Important: Before reading GB_PLANE, please read or at least skim the program for GB_MILES.

1. Introduction. This GraphBase module contains the *plane* subroutine, which constructs undirected planar graphs from vertices located randomly in a rectangle, as well as the *plane_miles* routine, which constructs planar graphs based on the mileage and coordinate data in `miles.dat`. Both routines use a general-purpose *delaunay* subroutine, which computes the Delaunay triangulation of a given set of points.

#define *plane_miles* *p_miles* /* abbreviation for Procrustean external linkage */

⟨ `gb_plane.h` 1 ⟩ ≡
#define *plane_miles* *p_miles*
  extern Graph **plane*( );
  extern Graph **plane_miles*( );
  extern void *delaunay*( );

See also sections 2 and 7.

2. The subroutine call *plane*(*n*, *x_range*, *y_range*, *extend*, *prob*, *seed*) constructs a planar graph whose vertices have integer coordinates uniformly distributed in the rectangle

$$\{\,(x,y)\ \mid\ 0 \le x < x\_range,\ 0 \le y < y\_range\,\}\,.$$

The values of *x_range* and *y_range* must be at most $2^{14} = 16384$; the latter value is the default, which is substituted if *x_range* or *y_range* is given as zero. If $extend \equiv 0$, the graph will have n vertices; otherwise it will have $n+1$ vertices, where the $(n+1)$st is assigned the coordinates $(-1,-1)$ and may be regarded as a point at ∞. Some of the n finite vertices might have identical coordinates, particularly if the point density $n/(x\_range * y\_range)$ is not very small.

The subroutine works by first constructing the Delaunay triangulation of the points, then discarding each edge of the resulting graph with probability *prob*/65536. Thus, for example, if *prob* is zero the full Delaunay triangulation will be returned; if $prob \equiv 32768$, about half of the Delaunay edges will remain. Each finite edge is assigned a length equal to the Euclidean distance between points, multiplied by 2^{10} and rounded to the nearest integer. If $extend \ne 0$, the Delaunay triangulation will also contain edges between ∞ and all points of the convex hull; such edges, if not discarded, are assigned length 2^{28}, otherwise known as `INFTY`.

If $extend \ne 0$ and $prob \equiv 0$, the graph will have $n+1$ vertices and $3(n-1)$ edges; this is the maximum number of edges that a planar graph on $n+1$ vertices can have. In such a case the average degree of a vertex will be $6(n-1)/(n+1)$, slightly less than 6; hence, if $prob \equiv 32768$, the average degree of a vertex will usually be near 3.

As with all other GraphBase routines that rely on random numbers, different values of *seed* will produce different graphs, in a machine-independent fashion that is reproducible on many different computers. Any *seed* value between 0 and $2^{31}-1$ is permissible.

#define `INFTY` #10000000$_L$ /* "infinite" length */

⟨ `gb_plane.h` 1 ⟩ +≡
#define `INFTY` #10000000$_L$

3. If the *plane* routine encounters a problem, it returns Λ (`NULL`), after putting a code number into the external variable *panic_code*. This code number identifies the type of failure. Otherwise *plane* returns a pointer to the newly created graph, which will be represented with the data structures explained in GB_GRAPH. (The external variable *panic_code* is itself defined in GB_GRAPH.)

#define *panic*(*c*) { *panic_code* = *c*; *gb_trouble_code* = 0; **return** Λ; }

4. Here is the overall shape of the C file `gb_plane.c`:

```
#include "gb_flip.h"  /* we will use the GB_FLIP routines for random numbers */
#include "gb_graph.h"  /* we will use the GB_GRAPH data structures */
#include "gb_miles.h"  /* and we might use GB_MILES for mileage data */
#include "gb_io.h"  /* and GB_MILES uses GB_IO, which has str_buf */
  ⟨ Preprocessor definitions ⟩
  ⟨ Type declarations 25 ⟩
  ⟨ Global variables 10 ⟩
  ⟨ Subroutines for arithmetic 13 ⟩
  ⟨ Other subroutines 12 ⟩
  ⟨ The delaunay routine 9 ⟩
  ⟨ The plane routine 5 ⟩
  ⟨ The plane_miles routine 41 ⟩
```

5. ⟨ The *plane* routine 5 ⟩ ≡

```
  Graph *plane(n, x_range, y_range, extend, prob, seed)
      unsigned long n;  /* number of vertices desired */
      unsigned long x_range, y_range;  /* upper bounds on rectangular coordinates */
      unsigned long extend;  /* should a point at infinity be included? */
      unsigned long prob;  /* probability of rejecting a Delaunay edge */
      long seed;  /* random number seed */
  { Graph *new_graph;  /* the graph constructed by plane */
    register Vertex *v;  /* the current vertex of interest */
    register long k;  /* the canonical all-purpose index */
    gb_init_rand(seed);
    if (x_range > 16384 ∨ y_range > 16384) panic(bad_specs);  /* range too large */
    if (n < 2) panic(very_bad_specs);  /* don't make n so small, you fool */
    if (x_range ≡ 0) x_range = 16384;  /* default */
    if (y_range ≡ 0) y_range = 16384;  /* default */
    ⟨ Set up a graph with n uniformly distributed vertices 6 ⟩;
    ⟨ Compute the Delaunay triangulation and run through the Delaunay edges; reject them with
        probability prob/65536, otherwise append them with their Euclidean length 11 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);  /* oops, we ran out of memory somewhere back there */
    }
    if (extend) new_graph→n++;  /* make the "infinite" vertex legitimate */
    return new_graph;
  }
```

This code is used in section 4.

6. The coordinates are placed into utility fields *x_coord* and *y_coord*. A random ID number is also stored in utility field *z_coord*; this number is used by the *delaunay* subroutine to break ties when points are equal or collinear or cocircular. No two vertices have the same ID number. (The header file `gb_miles.h` defines *x_coord*, *y_coord*, and *index_no* to be *x.I*, *y.I*, and *z.I* respectively.)

