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

1. Introduction. This program computes numerical coordinates so that I can experiment with some of the fascinating patterns that arise when the hyperbolic plane is tiled with 36°-45°-90° triangles. Such a tiling is unique, but it can be viewed in many different ways.

Maurice Margenstern discovered a beautiful way to assign numbers to the vertices, based on Fibonacci representations; his method is discussed in §6 of the paper "A universal cellular automaton in the hyperbolic plane," by Francine Herrmann and Maurice Margenstern, *Theoretical Computer Science* **296** (2003), 327–364. I want to play with those ideas further, and for this purpose I need to make special kinds of graph paper.

I'm writing this program for fun and experience. So I'm using basic brute force, together with data structures that ought to help me gain both a local and global understanding of the tiling.

```
#define maxn 300    /* this many triangles will be computed */
        /* in this implementation I assume that 3 * maxn < 1024 */
#define hprime 1009    /* a prime number, should be at least 2 * maxn */
#include <stdio.h>
#include <math.h>
  ⟨ Type definitions 2 ⟩
  ⟨ Global variables 6 ⟩
  ⟨ Subroutines 3 ⟩
  main()
  {
    register int a, b, c, j, k, t;
    register double phi;
    ⟨ Set up triangle 0 11 ⟩;
    for (k = 0; tptr < maxn; k++) ⟨ Compute the neighbors of triangle k 13 ⟩;
  }
```

2. Hyperbolic data structures. I think the most convenient way to deal with the hyperbolic plane is to consider its points to be those of the Euclidean upper half plane, namely the points (x, y) with $y > 0$. Its "lines" are half-circles centered on the x-axis, namely the sets of points $(c + r\cos\theta, r\sin\theta)$ for some center c and some radius $r > 0$, as θ runs from 0 to π. Given a point (x, y) and an angle θ between 0 and π, we can therefore can construct a corresponding hyperbolic line with center and radius

$$c = x - y\cot\theta, \qquad r = y\csc\theta.$$

⟨ Type definitions 2 ⟩ ≡

```
typedef struct { double x, y; } point;
typedef struct { double c, r; } circle;
```

See also section 7.

This code is used in section 1.

3. The unique hyperbolic line that passes through two given points (x, y) and (x', y') is centered at

$$c = \frac{x^2 + y^2 - x'^2 - y'^2}{2(x - x')} = \frac{x + x'}{2} + \frac{y^2 - y'^2}{2(x - x')}.$$

(If $x = x'$, we have $c = \infty$ and the "circle" is actually a straight vertical line. But I don't have to worry about that case, because it won't arise in this program.)

⟨ Subroutines 3 ⟩ ≡

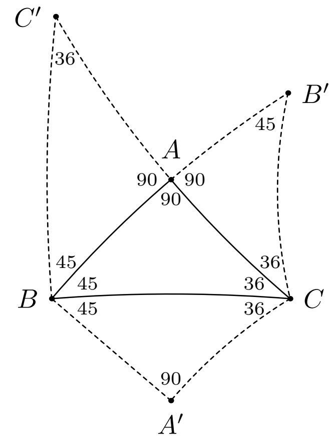
```
circle common(point z, point zp)
{
  circle t;
  t.c = (z.x + zp.x)/2.0 + ((z.y + zp.y)/2.0) * ((z.y - zp.y)/(z.x - zp.x));
  if (fabs(t.c) < 0.00001) t.c = 0.0;
  t.r = sqrt((z.x - t.c) * (z.x - t.c) + z.y * z.y);
  return t;
}
```

See also sections 4, 5, and 8.

This code is used in section 1.

4. The main technical operation in this program is to "reflect" a point with respect to a given hyperbolic line. If the line has center c and radius r, the reflection of $(c+s\cos\theta, s\sin\theta)$ is defined to be $(c+t\cos\theta, t\sin\theta)$, where $st = r^2$. One can show that this mapping is an automorphism of the hyperbolic plane.

