Important: Before reading GB_RAMAN, please read or at least skim the program for GB_GRAPH.

1. Introduction. This GraphBase module contains the *raman* subroutine, which creates a family of "Ramanujan graphs" based on a theory developed by Alexander Lubotzky, Ralph Phillips, and Peter Sarnak [see *Combinatorica* **8** (1988), 261–277].

Ramanujan graphs are defined by the following properties: They are connected, undirected graphs in which every vertex has degree k, and every eigenvalue of the adjacency matrix is either $\pm k$ or has absolute value $\le 2\sqrt{k-1}$. Such graphs are known to have good expansion properties, small diameter, and relatively small independent sets; they cannot be colored with fewer than $k/\bigl(2\sqrt{k-1}\,\bigr)$ colors unless they are bipartite. The particular examples of Ramanujan graphs constructed here are based on interesting properties of quaternions with integer coefficients.

An example of the use of this procedure can be found in the demo program called GIRTH.

```
⟨gb_raman.h 1⟩ ≡
  extern Graph *raman();
```

2. The subroutine call *raman*(p, q, *type*, *reduce*) constructs an undirected graph in which each vertex has degree $p+1$. The number of vertices is $q+1$ if *type* $= 1$, or $\frac{1}{2}q(q+1)$ if *type* $= 2$, or $\frac{1}{2}(q-1)q(q+1)$ if *type* $= 3$, or $(q-1)q(q+1)$ if *type* $= 4$. The graph will be bipartite if and only if it has type 4. Parameters p and q must be distinct prime numbers, and q must be odd. Furthermore there are additional restrictions: If $p = 2$, the other parameter q must satisfy $q \bmod 8 \in \{1, 3\}$ and $q \bmod 13 \in \{1, 3, 4, 9, 10, 12\}$; this rules out about one fourth of all primes. Moreover, if *type* $= 3$ the value of p must be a quadratic residue modulo q; in other words, there must be an integer x such that $x^2 \equiv p \pmod q$. If *type* $= 4$, the value of p must not be a quadratic residue.

If you specify *type* $= 0$, the procedure will choose the largest permissible type (either 3 or 4); the value of the type selected will appear as part of the string placed in the resulting graph's *id* field. For example, if *type* $= 0$, $p = 2$, and $q = 43$, a type 4 graph will be generated, because 2 is not a quadratic residue modulo 43. This graph will have $44 \times 43 \times 42 = 79464$ vertices, each of degree 3. (Notice that graphs of types 3 and 4 can be quite large even when q is rather small.)

The largest permissible value of q is 46337; this is the largest prime whose square is less than 2^{31}. Of course you would use it only for a graph of type 1.

If *reduce* is nonzero, loops and multiple edges will be suppressed. In this case the degrees of some vertices might turn out to be less than $p+1$, in spite of what was said above.

Although type 4 graphs are bipartite, the vertices are not separated into two blocks as in other bipartite graphs produced by GraphBase routines.

All edges of the graphs have length 1.

3. If the *raman* 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 *raman* 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; }
#define dead_panic(c)
        { gb_free(working_storage); panic(c); }
#define late_panic(c)
        { gb_recycle(new_graph); dead_panic(c); }
```

4. The C file `gb_raman.c` has the following general shape:

```
#include "gb_graph.h"    /* we will use the GB_GRAPH data structures */
  ⟨ Preprocessor definitions ⟩
  ⟨ Type declarations 18 ⟩
  ⟨ Private variables and routines 6 ⟩
  Graph *raman(p, q, type, reduce)
      long p;    /* one less than the desired degree; must be prime */
      long q;    /* size parameter; must be prime and properly related to type */
      unsigned long type;    /* selector between different possible constructions */
      unsigned long reduce;    /* if nonzero, multiple edges and self-loops won't occur */
  { ⟨ Local variables 5 ⟩
    ⟨ Prepare tables for doing arithmetic modulo q 7 ⟩;
    ⟨ Choose or verify the type, and determine the number n of vertices 12 ⟩;
    ⟨ Set up a graph with n vertices, and assign vertex labels 13 ⟩;
    ⟨ Compute p + 1 generators that will define the graph's edges 19 ⟩;
    ⟨ Append the edges 26 ⟩;
    if (gb_trouble_code) late_panic(alloc_fault);
        /* oops, we ran out of memory somewhere back there */
    gb_free(working_storage);
    return new_graph;
  }
```

5. ⟨ Local variables 5 ⟩ ≡

```
  Graph *new_graph;    /* the graph constructed by raman */
  Area working_storage;    /* place for auxiliary tables */
```

See also section 9.

This code is used in section 4.

6. Brute force number theory. Instead of using routines like Euclid's algorithm to compute inverses and square roots modulo q, we have plenty of time to build complete tables, since q is smaller than the number of vertices we will be generating.

We will make three tables: *q_sqr*[k] will contain k^2 modulo q; *q_sqrt*[k] will contain one of the values of $\sqrt{k}$ if k is a quadratic residue; and *q_inv*[k] will contain the multiplicative inverse of k.

