December 3, 2023 at 13:30

NumPSLA, a program for enumerating pseudoline arrangements and abstract order types

The purpose of this program is to enumerate ORIENTED abstract order types. (sometimes also called generalized configuration or a pseudoconfiguration)

The program enumerates the objects without repetition and with negligible storage.

We consider nondegenerate cases only: no three points on a line.

We abbreviate *oriented abstract order type* by OAOT.

(For statistics, can still report only one orientation of two mirror types)

0.1 Pseudoline arrangements and abstract order types

We consider everything *oriented*, i.e., the mirror object can be isomorphic or not. Also, only *simple*: No three curves through a point.

A *projective* pseudoline arrangement (PSLA) is a family of centrally symmetric closed Jordan curves on the sphere such that any two curves intersect in two points, and they intersect transversally at these points.

An *affine* PSLA is a family of Jordan curves in the plane that go to infinity at both ends and that intersect pairwise exactly once, and they intersect transversally at these points.

An x-*monotone* PSLA (*wiring diagram*, primitive sorting network) is an affine PSLA with x-monotone curves.

We consider two objects as equivalent under deformation by orientation-preserving isotopies of the sphere, or the plane, respectively. (An x-monotone PSLA must remain x-monotone throughout the deformation.)

A *marked* OAOT is an OAOT with a marked point on the convex hull.

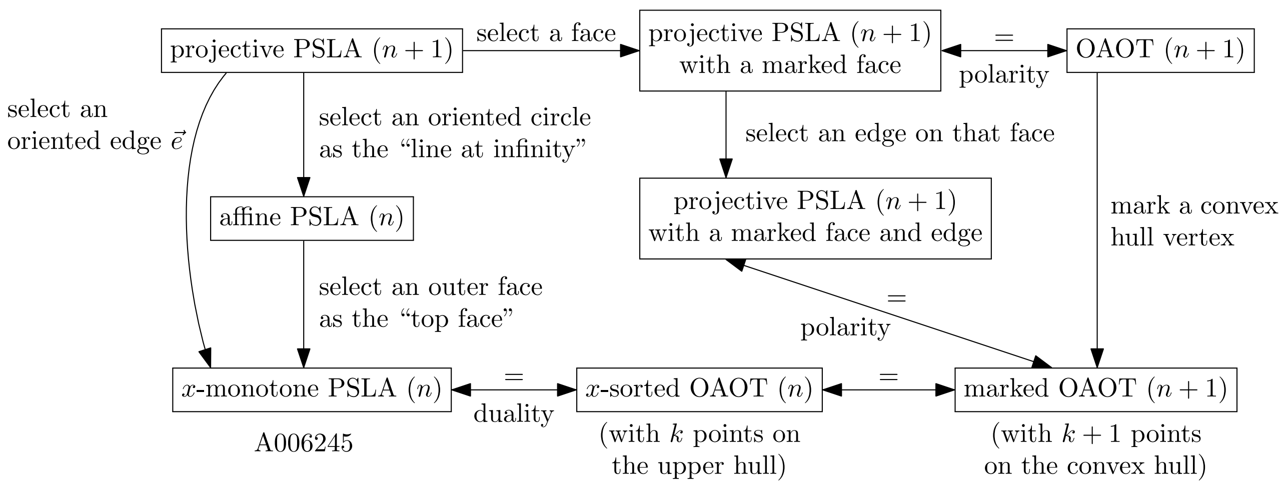


Figure 1: Relations between different concepts. There are different paths from the top left to the bottom right, which apply specialization or geometric reinterpretation in different order.

See Aichholzer and Krasser, Abstract order type extension and new results on the rectilinear crossing number. Comput. Geom. 36 (2007), 2–15, Table 1.

	[A006247]	[A063666]				[A006245]
n	#AOT	#realizable AOT	Δ	relative Δ	#mirror-symmetric AOT	#x-monotonePSLA
3	1	1	0	0	1	2
4	2	2	0	0	2	8
5	3	3	0	0	3	62
6	16	16	0	0	12	908
7	135	135	0	0	28	24,698
8	3,315	3,315	0	0	225	1,232,944
9	158,830	158,817	13	0,01 %	825	112,018,190
10	14,320,182	14,309,547	10,635	0,07 %	13,103	18,410,581,880
11	2,343,203,071	2,334,512,907	8,690,164	0,37 %	76,188	5,449,192,389,984
12	691,470,685,682					2,894,710,651,370,536

The last column counts the objects that the program actually enumerates one by one (almost, because we try to apply shortcuts). These numbers are known up to $n = 15$. For example, to get the 158,830 AOTs with 9 points, we go through all 1,232,944 xPSLAs with 8 pseudolines.

$$\#\text{OAOT} = 2 \times \#\text{AOT} - \#\text{mirror-symmetric AOT} \qquad \text{[A006246]}$$

#AOT equals the number of simple projective pseudoline arrangements with a marked cell.

According to OEIS, three different sequences give "the number of primitive sorting networks on n elements": A006245, A006246, A006248.

1 The main program

2 **#define** `MAXN` 11 /* The maximum number of pseudolines for which the program will work. */

⟨ Include standard libaries 5 ⟩
⟨ Types and data structures 4 ⟩
⟨ Global variables 7 ⟩
⟨ Subroutines 23 ⟩
⟨ Core subroutine for recursive generation 13 ⟩
int *main*(**int** *argc*, **char** **argv*[ ])
{
⟨ Parse the command line 8 ⟩;
#if *readdatabase* /* reading from the database */
⟨ Read all point sets of size *n_max* + 1 from the database and process them 69 ⟩
return 0;
#endif
#if *enumAOT*
⟨ Initialize statistics and open reporting file 50 ⟩;
⟨ Start the generation 14 ⟩;
⟨ Report statistics 52 ⟩;
#endif
return 0;
}

1.1 Preprocessor switches

The program has the enumeration procedure at its core, but it can be configured to perfom different tasks, by setting preprocessor switches at compile-time.

We assume that the program will anyway be modified and extended for specific counting or enumeration tasks, and it makes sense to set these options at compile-time.

(Other options, which are less permanent, can be set by command-line switches.)

3 **#define** *enumAOT* 1 /* purpose is enumeration of AOTs */
/* Other purposes might be enumeration of PSLAs */
#define *readdatabase* 0 /* version for reading point sets of the order-type database */
#define *generatelist* 0 /* List all PSLAs plus their IDs, as preparation for generating exclude-files of nonrealizable AOTs, requires *enumAOT* $\equiv$ 1. */
#define *profile* 1 /* gather statistics and profiling information */

¶ Type definitions.

4 ⟨ Types and data structures 4 ⟩ ≡
typedef enum { *false*, *true* } **boolean**;

See also chunks 10, 57, and 65*
This code is used in chunk 2*.

¶ Standard libraries

5 ⟨ Include standard libaries 5 ⟩ ≡

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```

See also chunk 67*.
This code is used in chunk 2*.