We're interested in reflection because every triangle in the tiling being computed has three neighbors, each of which is obtained by reflecting one of the vertices about the opposite edge. Consider, for example, the triangle ABC shown here:



Its neighbors $A'BC$, $AB'C$, and ABC' are found by reflecting A about BC, B about CA, and C about AB.

Repleated reflections will generate the whole tiling. Notice that, in this example, four triangles of the complete tiling will surround point A, eight triangles will surround point B, and ten triangles will surround point C. (The angles around any vertex of a tiling must sum to 360°, even though the angles of a hyperbolic triangle always sum to *less* than 180°.)

Incidentally, I could have stored r^2 instead of r in the **circle** nodes, because this program computes reflections from r^2. But what the heck, I prefer to work with r instead of r^2 when I'm looking at this stuff.

⟨ Subroutines 3 ⟩ +≡

```
point reflect(point z, circle l)
{
  point t;
  register double alpha;
  alpha = l.r * l.r / ((z.x - l.c) * (z.x - l.c) + z.y * z.y);
  t.x = l.c + alpha * (z.x - l.c);
  t.y = alpha * z.y;
  return t;
}
```

5. As our algorithm proceeds, it will repeatedly compute points and/or circles that have already been seen. Therefore we maintain a dictionary of what we know.

At first I tried using a hash table. But that was unsatisfactory, because near-but-unequal values should be considered equivalent. Therefore binary search trees are used in the present code.

In practice, I found that most of the equivalent values agreed to within 10^{-16} or so, although exact agreement was rather rare. Only two cases had an absolute error greater than 10^{-11}, and in those cases the error was $\approx 1.1 \times 10^{-10}$.

The following routines return an index to the saved copy of a given point or circle.

```
#define eps 0.000001   /* fuzziness for comparisons */
⟨Subroutines 3⟩ +≡
  int savepoint(point z)
  {
    register int p, *q = &pleft[0];
    for (p = *q; p; p = *q) {
      if (fabs(hpoint[p].x - z.x) < eps) {
        if (fabs(hpoint[p].y - z.y) < eps) goto found;
        if (hpoint[p].y < z.y) q = &pleft[p];
        else q = &pright[p];
      } else if (hpoint[p].x < z.x) q = &pleft[p];
      else q = &pright[p];
    }
    p = ++pptr;
    *q = p;
    printf("z%d=(%.15g,%.15g)\n", p, z.x, z.y);
    hpoint[p] = z;
  found: return p;
  }
  int savecircle(circle l)
  {
    register int p, *q = &cleft[0];
    for (p = *q; p; p = *q) {
      if (fabs(hcircle[p].c - l.c) < eps) {
        if (fabs(hcircle[p].r - l.r) < eps) goto found;
        if (hcircle[p].r < l.r) q = &cleft[p];
        else q = &cright[p];
      } else if (hcircle[p].c < l.c) q = &cleft[p];
      else q = &cright[p];
    }
    p = ++cptr;
    *q = p;
    printf("l%d=(%.15g,%.15g)\n", p, l.c, l.r);
    hcircle[p] = l;
  found: return p;
  }
```

6. ⟨ Global variables 6 ⟩ ≡

```
point hpoint[3 * maxn];   /* dictionary of known points */
int pptr;   /* the number of known points */
int pleft[3 * maxn], pright[3 * maxn];   /* links for binary tree search */
circle hcircle[3 * maxn];   /* dictionary of known hyperbolic lines */
int cptr;   /* the number of known lines */
int cleft[3 * maxn], cright[3 * maxn];   /* links for binary tree search */
```

See also sections 9 and 10.

This code is used in section 1.

7. The main component of our data structure is the table of all triangles that we have identified so far. Each triangle is represented by indices that point to its three vertices, its three edges, and its three neighbors.

The vertex indices are called *v36*, *v45*, and *v90*, because each triangle has a vertex with each of the angles $(36°, 45°, 90°)$. Edges *e36*, *e45*, and *e90* are *opposite* those vertices; triangles *t36*, *t45*, and *t90* are the neighbors on the other side of those edges.