```
⟨ Private variables and routines 6 ⟩ ≡
  static long *q_sqr;   /* squares */
  static long *q_sqrt;   /* square roots (or −1 if not a quadratic residue) */
  static long *q_inv;   /* reciprocals */
```

See also sections 15, 20, 22, and 30.

This code is used in section 4.

7.
```
⟨ Prepare tables for doing arithmetic modulo q 7 ⟩ ≡
  if (q < 3 ∨ q > 46337) panic(very_bad_specs);   /* q is way too small or way too big */
  if (p < 2) panic(very_bad_specs + 1);   /* p is way too small */
  init_area(working_storage);
  q_sqr = gb_typed_alloc(3 * q, long, working_storage);
  if (q_sqr ≡ 0) panic(no_room + 1);
  q_sqrt = q_sqr + q;
  q_inv = q_sqrt + q;   /* note that gb_alloc has initialized everything to zero */
  ⟨ Compute the q_sqr and q_sqrt tables 8 ⟩;
  ⟨ Find a primitive root a, modulo q, and its inverse aa 10 ⟩;
  ⟨ Compute the q_inv table 11 ⟩;
```

This code is used in section 4.

8.
```
⟨ Compute the q_sqr and q_sqrt tables 8 ⟩ ≡
  for (a = 1; a < q; a++) q_sqrt[a] = −1;
  for (a = 1, aa = 1; a < q; aa = (aa + a + a + 1) % q, a++) {
    q_sqr[a] = aa;
    q_sqrt[aa] = q − a;   /* the smaller square root will survive */
    q_inv[aa] = −1;   /* we make q_inv[aa] nonzero when aa can't be a primitive root */
  }
```

This code is used in section 7.

9.
```
⟨ Local variables 5 ⟩ +≡
  register long a, aa, k;   /* primary indices in loops */
  long b, bb, c, cc, d, dd;   /* secondary indices */
  long n;   /* the number of vertices */
  long n_factor;   /* either ½(q − 1) (type 3) or q − 1 (type 4) */
  register Vertex *v;   /* the current vertex of interest */
```

10. Here we implicitly test that q is prime, by finding a primitive root whose powers generate everything. If q is not prime, its smallest divisor will cause the inner loop in this step to terminate with $k \ge q$, because no power of that divisor will be congruent to 1.

```
⟨Find a primitive root a, modulo q, and its inverse aa 10⟩ ≡
  for ($a = 2$; ; $a{+}{+}$)
    if ($q\_inv[a] \equiv 0$) {
      for ($b = a, k = 1$; $b \ne 1 \land k < q$; $aa = b, b = (a * b) \% q, k{+}{+}$) $q\_inv[b] = -1$;
      if ($k \ge q$) $dead\_panic(bad\_specs + 1)$;      /* $q$ is not prime */
      if ($k \equiv q - 1$) break;      /* good, $a$ is the primitive root we seek */
    }
```

This code is used in section 7.

11. As soon as we have discovered a primitive root, it is easy to generate all the inverses. (We could also generate the discrete logarithms if we had a need for them.)

We set $q\_inv[0] = q$; this will be our internal representation of ∞.

```
⟨Compute the q_inv table 11⟩ ≡
  for ($b = a, bb = aa$; $b \ne bb$; $b = (a * b) \% q, bb = (aa * bb) \% q$) $q\_inv[b] = bb, q\_inv[bb] = b$;
  $q\_inv[1] = 1$;
  $q\_inv[b] = b$;      /* at this point $b$ must equal $q - 1$ */
  $q\_inv[0] = q$;
```

This code is used in section 7.

12. The conditions we stated for validity of q when $p = 2$ are equivalent to the existence of $\sqrt{-2}$ and $\sqrt{13}$ modulo q, according to the law of quadratic reciprocity (see, for example, *Fundamental Algorithms*, exercise 1.2.4–47).