#define *z_coord* *z.I*

⟨ Set up a graph with n uniformly distributed vertices 6 ⟩ ≡

```
if (extend) extra_n++;   /* allocate one more vertex than usual */
new_graph = gb_new_graph(n);
if (new_graph ≡ Λ) panic(no_room);   /* out of memory before we're even started */
sprintf(new_graph→id, "plane(%lu,%lu,%lu,%lu,%lu,%ld)", n, x_range, y_range, extend, prob, seed);
strcpy(new_graph→util_types, "ZZZIIIZZZZZZZZ");
for (k = 0, v = new_graph→vertices; k < n; k++, v++) {
  v→x_coord = gb_unif_rand(x_range);
  v→y_coord = gb_unif_rand(y_range);
  v→z_coord = ((long) (gb_next_rand()/n)) * n + k;
  sprintf(str_buf, "%ld", k); v→name = gb_save_string(str_buf);
}
if (extend) {
  v→name = gb_save_string("INF");
  v→x_coord = v→y_coord = v→z_coord = −1;
  extra_n −−;
}
```

This code is used in section 5.

7. ⟨ `gb_plane.h` 1 ⟩ +≡

#define *x_coord* *x.I*
#define *y_coord* *y.I*
#define *z_coord* *z.I*

8. Delaunay triangulation. The Delaunay triangulation of a set of vertices in the plane consists of all line segments uv such that there exists a circle passing through u and v containing no other vertices. Equivalently, uv is a Delaunay edge if and only if the Voronoi regions for u and v are adjacent; the Voronoi region of a vertex u is the polygon with the property that all points inside it are closer to u than to any other vertex. In this sense, we can say that Delaunay edges connect vertices with their "neighbors."

The definitions in the previous paragraph assume that no two vertices are equal, that no three vertices lie on a straight line, and that no four vertices lie on a circle. If those nondegeneracy conditions aren't satisfied, we can perturb the points very slightly so that the assumptions do hold.

Another way to characterize the Delaunay triangulation is to consider what happens when we map a given set of points onto the unit sphere via stereographic projection: Point (x, y) is mapped to

$$\bigl(2x/(r^2+1), 2y/(r^2+1), (r^2-1)/(r^2+1)\bigr)\,,$$

where $r^2 = x^2 + y^2$. If we now extend the configuration by adding $(0, 0, 1)$, which is the limiting point on the sphere when r approaches infinity, the Delaunay edges of the original points turn out to be edges of the polytope defined by the mapped points. This polytope, which is the 3-dimensional convex hull of $n + 1$ points on the sphere, also has edges from $(0, 0, 1)$ to the mapped points that correspond to the 2-dimensional convex hull of the original points. Under our assumption of nondegeneracy, the faces of this polytope are all triangles; hence its edges are said to form a triangulation.

A self-contained presentation of all the relevant theory, together with an exposition and proof of correctness of the algorithm below, can be found in the author's monograph *Axioms and Hulls*, Lecture Notes in Computer Science **606** (Springer-Verlag, 1992). For further references, see Franz Aurenhammer, *ACM Computing Surveys* **23** (1991), 345–405.

9. The *delaunay* procedure, which finds the Delaunay triangulation of a given set of vertices, is the key ingredient in *gb_plane*'s algorithms for generating planar graphs. The given vertices should appear in a GraphBase graph g whose edges, if any, are ignored by *delaunay*. The coordinates of each vertex appear in utility fields *x_coord* and *y_coord*, which must be nonnegative and less than $2^{14} = 16384$. The utility field *z_coord* must contain a unique ID number, distinct for every vertex, so that the algorithm can break ties in cases of degeneracy. (Note: These assumptions about the input data are the responsibility of the calling procedure; *delaunay* does not double-check them. If they are violated, catastrophic failure is possible.)

Instead of returning the Delaunay triangulation as a graph, *delaunay* communicates its answer implicitly by performing the procedure call $f(u,v)$ on every pair of vertices u and v joined by a Delaunay edge. Here f is a procedure supplied as a parameter; u and v are either pointers to vertices or Λ (i.e., `NULL`), where Λ denotes the vertex "∞." As remarked above, edges run between ∞ and all vertices on the convex hull of the given points. The graph of all edges, including the infinite edges, is planar.

For example, if the vertex at infinity is being ignored, the user can declare

```
void ins_finite(u, v)
    Vertex *u, *v;
{ if (u ∧ v) gb_new_edge(u, v, 1L); }
```

Then the procedure call *delaunay*(g, *ins_finite*) will add all the finite Delaunay edges to the current graph g, giving them all length 1.

If *delaunay* is unable to allocate enough storage to do its work, it will set *gb_trouble_code* nonzero and there will be no edges in the triangulation.

```
⟨ The delaunay routine 9 ⟩ ≡
  void delaunay(g, f)
      Graph *g;     /* vertices in the plane */
      void (*f)();     /* procedure that absorbs the triangulated edges */
  { ⟨ Local variables for delaunay 26 ⟩;
    ⟨ Find the Delaunay triangulation of g, or return with gb_trouble_code nonzero if out of memory 34 ⟩;
    ⟨ Call f(u, v) for each Delaunay edge uv 28 ⟩;
    gb_free(working_storage);
  }
```

This code is used in section 4.

10. The procedure passed to *delaunay* will communicate with *plane* via global variables called *gprob* and *inf_vertex*.

```
⟨ Global variables 10 ⟩ ≡
  static unsigned long gprob;     /* copy of the prob parameter */
  static Vertex *inf_vertex;     /* pointer to the vertex ∞, or Λ */
```

This code is used in section 4.

11. ⟨ Compute the Delaunay triangulation and run through the Delaunay edges; reject them with probability *prob*/65536, otherwise append them with their Euclidean length 11 ⟩ ≡

```
  gprob = prob;
  if (extend) inf_vertex = new_graph→vertices + n;
  else inf_vertex = Λ;
  delaunay(new_graph, new_euclid_edge);
```

This code is used in section 5.

12. ⟨ Other subroutines 12 ⟩ ≡

```
void new_euclid_edge(u, v)
    Vertex *u, *v;
{ register long dx, dy;
  if ((gb_next_rand() >> 15) >= gprob) {
    if (u) {
      if (v) {
        dx = u->x_coord - v->x_coord;
        dy = u->y_coord - v->y_coord;
        gb_new_edge(u, v, int_sqrt(dx * dx + dy * dy));
      } else if (inf_vertex) gb_new_edge(u, inf_vertex, INFTY);
    } else if (inf_vertex) gb_new_edge(inf_vertex, v, INFTY);
  }
}
```

See also sections 20, 21, 40, and 44.