⟨ Type definitions 2 ⟩ +≡

```
typedef struct {
   int v36, v45, v90;   /* where the vertices occur in hpoint */
   int e36, e45, e90;   /* where the edges occur in hcircle */
   int t36, t45, t90;   /* where the neighbors occur in triang */
} triangle;
```

8. An auxiliary hash table keeps track of the triangles we've seen.

(I've imposed the restriction $3 * maxn < 1024$ simply because I want to pack the values (*v36*, *v45*, *v90*) into a single word on my old 32-bit computer.)

⟨ Subroutines 3 ⟩ +≡

```
int savetriangle(int v36, int v45, int v90)
{
   register unsigned int w = (((v36 << 10) + v45) << 10) + v90;
   register int h = w % hprime;
   while (triple[h]) {
      if (triple[h] == w) goto found;
      h = (h ? h : hprime) - 1;
   }
   triple[h] = w, tripnum[h] = tptr;
   triang[tptr].v36 = v36, triang[tptr].v45 = v45, triang[tptr].v90 = v90;
   tptr++;
found: return tripnum[h];
}
```

9. ⟨ Global variables 6 ⟩ +≡

```
int triple[hprime];   /* the vertex triples that we've seen */
int tripnum[hprime];   /* their serial numbers */
int tptr;   /* the number of triangles we've seen */
triangle triang[maxn + 3];   /* their details */
```

10. Getting started. To prime the pump, I need one 36°-45°-90° triangle to begin the process. The simplest one that I could think of has vertices $e^{i\theta}$, i/r, and i in the complex plane, where

$$r = \sqrt{\phi + \sqrt{\phi}}, \qquad \cos\theta = 1/\sqrt{2\phi},$$

and $\phi = (1 + \sqrt{5}\,)/2$ is the golden ratio.

#define *makepoint*(v, xx, yy) $z.x = xx, z.y = yy, v = savepoint(z)$
#define *makecircle*(v, cc, rr) $l.c = cc, l.r = rr, v = savecircle(l)$

⟨ Global variables 6 ⟩ +≡
  point z;   /* staging area for *makepoint* */
  circle l;   /* staging area for *makecircle* */

11. ⟨ Set up triangle 0 11 ⟩ ≡
  $phi = (1.0 + sqrt(5.0))/2.0;$
  $makepoint(a, sqrt(0.5/phi), sqrt(1.0 - 0.5/phi));$
  $makepoint(b, 0.0, 1.0/sqrt(phi + sqrt(phi)));$
  $makepoint(c, 0.0, 1.0);$
  ⟨ Compute the edges of triangle 0 12 ⟩;
  savetriangle(a, b, c);   /* now *tptr* will equal 1 */
  printf("triangle␣0␣=␣(%d,%d,%d),", a, b, c);
  printf("␣edges␣(*,%d,%d)\n", *triang*[0].*e45*, *triang*[0].*e90*);

This code is used in section 1.

12. Edge *e36* of the starting triangle is a vertical line from i/r to i. This is the only vertical line that we will need in the present program. (In fact, I'll explain shortly that the computations will be limited to the subtiling that appears in an annulus. Then one can prove a strict upper bound on the size of the c values that occur, even if the computation proceeds indefinitely.)

It turns out easiest to handle the exceptional vertical case by setting $e36 = e45$. I do admit however that this is a sneaky trick, hard to justify on moral grounds.

⟨ Compute the edges of triangle 0 12 ⟩ ≡
  makecircle(*triang*[0].*e45*, 0.0, 1.0);
  triang[0].*e36* = *triang*[0].*e45*;
  makecircle(*triang*[0].*e90*, *hpoint*[b].y, *sqrt*(2.0) ∗ *hpoint*[b].y);

This code is used in section 11.

13. The algorithm. One more important thing needs to be mentioned, before we put all the pieces together: I don't actually want to compute the entire tiling. I only want to compute it in the quarter-annulus that consists of the points between circles $|z| = 1$ and $|z| = 1/r$, in the upper right quadrant of the complex plane.

