_\infty(x,r)$ is the set of all points y with $d_\infty(x,y) < r$. The way we draw this is to first fill in $[0,1]^2$ in red, then superimpose various balls in blue. The remaining red area is the locus of all points that is further from each committee member than the candidate is.

```
⟨ Locus computation 22 ⟩ ≡
  void draw_locus(FILE *file, int committee_size, double **committee, double *candidate, int radius)
  {
    draw_square(file, 1, 0, 0, 0, 0, radius);    /* background drawn in red */
    for (int i = 0; i < committee_size; ++i) {    /* complement of locus drawn in blue */
      draw_square(file, 0.2, 0, 1, committee[i][1] * radius, committee[i][2] * radius, p_dist(committee[i],
          candidate) * radius);
    }
    for (int i = 0; i < committee_size; ++i) {    /* draw committee in black */
      plot_single_point(file, 0, 0, 0, committee[i][1] * radius, committee[i][2] * radius);
    }
    plot_single_point(file, 0, 1, 0, candidate[1] * radius, candidate[2] * radius);    /* candidate in green */
    draw_square(file, 1, 1, 1, 2 * radius, 0, radius);    /* clean up with white squares */
    draw_square(file, 1, 1, 1, 0, 2 * radius, radius);
    draw_square(file, 1, 1, 1, 2 * radius, 2 * radius, radius);
  }
```

This code is used in section 20.

23. Lastly, we sample random points and generate the corresponding drawing.

```
⟨ Generate a drawing with random points 23 ⟩ ≡
  void random_locus(int committee_size, const char *filename, int radius, int num_pages)
  {
    double *candidate = malloc((d + 1) * sizeof(double));
    double **committee = malloc(committee_size * sizeof(double *));
    for (int i = 0; i < committee_size; ++i) committee[i] = malloc((d + 1) * sizeof(double));
    if (d ≠ 2) {
      printf("I␣cannot␣output␣PostScript␣unless␣d␣equals␣2!\n");
      return;
    }
    FILE *file = fopen(filename, "w");
    ⟨ Preamble 13 ⟩;
    for (int page = 0; page < num_pages; ++page) {
      for (int i = 0; i < committee_size; ++i) random_point(committee[i]);
      random_point(candidate);
      if (page ≠ 0) fprintf(file, "%d␣%d␣translate\n", radius, radius);
      draw_locus(file, committee_size, committee, candidate, radius);
      fprintf(file, "showpage\n");
    }
    fclose(file);
    free(candidate);
    for (int i = 0; i < committee_size; ++i) free(committee[i]);
  }
```

This code is used in section 20.

24. Handling user input. These components of the *main* function deal with command-line input. The program takes up to three command-line arguments in the following order: p, d, and *max_points*. If some or all of these arguments are missing, we default to $p = 2$, $d = 2$, and *max_points* = 300.

⟨ Gather input and initialise global variables 24 ⟩ ≡

```
p = 2.0;
d = 2;
max_points = 300;
num_points = 0;
if (argc ≥ 2) {
  p = atof(argv[1]);
  p_inv = 1.0/p;
}
if (argc ≥ 3) {
  d = atoi(argv[2]);
  if (d < 0) {
    printf("I expect d to be >= 0.\n");
    return 1;
  }
}
if (argc ≥ 4) {
  max_points = atoi(argv[3]);
  if (max_points < 10) {
    printf("I expect max_points to be >= 0.\n");
    return 1;
  }
}
points = malloc(max_points * sizeof(double *));
for (int i = 0; i < max_points; ++i) points[i] = malloc((d + 1) * sizeof(double));
printf("Program started with p = %f, d = %d, and max_points = %d\n", p, d, max_points);
```

This code is used in section 2.

25. Miscellaneous tests. Here are collected some snippets that were used during testing phases.

```
⟨ Sample mean test 25 ⟩ ≡
  double sum = 0.0;
  int num_samples = 0;
  for (int i = 0; i < num_samples; ++i) {
    double sample = exponential1 ( );
    sum += sample;
    printf ("%f\n", sample);
  }
  printf ("Sample␣mean:␣%f\n", sum/num_samples);
```

26. ⟨ Generate random point cloud 26 ⟩ ≡

```
  double *point = malloc((d + 1) * sizeof(double));
  for (int i = 0; i < max_points; ++i) {
    random_point(point);
    add_point(point);
  }
  free(point);
```

27. ⟨ Test L^p 27 ⟩ ≡

```
  int rounds = max_points * max_points;
  printf ("L^p-distance␣voting␣for␣%d␣rounds␣produced␣a␣committee␣with␣%d␣members.\n",
      rounds, consensus(rounds, p_dist));
```

28. ⟨ Test orthogonal 28 ⟩ ≡

```
  int rounds = max_points;
  printf ("Orthogonal␣voting␣for␣%d␣rounds␣produced␣a␣committee␣with␣%d␣members.\n", rounds,
      consensus(rounds, abs_inner_product));
```

29. ⟨ Test close inner product 29 ⟩ ≡

```
  int rounds = max_points;
  printf ("Close␣inner␣product␣voting␣for␣%d␣rounds␣produced␣a␣committee␣with␣%d␣members.\n",
      rounds, consensus(rounds, one_minus_inner_product));
```