```
⟨Choose or verify the type, and determine the number n of vertices 12⟩ ≡
  if ($p \equiv 2$) {
    if ($q\_sqrt[13 \% q] < 0 \lor q\_sqrt[q - 2] < 0$) $dead\_panic(bad\_specs + 2)$;
        /* improper prime to go with $p = 2$ */
  }
  if ($(a = p \% q) \equiv 0$) $dead\_panic(bad\_specs + 3)$;      /* $p$ divisible by $q$ */
  if ($type \equiv 0$) $type = (q\_sqrt[a] > 0 \; ? \; 3 : 4)$;
  $n\_factor = (type \equiv 3 \; ? \; (q - 1)/2 : q - 1)$;
  switch ($type$) {
  case 1: $n = q + 1$; break;
  case 2: $n = q * (q + 1)/2$; break;
  default:
    if ($(q\_sqrt[a] > 0 \land type \ne 3) \lor (q\_sqrt[a] < 0 \land type \ne 4)$)
      $dead\_panic(bad\_specs + 4)$;      /* wrong type for $p$ modulo $q$ */
    if ($q > 1289$) $dead\_panic(bad\_specs + 5)$;      /* way too big for types 3, 4 */
    $n = n\_factor * q * (q + 1)$;
    break;
  }
  if ($p \ge$ (long) (#3fffffff/$n$)) $dead\_panic(bad\_specs + 6)$;      /* $(p + 1)n \ge 2^{30}$ */
```

This code is used in section 4.

13. The vertices. Graphs of type 1 have vertices from the set $\{0, 1, \ldots, q-1, \infty\}$, namely the integers modulo q with an additional "infinite" element thrown in. The idea is to operate on these quantities by adding constants, and/or multiplying by constants, and/or taking reciprocals, modulo q.

Graphs of type 2 have vertices that are unordered pairs of distinct elements from that same $(q+1)$-element set.

Graphs of types 3 and 4 have vertices that are 2×2 matrices having nonzero determinants modulo q. The determinants of type 3 matrices are, in fact, nonzero quadratic residues. We consider two matrices to be equivalent if one is obtained from the other by multiplying all entries by a constant (modulo q); therefore we will normalize all the matrices so that the second row is either $(0, 1)$ or has the form $(1, x)$ for some x. The total number of equivalence classes of type 4 matrices obtainable in this way is $(q+1)q(q-1)$, because we can choose the second row in $q+1$ ways, after which there are two cases: Either the second row is $(0, 1)$, and we can select the upper right corner element arbitrarily and choose the upper left corner element nonzero; or the second row is $(1, x)$, and we can select the upper left corner element arbitrarily and then choose an upper right corner element to make the determinant nonzero. For type 3 the counting is similar, except that "nonzero" becomes "nonzero quadratic residue," hence there are exactly half as many choices.

It is easy to verify that the equivalence classes of matrices that correspond to vertices in these graphs of types 3 and 4 are closed under matrix multiplication. Therefore the vertices can be regarded as the elements of finite groups. The type 3 group for a given q is often called the linear fractional group $LF(2, \mathbf{F}_q)$, or the projective special linear group $PSL(2, \mathbf{F}_q)$, or the linear simple group $L_2(q)$; it can also be regarded as the group of 2×2 matrices with determinant 1 (mod q), when the matrix A is considered equivalent to $-A$. (This group is a simple group for all primes $q > 2$.) The type 4 group is officially known as the projective general linear group of degree 2 over the field of q elements, $PGL(2, \mathbf{F}_q)$.

```
⟨ Set up a graph with n vertices, and assign vertex labels 13 ⟩ ≡
  new_graph = gb_new_graph(n);
  if (new_graph ≡ Λ) dead_panic(no_room);    /* out of memory before we try to add edges */
  sprintf(new_graph→id, "raman(%ld,%ld,%lu,%lu)", p, q, type, reduce);
  strcpy(new_graph→util_types, "ZZZIIZIZZZZZZZ");
  v = new_graph→vertices;
  switch (type) {
  case 1: ⟨ Assign labels from the set {0, 1, ..., q − 1, ∞} 14 ⟩; break;
  case 2: ⟨ Assign labels for pairs of distinct elements 16 ⟩; break;
  default: ⟨ Assign projective matrix labels 17 ⟩; break;
  }
```

This code is used in section 4.

14. Type 1 graphs are the easiest to label. We store a serial number in utility field $x.I$, using q to represent ∞.

```
⟨ Assign labels from the set {0, 1, ..., q − 1, ∞} 14 ⟩ ≡
  new_graph→util_types[4] = 'Z';
  for (a = 0; a < q; a++) {
    sprintf(name_buf, "%ld", a);
    v→name = gb_save_string(name_buf);
    v→x.I = a;
    v++;
  }
  v→name = gb_save_string("INF");
  v→x.I = q;
  v++;
```

This code is used in section 13.

15. ⟨ Private variables and routines 6 ⟩ +≡

```
static char name_buf[] = "(1111,1111;1,1111)";    /* place to form vertex names */
```

16. The type 2 labels run from $\{0,1\}$ to $\{q-1,\infty\}$; we put the coefficients into $x.I$ and $y.I$, where they might prove useful in some applications.

⟨ Assign labels for pairs of distinct elements 16 ⟩ ≡