The reason is that the tiling between this annulus and the next-smaller one, $|z| = 1/r^2$, is just the reflection of the first tiling about the line $|z| = 1/r$. Then in the next annulus, we shrink the outer-annulus tiling by a factor of $1/r^2$, and so on.

Indeed, this restriction of the tiling accounts for my claims that triangle 0 is the only triangle with a vertical edge.

To implement the restriction, we simply refrain from computing the neighbor at any edge whose center c is zero. (And that is why the sneaky trick mentioned on the previous page actually works.)

```
⟨ Compute the neighbors of triangle k 13 ⟩ ≡
  {
    if (hcircle[triang[k].e36].c) ⟨ Compute t36 14 ⟩;
    if (hcircle[triang[k].e45].c) ⟨ Compute t45 15 ⟩;
    if (hcircle[triang[k].e90].c) ⟨ Compute t90 16 ⟩;
    printf("Triangle␣%d␣neighbors:", k);
    if (hcircle[triang[k].e36].c) printf("␣t36=%d", triang[k].t36);
    if (hcircle[triang[k].e45].c) printf("␣t45=%d", triang[k].t45);
    if (hcircle[triang[k].e90].c) printf("␣t90=%d", triang[k].t90);
    printf("\n");
  }
```

This code is used in section 1.

14.
```
⟨ Compute t36 14 ⟩ ≡
  {
    j = savepoint(reflect(hpoint[triang[k].v36], hcircle[triang[k].e36]));
    t = tptr;
    triang[k].t36 = savetriangle(j, triang[k].v45, triang[k].v90);
    if (tptr > t) {   /* that triangle is new */
      triang[t].e36 = triang[k].e36;
      triang[t].e45 = savecircle(common(hpoint[triang[t].v36], hpoint[triang[t].v90]));
      triang[t].e90 = savecircle(common(hpoint[triang[t].v36], hpoint[triang[t].v45]));
      printf("triangle␣%d␣=␣(z%d,z%d,z%d),", t, triang[t].v36, triang[t].v45, triang[t].v90);
      printf("␣edges␣(%d,%d,%d)\n", triang[t].e36, triang[t].e45, triang[t].e90);
    }
  }
```

This code is used in section 13.

15. ⟨ Compute *t45* 15 ⟩ ≡

```
{
  j = savepoint(reflect(hpoint[triang[k].v45], hcircle[triang[k].e45]));
  t = tptr;
  triang[k].t45 = savetriangle(triang[k].v36, j, triang[k].v90);
  if (tptr > t) {    /* that triangle is new */
    triang[t].e45 = triang[k].e45;
    triang[t].e36 = savecircle(common(hpoint[triang[t].v45], hpoint[triang[t].v90]));
    triang[t].e90 = savecircle(common(hpoint[triang[t].v36], hpoint[triang[t].v45]));
    printf("triangle␣%d␣=␣(z%d,z%d,z%d),", t, triang[t].v36, triang[t].v45, triang[t].v90);
    printf("␣edges␣(%d,%d,%d)\n", triang[t].e36, triang[t].e45, triang[t].e90);
  }
}
```