1.2 Auxiliary macros for for-loops

6
```
#define for_int_from_to(x, first, last) for (int x ← first; x ≤ last; x++)
  format for_int_from_to for
#define print_array(a, length, begin, separator, end)
        {   /* for reporting and debugging */
          printf(begin);
          for_int_from_to (j, 0, length − 1) {
            if (j > 0) printf(separator);
            printf("%d", a[j]);
          }
          printf(end);
        }   /* for gcc, compile with -Wno-format-zero-length to suppress warnings */
```

1.3 Command-line arguments

7
```
#define PRINT_INSTRUCTIONS
        printf("Usage:␣%s␣n␣[-exclude␣excludefile]␣[splitlevel␣parts␣part]␣[fileprefix]\n",
            argv[0]);
```

⟨ Global variables 7 ⟩ ≡

```
  small_int n_max, split_level;
  unsigned int parts ← 1000, part ← 0;
  char *fileprefix ← "reportPSLA";   /* default name */
  char *exclude_file_name ← 0;
  char fname[200] ← "";
  FILE *reportfile ← 0;
```

See also chunks 11, 17, 27, 32, 36, 45, 49*, and 63*
This code is used in chunk 2*.

8 ¶⟨ Parse the command line 8 ⟩ ≡

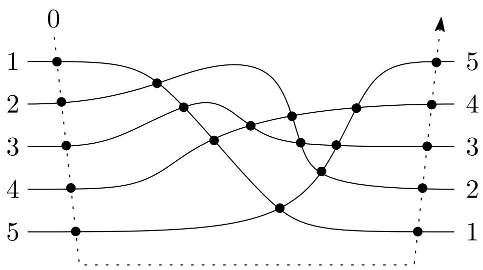
```
  if (argc < 2) n_max ← 7;
  else {
    if (argv[1][0] ≡ '-') {   /* first argument "--help" gives help message. */
      PRINT_INSTRUCTIONS;
      exit(0);
    }
    n_max ← atoi(argv[1]);
  }
  printf("Enumeration␣up␣to␣n␣=␣%d␣pseudolines,␣%d␣points.\n", n_max, n_max + 1);
  if (n_max > MAXN) {
    printf("The␣largest␣allowed␣value␣is␣%d.␣Aborting.\n", MAXN);
    exit(1);
  }
  int argshift ← 0;
```

```
if (argc ≥ 3) {
  if (strcmp(argv[2], "-exclude") ≡ 0) {
    if (argc ≥ 4) {
      exclude_file_name ← argv[3];
      argshift ← 2;
      printf("Excluding␣entries␣from␣file␣%s.\n", exclude_file_name);
      ⟨Open the exclude-file and read first line 19⟩
    }
    else {
      PRINT_INSTRUCTIONS;
      exit(1);
    }
  }
}
if (argc ≥ 3 + argshift) {
  split_level ← atoi(argv[2 + argshift]);
  if (split_level ≡ 0) {
    if (argv[2 + argshift][0] ≠ '-') fileprefix ← argv[2 + argshift];
    snprintf(fname, sizeof (fname) − 1, "%s-%d.txt", fileprefix, n_max);
    parts ← 1;
  }
  else {
    if (argc ≥ 4 + argshift) parts ← atoi(argv[3 + argshift]);
    if (argc ≥ 5 + argshift) part ← atoi(argv[4 + argshift]);
    part ← part % parts;
    if (argc ≥ 6 + argshift) fileprefix ← argv[5 + argshift];
    snprintf(fname, sizeof (fname) − 1, "%s-%d-S%d-part_%d_of_%d.txt", fileprefix, n_max, split_level,
        part, parts);
    printf("Partial␣enumeration:␣split␣at␣level␣n␣=␣%d.␣Part␣%d␣of␣%d.\n", split_level, part,
        parts);
  }
  printf("Results␣will␣be␣reported␣to␣file␣%s.\n", fname);
  fflush(stdout);
}
```

This code is used in chunk 2*.

2 Representations of pseudoline arrangements

Here is an x-monotone pseudoline arrangement with $n = 5$ pseudolines, together with a primitive graphic representation as produced by the program *print_wiring_diagram*:




```
1-1-1-1 4-4 5-5
       X   X
2 3-3 4 1 5 4-4
 X   X   X
3 2 4 3 5 1 3-3
   X   X   X
4-4 2 5 3-3 1 2
     X       X
5-5-5 2-2-2-2 1
```

Pseudoline 1 starts topmost and ends bottommost. On the right end, the order of all pseudolines is reversed. There is an imaginary pseudoline 0 of very negative slope that intersects all other pseudolines from top to bottom at the very left and again intersects all pseudolines from bottom to top at the very right.

2.1 The local sequences matrix and its inverse

Here is a representation as a two-dimensional array, indicating for each pseudoline i the sequence P_i of crossings with the other lines.

local sequences matrix

$P_0 = [1,2,3,4,5]$	$\bar{P}_0 = [\text{-},0,1,2,3,4]$	
$P_1 = [0,4,5,3,2]$	$\bar{P}_1 = [0,\text{-},4,3,1,2]$	$B_1 = [0,0,0,0,0]$
$P_2 = [0,3,4,5,1]$	$\bar{P}_2 = [0,4,\text{-},1,2,3]$	$B_2 = [0,0,0,0,1]$
$P_3 = [0,2,4,5,1]$	$\bar{P}_3 = [0,4,1,\text{-},2,3]$	$B_3 = [0,1,0,0,1]$
$P_4 = [0,2,3,1,5]$	$\bar{P}_4 = [0,3,1,2,\text{-},4]$	$B_4 = [0,1,1,1,0]$
$P_5 = [0,2,3,1,4]$	$\bar{P}_5 = [0,3,1,2,4,\text{-}]$	$B_5 = [0,1,1,1,1]$

The first row and the first column are determined. Each row has n elements. We also use the data structure for an inverse array $\bar{P}$, which is essentially the inverse permutation of the rows. The j-th element of $\bar{P}_i$ gives the position in P_i where the crossing with j occurs. The diagonal entries are irrelevant. The column indices in $\bar{P}$ range from 0 to n; therefore we define the rows to have maximum length `MAXN` + 1.

10 ⟨ Types and data structures 4 ⟩ +≡
typedef int PSLA [**MAXN** + 1] [**MAXN** +1];

2.2 Linked representation

For modifying and extending PSLAs, it is best to work with a linked representation.

Point (j,k) describes the crossing with line k along the line j. `SUCC`(j,k) and `PRED`(j,k) point to the next and previous crossing on line j. For (k,j) we get the corresponding information for the line k. In the example, we have `SUCC`$(2,3) = 5$ and accordingly `PRED`$(2,5) = 3$.

The infinite rays on line j are represented by the additional line 0: `SUCC`$(j,0)$ is the first (leftmost) crossing on line j, and `PRED`$(j,0)$ is the last crossing. The intersections on line 0 are cyclically ordered $1,\ldots,n$. Thus, `SUCC`$(0,i) \leftarrow i+1$ and `SUCC`$(0,n) = 1$.