```
for (a = 0; a < q; a++)
  for (aa = a + 1; aa ≤ q; aa++) {
    if (aa ≡ q) sprintf(name_buf, "{%ld,INF}", a);
    else sprintf(name_buf, "{%ld,%ld}", a, aa);
    v→name = gb_save_string(name_buf);
    v→x.I = a; v→y.I = aa;
    v++;
  }
```

This code is used in section 13.

17. For graphs of types 3 and 4, we set the $x.I$ and $y.I$ fields to the elements of the first row of the matrix, and we set the $z.I$ field equal to the ratio of the elements of the second row (again with q representing ∞).

The vertices in this case consist of $q(q+1)$ blocks of vertices having a given second row and a given element in the upper left or upper right position. Within each block of vertices, the determinants are respectively congruent modulo q to 1^2, 2^2, …, $(\frac{q-1}{2})^2$ in the case of type 3 graphs, or to 1, 2, …, $q-1$ in the case of type 4.

⟨ Assign projective matrix labels 17 ⟩ ≡

```
new_graph→util_types[5] = 'I';
for (c = 0; c ≤ q; c++)
  for (b = 0; b < q; b++)
    for (a = 1; a ≤ n_factor; a++) {
      v→z.I = c;
      if (c ≡ q) {    /* second row of matrix is (0,1) */
        v→y.I = b;
        v→x.I = (type ≡ 3 ? q_sqr[a] : a);    /* determinant is a^2 or a */
        sprintf(name_buf, "(%ld,%ld;0,1)", v→x.I, b);
      } else {    /* second row of matrix is (1,c) */
        v→x.I = b;
        v→y.I = (b * c + q − (type ≡ 3 ? q_sqr[a] : a)) % q;    /* determinant is a^2 or a */
        sprintf(name_buf, "(%ld,%ld;1,%ld)", b, v→y.I, c);
      }
      v→name = gb_save_string(name_buf);
      v++;
    }
```

This code is used in section 13.

18. Group generators. We will define a set of $p+1$ permutations $\{\pi_0, \pi_1, \ldots, \pi_p\}$ of the vertices, such that the arcs of our graph will go from v to $v\pi_k$ for $0 \le k \le p$. Thus, each path in the graph will be defined by a product of permutations; the cycles of the graph will correspond to vertices that are left fixed by a product of permutations. The graph will be undirected, because the inverse of each π_k will also be one of the permutations of the generating set.

In fact, each permutation π_k will be defined by a 2×2 matrix. For graphs of types 3 and 4, the permutations will therefore correspond to certain vertices, and the vertex $v\pi_k$ will simply be the product of matrix v by matrix π_k.

For graphs of type 1, the permutations will be defined by linear fractional transformations, which are mappings of the form

$$v \longmapsto \frac{av+b}{cv+d} \bmod q\,.$$

This transformation applies to all $v \in \{0, 1, \ldots, q-1, \infty\}$, under the usual conventions that $x/0 = \infty$ when $x \ne 0$ and $(x\infty + x')/(y\infty + y') = x/y$. The composition of two such transformations is again a linear fractional transformation, corresponding to the product of the two associated matrices $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$.

Graphs of type 2 will be handled just like graphs of type 1, except that we will compute the images of two distinct points $v = \{v_1, v_2\}$ under the linear fractional transformation. The two images will be distinct, because the transformation is invertible.

When $p = 2$, a special set of three generating matrices π_0, π_1, π_2 can be shown to define Ramanujan graphs; these matrices are described below. Otherwise p is odd, and the generators are based on the theory of integral quaternions. Integral quaternions, for our purposes, are quadruples of the form $\alpha = a_0 + a_1 i + a_2 j + a_3 k$, where a_0, a_1, a_2, and a_3 are integers; we multiply them by using the associative but noncommutative multiplication rules $i^2 = j^2 = k^2 = ijk = -1$. If we write $\alpha = a + A$, where a is the "scalar" a_0 and A is the "vector" $a_1 i + a_2 j + a_3 k$, the product of quaternions $\alpha = a + A$ and $\beta = b + B$ can be expressed as

$$(a + A)(b + B) = ab - A\cdot B + aB + bA + A\times B\,,$$

where $A\cdot B$ and $A\times B$ are the usual dot product and cross product of vectors. The conjugate of $\alpha = a + A$ is $\overline{\alpha} = a - A$, and we have $\alpha\overline{\alpha} = a_0^2 + a_1^2 + a_2^2 + a_3^2$. This important quantity is called $N(\alpha)$, the norm of α. It is not difficult to verify that $N(\alpha\beta) = N(\alpha)N(\beta)$, because of the basic identity $\overline{\alpha\beta} = \overline{\beta}\,\overline{\alpha}$ and the fact that $\alpha x = x\alpha$ when x is scalar.

Integral quaternions have a beautiful theory; for example, there is a nice variant of Euclid's algorithm by which we can compute the greatest common left divisor of any two integral quaternions having odd norm. This algorithm makes it possible to prove that integral quaternions whose coefficients are relatively prime can be canonically factored into quaternions whose norm is prime. However, the details of that theory are beyond the scope of this documentation. It will suffice for our purposes to observe that we can use quaternions to define the finite groups $PSL(2, \mathbf{F}_q)$ and $PGL(2, \mathbf{F}_q)$ in a different way from the definitions given earlier: Suppose we consider two quaternions to be equivalent if one is a nonzero scalar multiple of the other, modulo q. Thus, for example, if $q = 3$ we consider $1 + 4i - j$ to be equivalent to $1 + i + 2j$, and also equivalent to $2 + 2i + j$. It turns out that there are exactly $(q+1)q(q-1)$ such equivalence classes, when we omit quaternions whose norm is a multiple of q; and they form a group under quaternion multiplication that is the same as the projective group of 2×2 matrices under matrix multiplication, modulo q. One way to prove this is by means of the one-to-one correspondence

$$a_0 + a_1 i + a_2 j + a_3 k \longleftrightarrow \begin{pmatrix} a_0 + a_1 g + a_3 h & a_2 + a_3 g - a_1 h \\ -a_2 + a_3 g - a_1 h & a_0 - a_1 g - a_3 h \end{pmatrix},$$

where g and h are integers with $g^2 + h^2 \equiv -1 \pmod{q}$.

Jacobi proved that the number of ways to represent any odd number p as a sum of four squares $a_0^2 + a_1^2 + a_2^2 + a_3^2$ is 8 times the sum of divisors of p. [This fact appears in the concluding sentence of his monumental work *Fundamenta Nova Theoriæ Functionum Ellipticorum*, Königsberg, 1829.] In particular, when p is prime, the number of such representations is $8(p+1)$; in other words, there are exactly $8(p+1)$ quaternions

$\alpha = a_0+a_1i+a_2j+a_3k$ with $N(\alpha) = p$. These quaternions form $p+1$ equivalence classes under multiplication by the eight "unit quaternions" $\{\pm1, \pm i, \pm j, \pm k\}$. We will select one element from each equivalence class, and the resulting $p + 1$ quaternions will correspond to $p + 1$ matrices, which will generate the $p + 1$ arcs leading from each vertex in the graphs to be constructed.