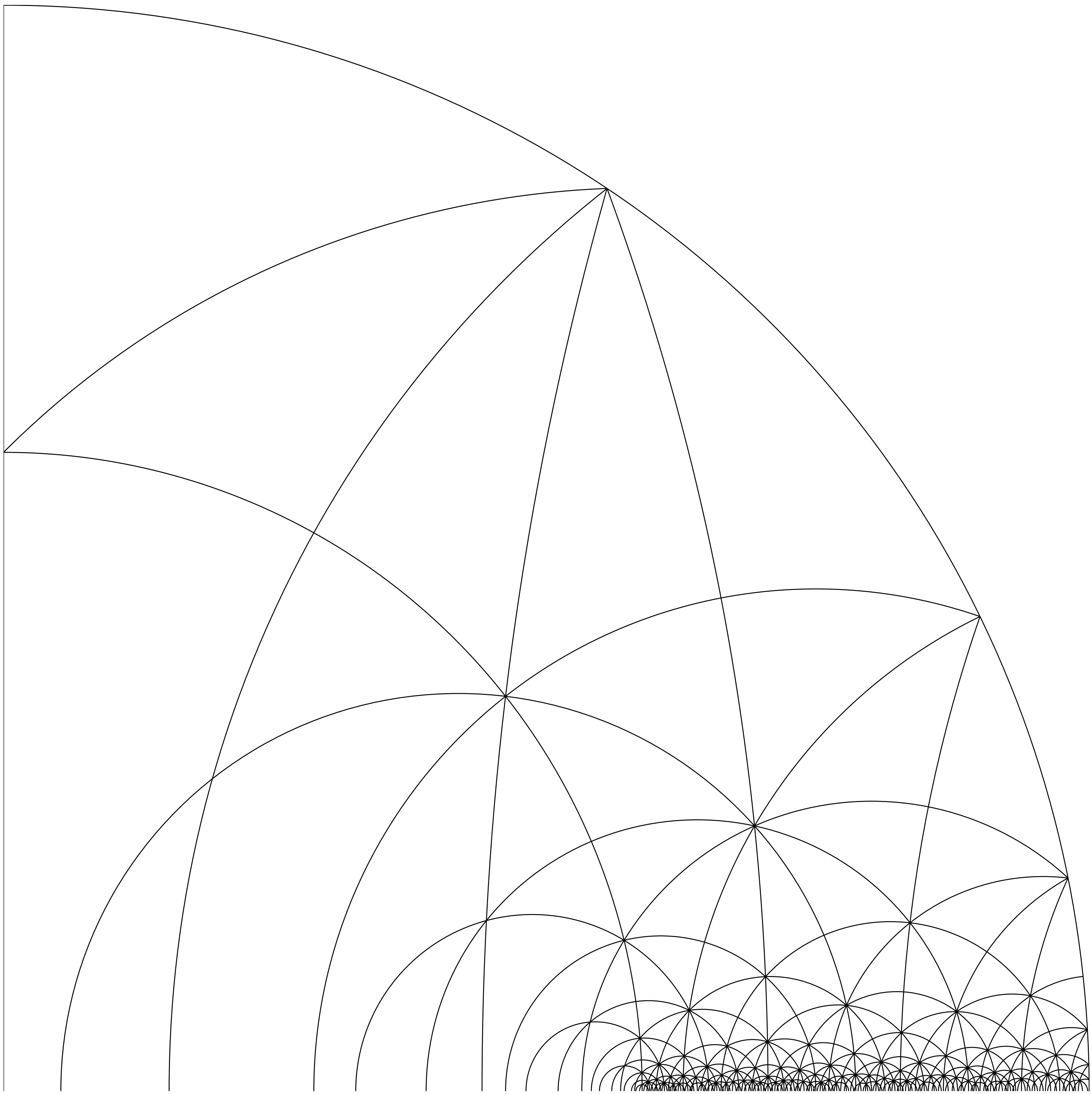
This code is used in section 13.

