December 11, 2023 at 06:49
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.

3
```
#define MAXN 11   /* The maximum number of pseudolines for which the program will work. */
  ⟨ Include standard libaries 6 ⟩
  ⟨ Types and data structures 5 ⟩
  ⟨ Global variables 8 ⟩
  ⟨ Subroutines 25 ⟩
  ⟨ Core subroutine for recursive generation 14 ⟩
  int main(int argc, char *argv[])
  {
    ⟨ Parse the command line 9 ⟩;
#if readdatabase   /* reading from the database */
    ⟨ Read all point sets of size n_max + 1 from the database and process them 73 ⟩
    return 0;
#endif
#if enumAOT
    ⟨ Initialize statistics and open reporting file 50 ⟩;
    ⟨ Start the generation 15 ⟩;
    ⟨ Report statistics 52 ⟩;
#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*.

PSLAcount gives OAOT of point sets with a marked point on the convex hull. http://oeis.org/A006245 (see below) is the same sequence with n shifted by 0.

49
```
#define NO_MIRROR 0
#define MIRROR_WITH_FIXED_VERTEX 1
#define MIRROR_WITHOUT_FIXED_VERTEX 2
⟨ Global variables 8 ⟩ +≡
  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. Also the number of oriented matroids of rank 3 on
        n(?) elements. */
     /* 1, 1, 2, 8, 62, 908, 24698, 1232944, 112018190, 18410581880, 5449192389984 ... until n = 15. */
  long long unsigned xPSLAcount[MAXN + 2];
  long long unsigned classcount[MAXN + 2][MAXN + 2][MAXN
        + 2][3][MAX_HALVING_LINES + 1][MAX_CROSSINGS + 1];
  int num_halving_lines;   /* global variable; this is not clean */
  long long unsigned numComparisons ← 0, numTests ← 0;   /* profiling */
```

51 ¶ ⟨ Gather statistics about the AOT, collect output 51 ⟩ ≡ /* Determine the extreme points: */

```
int hulledges[MAXN +1];
int hullsize ← upper_hull_PSLA(n, hulledges);
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]++;
  if (has_mirror_symmetry) {
    countO[n_points]++;
    PSLAcount[n] += rotation_period;
    if (has_fixed_vertex) xPSLAcount[n] += rotation_period/2 + 1;
        /* works for even and odd rotation_period */
    else xPSLAcount[n] += rotation_period/2;
  }
  else {
    countO[n_points] += 2;
    PSLAcount[n] += 2 * rotation_period;
    xPSLAcount[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 14.

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

53 ⟨ Report statistics 52 ⟩ +≡

```
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_max + 1)
      for_int_from_to (p, 0, n_max + 1)
        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 (the maximum that the program is set up to deal with), 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. 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.)

56
```
#define MAX_HALVING_LINES 24
#define MAX_CROSSINGS (MAXN +1) ∗MAXN ∗(MAXN −1)∗(MAXN −2) / 24
        /∗ crossing-number goes up to (n choose 4) for n points ∗/
```

¶ How to check for a crossing.

This algorithm is like the program for drawing the wiring diagram, except that it does not draw anything.

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 crossings between consecutive tracks p and $p+1$, $0 \le p \le n-1$.

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

57
```
#define CHECK_CROSSING(p)
    {
      {
        int i ← line_at[p];
        int j ← line_at[p + 1];
        if (i < j ∧ next_crossing[i] > i ∧ next_crossing[j] < j ∧ next_crossing[j] ≠ 0)
             /∗ Line i wants to cross down and line j wants to cross up. ∗/
             /∗ (In this case, we must actually have next_crossing[i] ≡ j and next_crossing[j] ≡ i.) ∗/
          crossings[num_crossings++] ← p;
             /∗ The value p indicates a crossing between tracks p and p + 1. ∗/
      }
    }
⟨ Subroutines 25 ⟩ +≡
  int count_crossings(int n)
  {
    int next_crossing[MAXN +1];
    int line_at[MAXN +1];
    int num_crossings_on_level[MAXN −1];
    int crossings[MAXN];      /∗ stack ∗/
    int num_crossings ← 0;    /∗ Initialize ∗/
    for_int_from_to (i, 1, n) {
      next_crossing[i] ← SUCC(i, 0);
         /∗ current crossing on each line; The first crossing with line 0 “at ∞” is not considered. ∗/
      line_at[i − 1] ← i;     /∗ which line is on the p-th track, 0 ≤ p < n. tracks are numbered p = 0 . . . n − 1
          from top to bottom. ∗/
    }
    for_int_from_to (p, 0, n − 1) num_crossings_on_level[p] ← 1;     /∗ counting the crossing with line 0 ∗/
         /∗ maintain a stack crossings of available crossings. p ∈ crossings means that tracks p and p + 1 are
          ready to cross ∗/
    for_int_from_to (p, 0, n − 2) CHECK_CROSSING(p)
    while (num_crossings) {     /∗ Main loop ∗/
      int p ← crossings[−−num_crossings];
      num_crossings_on_level[p]++;
      xE     /∗ update the data structures to CARRY OUT the crossing ∗/
          int i ← line_at[p];
          int j ← line_at[p + 1];
      next_crossing[i] ← SUCC(i, next_crossing[i]);
```

```
    next_crossing[j] ← SUCC(j, next_crossing[j]);
    line_at[p] ← j;
    line_at[p + 1] ← i;      /* Look for new crossings: */
    if (p > 0) CHECK_CROSSING(p − 1)
    if (p < n − 1) CHECK_CROSSING(p + 1)
  }      /* compute result */
  int crossing_formula ← −(n + 1) ∗ n ∗ (n − 1)/2;
  for_int_from_to (p, 0, n − 1)
    crossing_formula += num_crossings_on_level[p] ∗ (n − 1 − 2 ∗ p) ∗ (n − 1 − 2 ∗ p);
        /* global variable num_halving_lines is set. */
  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];
  return crossing_formula/4;
}
```