The program works with a single linked-list representation, which is stored in the global arrays *succ* and *pred*. A single pair of these arrays is sufficient for the whole program.

11 **#define** `SUCC`(i,j) $succ[i][j]$ /∗ access macros ∗/
#define `PRED`(i,j) $pred[i][j]$
#define `LINK`$(j, k1, k2)$
{ /∗ make crossing with k_1 and k_2 adjacent on line j ∗/
`SUCC`$(j, k1) \leftarrow k2$;
`PRED`$(j, k2) \leftarrow k1$;
}

⟨ Global variables 7 ⟩ +≡
int *succ*[**MAXN** +1][**MAXN** +1];
int *pred*[**MAXN** +1][**MAXN** +1];

3 Recursive Enumeration

We extend an x-monotone pseudoline arrangement of $n-1$ lines $1,\ldots,n-1$, by threading an additional line n through it from the bottom face to the top face. The new line gets the largest slope of all lines.

Line 0 crosses the other lines in the order $1,2,\ldots,n$.

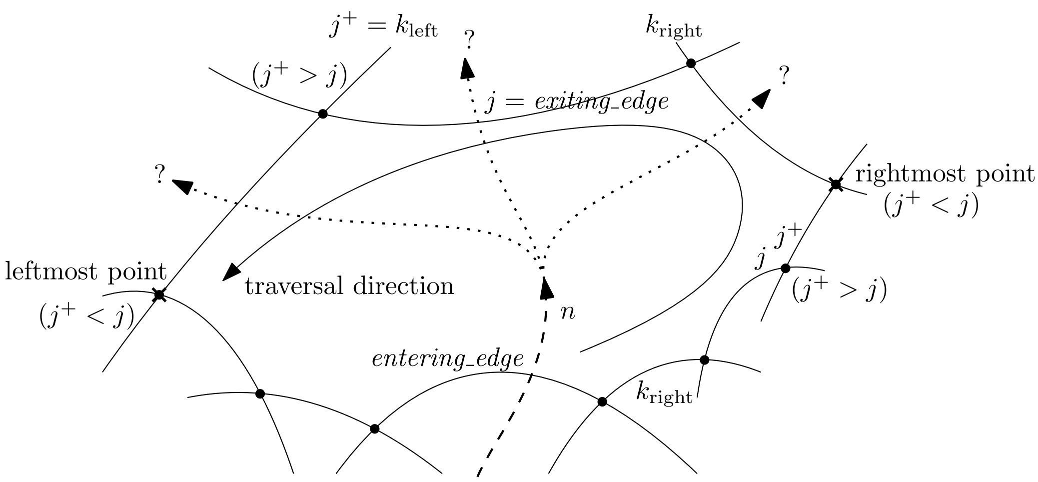


Figure 2: Threading line n through a face

13 ¶⟨ Core subroutine for recursive generation 13 ⟩ ≡

```
void recursive_generate_PSLA_start(int n);

void recursive_generate_PSLA(int entering_edge, int k_right, int n)
{   /* The new line enters a face F from the bottom. The edge through which it crosses is part of line
       entering_edge, and its endpoint is the crossing with k_right. */
  int j ← entering_edge;
  int j⁺ ← k_right;
  while (j⁺ > j) {   /* find right vertex of the current cell F */
    int j⁺_old ← j⁺;
    j⁺ ← SUCC(j⁺, j);
    j ← j⁺_old;
  }   /* the right vertex is the intersection of j and j⁺ */
  if (j⁺ ≡ 0) {   /* F is unbounded */
    if (j ≡ n − 1) {   /* F is the top face. */
      LINK(n, entering_edge, 0);   /* complete the insertion of line n */
      ⟨ Update counters 16 ⟩
      ⟨ Indicate Progress 15 ⟩;
      ⟨ Check for exclusion and set the flag is_excluded 18 ⟩
      if (is_excluded) return;
      ⟨ Gather statistics about the AOT, collect output 51 ⟩
      ⟨ Further processing of the AOT 53 ⟩
      if (n < n_max)
        if (n ≠ split_level ∨ countPSLA[n] % parts ≡ part) {
#if enumAOT   /* screening one level below */
          boolean hopeful ← true;
          if (n ≡ n_max − 1) {
            ⟨ Screen one level below level n_max 44 ⟩
          }
          if (hopeful)
#endif
          {
            localCountPSLA[n + 1] ← 0;   /* reset child counter */
            recursive_generate_PSLA_start(n + 1);   /* thread the next pseudoline */
          }
        }
      return;
    }
    else {   /* jump to the upper bounding ray of F */
      j⁺ ← j + 1;
      j ← 0;
    }
  }   /* Now the crossing j×j⁺ is the rightmost vertex of the face F. The edge j⁺ is on the upper side.
         If F is bounded, j is on the lower side; otherwise, j = 0. */
  do {   /* scan the upper edges of F from right to left and try them out. */
    k_right ← j;
    j ← j⁺;
    int k_left ← j⁺ ← PRED(j, k_right);   /* j is the exiting edge */
    LINK(j, k_left, n);   /* insert the crossing to prepare for the recursive call */
    LINK(j, n, k_right);
    LINK(n, entering_edge, j);
    recursive_generate_PSLA(j, k_right, n);   /* enter the recursion */
    LINK(j, k_left, k_right);   /* undo the changes */
  } while (j⁺ > j);   /* terminate at left endpoint of the face F or at unbounded ray (j⁺=0) */
  return;
}

void recursive_generate_PSLA_start(int n)
{
  LINK(0, n − 1, n);   /* insert line n on line 0 */
  LINK(0, n, 1);
```

```
    recursive_generate_PSLA(0, 0, n);      /* enter the recursion. */
      /* There us a little trick: With these parameters 0, 0, the procedure recursive_generate_PSLA will skip
         the first loop and will then correctly scan the edges of the bottom face F from right to left. */
    LINK(0, n − 1, 1);      /* undo the insertion of line n */
  }
```

This code is used in chunk 2*.

¶ Start with 2 pseudolines.

14 ⟨ Start the generation 14 ⟩ ≡

```
  LINK(1, 0, 2);
  LINK(1, 2, 0);
  LINK(2, 0, 1);
  LINK(2, 1, 0);
  LINK(0, 1, 2);      /* LINK(0, 2, 3) and LINK(0, 3, 1) will be established shortly in the first recursive call. */
  recursive_generate_PSLA_start(3);
```

This code is used in chunk 2*.

15 ¶⟨ Indicate Progress 15 ⟩ ≡

```
  if (n ≡ n_max ∧ countPSLA[n] % 50000000000 ≡ 0) {      /* 5 × 10^10 */
    printf("..%Ld..␣", countPSLA[n]);
    PSLA P;
    convert_to_PS_array(&P, n);
    print_pseudolines_short(&P, n);
    fflush(stdout);
  }
```

This code is used in chunk 13.