This code is used in section 4.

13. Arithmetic. Before we lunge into the world of geometric algorithms, let's build up some confidence by polishing off some subroutines that will be needed to ensure correct results. We assume that **long** integers are less than 2^{31}.

First is a routine to calculate $s = \lfloor 2^{10}\sqrt{x} + \frac{1}{2} \rfloor$, the nearest integer to 2^{10} times the square root of a given nonnegative integer x. If $x > 0$, this is the unique integer such that $2^{20}x - s \le s^2 < 2^{20}x + s$.

The following routine appears to work by magic, but the mystery goes away when one considers the invariant relations

$$m = \lfloor 2^{2k-21} \rfloor, \qquad 0 < y = \lfloor 2^{20-2k}x \rfloor - s^2 + s \le q = 2s.$$

(Exception: We might actually have $y = 0$ for a short time when $q = 2$.)

⟨ Subroutines for arithmetic 13 ⟩ ≡

```
long int_sqrt(x)
    long x;
{ register long y, m, q = 2;
  long k;
  if (x <= 0) return 0;
  for (k = 25, m = #20000000; x < m; k--, m >>= 2) ;   /* find the range */
  if (x >= m + m) y = 1;
  else y = 0;
  do ⟨ Decrease k by 1, maintaining the invariant relations between x, y, m, and q 14 ⟩ while (k);
  return q >> 1;
}
```

See also sections 15 and 24.

This code is used in section 4.

14. ⟨ Decrease k by 1, maintaining the invariant relations between x, y, m, and q 14 ⟩ ≡

```
{
  if (x & m) y += y + 1;
  else y += y;
  m >>= 1;
  if (x & m) y += y - q + 1;
  else y += y - q;
  q += q;
  if (y > q) y -= q, q += 2;
  else if (y <= 0) q -= 2, y += q;
  m >>= 1;
  k--;
}
```

This code is used in section 13.

15. We are going to need multiple-precision arithmetic in order to calculate certain geometric predicates properly, but it turns out that we do not need to implement general-purpose subroutines for bignums. It suffices to have a single special-purpose routine called $sign\_test(x1, x2, x3, y1, y2, y3)$, which computes a single-precision integer having the same sign as the dot product

$$x1 * y1 + x2 * y2 + x3 * y3$$

when we have $-2^{29} < x1, x2, x3 < 2^{29}$ and $0 \le y1, y2, y3 < 2^{29}$.

```
⟨ Subroutines for arithmetic 13 ⟩ +≡
  long sign_test(x1, x2, x3, y1, y2, y3)
      long x1, x2, x3, y1, y2, y3;
  { long s1, s2, s3;    /* signs of individual terms */
    long a, b, c;    /* components of a redundant representation of the dot product */
    register long t;    /* temporary register for swapping */
    ⟨ Determine the signs of the terms 16 ⟩;
    ⟨ If the answer is obvious, return it without further ado; otherwise, arrange things so that x3 * y3 has
        the opposite sign to x1 * y1 + x2 * y2 17 ⟩;
    ⟨ Compute a redundant representation of x1 * y1 + x2 * y2 + x3 * y3 18 ⟩;
    ⟨ Return the sign of the redundant representation 19 ⟩;
  }
```

16. ⟨ Determine the signs of the terms 16 ⟩ ≡

```
  if (x1 ≡ 0 ∨ y1 ≡ 0) s1 = 0;
  else {
    if (x1 > 0) s1 = 1;
    else x1 = −x1, s1 = −1;
  }
  if (x2 ≡ 0 ∨ y2 ≡ 0) s2 = 0;
  else {
    if (x2 > 0) s2 = 1;
    else x2 = −x2, s2 = −1;
  }
  if (x3 ≡ 0 ∨ y3 ≡ 0) s3 = 0;
  else {
    if (x3 > 0) s3 = 1;
    else x3 = −x3, s3 = −1;
  }
```

This code is used in section 15.

17. The answer is obvious unless one of the terms is positive and one of the terms is negative.

⟨ If the answer is obvious, return it without further ado; otherwise, arrange things so that $x3 * y3$ has the opposite sign to $x1 * y1 + x2 * y2$ 17 ⟩ ≡

```
if ((s1 ≥ 0 ∧ s2 ≥ 0 ∧ s3 ≥ 0) ∨ (s1 ≤ 0 ∧ s2 ≤ 0 ∧ s3 ≤ 0)) return (s1 + s2 + s3);
if (s3 ≡ 0 ∨ s3 ≡ s1) {
  t = s3; s3 = s2; s2 = t;
  t = x3; x3 = x2; x2 = t;
  t = y3; y3 = y2; y2 = t;
} else if (s3 ≡ s2) {
  t = s3; s3 = s1; s1 = t;
  t = x3; x3 = x1; x1 = t;
  t = y3; y3 = y1; y1 = t;
}
```

This code is used in section 15.

18. We make use of a redundant representation $2^{28}a + 2^{14}b + c$, which can be computed by brute force. (Everything is understood to be multiplied by $-s3$.)

⟨ Compute a redundant representation of $x1 * y1 + x2 * y2 + x3 * y3$ 18 ⟩ ≡

```
{ register long lx, rx, ly, ry;
  lx = x1/#4000; rx = x1 % #4000;     /* split off the least significant 14 bits */
  ly = y1/#4000; ry = y1 % #4000;
  a = lx * ly; b = lx * ry + ly * rx; c = rx * ry;
  lx = x2/#4000; rx = x2 % #4000;
  ly = y2/#4000; ry = y2 % #4000;
  a += lx * ly; b += lx * ry + ly * rx; c += rx * ry;
  lx = x3/#4000; rx = x3 % #4000;
  ly = y3/#4000; ry = y3 % #4000;
  a −= lx * ly; b −= lx * ry + ly * rx; c −= rx * ry;
}
```

This code is used in section 15.

