December 20, 2023 at 16:55
Changed sections for computing the crossing number.

1 The main program

Each PSLA for n lines has a unique parent with $n-1$ lines. This defines a tree structure on the PSLAs. The principle of the enumeration algorithm is a depth-first traversal of this tree.

Change: We are keeping statistics for several independent characteristics, one of which (the *crossing_number*) can rise to high values (see `MAX_CROSSINGS` in Section 3). Therefore, we reduce the maximum number `MAXN` of pseudolines from 15 to what we really need.

```
6  #define MAXN 12  /* The maximum number of pseudolines for which the program will work. */
   ⟨ Include standard libaries 10 ⟩
   ⟨ Types and data structures 4 ⟩
   ⟨ Global variables 5 ⟩
   ⟨ Subroutines 27 ⟩
   ⟨ Core subroutines for recursive generation 15 ⟩
   int main(int argc, char *argv[])
   {
     ⟨ Parse the command line 13 ⟩;
#if readdatabase  /* reading from the database */
     ⟨ Read all point sets of size n_max + 1 from the database and process them 78 ⟩
     return 0;
#endif
#if enumAOT
     ⟨ Initialize statistics and open reporting file 52 ⟩;
     ⟨ Start the generation 16 ⟩;
     ⟨ Report statistics 54 ⟩;
#endif
     return 0;
   }
```

2 Statistics

Characteristics:

- number h of hull points.
- period p of rotational symmetry on the hull. (The order of the rotation group is h/p.)
- mirror symmetry, with or without fixed vertex on the hull (3 possibilities).

In addition, we keep

- the number of halving-lines, *num_halving_lines*.
- the crossing number, *crossing_number*.

U_PSLAcount counts OAOT of point sets with a marked point on the convex hull, but no specified traversal direction. (U stands for unoriented.) Equivalently, it counts the x-monotone PSLAs when a PSLA is identified with its left-right mirror.

PSLAcount counts OAOT of point sets with a marked point on the convex hull and a specified traversal direction. Equivalently, it counts the x-monotone PSLAs, see http://oeis.org/A006245. This gives always the correct number, even if the program does not visit all these PSLAs due to the pre-screening.

```
51 #define NO_MIRROR 0
   #define MIRROR_WITH_FIXED_VERTEX 1
   #define MIRROR_WITHOUT_FIXED_VERTEX 2
   ⟨ Global variables 5 ⟩ +≡
     long long unsigned countPSLA[MAXN + 2], countO[MAXN + 2], countU[MAXN + 2];
     long long unsigned PSLAcount[MAXN + 2];  /* A006245, Number of primitive sorting networks on n
         elements; also number of rhombic tilings of 2n-gon. */
       /* 1, 1, 2, 8, 62, 908, 24698, 1232944, 112018190, 18410581880, 5449192389984 ... until n = 16. */
```

```
long long unsigned U_PSLAcount[MAXN + 2];
long long unsigned classcount[MAXN+2][MAXN+2][MAXN+2][3][MAX_HALVING_LINES+1][MAX_CROSSINGS+1];
    /* This is a huge array: If it were full it would take about 2 GBytes of main memory. But it is very
      sparsely filled. With the gcc compiler, the system reports less than 30 MBytes of used storage, and
      often only 2 or 3 MBytes. This might just depend on the available memory for swapping. With clang
      compiler, the observed memory usage was also less than 3 MBytes. Only in ⟨ Report statistics 54 ⟩ is the
      full array scanned once. */
int num_halving_lines;      /* global variable; (This is not the cleanest way to do it.) */
long long unsigned numComparisons ← 0, numTests ← 0;      /* profiling */
```