16 ¶⟨ Update counters 16 ⟩ ≡

```
  countPSLA[n]++;      /* update accession number counter */
  localCountPSLA[n]++;      /* update local counter */
```

This code is used in chunk 13.

4 Handling the exclude-file

The array *excluded_code*[3 . . . *excluded_length*] contains the decimal code of the next PSLA that should be excluded from the enumeration. During the enumeration, the decimal code of the currently visited tree node (as stored in *localCountPSLA*) agrees with *excluded_code* up to position *matched_length*.

It is assumed that the codes in the exclude-file are sorted in strictly increasing lexicographic order, and no code is a prefix of another code.

To give an example, here are a few lines from the middle of the file `exclude10.txt`:

```
1.3.7.12.9.17.45
1.3.7.12.9.18.35
1.3.7.12.9.18.37
1.3.7.12.9.19
1.3.7.12.9.20
1.3.7.12.9.21.36
1.3.7.12.9.21.37
```

NOTE: As currently implemented, the handling of the exclude-file does not work together with the parallelization through the *splitlevel* option. This is not checked.

17 ⟨ Global variables 7 ⟩ +≡

```
  unsigned excluded_code[MAXN + 3];
  int excluded_length ← 0;
  int matched_length ← 0;      /* These initial values will never lead to any match. */
  FILE *exclude_file;
  char exclude_file_line[100];
```

18 ¶⟨ Check for exclusion and set the flag *is_excluded* 18 ⟩ ≡

```
boolean is_excluded ← false;
if (n ≡ matched_length + 1 ∧ localCountPSLA[n] ≡ excluded_code[n]) {
  matched_length ← n;    /* one more matching entry was found. */
  if (matched_length ≡ excluded_length) {    /* skip this PSLA and the whole subtree */
    is_excluded ← true;
    ⟨ Get the next excluded decimal code from the exclude-file 20 ⟩
    ⟨ Determine the matched length matched_length 21 ⟩
  }
}
```

This code is used in chunk 13.

19 ¶⟨ Open the exclude-file and read first line 19 ⟩ ≡

```
exclude_file ← fopen(exclude_file_name, "r");
⟨ Get the next excluded decimal code from the exclude-file 20 ⟩
matched_length ← 2;
```

This code is used in chunk 8.

¶ I don't know why the following program piece is so badly formatted by `cweave`.

20 ⟨ Get the next excluded decimal code from the exclude-file 20 ⟩ ≡

```
do { if (fscanf(exclude_file, "%s\n", exclude_file_line) ≠ EOF) { char *str1 ← exclude_file_line;
char *token, *saveptr;
excluded_length ← 2;
while (true) { token ← strtok_r(str1, ".", &saveptr);
if (token ≡ Λ) break;
assert ( excluded_length < MAXN +3−1 ) ;
excluded_code[++excluded_length] ← atoi(token);
str1 ← Λ; } }
else {
  excluded_length ← 0;    /* end of file reached. */
  fclose(exclude_file);
}
}
while (excluded_length > n_max) ;    /* patterns longer than n_max are filtered. */
```

This code is used in chunks 18 and 19

¶ (The following program piece could be accelerated if the exclude-file would not store every decimal code completely but indicate only the deviation from the previous code.)

21 ⟨ Determine the matched length *matched_length* 21 ⟩ ≡

```
matched_length ← 2;
while (excluded_code[matched_length + 1] ≡ localCountPSLA[matched_length + 1] ∧ matched_length <
      excluded_length ∧ matched_length < n)
  matched_length++;
```

This code is used in chunk 18.

5 Conversion between different representations

¶ Convert from linked list to array.

Input: PSLA with n lines $1 \,..\, n$, stored in *succ*. Output: PSLA-Array P of size $(n+1)\times(n-1)$ for pseudoline arrangement on n pseudolines.

23 ⟨ Subroutines 23 ⟩ ≡

```
void convert_to_PS_array(PSLA *P, int n)
{
  int j ← 1;
```

```
    for_int_from_to (i, 0, n) {
      for_int_from_to (p, 0, n − 1) {
        (∗P)[i][p] ← j;
        j ← SUCC(i, j);
      }
      j ← 0;      /∗ j starts at 0 except for the very first line. ∗/
    }
  }
```

See also chunks 24, 26, 28, 30, 31, 33, 37, 38, 42, 46, 58, 60*, 62*, 64*, 66*, 68*, and 70*
This code is used in chunk 2*.

¶ The inverse PSLA matrix $\bar{P}$ gives the following information: $\bar{P}_{jk} = p$ if the intersection between line j and line k is the p-th intersection on line j ($p = 0, .., n-1$). This is used to answer orientation queries about the pseudoline arrangement, and about the dual point set, see Section 1.8.

24 ⟨ Subroutines 23 ⟩ +≡

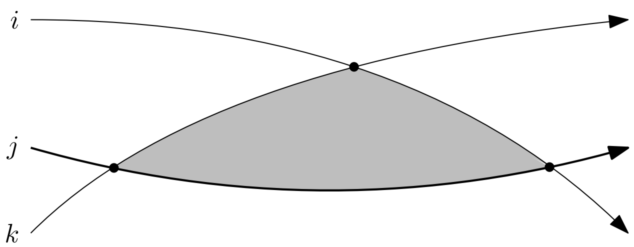
```
  void convert_to_inverse_PS_array(PSLA ∗P̄, int n)
  {
    int j ← 1;
    for_int_from_to (i, 0, n) {
      for_int_from_to (p, 0, n − 1) {
        (∗P̄)[i][j] ← p;
        j ← SUCC(i, j);
      }
      j ← 0;      /∗ j starts at 0 except for the very first line. ∗/
    }
  }
```

6 The orientation predicate

We compute the orientation predicate in constant time from the inverse permutation array $\bar{P}$. It is a **boolean** predicate that returns *true* if the points i, j, k are in counterclockwise order. It works only when the three indices are distinct.

It is computed by comparing the intersections on line j.

If $i < j < k$, this predicate is *true* if the intersection of lines i and k lies above line j. When i, j, k are permuted, the predicate must change according to the sign of the permutation. For documentation purposes, we specify an expression *getOrientation_explicit* that distinguishes all 3! possibilities in which the indices i, j, k can be ordered. *getOrientation* is a simpler, equivalent, expression.



25 **#define** *getOrientation_explicit*$(\bar{P}, i, j, k)$

$$(i < j \land j < k \,?\, \bar{P}[i][j] > \bar{P}[i][k] : i < k \land k < j \,?\, \bar{P}[i][j] > \bar{P}[i][k] : j < i \land i < k \,?\, \bar{P}[i][j] < \bar{P}[i][k] : j < k \land k < i \,?\, \bar{P}[i][j] > \bar{P}[i][k] : k < j \land j < i \,?\, \bar{P}[i][j] > \bar{P}[i][k] : k < i \land i < j \,?\, \bar{P}[i][j] < \bar{P}[i][k] : 0)$$

#define *getOrientation*$(\bar{P}, i, j, k)$ $((i < j) \oplus (j < k) \oplus (\bar{P}[j][i] > \bar{P}[j][k]))$

¶ extreme points from the PSLA.