```
⟨ Type declarations 18 ⟩ ≡
  typedef struct {
    long a0, a1, a2, a3;     /* coefficients of a quaternion */
    unsigned long bar;    /* the index of the inverse (conjugate) quaternion */
  } quaternion;
```

This code is used in section 4.

19. A global variable *gen_count* will be declared below, indicating the number of generators found so far. When p isn't prime, we will find more than $p + 1$ solutions, so we allocate an extra slot in the *gen* table to hold a possible overflow entry.

```
⟨ Compute $p+1$ generators that will define the graph's edges 19 ⟩ ≡
  gen = gb_typed_alloc(p + 2, quaternion, working_storage);
  if (gen ≡ Λ) late_panic(no_room + 2);      /* not enough memory */
  gen_count = 0; max_gen_count = p + 1;
  if (p ≡ 2) ⟨ Fill the gen table with special generators 25 ⟩
  else ⟨ Fill the gen table with representatives of all quaternions having norm p 21 ⟩;
  if (gen_count ≠ max_gen_count) late_panic(bad_specs + 7);      /* p is not prime */
```

This code is used in section 4.

20.
```
⟨ Private variables and routines 6 ⟩ +≡
  static quaternion *gen;      /* table of the p + 1 generators */
```

21. As mentioned above, quaternions of norm p come in sets of 8, differing from each other only by unit multiples; we need to choose one of the 8. Suppose $a_0^2 + a_1^2 + a_2^2 + a_3^2 = p$. If $p \bmod 4 = 1$, exactly one of the a's will be odd; so we call it a_0 and assign it a positive sign. When $p \bmod 4 = 3$, exactly one of the a's will be even; we call it a_0, and if it is nonzero we make it positive. If $a_0 = 0$, we make sure that one of the others—say the rightmost appearance of the largest one—is positive. In this way we obtain a unique representative from each set of 8 equivalent quaternions.

For example, the four quaternions of norm 3 are $\pm i \pm j + k$; the six of norm 5 are $1 \pm 2i$, $1 \pm 2j$, $1 \pm 2k$.

In the program here we generate solutions to $a^2 + b^2 + c^2 + d^2 = p$ when $a \not\equiv b \equiv c \equiv d \pmod 2$ and $b \le c \le d$. The variables *aa*, *bb*, and *cc* hold the respective values $p - a^2 - b^2 - c^2 - d^2$, $p - a^2 - 3b^2$, and $p - a^2 - 2c^2$. The **for** statements use the fact that a^2 increases by $4(a + 1)$ when a increases by 2.