19. Here we use the fact that $|c| < 2^{29}$.

⟨ Return the sign of the redundant representation 19 ⟩ ≡

```
  if (a ≡ 0) goto ez;
  if (a < 0) a = −a, b = −b, c = −c, s3 = −s3;
  while (c < 0) {
    a−−; c += #10000000;
    if (a ≡ 0) goto ez;
  }
  if (b ≥ 0) return −s3;     /* the answer is clear when a > 0 ∧ b ≥ 0 ∧ c ≥ 0 */
  b = −b;
  a −= b/#4000;
  if (a > 0) return −s3;
  if (a ≤ −2) return s3;
  return −s3 * ((a * #4000 − b % #4000) * #4000 + c);
ez: if (b ≥ #8000) return −s3;
  if (b ≤ −#8000) return s3;
  return −s3 * (b * #4000 + c);
```

This code is used in section 15.

20. Determinants. The *delaunay* routine bases all of its decisions on two geometric predicates, which depend on whether certain determinants are positive or negative.

The first predicate, $ccw(u, v, w)$, is true if and only if the three points (u, v, w) have a counterclockwise orientation. This means that if we draw the unique circle through those points, and if we travel along that circle in the counterclockwise direction starting at u, we will encounter v before w.

It turns out that $ccw(u, v, w)$ holds if and only if the determinant

$$\begin{vmatrix} x_u & y_u & 1 \\ x_v & y_v & 1 \\ x_w & y_w & 1 \end{vmatrix} = \begin{vmatrix} x_u - x_w & y_u - y_w \\ x_v - x_w & y_v - y_w \end{vmatrix}$$

is positive. The evaluation must be exact; if the answer is zero, a special tie-breaking rule must be used because the three points were collinear. The tie-breaking rule is tricky (and necessarily so, according to the theory in *Axioms and Hulls*).

Integer evaluation of that determinant will not cause **long** integer overflow, because we have assumed that all x and y coordinates lie between 0 and $2^{14} - 1$, inclusive. In fact, we could go up to $2^{15} - 1$ without risking overflow; but the limitation to 14 bits will be helpful when we consider a more complicated determinant below.

```
⟨ Other subroutines 12 ⟩ +≡
  long ccw(u, v, w)
      Vertex *u, *v, *w;
  { register long wx = w→x_coord, wy = w→y_coord;     /* x_w, y_w */
    register long det = (u→x_coord − wx) * (v→y_coord − wy) − (u→y_coord − wy) * (v→x_coord − wx);
    Vertex *t;
    if (det ≡ 0) {
      det = 1;
      if (u→z_coord > v→z_coord) {
        t = u; u = v; v = t; det = −det;
      }
      if (v→z_coord > w→z_coord) {
        t = v; v = w; w = t; det = −det;
      }
      if (u→z_coord > v→z_coord) {
        t = u; u = v; v = t; det = −det;
      }
      if (u→x_coord > v→x_coord ∨ (u→x_coord ≡ v→x_coord ∧
            (u→y_coord > v→y_coord ∨ (u→y_coord ≡ v→y_coord ∧
            (w→x_coord > u→x_coord ∨ (w→x_coord ≡ u→x_coord ∧ w→y_coord ≥ u→y_coord)))))) det = −det;
    }
    return (det > 0);
  }
```

21. The other geometric predicate, *incircle*(t,u,v,w), is true if and only if point t lies outside the circle passing through u, v, and w, assuming that *ccw*(u,v,w) holds. This predicate makes us work harder, because it is equivalent to the sign of a 4×4 determinant that requires twice as much precision:

$$\begin{vmatrix} x_t & y_t & x_t^2+y_t^2 & 1\\ x_u & y_u & x_u^2+y_u^2 & 1\\ x_v & y_v & x_v^2+y_v^2 & 1\\ x_w & y_w & x_w^2+y_w^2 & 1\end{vmatrix} = \begin{vmatrix} x_t-x_w & y_t-y_w & (x_t-x_w)^2+(y_t-y_w)^2\\ x_u-x_w & y_u-y_w & (x_u-x_w)^2+(y_u-y_w)^2\\ x_v-x_w & y_v-y_w & (x_v-x_w)^2+(y_v-y_w)^2\end{vmatrix}.$$

The sign can, however, be deduced by the *sign_test* subroutine we had the foresight to provide earlier.

```
⟨Other subroutines 12⟩ +≡
long incircle(t, u, v, w)
    Vertex *t, *u, *v, *w;
{ register long wx = w->x_coord, wy = w->y_coord;     /* x_w, y_w */
  long tx = t->x_coord - wx, ty = t->y_coord - wy;     /* x_t - x_w, y_t - y_w */
  long ux = u->x_coord - wx, uy = u->y_coord - wy;     /* x_u - x_w, y_u - y_w */
  long vx = v->x_coord - wx, vy = v->y_coord - wy;     /* x_v - x_w, y_v - y_w */
  register long det = sign_test(tx * uy - ty * ux, ux * vy - uy * vx, vx * ty - vy * tx,
      vx * vx + vy * vy, tx * tx + ty * ty, ux * ux + uy * uy);
  Vertex *s;
  if (det ≡ 0) {
    ⟨Sort (t, u, v, w) by ID number 22⟩;
    ⟨Remove incircle degeneracy 23⟩;
  }
  return (det > 0);
}
```

22.
```
⟨Sort (t, u, v, w) by ID number 22⟩ ≡
  det = 1;
  if (t->z_coord > u->z_coord) {
    s = t; t = u; u = s; det = -det;
  }
  if (v->z_coord > w->z_coord) {
    s = v; v = w; w = s; det = -det;
  }
  if (t->z_coord > v->z_coord) {
    s = t; t = v; v = s; det = -det;
  }
  if (u->z_coord > w->z_coord) {
    s = u; u = w; w = s; det = -det;
  }
  if (u->z_coord > v->z_coord) {
    s = u; u = v; v = s; det = -det;
  }
```
This code is used in section 21.