This is easy; we just scan the top face. We know that 0, 1, and n belong to the convex hull. 0 represents the line at ∞).

The input is taken from the global variable *succ*. (*pred* is not used.)

26 ⟨Subroutines 23⟩ +≡

```
small_int upper_hull_PSLA(int n, small_int *hulledges)
{
  hulledges[0] ← 0;
  small_int hullsize ← 1;
  int k ← 0, k_left, k_right ← 1;
  do {    /* scan the edges of the top face F from left to right */
    k_left ← k;
    k ← k_right;
    k_right ← SUCC(k, k_left);
    hulledges[hullsize++] ← k;
  } while (k_right ≠ 0);
  return hullsize;    /* Result is the number of extreme points. */
}
```

7 Unique identifiers, accession numbers, Dewey decimal notation

The recursive enumeration algorithm imposes an implicit tree structure on PSLAs: the parents of a PSLA with n lines is the unique PSLA on $n-1$ lines from which it is generated. We number the children of each node in the order in which they are generated, starting from 1. The sequence of labels on the path from the root to a node gives a unique identifier to each node in the tree. (This is, however, specific to details of the enumeration algorithm: in which order edges are considered for crossing in the insertion, the choice of lexicographic criterion.)

The purpose of this scheme is that it allows to identify a PSLA even if we parallelize the computation, and one thread of the program only visits certain branches of the tree.

27 ⟨Global variables 7⟩ +≡

```
unsigned localCountPSLA[MAXN +3];
```

28 ¶ ⟨Subroutines 23⟩ +≡

```
void print_id(int n)
{
  printf("%d", localCountPSLA[3]);
  for_int_from_to (i, 4, n) printf(".%d", localCountPSLA[i]);
}
```

8 Output

¶ Prettyprinting of a wiring diagram. Fill a buffer of lines columnwise from left to right.

30 #define TO_CHAR(i) ((char)((i < 10 ? (int) '0' : ((int) 'A' − 10)) + i))

⟨Subroutines 23⟩ +≡

```
void print_wiring_diagram(int n) {    /* ASCII, horizontal, column-wise */
    int next_crossing[MAXN +1];    /* current crossing on each line */
    int line_at[MAXN +1];    /* which line is on the i-th track */
    boolean crossing[MAXN];    /* is there a crossing between track i and i + 1 */
    char buffer [ 2 *MAXN ] [ MAXN * MAXN ] ;
        for_int_from_to (j, 0, n − 1) {
          next_crossing[j + 1] ← SUCC(j + 1, 0);
              /* crossing #0 with line 0 “at ∞” is not considered. */
          line_at[j] ← j + 1;
        }
        crossing[n − 1] ← false;
        int n_crossings ← 0;
        int column ← 0;
        for_int_from_to (p, 0, 2 * n − 1) buffer[p][column] ← '␣'; column++;    /* empty column */
        while (true) {
              /* find where crossings occur, set boolean array crossing[0 . . n − 2] accordingly. */
          boolean something_done ← false;
```

```
        for_int_from_to (p, 0, n − 2) {
          int i ← line_at[p];
          int j ← line_at[p + 1];
          crossing[p] ← next_crossing[i] ≡ j ∧ next_crossing[j] ≡ i;
          if (crossing[p]) {
            something_done ← true;
            n_crossings ++;
          }
        }
        for_int_from_to (p, 0, n − 1) {
          buffer[2 ∗ p][column] ← TO_CHAR(line_at[p]);
          buffer[2 ∗ p + 1][column] ← '␣';
        }
        column ++;
        if (¬something_done) break;
        for_int_from_to (p, 0, n − 1) {
          buffer[2 ∗ p][column] ← '-';
          buffer[2 ∗ p + 1][column] ← '␣';
        }
        for_int_from_to (p, 0, n − 2) {
          if (crossing[p]) {     /∗ print the crossing as an 'X' ∗/
            buffer[2 ∗ p][column] ← buffer[2 ∗ p + 2][column] ← '␣';
                /∗ erase the adjacent lines ∗/
            buffer[2 ∗ p + 1][column] ← 'X';
          }
        }
        column ++;
        for_int_from_to (p, 0, n − 2) {     /∗ carry out the crossings ∗/
          if (crossing[p]) {
            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;
          }
        }
      }
      for_int_from_to (p, 0, 2 ∗ n − 2) {
        buffer[p][column] ← 0;     /∗ finish the lines ∗/
        printf("%s\n", buffer[p]);     /∗ and print them ∗/
      }
      assert(n_crossings ∗ 2 ≡ n ∗ (n − 1)); }
```

8.1 Fingerprints

31 ⟨ Subroutines 23 ⟩ +≡

```
  void print_pseudolines_short(PSLA ∗P, int n)
  {
    printf("P");
    for_int_from_to (i, 0, n) {
      printf("!");
      for_int_from_to (j, 0, n − 1) printf("%c", TO_CHAR((∗P)[i][j]));
    }
    printf("\n");
  }
  void print_pseudolines_compact(PSLA ∗P, int n)
  {     /∗ line 0 is always 1234. . ∗/
    for_int_from_to (i, 1, n) {     /∗ line P_i starts with 0 and is a permutation that misses i. ∗/
      if (i > 1) printf("!");
```

```
      for_int_from_to (j, 1, n − 2)  printf ("%c", TO_CHAR((∗P)[i][j]));
    }
  }
```

8.1.1 A more compact fingerprint

Sufficient to know

$B_i[j] = 1$ if $P_i[j] < i$, see Felsner, Chapter 6.

binary arrays $B_1, \ldots, B_n$. The first column is fixed. The first row B_1 and the last row B_n is fixed, and they need not be coded. Also, since row B_i contains $i - 1$ ones, we can omit the last entry per row, since it can be reconstructed from the remaining entries. Thus we encode the $(n-2) \times (n-2)$ array obtained removing the bordere from the original $n \times n$ array.

We code 6 bits into an ASCII symbol, using the small and capital letters, the digits, and the symbols `+` and `-`.

Since we use this encoding for the case when n is known, we need not worry about terminating the code.

(Replace matrices would offer even more savings.)