16. ⟨ Compute *t90* 16 ⟩ ≡

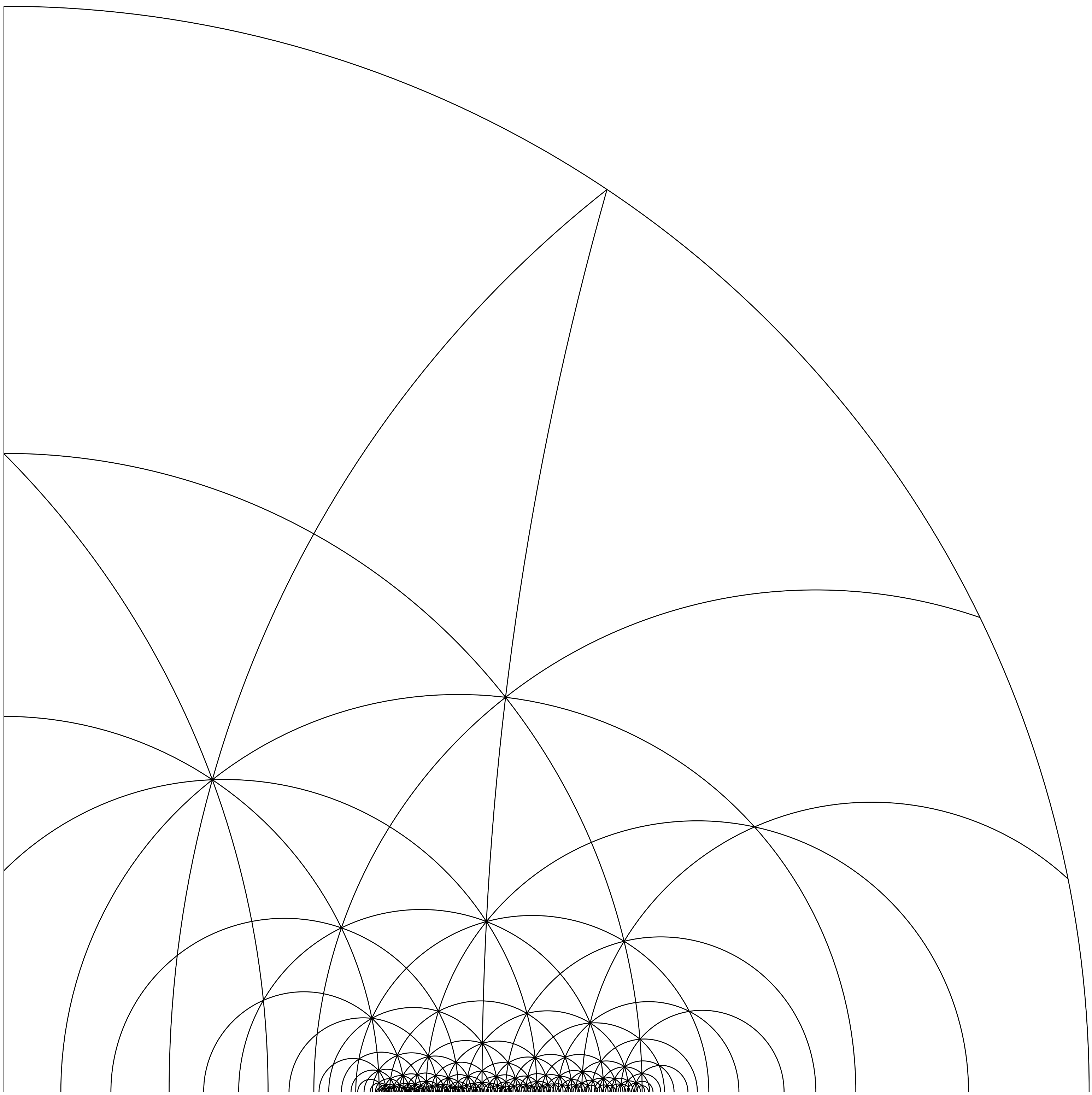
```
{
  j = savepoint(reflect(hpoint[triang[k].v90], hcircle[triang[k].e90]));
  t = tptr;
  triang[k].t90 = savetriangle(triang[k].v36, triang[k].v45, j);
  if (tptr > t) {    /* that triangle is new */
    triang[t].e90 = triang[k].e90;
    triang[t].e36 = savecircle(common(hpoint[triang[t].v45], hpoint[triang[t].v90]));
    triang[t].e45 = savecircle(common(hpoint[triang[t].v36], hpoint[triang[t].v90]));
    printf("triangle␣%d␣=␣(z%d,z%d,z%d),", t, triang[t].v36, triang[t].v45, triang[t].v90);
    printf("␣edges␣(%d,%d,%d)\n", triang[t].e36, triang[t].e45, triang[t].e90);
  }
}
```

This code is used in section 13.