30. Index.

a: 7.
abs_inner_product: 19, 28.
accept: 7.
add_point: 3, 17, 26.
argc: 2, 24.
argv: 2, 24.
atof: 24.
atoi: 24.
b: 7, 14.
blue: 12, 21.
c: 7.
candidate: 22, 23.
cand1: 17.
cand2: 17.
clear_points: 3, 17.
colour1: 14.
colour2: 14.
committee: 22, 23.
committee_size: 22, 23.
consensus: 17, 27, 28, 29.
cos: 11.
d: 2.
DBL_MIN: 16.
diff: 16.
dist: 17.
dist1: 17.
dist2: 17.
draw_locus: 22, 23.
draw_square: 21, 22.
exp: 7.
exponential1: 6, 10, 25.
fabs: 10, 16, 19.
fclose: 12, 23.
file: 12, 13, 14, 21, 22, 23.
filename: 12, 23.
fopen: 12, 23.
fprintf: 12, 13, 21, 23.
free: 8, 17, 23, 26.
g: 14.
gamma_dist: 7, 10.
green: 12, 21.
i: 3, 5, 8, 9, 10, 14, 16, 17, 19, 22, 23, 24, 25, 26.
inner_product: 19.
j: 3.
k: 6.
lambda: 7.
list_points: 3.
log: 7, 11.
M_PI: 11.
main: 2, 5, 24.
malloc: 8, 17, 23, 24, 26.
max: 16.
max_points: 2, 3, 24, 26, 27, 28, 29.
max_t: 17.
normal01: 11.
num_pages: 23.
num_points: 2, 3, 14, 17, 24.
num_samples: 25.
one_minus_inner_product: 19, 29.
p: 2.
p_dist: 16, 22, 27.
p_inv: 2, 10, 24.
page: 23.
plot_single_point: 12, 14, 22.
point: 3, 8, 9, 10, 26.
points: 2, 3, 14, 17, 24.
pow: 10, 16.
pow_sum: 10.
preamble: 13.
printf: 3, 12, 19, 23, 24, 25, 27, 28, 29.
R: 7.
r: 14, 21.
radius: 12, 13, 14, 22, 23.
rand: 5, 6.
RAND_MAX: 6.
random_locus: 2, 23.
random_point: 8, 17, 23, 26.
red: 12, 21.
rounds: 27, 28, 29.
S: 7.
sample: 25.
scale: 11.
scale_factor: 10.
sqrt: 7, 11.
srand: 5.
stop: 6.
sum: 16, 19, 25.
t: 5, 14, 17.
time: 5.
to_postscript: 12.
U: 6, 7.
uniform01: 6, 7, 9, 10, 11.
U1: 11.
U2: 11.
V: 7, 10.
voted1: 17.
voted2: 17.
W: 6.
X: 7, 8.
x: 12, 16, 19, 21.
Y: 6, 7, 10.
y: 12, 16, 19, 21.

Z: 6, 7.

⟨ Committee-generating algorithm 15 ⟩ Used in section 2.
⟨ Consensus algorithm 17 ⟩ Used in section 15.
⟨ Distance computation 16 ⟩ Used in section 15.
⟨ Draw loci when $p = \infty$ 20 ⟩ Used in section 2.
⟨ Draw squares in PostScript 21 ⟩ Used in section 20.
⟨ Gamma random variate 7 ⟩ Used in section 4.
⟨ Gather input and initialise global variables 24 ⟩ Used in section 2.
⟨ Generate a drawing with random points 23 ⟩ Used in section 20.
⟨ Generate random point cloud 26 ⟩
⟨ Inner product computation 19 ⟩ Used in section 2.
⟨ Locus computation 22 ⟩ Used in section 20.
⟨ Normal random variate 11 ⟩ Used in section 4.
⟨ Plot the points array 14 ⟩ Used in section 12.
⟨ Point set functions 3 ⟩ Used in section 2.
⟨ Preamble 13 ⟩ Used in sections 12 and 23.
⟨ Print points to PostScript file 12 ⟩ Used in section 2.
⟨ Random variate generation 4 ⟩ Used in section 2.
⟨ Sample mean test 25 ⟩
⟨ Seed the pseudorandom number generator 5 ⟩ Used in section 2.
⟨ Test L^p 27 ⟩
⟨ Test close inner product 29 ⟩
⟨ Test orthogonal 28 ⟩
⟨ The case $p = \infty$ 9 ⟩ Used in section 8.
⟨ The case $p \neq \infty$ 10 ⟩ Used in section 8.
⟨ Uniform and exponential variates 6 ⟩ Used in section 4.
⟨ Uniform random point in unit ball 8 ⟩ Used in section 4.

LP-BALLS

	Section	Page
Introduction	1	1
The point set	3	2
Random variate generation	4	3
File output	12	8
Growing a committee by consensus voting	15	9
Illustrations and examples	18	11
Orthogonal committees	19	14
Locus of points further to committee than candidate	20	15
Handling user input	24	17
Miscellaneous tests	25	18
Index	30	19