32 **#define** `FINGERPRINT_LENGTH` 30 /∗ enough for 13×13 bits plus terminating null ∗/

```
⟨ Global variables 7 ⟩ +≡
  char fingerprint[FINGERPRINT_LENGTH];
```

33 ¶⟨ Subroutines 23 ⟩ +≡

```
  char encode_bits(int acc)
  {
    if (acc < 26) return (char)(acc + (int) 'A');
    else if (acc < 52) return (char)(acc − 26 + (int) 'a');
    else if (acc < 62) return (char)(acc − 52 + (int) '0');
    else if (acc ≡ 62) return '+';
    else return '-';
  }
  void compute_fingerprint(PSLA ∗P, int n)
  {
    int charpos ← 0;
    int bit_num ← 0;
    int acc ← 0;
    for_int_from_to (i, 1, n − 1)
      for_int_from_to (j, 1, n − 1) {
        acc ≪= 1;
        if ((∗P)[i][j] < i) acc |= 1;
        bit_num += 1;
        if (bit_num ≡ 6) {
          fingerprint[charpos++] ← encode_bits(acc);
          assert(charpos < FINGERPRINT_LENGTH − 1);
          bit_num ← acc ← 0;
        }
      }
    if (bit_num) fingerprint[charpos++] ← encode_bits(acc ≪ (6 − bit_num));
    assert(charpos < FINGERPRINT_LENGTH − 1);
    fingerprint[charpos++] ← '\0';
  }
```

34 ¶⟨ Print PSLA-fingerprint 34 ⟩ ≡

```
  {
    PSLA P;
    convert_to_PS_array(&P, n);
    compute_fingerprint(&P, n);
    printf ("%s:", fingerprint);
  }
```

This code is used in chunk 53.

9 Abstract order types

9.1 Lexmin for PSLA representation

In order to generate every AOT only once, we check whether the representation is smallest among all PSLAs that produce AOTs, that are *equivalent* by rotation and reflection.

Lexicographically smallest. We have to try all "boundary points"(?) as pivot points. The average number of extreme vertices is slightly less than 4. It does not pay off to shorten the loop considerably. (The average *squared* face size matters!)

To determine !!!! whether a PSLA is the lex-smallest among all PSLAs representing an AOT, we scan the PSLA matrix row-wise *from right to left*. In comparison with the more natural left-to-right order, this gives, experimentally, a quicker way to eliminate tentative PSLA than the left-to-right order.

36 ⟨Global variables 7⟩ +≡

```
int Sequence[MAXN +1][MAXN +1];
   /* Sequence[r][p] gives the p-th crossing on the r-th hull edge. */
int new_label[MAXN +1][MAXN +1];      /* When the r-th hull edge is used in the role of line 0,
    new_label[r][j] gives index that is use for the (original) line j. */
int candidate[2*(MAXN +1)];      /* list of candidates, gives index r into hulledges */
int current_crossing[2*(MAXN +1)];      /* indexed by candidate number */
int P_1_n_forward[MAXN +1];
int P_1_n_reverse[MAXN +1];
```

37 ¶⟨Subroutines 23⟩ +≡

```
void prepare_label_arrays(small_int n, small_int *hulledges, small_int hullsize)
{
  for_int_from_to (r, 0, hullsize − 1)
    if (P_1_n_reverse[r] ≡ P_1_n_forward[0] ∨ (r > 0 ∧ P_1_n_forward[r] ≡ P_1_n_forward[0])) {
        /* otherwise not needed. */
      int line0 ← hulledges[r];
      new_label[r][line0] ← 0;
      int i ← (r < hullsize − 1) ? hulledges[r + 1] : 0;      /* 0 ≡ hulledges[0] */
      for_int_from_to (p, 1, n) {
        new_label[r][i] ← p;
        Sequence[r][p] ← i;
        i ← SUCC(line0, i);
      }
    }
}
```

9.2 Compute the lex-smallest representation

The input is taken from the global *succ* and *pred* arrays. The function assumes that *hulledges* and *hullsize* have been computed.l)

38 ⟨Subroutines 23⟩ +≡

```
void compute_lex_smallest_PSLA(PSLA *P, small_int n, small_int *hulledges, small_int hullsize)
{
  for_int_from_to (q, 0, n − 1) (*P)[0][q] ← q + 1;      /* row 0 */
  for_int_from_to (r, 0, hullsize − 1) P_1_n_forward[r] ← P_1_n_reverse[r] ← 0;
      /* no screening. dummy values ensure that prepare_label_arrays will prepare all label arrays */
  prepare_label_arrays(n, hulledges, hullsize);
  int numcandidates ← 0;
  for_int_from_to (r, 0, hullsize − 1) candidate[numcandidates++] ← r;
  int numcandidates_forward ← numcandidates;
  for_int_from_to (r, 0, hullsize − 1) candidate[numcandidates++] ← r;
  for_int_from_to (p, 1, n) {      /* compute row P_p of the PSLA array P */
    (*P)[p][0] ← 0;
    for_int_from_to (c, 0, numcandidates − 1) {
      int r ← candidate[c];
```

```
        current_crossing[c] ← hulledges[r];     /* plays the role of line 0 */
      }
      for_int_from_to (q, 1, n − 1) {
          /* Compute P_{p,n−q} by taking the minimum over all candidate choices of line 0. */
        int c;
        int new_candidates, new_candidates_forward;
        int current_min ← n + 1;     /* essentially ∞ */
        boolean reversed ← false;
        int pos ← p;     /* position of line 0; the line we are currently searching in Sequence */
        for (c ← 0; c < numcandidates_forward; c++) {
          ⟨ Process candidate c, keep in list and advance new_candidates if equal; reset new_candidates if
              better value than current_min 39 ⟩
        }
        new_candidates_forward ← new_candidates;     /* can be reset in the next loop */
        reversed ← true;
        pos ← n + 1 − p;
        for ( ; c < numcandidates; c++) {
          ⟨ Process candidate c, keep in list and advance new_candidates if equal; reset new_candidates if
              better value than current_min 39 ⟩
        }
        numcandidates_forward ← new_candidates_forward;
        numcandidates ← new_candidates;
        (*P)[p][n − q] ← current_min;     /* could enter a shortcut as soon as numcandidates ≡ 1 */
      }
    }
  }
```

¶ The list of candidates is scanned and simultaneously overwritten with new values.

39 ⟨ Process candidate c, keep in list and advance *new_candidates* if equal; reset *new_candidates* if better value than *current_min* 39 ⟩ ≡

```
  int r ← candidate[c];
  int i ← Sequence[r][pos];     /* We are proceeding on line i */
  int j ← current_crossing[c];
  j ← reversed ? SUCC(i, j) : PRED(i, j);
  int a ← new_label[r][j];
  if (reversed ∧ a ≠ 0) a ← n + 1 − a;
  if (a < current_min)     /* new record: */
  {
    new_candidates ← new_candidates_forward ← 0;
    current_min ← a;
  }
  if (a ≡ current_min) {     /* candidate survives. */
    candidate[new_candidates] ← r;
    current_crossing[new_candidates] ← j;
    new_candidates++;
  }     /* Otherwise the candidate is skipped. */
```

This code is used in chunk 38.

¶ The output parameters have only a meaning if the test returns *true*. *has_fixpoint* is only set if the PSLA is mirror-symmetric.

We scan the entries of P row-wise from right to left. We maintain a list of solutions, which are still *candidates* to be lex-smallest. Initially we have $2 \times$ *hullsize* candidates, *hullsize* "forward" candidates and the same number of mirror-symmetric, reversed candidates.

Candidates $0 \,..\,$ *numcandidates_forward* -1 are forward candidates. The remaining candidates up to *numcandidates* -1 are reverse (mirror) candidates.