```
⟨ Fill the gen table with representatives of all quaternions having norm p 21 ⟩ ≡
  { long sa, sb;      /* p − a^2, p − a^2 − b^2 */
    long pp = (p ≫ 1) & 1;      /* 0 if p mod 4 = 1, 1 if p mod 4 = 3 */
    for (a = 1 − pp, sa = p − a; sa > 0; sa −= (a + 1) ≪ 2, a += 2)
      for (b = pp, sb = sa − b, bb = sb − b − b; bb ≥ 0; bb −= 12 * (b + 1), sb −= (b + 1) ≪ 2, b += 2)
        for (c = b, cc = bb; cc ≥ 0; cc −= (c + 1) ≪ 3, c += 2)
          for (d = c, aa = cc; aa ≥ 0; aa −= (d + 1) ≪ 2, d += 2)
            if (aa ≡ 0) ⟨ Deposit the quaternions associated with a + bi + cj + dk 23 ⟩;
    ⟨ Change the gen table to matrix format 24 ⟩;
  }
```

This code is used in section 19.

22. If $a > 0$ and $0 < b < c < d$, we obtain 48 different classes of quaternions having the same norm by permuting $\{b, c, d\}$ in six ways and attaching signs to each permutation in eight ways. This happens, for example, when $p = 71$ and $(a, b, c, d) = (6, 1, 3, 5)$. Fewer quaternions arise when $a = 0$ or $0 = b$ or $b = c$ or $c = d$.

The inverse of the matrix corresponding to a quaternion is the matrix corresponding to the conjugate quaternion. Therefore a generating matrix π_k will be its own inverse if and only if it comes from a quaternion with $a = 0$.

It is convenient to have a subroutine that deposits a new quaternion and its conjugate into the table of generators.

```
⟨ Private variables and routines 6 ⟩ +≡
  static unsigned long gen_count;    /* the next available quaternion slot */
  static unsigned long max_gen_count;    /* p + 1, stored as a global variable */
  static void deposit(a, b, c, d)
      long a, b, c, d;    /* a solution to a^2 + b^2 + c^2 + d^2 = p */
  {
    if (gen_count ≥ max_gen_count)    /* oops, we already found p + 1 solutions */
      gen_count = max_gen_count + 1;    /* this will happen only if p isn't prime */
    else {
      gen[gen_count].a0 = gen[gen_count + 1].a0 = a;
      gen[gen_count].a1 = b; gen[gen_count + 1].a1 = −b;
      gen[gen_count].a2 = c; gen[gen_count + 1].a2 = −c;
      gen[gen_count].a3 = d; gen[gen_count + 1].a3 = −d;
      if (a) {
        gen[gen_count].bar = gen_count + 1;
        gen[gen_count + 1].bar = gen_count;
        gen_count += 2;
      } else {
        gen[gen_count].bar = gen_count;
        gen_count++;
      }
    }
  }
```

23. ⟨ Deposit the quaternions associated with $a + bi + cj + dk$ 23 ⟩ ≡

```
{
  deposit(a, b, c, d);
  if (b) {
    deposit(a, −b, c, d); deposit(a, −b, −c, d);
  }
  if (c) deposit(a, b, −c, d);
  if ($b < c$) {
    deposit(a, c, b, d); deposit(a, −c, b, d); deposit(a, c, d, b); deposit(a, −c, d, b);
    if (b) {
      deposit(a, c, −b, d); deposit(a, −c, −b, d); deposit(a, c, d, −b); deposit(a, −c, d, −b);
    }
  }
  if ($c < d$) {
    deposit(a, b, d, c); deposit(a, d, b, c);
    if (b) {
      deposit(a, −b, d, c); deposit(a, −b, d, −c); deposit(a, d, −b, c); deposit(a, d, −b, −c);
    }
    if (c) {
      deposit(a, b, d, −c); deposit(a, d, b, −c);
    }
    if ($b < c$) {
      deposit(a, d, c, b); deposit(a, d, −c, b);
      if (b) {
        deposit(a, d, c, −b); deposit(a, d, −c, −b);
      }
    }
  }
}
```

This code is used in section 21.

24. Once we've found the generators in quaternion form, we want to convert them to 2×2 matrices, using the correspondence mentioned earlier:

$$a_0 + a_1 i + a_2 j + a_3 k \;\longleftrightarrow\; \begin{pmatrix} a_0 + a_1 g + a_3 h & a_2 + a_3 g - a_1 h \\ -a_2 + a_3 g - a_1 h & a_0 - a_1 g - a_3 h \end{pmatrix},$$

where g and h are integers with $g^2 + h^2 \equiv -1 \pmod q$. Appropriate values for g and h can always be found by the formulas

$$g = \sqrt{k} \qquad \text{and} \qquad h = \sqrt{q-1-k},$$

where k is the largest quadratic residue modulo q. For if $q-1$ is not a quadratic residue, and if $k+1$ isn't a residue either, then $q-1-k$ must be a quadratic residue because it is congruent to the product $(q-1)(k+1)$ of nonresidues. (We will have $h = 0$ if and only if $q \bmod 4 = 1$; $h = 1$ if and only if $q \bmod 8 = 3$; $h = \sqrt{2}$ if and only if $q \bmod 24 = 7$ or 15; etc.)