23. By slightly perturbing the points, we can always make them nondegenerate, although the details are complicated. A sequence of 12 steps, involving up to four auxiliary functions

$$ff(t,u,v,w) = \begin{vmatrix} x_t - x_v & (x_t - x_w)^2 + (y_t - y_w)^2 - (x_v - x_w)^2 - (y_v - y_w)^2 \\ x_u - x_v & (x_u - x_w)^2 + (y_u - y_w)^2 - (x_v - x_w)^2 - (y_v - y_w)^2 \end{vmatrix},$$

$$gg(t,u,v,w) = \begin{vmatrix} y_t - y_v & (x_t - x_w)^2 + (y_t - y_w)^2 - (x_v - x_w)^2 - (y_v - y_w)^2 \\ y_u - y_v & (x_u - x_w)^2 + (y_u - y_w)^2 - (x_v - x_w)^2 - (y_v - y_w)^2 \end{vmatrix},$$

$$hh(t,u,v,w) = (x_u - x_t)(y_v - y_w),$$

$$jj(t,u,v,w) = (x_u - x_v)^2 + (y_u - y_w)^2 - (x_t - x_v)^2 - (y_t - y_w)^2,$$

does the trick, as explained in *Axioms and Hulls.*

⟨ Remove incircle degeneracy 23 ⟩ ≡
{ **long** dd;
  if $((dd = ff(t,u,v,w)) < 0 \lor (dd \equiv 0 \land$
    $((dd = gg(t,u,v,w)) < 0 \lor (dd \equiv 0 \land$
    $((dd = ff(u,t,w,v)) < 0 \lor (dd \equiv 0 \land$
    $((dd = gg(u,t,w,v)) < 0 \lor (dd \equiv 0 \land$
    $((dd = ff(v,w,t,u)) < 0 \lor (dd \equiv 0 \land$
    $((dd = gg(v,w,t,u)) < 0 \lor (dd \equiv 0 \land$
    $((dd = hh(t,u,v,w)) < 0 \lor (dd \equiv 0 \land$
    $((dd = jj(t,u,v,w)) < 0 \lor (dd \equiv 0 \land$
    $((dd = hh(v,t,u,w)) < 0 \lor (dd \equiv 0 \land$
    $((dd = jj(v,t,u,w)) < 0 \lor (dd \equiv 0 \land jj(t,w,u,v) < 0))))))))))))))))))))$ $det = -det$;
}

This code is used in section 21.

24. ⟨ Subroutines for arithmetic 13 ⟩ +≡

```
long ff(t, u, v, w)
    Vertex *t, *u, *v, *w;
{ register long wx = w->x_coord, wy = w->y_coord;    /* x_w, y_w */
  long tx = t->x_coord - wx, ty = t->y_coord - wy;    /* x_t - x_w, y_t - y_w */
  long ux = u->x_coord - wx, uy = u->y_coord - wy;    /* x_u - x_w, y_u - y_w */
  long vx = v->x_coord - wx, vy = v->y_coord - wy;    /* x_v - x_w, y_v - y_w */
  return sign_test(ux - tx, vx - ux, tx - vx, vx * vx + vy * vy, tx * tx + ty * ty, ux * ux + uy * uy);
}
long gg(t, u, v, w)
    Vertex *t, *u, *v, *w;
{ register long wx = w->x_coord, wy = w->y_coord;    /* x_w, y_w */
  long tx = t->x_coord - wx, ty = t->y_coord - wy;    /* x_t - x_w, y_t - y_w */
  long ux = u->x_coord - wx, uy = u->y_coord - wy;    /* x_u - x_w, y_u - y_w */
  long vx = v->x_coord - wx, vy = v->y_coord - wy;    /* x_v - x_w, y_v - y_w */
  return sign_test(uy - ty, vy - uy, ty - vy, vx * vx + vy * vy, tx * tx + ty * ty, ux * ux + uy * uy);
}
long hh(t, u, v, w)
    Vertex *t, *u, *v, *w;
{
  return (u->x_coord - t->x_coord) * (v->y_coord - w->y_coord);
}
long jj(t, u, v, w)
    Vertex *t, *u, *v, *w;
{ register long vx = v->x_coord, wy = w->y_coord;
  return (u->x_coord - vx) * (u->x_coord - vx) + (u->y_coord - wy) * (u->y_coord - wy)
      - (t->x_coord - vx) * (t->x_coord - vx) - (t->y_coord - wy) * (t->y_coord - wy);
}
```

25. Delaunay data structures. Now we have the primitive predicates we need, and we can get on with the geometric aspects of *delaunay*. As mentioned above, each vertex is represented by two coordinates and an ID number, stored in the utility fields *x_coord*, *y_coord*, and *z_coord*.

Each edge of the current triangulation is represented by two arcs pointing in opposite directions; the two arcs are called *mates*. Each arc conceptually has a triangle on its left and a mate on its right.

An **arc** record differs from an **Arc**; it has three fields:

vert is the vertex this arc leads to, or Λ if that vertex is ∞;

next is the next arc having the same triangle at the left;

inst is the branch node that points to the triangle at the left, as explained below.

If p points to an arc, then $p\to next\to next\to next \equiv p$, because a triangle is bounded by three arcs. We also have $p\to next\to inst \equiv p\to inst$, for all arcs p.

⟨ Type declarations 25 ⟩ ≡

```
typedef struct a_struct {
  Vertex *vert;   /* v, if this arc goes from u to v */
  struct a_struct *next;   /* the arc from v that shares a triangle with this one */
  struct n_struct *inst;   /* instruction to change when the triangle is modified */
} arc;
```

See also section 29.

This code is used in section 4.

26. Storage is allocated in such a way that, if p and q point respectively to an arc and its mate, then $p + q = \&arc\_block[0] + \&arc\_block[m - 1]$, where m is the total number of arc records allocated in the *arc_block* array. This convention saves us one pointer field in each arc.

When setting q to the mate of p, we need to do the calculation cautiously using an auxiliary register, because the constant $\&arc\_block[0] + \&arc\_block[m - 1]$ might be too large to evaluate without integer overflow on some systems.

```
#define mate(a, b)
        {   /* given a, set b to its mate */
          reg = max_arc − (siz_t) a;
          b = (arc *) (reg + min_arc);
        }
```

⟨ Local variables for *delaunay* 26 ⟩ ≡

```
register siz_t reg;   /* used while computing mates */
siz_t min_arc, max_arc;   /* &arc_block[0], &arc_block[m − 1] */
arc *next_arc;   /* the first arc record that hasn't yet been used */
```

See also sections 30 and 32.

This code is used in section 9.

27. ⟨ Initialize the array of arcs 27 ⟩ ≡

```
next_arc = gb_typed_alloc(6 * g→n − 6, arc, working_storage);
if (next_arc ≡ Λ) return;   /* gb_trouble_code is nonzero */
min_arc = (siz_t) next_arc;
max_arc = (siz_t) (next_arc + (6 * g→n − 7));
```

This code is used in section 31.