If information about mirror symmetry is not necessary, then the mirror candidates can be omitted.

9.3 Streamlined version

Fast screening of candidates

Let i and j be two consecutive edges on the upper envelope. The quantity $Q(i,j)$ is defined as follows, see Figure 3a.

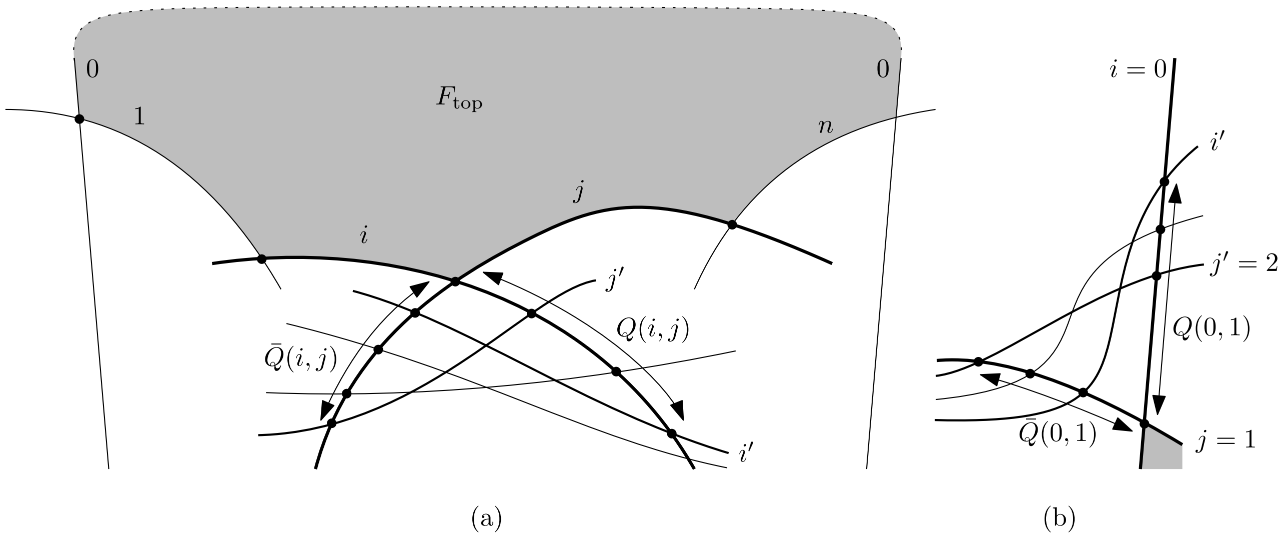


Figure 3: (a) An example with $Q(i,j)=4$ and $\bar{Q}(i,j)=5$; (b) an example with $Q(0,1)=\bar{Q}(0,1)=4$

Let $i' = \texttt{PRED}(i,j)$. Walk on line i to the right (by `SUCC`) from the intersection between i and j until meeting the intersection with i'. Then $Q(i,j)$ is the number of visited points on i, including the endpoints. This convention ensures that $Q(i,j)$ is the value P_{1n} when line i is chosen to play the role of line 0, (and j will become line 1). In the walk along i, we may cross line 0 and wrap around to the left end.

The quantity $\bar{Q}(i,j)$ is defined with switched roles of i and j and with left and right exchanged, and it gives the value P_{1n} in the mirror situation (the *backward* direction) when line j is chosen to play the role of line 0: Let $j' = \texttt{SUCC}(i,j)$. Walk on line j to the left (by `PRED`) until meeting line j'.

We apply this definition two all pairs (i,j) of consecutive edges on the upper envelope, starting with $(0,1)$ and ending with $(n,0)$. (The last pair is the only pair with $i>j$.)

The numbers $Q(i,j)$ and $\bar{Q}(i,j)$ are between 2 and n, and $Q(i,j)=2 \iff \bar{Q}(i,j)=2$.

For $(i,j)=(0,1)$, the wedge between lines i and j appears actually at the bottom right of the wiring diagram, see Figure 3b. Here we have $Q(0,1) = \texttt{PRED}(1,0) = P_{1n}$, since this is the original situation where line 0 is where it should be. Similarly, for $(i,j)=(n,0)$, we have to look at the bottom left corner.

...

Our primary criterion in comparing candidates is P_{1n} which is given by $Q(i,j)$ and $\bar{Q}(i,j)$ for the pairs (i,j) of consecutive edges on the upper envelope. This has to be compared against. $Q(0,1)$.

¶ Screen candidates by comparing the leading entry P_{1n},

Compute the leading entry P_{1n} for all candidates directly, without first computing the *label_arrays*. The *label_arrays* are computed afterwards (if at all), and only those that are still necessary. This saves about 20 % of the runtime for enumerating AOTs. If $P_{1n}=2$ for line 0, the screening has no effect, but otherwise there is a high chance for finding a smaller value P_{1n} for some of the other candidates.

[Observation. The relative frequence of P_{1n} over all PSLAs is about 26 % for 2 and n, about 11 % for 3 and $n-1$ and decreases towards the middle values. The symmetry can be explained as follows. An xPSLA is essentially a projective oriented PSLA with a marked angle. Going to an adjacent angle and mirroring the PSLA exchanges a with $n+2-a$.]

The following program treats each forward candidate i together with the corresponding mirror candidate j. it uses the condition $Q(i,j)=2 \iff \bar{Q}(i,j)=2$ to shortcut the computation. (not sure if it brings any advantage.)

For example there are 18,410,581,880 PSLAs with $n=10$ lines. Of these, only 5,910,452,118 pass the screening test. Eventually, only 2,343,203,071 PSLA are really lex-min, and this is the number of AOTs that we really want.

42 ⟨Subroutines 23⟩ +≡

```
boolean screen(small_int n, small_int *hulledges, small_int hullsize)
{
  P_1_n_forward[0] ← PRED(1,0);      /* because hulledges[1] ≡ 1 */
  for_int_from_to (r, 1, hullsize − 1) {
    int r_next ← (r + 1) % hullsize;
    int i ← hulledges[r];
    int j ← hulledges[r_next];      /* i or j plays the role of line 0 */
    int i′ ← PRED(j, i);
    int a ← 2; int j2 ← SUCC(i, j);
    while (j2 ≠ i′) {      /* compute a by running along i */
      j2 ← SUCC(i, j2);
      a++;
      if (a > P_1_n_forward[0]) break;      /* shortcut */
    }
    if (a < P_1_n_forward[0]) return false;
    P_1_n_forward[r] ← a;      /* This may not be the precise value if a > P_1_n_forward[0] */
  }
  for_int_from_to (r, 0, hullsize − 1) {
    int r_next ← (r + 1) % hullsize;
    if (P_1_n_forward[r] ≡ 2) {
      P_1_n_reverse[r_next] ← 2;
         /* The wedge between i and i is a triangle; Q(i, j) and Q̄(i, j) are both 2. */
      continue;
    }
    int i ← hulledges[r];
    int j ← hulledges[r_next];      /* i or j plays the role of line 0 */
    int j′ ← SUCC(i, j);
    int a ← 2; int i2 ← PRED(j, i);
    do {      /* compute a by running along j */
      i2 ← PRED(j, i2);
      a++;
      if (a > P_1_n_forward[0]) break;
    } while (i2 ≠ j′);
    if (a < P_1_n_forward[0]) return false;
    P_1_n_reverse[r_next] ← a;
  }
  return true;
}
```

¶ More effective screening at the previous level.