53 ¶ ⟨ Gather statistics about the AOT, collect output 53 ⟩ ≡

```
int hulledges[MAXN + 1];
int hullsize ← upper_hull_PSLA(n, hulledges);      /* Determine the extreme points: */
int rotation_period;
boolean has_fixed_vertex;
boolean has_mirror_symmetry;
int n_points ← n + 1;      /* number of points of the AOT */
boolean lex_smallest ← is_lex_smallest_P_matrix(n, hulledges, hullsize, &rotation_period,
    &has_mirror_symmetry, &has_fixed_vertex);
if (lex_smallest) {
  countU[n_points]++;      /* We count to contribution from this AOT to the various counters countO,
      PSLAcount, U_PSLAcount according to the symmetry information. */
  if (has_mirror_symmetry) {
    countO[n_points]++;
    PSLAcount[n] += rotation_period;
    if (has_fixed_vertex) U_PSLAcount[n] += rotation_period/2 + 1;
        /* works for even and odd rotation_period */
    else U_PSLAcount[n] += rotation_period/2;
  }
  else {
    countO[n_points] += 2;
    PSLAcount[n] += 2 * rotation_period;
    U_PSLAcount[n] += rotation_period;
  }
  int crossing_number ← count_crossings(n);
  assert(num_halving_lines ≤ MAX_HALVING_LINES);
  classcount[n_points][hullsize][rotation_period][¬has_mirror_symmetry ?
      NO_MIRROR : has_fixed_vertex ? MIRROR_WITH_FIXED_VERTEX :
      MIRROR_WITHOUT_FIXED_VERTEX][num_halving_lines][crossing_number]++;
}
#if 0      /* debugging */
  printf("found n=%d. %Ld ", n_points, countO[n_points]);
  print_small(S, n_points);
#endif
```

This code is used in chunk 18.

¶ The statistics gathered in the *classcount* array are written to a *reportfile* so that a subsequent program can conveniently read and process it.

55 ⟨ Report statistics 54 ⟩ +≡

```
if (strlen(fname)) {
  fprintf(reportfile, "# N_max=%d/%d", n_max, n_max + 1);
  if (parts ≠ 1) fprintf(reportfile, ", split-level=%d, part %d of %d", split_level, part, parts);
  fprintf(reportfile, "\n#x N hull period mirror-type halving-lines crossing-number NUM\n");
  for_int_from_to (n, 0, n_max + 1) {
    char c ← 'T';      /* total count */
```

```
    if (parts ≠ 1 ∧ n > split_level + 1) c ← 'P';      /* partial count */
    for_int_from_to (k, 0, n)
      for_int_from_to (p, 0, n)
        for_int_from_to (t, 0, 2)
          for_int_from_to (h, 0, MAX_HALVING_LINES)
            for_int_from_to (cr, 0, MAX_CROSSINGS)
              if (classcount[n][k][p][t][h][cr]) fprintf(reportfile, "%c␣%d␣%d␣%d␣%d␣%d␣%d␣␣%Ld\n", c, n, k,
                    p, t, h, cr, classcount[n][k][p][t][h][cr]);
  }
  if (parts ≡ 1) fprintf(reportfile, "EOF\n");
  else fprintf(reportfile, "EOF␣%d,␣part␣%d␣of␣%d\n", split_level, part, parts);
  fclose(reportfile);
  printf("Results␣have␣been␣written␣to␣file␣%s.\n", fname);
}
```

3 Extension: Compute crossing-number for each AOT

What range of values should we anticipate for the number of halving-lines? By https://oeis.org/A076523, a set with $n = 12$ points has at most 18 halving-lines. According to S. Bereg and M. Haghpanah, New algorithms and bounds for halving pseudolines, Discrete Applied Mathematics 319 (2022) 194–206, https://doi.org/10.1016/j.dam.2021.05.029, Table 1 on p. 196, the number of halving lines-with for odd numbers n of points are nearly 70 % higher than for the adjacent even values. I could not find the bounds for small odd n in the literature or on the internet. After running the program once with a larger safety margin, it was found that a set with $n = 11$ points has at most 24 halving-lines. (The program checks if the bound is not violated.) With $n = 13$ points and a bound 40 on the halving-lines we should be on the safe side.

```
59 #define MAX_HALVING_LINES 40      /* For n = 12 point, MAX_HALVING_LINES ← 24 would be sufficient. */
   #define MAX_CROSSINGS (MAXN + 1) * MAXN * (MAXN − 1) * (MAXN − 2)/24
            /* crossing-number goes up to (n choose 4) for n points */
```

¶ Find the number of crossings on each level of the wiring diagram.