28. ⟨ Call $f(u, v)$ for each Delaunay edge uv 28 ⟩ ≡

```
a = (arc *) min_arc;
b = (arc *) max_arc;
for ( ; a < next_arc; a++, b−−) (*f)(a→vert, b→vert);
```

This code is used in section 9.

29. The last and probably most crucial component of the data structure is the collection of *branch nodes*, which will be linked together into a binary tree. Given a new vertex w, we will ascertain what triangle it belongs to by starting at the root of this tree and executing a sequence of instructions, each of which has the form 'if w lies to the right of the straight line from u to v then go to α else go to β', where α and β are nodes that continue the search. This process continues until we reach a terminal node, which says 'congratulations, you're done, w is in triangle such-and-such'. The terminal node points to one of the three arcs bounding that triangle. If a vertex of the triangle is ∞, the terminal node points to the arc whose *vert* pointer is Λ.

⟨ Type declarations 25 ⟩ +≡
```
typedef struct n_struct {
  Vertex *u;   /* first vertex, or Λ if this is a terminal node */
  Vertex *v;   /* second vertex, or pointer to the triangle corresponding to a terminal node */
  struct n_struct *l;   /* go here if w lies to the left of uv */
  struct n_struct *r;   /* go here if w lies to the right of uv */
} node;
```

30. The search tree just described is actually a dag (a directed acyclic graph), because it has overlapping subtrees. As the algorithm proceeds, the dag gets bigger and bigger, since the number of triangles keeps growing. Instructions are never deleted; we just extend the dag by substituting new branches for nodes that once were terminal.

The expected number of nodes in this dag is $O(n)$ when there are n vertices, if we input the vertices in random order. But it can be as high as order n^2 in the worst case. So our program will allocate blocks of nodes dynamically instead of assuming a maximum size.

```
#define nodes_per_block 127   /* on most computers we want it ≡ 15 (mod 16) */
#define new_node(x)
        if (next_node ≡ max_node) {
          x = gb_typed_alloc(nodes_per_block, node, working_storage);
          if (x ≡ Λ) {
            gb_free(working_storage);   /* release delaunay's auxiliary memory */
            return;   /* gb_trouble_code is nonzero */
          }
          next_node = x + 1;
          max_node = x + nodes_per_block;
        } else x = next_node++;
#define terminal_node(x, p)
        { new_node(x);   /* allocate a new node */
          x→v = (Vertex *) (p);   /* make it point to a given arc from the triangle */
        }   /* note that x→u ≡ Λ, representing a terminal node */
```

⟨ Local variables for *delaunay* 26 ⟩ +≡
```
  node *next_node;   /* the first yet-unused node slot in the current block of nodes */
  node *max_node;   /* address of nonexistent node following the current block of nodes */
  node root_node;   /* start here to locate a vertex in its triangle */
  Area working_storage;   /* where delaunay builds its triangulation */
```

31. The algorithm begins with a trivial triangulation that contains only the first two vertices, together with two "triangles" extending to infinity at their left and right.

⟨ Initialize the data structures 31 ⟩ ≡
  $\mathit{next\_node} = \mathit{max\_node} = \Lambda$;
  $\mathit{init\_area}(\mathit{working\_storage})$;
  ⟨ Initialize the array of arcs 27 ⟩;
  $u = g\to\mathit{vertices}$;
  $v = u + 1$;
  ⟨ Make two "triangles" for u, v, and ∞ 33 ⟩;

This code is used in section 34.

32. We'll need a bunch of local variables to do elementary operations on data structures.

⟨ Local variables for *delaunay* 26 ⟩ +≡
  Vertex $*p, *q, *r, *s, *t, *\mathit{tp}, *\mathit{tpp}, *u, *v$;
  arc $*a, *\mathit{aa}, *b, *c, *d, *e$;
  node $*x, *y, *\mathit{yp}, *\mathit{ypp}$;

33. ⟨ Make two "triangles" for u, v, and ∞ 33 ⟩ ≡
  $\mathit{root\_node}.u = u$;
  $\mathit{root\_node}.v = v$;
  $a = \mathit{next\_arc}$;
  $\mathit{terminal\_node}(x, a + 1)$;
  $\mathit{root\_node}.l = x$;
  $a\to\mathit{vert} = v$; $a\to\mathit{next} = a + 1$; $a\to\mathit{inst} = x$;
  $(a + 1)\to\mathit{next} = a + 2$; $(a + 1)\to\mathit{inst} = x$;   /* $(a + 1)\to\mathit{vert} = \Lambda$, representing ∞ */
  $(a + 2)\to\mathit{vert} = u$; $(a + 2)\to\mathit{next} = a$; $(a + 2)\to\mathit{inst} = x$;
  $\mathit{mate}(a, b)$;
  $\mathit{terminal\_node}(x, b - 2)$;
  $\mathit{root\_node}.r = x$;
  $b\to\mathit{vert} = u$; $b\to\mathit{next} = b - 2$; $b\to\mathit{inst} = x$;
  $(b - 2)\to\mathit{next} = b - 1$; $(b - 2)\to\mathit{inst} = x$;   /* $(b - 2)\to\mathit{vert} = \Lambda$, representing ∞ */
  $(b - 1)\to\mathit{vert} = v$; $(b - 1)\to\mathit{next} = b$; $(b - 1)\to\mathit{inst} = x$;
  $\mathit{next\_arc} \mathrel{+}= 3$;

This code is used in section 31.

34. Delaunay updating. The main loop of the algorithm updates the data structure incrementally by adding one new vertex at a time. The new vertex will always be connected by an edge (i.e., by two arcs) to each of the vertices of the triangle that previously enclosed it. It might also deserve to be connected to other nearby vertices.

⟨ Find the Delaunay triangulation of g, or return with *gb_trouble_code* nonzero if out of memory 34 ⟩ ≡
```
if (g→n < 2) return;    /* no edges unless there are at least 2 vertices */
⟨ Initialize the data structures 31 ⟩;
for (p = g→vertices + 2; p < g→vertices + g→n; p++) {
   ⟨ Find an arc a on the boundary of the triangle containing p 35 ⟩;
   ⟨ Divide the triangle left of a into three triangles surrounding p 36 ⟩;
   ⟨ Explore the triangles surrounding p, "flipping" their neighbors until all triangles that should touch p
        are found 39 ⟩;
}
```
This code is used in section 9.