```
⟨ Change the gen table to matrix format 24 ⟩ ≡
  { register long g, h;
    long a00, a01, a10, a11;      /* entries of 2 × 2 matrix */
    for (k = q − 1; q_sqrt[k] < 0; k−−) ;      /* find the largest quadratic residue, k */
    g = q_sqrt[k]; h = q_sqrt[q − 1 − k];
    for (k = p; k ≥ 0; k−−) {
      a00 = (gen[k].a0 + g ∗ gen[k].a1 + h ∗ gen[k].a3) % q;
      if (a00 < 0) a00 += q;
      a11 = (gen[k].a0 − g ∗ gen[k].a1 − h ∗ gen[k].a3) % q;
      if (a11 < 0) a11 += q;
      a01 = (gen[k].a2 + g ∗ gen[k].a3 − h ∗ gen[k].a1) % q;
      if (a01 < 0) a01 += q;
      a10 = (−gen[k].a2 + g ∗ gen[k].a3 − h ∗ gen[k].a1) % q;
      if (a10 < 0) a10 += q;
      gen[k].a0 = a00; gen[k].a1 = a01; gen[k].a2 = a10; gen[k].a3 = a11;
    }
  }
```

This code is used in section 21.

25. When $p = 2$, the following three appropriate generating matrices have been found by Patrick Chiu:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad \begin{pmatrix} 2+s & t \\ t & 2-s \end{pmatrix}, \qquad \text{and} \qquad \begin{pmatrix} 2-s & -t \\ -t & 2+s \end{pmatrix},$$

where $s^2 \equiv -2$ and $t^2 \equiv -26 \pmod q$. The determinants of these matrices are respectively -1, 32, and 32; the product of the second and third matrices is 32 times the identity matrix. Notice that when 2 is a quadratic residue (this happens when $q = 8k + 1$), the determinants are all quadratic residues, so we get a graph of type 3. When 2 is a quadratic nonresidue (which happens when $q = 8k + 3$), the determinants are all nonresidues, so we get a graph of type 4.

```
⟨ Fill the gen table with special generators 25 ⟩ ≡
  { long s = q_sqrt[q − 2], t = (q_sqrt[13 % q] ∗ s) % q;
    gen[0].a0 = 1; gen[0].a1 = gen[0].a2 = 0; gen[0].a3 = q − 1; gen[0].bar = 0;
    gen[1].a0 = gen[2].a3 = (2 + s) % q;
    gen[1].a1 = gen[1].a2 = t;
    gen[2].a1 = gen[2].a2 = q − t;
    gen[1].a3 = gen[2].a0 = (q + 2 − s) % q;
    gen[1].bar = 2; gen[2].bar = 1;
    gen_count = 3;
  }
```

This code is used in section 19.

26. Constructing the edges. The remaining task is to use the permutations defined by the *gen* table to create the arcs of the graph and their inverses.

The *ref* fields in each arc will refer to the permutation leading to the arc. In most cases each vertex v will have degree exactly $p+1$, and the edges emanating from it will appear in a linked list having the respective *ref* fields $0, 1, \ldots, p$ in order. However, if *reduce* is nonzero, self-loops and multiple edges will be eliminated, so the degree might be less than $p+1$; in this case the *ref* fields will still be in ascending order, but some generators won't be referenced.

There is a subtle case where $reduce = 0$ but the degree of a vertex might actually be greater than $p+1$. We want the graph g generated by *raman* to satisfy the conventions for undirected graphs stated in GB_GRAPH; therefore, if any of the generating permutations has a fixed point, we will create two arcs for that fixed point, and the corresponding vertex v will have an edge running to itself. Since each edge consists of two arcs, such an edge will produce two consecutive entries in the list $v\to arcs$. If the generating permutation happens to be its own inverse, there will be two consecutive entries with the same *ref* field; this means there will be more than $p+1$ entries in $v\to arcs$, and the total number of arcs $g\to m$ will exceed $(p+1)n$. Self-inverse generating permutations arise only when $p=2$ or when p is expressible as a sum of three odd squares (hence $p \bmod 8 = 3$); and such permutations will have fixed points only when $type < 3$. Therefore this anomaly does not arise often. But it does occur, for example, in the smallest graph generated by *raman*, namely when $p=2$, $q=3$, and $type=1$, when there are 4 vertices and 14 (not 12) arcs.