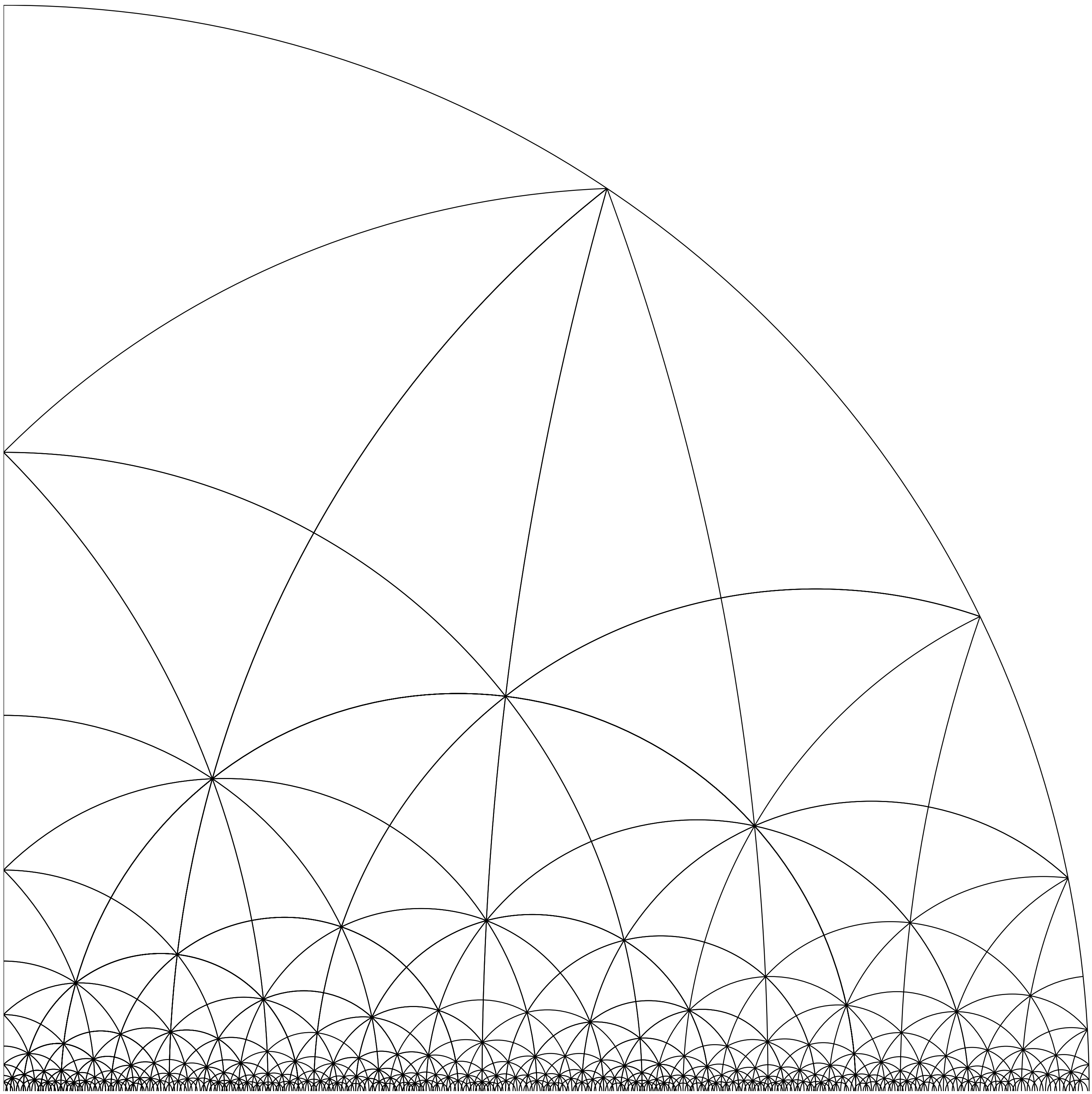
17. Output. Here's what we get when the circles `l1`, `l2`, ... are plotted:



18. Dual output. And here's what happens when those circles are reflected with respect to $|z| = 1/r$:



19. The overall tiling. Finally, the right half of the whole thing, omitting circles of radius < 0.007:



20. Index.

HYPERBOLIC