35. We have set up the branch nodes so that they solve the triangle location problem.

⟨ Find an arc a on the boundary of the triangle containing p 35 ⟩ ≡
```
x = &root_node;
do {
   if (ccw(x→u, x→v, p)) x = x→l;
   else x = x→r;
} while (x→u);
a = (arc *) x→v;    /* terminal node points to the arc we want */
```
This code is used in section 34.

36. Subdividing a triangle is an easy exercise in data structure manipulation, except that we must do something special when one of the vertices is infinite. Let's look carefully at what needs to be done.

Suppose the triangle containing p has the vertices q, r, and s in counterclockwise order. Let x be the terminal node that points to the triangle Δqrs. We want to change x so that we will be able to locate a future point of Δqrs within either Δpqr, Δprs, or Δpsq.

If q, r, and s are finite, we will change x and add five new nodes as follows:

x: if left of rp, go to x'', else go to x';
x': if left of sp, go to y, else go to y';
x'': if left of qp, go to y', else go to y'';
y: you're in Δprs;
y': you're in Δpsq;
y'': you're in Δpqr.

But if, say, $q=\infty$, such instructions make no sense, because there are lines in all directions that run from ∞ to any point. In such a case we use "wedges" instead of triangles, as explained below.

At the beginning of the following code, we have $x \equiv a{\to}inst$.

⟨ Divide the triangle left of a into three triangles surrounding p 36 ⟩ ≡
$b = a{\to}next$; $c = b{\to}next$;
$q = a{\to}vert$; $r = b{\to}vert$; $s = c{\to}vert$;
⟨ Create new terminal nodes y, yp, ypp, and new arcs pointing to them 37 ⟩;
if $(q \equiv \Lambda)$ ⟨ Compile instructions to update convex hull 38 ⟩
else { **register node** $*xp$;
$x{\to}u = r$; $x{\to}v = p$;
$new\_node(xp)$;
$xp{\to}u = q$; $xp{\to}v = p$; $xp{\to}l = yp$; $xp{\to}r = ypp$; /* instruction x'' above */
$x{\to}l = xp$;
$new\_node(xp)$;
$xp{\to}u = s$; $xp{\to}v = p$; $xp{\to}l = y$; $xp{\to}r = yp$; /* instruction x' above */
$x{\to}r = xp$;
}

This code is used in section 34.

37. The only subtle point here is that $q = a{\to}vert$ might be Λ. A terminal node must point to the proper arc of an infinite triangle.

⟨ Create new terminal nodes y, yp, ypp, and new arcs pointing to them 37 ⟩ ≡
$terminal\_node(yp, a)$; $terminal\_node(ypp, next\_arc)$; $terminal\_node(y, c)$;
$c{\to}inst = y$; $a{\to}inst = yp$; $b{\to}inst = ypp$;
$mate(next\_arc, e)$;
$a{\to}next = e$; $b{\to}next = e-1$; $c{\to}next = e-2$;
$next\_arc{\to}vert = q$; $next\_arc{\to}next = b$; $next\_arc{\to}inst = ypp$;
$(next\_arc+1){\to}vert = r$; $(next\_arc+1){\to}next = c$; $(next\_arc+1){\to}inst = y$;
$(next\_arc+2){\to}vert = s$; $(next\_arc+2){\to}next = a$; $(next\_arc+2){\to}inst = yp$;
$e{\to}vert = (e-1){\to}vert = (e-2){\to}vert = p$;
$e{\to}next = next\_arc+2$; $(e-1){\to}next = next\_arc$; $(e-2){\to}next = next\_arc+1$;
$e{\to}inst = yp$; $(e-1){\to}inst = ypp$; $(e-2){\to}inst = y$;
$next\_arc \mathrel{+}= 3$;

This code is used in section 36.

38. Outside of the current convex hull, we have "wedges" instead of triangles. Wedges are exterior angles whose points lie outside an edge rs of the convex hull, but not outside the next edge on the other side of point r. When a new point lies in such a wedge, we have to see if it also lies outside the edges st, tu, etc., in the clockwise direction, in which case the convex hull loses points s, t, etc., and we must update the new wedges accordingly.

This was the hardest part of the program to prove correct; a complete proof can be found in *Axioms and Hulls*.

⟨ Compile instructions to update convex hull 38 ⟩ ≡

```
{ register node *xp;
  x->u = r; x->v = p; x->l = ypp;
  new_node(xp);
  xp->u = s; xp->v = p; xp->l = y; xp->r = yp;
  x->r = xp;
  mate(a, aa); d = aa->next; t = d->vert;
  while (t ≠ r ∧ (ccw(p, s, t))) { register node *xpp;
    terminal_node(xpp, d);
    xp->r = d->inst;
    xp = d->inst;
    xp->u = t; xp->v = p; xp->l = xpp; xp->r = yp;
    flip(a, aa, d, s, Λ, t, p, xpp, yp);
    a = aa->next; mate(a, aa); d = aa->next;
    s = t; t = d->vert;
    yp->v = (Vertex *) a;
  }
  terminal_node(xp, d->next);
  x = d->inst; x->u = s; x->v = p; x->l = xp; x->r = yp;
  d->inst = xp; d->next->inst = xp; d->next->next->inst = xp;
  r = s;     /* this value of r shortens the exploration step that follows */
}
```

This code is used in section 36.

39. The updating process finishes by walking around the triangles that surround p, making sure that none of them are adjacent to triangles containing p in their circumcircle. (Such triangles are no longer in the Delaunay triangulation, by definition.)

⟨ Explore the triangles surrounding p, "flipping" their neighbors until all triangles that should touch p are found 39 ⟩ ≡

```
while (1) {
  mate(c, d); e = d→next;
  t = d→vert; tp = c→vert; tpp = e→vert;
  if (tpp ∧ incircle(tpp, tp, t, p)) {    /* triangle tt″t′ no longer Delaunay */
    register node *xp, *xpp;

    terminal_node(xp, e);
    terminal_node(xpp, d);
    x = c→inst; x→u = tpp; x→v = p; x→l = xp; x→r = xpp;
    x = d→inst; x→u = tpp; x→v = p; x→l = xp; x→r = xpp;
    flip(c, d, e, t, tp, tpp, p, xp, xpp);
    c = e;
  }
  else if (tp ≡ r) break;
  else {
    mate(c→next, aa);
    c = aa→next;
  }
}
```

This code is used in section 34.