```
#define ref a.I    /* the ref field of an arc refers to its permutation number */
⟨Append the edges 26⟩ ≡
  for (k = p; k ≥ 0; k−−) { long kk;
    if ((kk = gen[k].bar) ≤ k)    /* we assume that kk = k or kk = k − 1 */
      for (v = new_graph→vertices; v < new_graph→vertices + n; v++) {
        register Vertex *u;
        ⟨Compute the image, u, of v under the permutation defined by gen[k] 27⟩;
        if (u ≡ v) {
          if (¬reduce) {
            gb_new_edge(v, v, 1L);
            v→arcs→ref = kk; (v→arcs + 1)→ref = k;
              /* see the remarks above regarding the case kk = k */
          }
        } else { register Arc *ap;
          if (u→arcs ∧ u→arcs→ref ≡ kk) continue;
              /* kk = k and we've already done this two-cycle */
          else if (reduce)
            for (ap = v→arcs; ap; ap = ap→next)
              if (ap→tip ≡ u) goto done;    /* there's already an edge between u and v */
          gb_new_edge(v, u, 1L);
          v→arcs→ref = k; u→arcs→ref = kk;
          if ((ap = v→arcs→next) ≠ Λ ∧ ap→ref ≡ kk) {
            v→arcs→next = ap→next; ap→next = v→arcs; v→arcs = ap;
          }    /* now the v→arcs list has ref fields in order again */
        done: ;
        }
      }
  }
```

This code is used in section 4.

27. For graphs of types 3 and 4, our job is to compute a 2×2 matrix product, reduce it modulo q, and find the appropriate equivalence class u.

```
⟨ Compute the image, $u$, of $v$ under the permutation defined by $gen[k]$ 27 ⟩ ≡
  **if** $(type < 3)$
    ⟨ Compute the image, $u$, of $v$ under the linear fractional transformation defined by $gen[k]$ 31 ⟩
  **else** { **long** $a00 = gen[k].a0$, $a01 = gen[k].a1$, $a10 = gen[k].a2$, $a11 = gen[k].a3$;
    $a = v\to x.I$; $b = v\to y.I$;
    **if** $(v\to z.I \equiv q)$ $c = 0, d = 1$;
    **else** $c = 1, d = v\to z.I$;
    ⟨ Compute the matrix product $(aa, bb;\ cc, dd) = (a, b;\ c, d) * (a00, a01;\ a10, a11)$ 28 ⟩;
    $a = (cc\ ?\ q\_inv[cc] : q\_inv[dd])$;   /* now $a$ is a normalization factor */
    $d = (a * dd) \% q$; $c = (a * cc) \% q$; $b = (a * bb) \% q$; $a = (a * aa) \% q$;
    ⟨ Set $u$ to the vertex whose label is $(a, b;\ c, d)$ 29 ⟩;
  }
```

This code is used in section 26.

28. ⟨ Compute the matrix product $(aa, bb;\ cc, dd) = (a, b;\ c, d) * (a00, a01;\ a10, a11)$ 28 ⟩ ≡

```
  $aa = (a * a00 + b * a10) \% q$;
  $bb = (a * a01 + b * a11) \% q$;
  $cc = (c * a00 + d * a10) \% q$;
  $dd = (c * a01 + d * a11) \% q$;
```

This code is used in section 27.

29. ⟨ Set u to the vertex whose label is $(a, b;\ c, d)$ 29 ⟩ ≡

```
  **if** $(c \equiv 0)$ $d = q, aa = a$;
  **else** {
    $aa = (a * d - b) \% q$;
    **if** $(aa < 0)$ $aa$ += $q$;
    $b = a$;
  }   /* now $aa$ is the determinant of the matrix */
  $u = new\_graph\to vertices + ((d * q + b) * n\_factor + (type \equiv 3\ ?\ q\_sqrt[aa] : aa) - 1)$;
```

This code is used in section 27.

30. Linear fractional transformations. Given a nonsingular 2×2 matrix $\binom{a\,b}{c\,d}$, the linear fractional transformation $z \mapsto (az+b)/(cz+d)$ is defined modulo q by the following subroutine. We assume that the matrix $\binom{a\,b}{c\,d}$ appears in row k of the *gen* table.

```
⟨ Private variables and routines 6 ⟩ +≡
  static long lin_frac(a, k)
      long a;     /* the number being transformed; q represents ∞ */
      long k;     /* index into gen table */
  { register long q = q_inv[0];     /* the modulus */
    long a00 = gen[k].a0, a01 = gen[k].a1, a10 = gen[k].a2, a11 = gen[k].a3;     /* the coefficients */
    register long num, den;     /* numerator and denominator */
    if (a ≡ q) num = a00, den = a10;
    else num = (a00 * a + a01) % q, den = (a10 * a + a11) % q;
    if (den ≡ 0) return q;
    else return (num * q_inv[den]) % q;
  }
```

31. We are computing the same values of *lin_frac* over and over again in type 2 graphs, but the author was too lazy to optimize this.

```
⟨ Compute the image, u, of v under the linear fractional transformation defined by gen[k] 31 ⟩ ≡
  if (type ≡ 1) u = new_graph→vertices + lin_frac(v→x.I, k);
  else {
    a = lin_frac(v→x.I, k); aa = lin_frac(v→y.I, k);
    u = new_graph→vertices + (a < aa ? (a * (2 * q − 1 − a))/2 + aa − 1 : (aa * (2 * q − 1 − aa))/2 + a − 1);
  }
```

This code is used in section 27.

32. Index. Here is a list that shows where the identifiers of this program are defined and used.

January 9, 2001 at 12:11

GB_RAMAN

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