This algorithm is like the program for drawing the wiring diagram, except that it does not draw anything. The wires run on n *tracks*, which are labeled from 1 to n from top to bottom.

The program computes the number of crossings *num_crossings_on_level*[p] at each level p including the crossings with line 0. A crossing at level p is a crossing between consecutive tracks p and $p + 1$, $1 \le p \le n - 1$.

From this information, the crossing number of the complete graph K_n when it is drawn on this point set can be computed by easy explicit formula, see Lovász, Vesztergombi, Wagner, and Welzl, *Convex quadrilaterals and k-sets*, DOI:10.1090/conm/342/06138.

In addition, the program stores the number of halving-lines in the global variable *num_halving_lines*.

```
60 ⟨ Subroutines 27 ⟩ +≡
  int count_crossings(int n)
  {
    int line_at[MAXN + 1];
    int next_crossing[MAXN + 1];
    int num_crossings_on_level[MAXN + 1];
    for_int_from_to (i, 0, n) {      /* Initialize: */
      line_at[i] ← i;      /* which line is on the i-th track, 1 ≤ i ≤ n. Tracks are numbered p = 1 . . . n from
          top to bottom. */
      next_crossing[i] ← SUCC(i, 0);
          /* current crossing on each line; the first crossing with line 0 "at ∞" is counted separately: */
      num_crossings_on_level[i] ← 1;      /* The initial value counts the crossing with line 0 */
    }
    next_crossing[0] ← 1;      /* sentinel values to ensure that the loop breaks at p = 0 */
    int p ← n;
    while (true) {      /* Main loop */
          /* Invariant throughout: The lines at levels p + 1, . . . , n want to cross upwards (or are finished.) */
      while (next_crossing[p] < line_at[p]) p−−;      /* while p wants to cross upwards */
      if (p ≡ 0) break;
```

```
    do {    /* The line i ≡ line_at[p] on track p wants to cross down and the line j ≡ line_at[p+1] on
               track p+1 wants to cross up. (In this case, we must actually have next_crossing[i] ≡ j and
               next_crossing[j] ≡ i.) */
      num_crossings_on_level[p]++;
      /* update the data structures to CARRY OUT the crossing p ↔ p+1: */
      int i ← line_at[p];
      int j ← line_at[p+1];
      int temp ← SUCC(i, next_crossing[p]);
      next_crossing[p] ← SUCC(j, next_crossing[p+1]);
      next_crossing[p+1] ← temp;
      line_at[p] ← j;
      line_at[p+1] ← i;
      p++;
    } while (next_crossing[p] > line_at[p]);    /* while p wants to cross downwards */
    p −= 2;    /* We need not check p−1, because this would be an immediate back-crossing at the same
          level. */
  }
  /* Number of halving-lines is stored in the global variable num_halving_lines: */
  if (n % 2)    /* n odd, number of points even: */
    num_halving_lines ← num_crossings_on_level[(n+1)/2];
  else    /* n even, number of points odd: */
    num_halving_lines ← num_crossings_on_level[n/2] + num_crossings_on_level[n/2+1];
  /* compute crossing number according to the formula: */
  int crossing_formula ← −(n+1) * n * (n−1)/2;
  for_int_from_to (p, 1, n) crossing_formula += num_crossings_on_level[p] * (n+1−2*p) * (n+1−2*p);
  return crossing_formula/4;
}
```