40. Here d is the mate of c, $e = d\to next$, $t = d\to vert$, $tp = c\to vert$, and $tpp = e\to vert$. The triangles $\Delta tt'p$ and $\Delta t'tt''$ to the left and right of arc c are being replaced in the current triangulation by $\Delta ptt''$ and $\Delta t''t'p$, corresponding to terminal nodes xp and xpp. (The values of t and tp are not actually used, so some optimization is possible.)

⟨ Other subroutines 12 ⟩ +≡

```
void flip(c, d, e, t, tp, tpp, p, xp, xpp)
    arc *c, *d, *e;
    Vertex *t, *tp, *tpp, *p;
    node *xp, *xpp;
{ register arc *ep = e→next, *cp = c→next, *cpp = cp→next;

  e→next = c; c→next = cpp; cpp→next = e;
  e→inst = c→inst = cpp→inst = xp;
  c→vert = p;
  d→next = ep; ep→next = cp; cp→next = d;
  d→inst = ep→inst = cp→inst = xpp;
  d→vert = tpp;
}
```

41. Use of mileage data. The *delaunay* routine is now complete, and the only missing piece of code is the promised routine that generates planar graphs based on data from the real world.

The subroutine call *plane_miles*(n, *north_weight*, *west_weight*, *pop_weight*, *extend*, *prob*, *seed*) will construct a planar graph with $\min(128, n)$ vertices, where the vertices are exactly the same as the cities produced by the subroutine call *miles*(n, *north_weight*, *west_weight*, *pop_weight*, 0, 0, *seed*). (As explained in module GB_MILES, the weight parameters *north_weight*, *west_weight*, and *pop_weight* are used to rank the cities by location and/or population.) The edges of the new graph are obtained by first constructing the Delaunay triangulation of those cities, based on a simple projection onto the plane using their latitude and longitude, then discarding each Delaunay edge with probability *prob*/65536. The length of each surviving edge is the same as the mileage between cities that would appear in the complete graph produced by *miles*.

If $extend \neq 0$, an additional vertex representing ∞ is also included. The Delaunay triangulation includes edges of length `INFTY` connecting this vertex with all cities on the convex hull; these edges, like the others, are subject to being discarded with probability *prob*/65536. (See the description of *plane* for further comments about using *prob* to control the sparseness of the graph.)

The weight parameters must satisfy

$$|north\_weight| \leq 100{,}000, \quad |west\_weight| \leq 100{,}000, \quad |pop\_weight| \leq 100.$$

Vertices of the graph will appear in order of decreasing weight. The *seed* parameter defines the pseudo-random numbers used wherever a “random” choice between equal-weight vertices needs to be made, or when deciding whether to discard a Delaunay edge.

```
⟨ The plane_miles routine 41 ⟩ ≡
  Graph *plane_miles(n, north_weight, west_weight, pop_weight, extend, prob, seed)
      unsigned long n;    /* number of vertices desired */
      long north_weight;    /* coefficient of latitude in the weight function */
      long west_weight;    /* coefficient of longitude in the weight function */
      long pop_weight;    /* coefficient of population in the weight function */
      unsigned long extend;    /* should a point at infinity be included? */
      unsigned long prob;    /* probability of rejecting a Delaunay edge */
      long seed;    /* random number seed */
  { Graph *new_graph;    /* the graph constructed by plane_miles */
    ⟨ Use miles to set up the vertices of a graph 42 ⟩;
    ⟨ Compute the Delaunay triangulation and run through the Delaunay edges; reject them with
        probability prob/65536, otherwise append them with the road length in miles 43 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);    /* oops, we ran out of memory somewhere back there */
    }
    gb_free(new_graph→aux_data);    /* recycle special memory used by miles */
    if (extend) new_graph→n++;    /* make the “infinite” vertex legitimate */
    return new_graph;
  }
```

This code is used in section 4.

42. By setting the *max_distance* parameter to 1, we cause *miles* to produce a graph having the desired vertices but no edges. The vertices of this graph will have appropriate coordinate fields *x_coord*, *y_coord*, and *z_coord*.

⟨ Use *miles* to set up the vertices of a graph 42 ⟩ ≡

```
if (extend) extra_n++;   /* allocate one more vertex than usual */
if (n ≡ 0 ∨ n > MAX_N) n = MAX_N;   /* compute true number of vertices */
new_graph = miles(n, north_weight, west_weight, pop_weight, 1L, 0L, seed);
if (new_graph ≡ Λ) return Λ;   /* panic_code has been set by miles */
sprintf(new_graph→id, "plane_miles(%lu,%ld,%ld,%ld,%lu,%lu,%ld)", n, north_weight, west_weight,
    pop_weight, extend, prob, seed);
if (extend) extra_n−−;   /* restore extra_n to its previous value */
```

This code is used in section 41.

43. ⟨ Compute the Delaunay triangulation and run through the Delaunay edges; reject them with probability *prob*/65536, otherwise append them with the road length in miles 43 ⟩ ≡

```
gprob = prob;
if (extend) {
  inf_vertex = new_graph→vertices + new_graph→n;
  inf_vertex→name = gb_save_string("INF");
  inf_vertex→x_coord = inf_vertex→y_coord = inf_vertex→z_coord = −1;
} else inf_vertex = Λ;
delaunay(new_graph, new_mile_edge);
```

This code is used in section 41.

44. The mileages will all have been negated by *miles*, so we make them positive again.

⟨ Other subroutines 12 ⟩ +≡

```
void new_mile_edge(u, v)
    Vertex *u, *v;
{
  if ((gb_next_rand() ≫ 15) ≥ gprob) {
    if (u) {
      if (v) gb_new_edge(u, v, −miles_distance(u, v));
      else if (inf_vertex) gb_new_edge(u, inf_vertex, INFTY);
    } else if (inf_vertex) gb_new_edge(inf_vertex, v, INFTY);
  }
}
```

45. Index. As usual, we close with an index that shows where the identifiers of *gb_plane* are defined and used.

GB_PLANE

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.