Rather than generating many PSLAs with n lines and eliminating them by screening, it is better not to generate them at all, or to generate only those that have a change of surviving the screening test.

To do this, we apply a test at the previous level.

When adding a new line n, the quantities $Q(i,j)$ can change in a few ways.

1. We cut off some hull vertices. In particular, $(n-1,0)$ will always disappear.
2. We generate two new hull vertices: (i,n) with $1 \le i \le n-1$, and $(n,0)$.
3. In the definition of $Q(i,j)$, line n could take the role of i'. (or j' in the case of $\bar{Q}(i,j)$).
4. In the definition of $Q(i,j)$, line n could intervene between the intersections with j and i' on line i, thus increasing $Q(i,j)$ by 1. (or a similar situation for $\bar{Q}(i,j)$).

A very rudimentary pre-screening test has been implemented, namely for the comparison between $Q(0,1)$ and $\bar{Q}(1,0)$:

If $\bar{Q}(0,1) < Q(1,0) - 1$ in the arrangement with $n-1$ lines, then there is no chance to augment this to a lex-min PSLA.

Proof: See Figure 3b. There are two cases. If line n does not intersect the segment between 1×0 and $1\times \mathtt{PRED}(1,0)$, then $Q(0,1)=P_{1n}$ is unchanged. $\bar{Q}(1,0)$ can increase by at most 1. Thus $\bar{Q}(1,0)$ will beat $Q(1,0)$.

If line n intersects line 1 between 1×0 and $1\times \mathtt{PRED}(1,0)$, then n becomes the new $i'=\mathtt{PRED}(1,0)=Q(0,1)=P_{1n}$, and thus P_{1n} has the maximum possible value, n, and is certainly larger than before. $\bar{Q}(1,0)$ can still increase by at most 1. Thus $\bar{Q}(1,0)$ will beat $Q(1,0)$.

For example, with $n=9$ lines there are 112,018,190 PSLAs, and they generate as children 18,410,581,880 PSLAs with $n=10$ lines, as mentioned above. The screening test at level $n=9$ eliminates 22,023,041 out of the 112,018,190 PSLAs (19.66%) because they are not able to produce a lex-min AOT in the next generation. The remaining 89,995,149 PSLAs produce 15,409,623,219 offspring PSLAs with $n=10$ lines. as opposed to 18,410,581,880 without this pruning procedure. These remaining PSLAs are subject to the screening as before.

44 ¶⟨ Screen one level below level *n_max* 44 ⟩ ≡

```
int P_1_n ← PRED(1,0);      /* insertion of last line n can only make this larger. */
if (P_1_n > 3) {
  int a ← 2;
  int i2 ← P_1_n;      /* ≡ i' */
  while (i2 ≠ 2) {      /* compute a by running along j ≡ 1 */
    i2 ← PRED(1, i2);
    a++;
  }      /* Now P_1_n_reverse ≡ a but insertion of line n could increase this by 1. */
  if (a + 1 < P_1_n) hopeful ← false;
}
if (hopeful) cpass++; else csaved++;
```

This code is used in chunk 13.

¶ We maintain statistics about the effectiveness of this test:

45 ⟨ Global variables 7 ⟩ +≡

```
long long unsigned cpass, csaved;
```

46 ¶⟨ Subroutines 23 ⟩ +≡

```
boolean is_lex_smallest_PSLA(small_int n, small_int *hulledges, small_int hullsize, small_int
        *rotation_period, boolean *is_symmetric, boolean *has_fixpoint)
{
  if (¬screen(n, hulledges, hullsize)) return false;
#if profile
  numTests++;
#endif
  prepare_label_arrays(n, hulledges, hullsize);
  int numcandidates ← 0;
  for_int_from_to (r, 1, hullsize − 1)
    if (P_1_n_forward[r] ≡ P_1_n_forward[0]) candidate[numcandidates++] ← r;
  int numcandidates_forward ← numcandidates;
  for_int_from_to (r, 0, hullsize − 1)
    if (P_1_n_reverse[r] ≡ P_1_n_forward[0]) candidate[numcandidates++] ← r;
  for_int_from_to (p, 1, n) {      /* explore row P_p of the PSLA array P */
    int current_crossing_0 ← 0;      /* candidate c = 0 is treated specially. */
    for_int_from_to (c, 0, numcandidates − 1) {
      int r ← candidate[c];      /* plays the role of line 1 */
      current_crossing[c] ← hulledges[r];      /* plays the role of line 0 */
    }
    for_int_from_to (q, 1, n − 2) {      /* Compute P_{p,n−q} for all choices of line 0. The last entry q = n − 1
          can be omitted, because every row is a permutation. */
      int target_value ← current_crossing_0 ← PRED(p, current_crossing_0);
        /* special treatment of candidate 0: current line i is line p; no relabeling necessary. */
```

```
        int c;
        int new_candidates ← 0;
        boolean reversed ← false;
        int pos ← p;      /* position of line 0 */
        for (c ← 0; c < numcandidates_forward; c++) {
          ⟨Process candidate c, keep in list and advance new_candidates if successful; return false if better
              value than target_value is found 47⟩
        }
        numcandidates_forward ← new_candidates;
        reversed ← true;
        pos ← n + 1 − p;
        for ( ; c < numcandidates; c++) {      /* continue the previous loop */
          ⟨Process candidate c, keep in list and advance new_candidates if successful; return false if better
              value than target_value is found 47⟩
        }
        numcandidates ← new_candidates;
        if (numcandidates ≡ 0) {      /* early return */
          *rotation_period ← hullsize;
          *is_symmetric ← false;
          return true;
        }
      }
    }
    ⟨Determine the result parameters, depending on the remaining candidates. 48⟩
    return true;
  }
```

47 ¶⟨Process candidate c, keep in list and advance *new_candidates* if successful; return *false* if better value than *target_value* is found 47⟩ ≡

```
#if profile
  numComparisons++;
#endif
  int r ← candidate[c];
  int i ← Sequence[r][pos];
  int j ← current_crossing[c];
  j ← reversed ? SUCC(i, j) : PRED(i, j);
  int a ← new_label[r][j];
  if (reversed ∧ a ≠ 0) a ← n + 1 − a;
  if (a < target_value) return false;
  if (a ≡ target_value) {
    candidate[new_candidates] ← r;
    current_crossing[new_candidates] ← j;
    new_candidates++;
  }
```

This code is used in chunk 46.

48 ¶⟨Determine the result parameters, depending on the remaining candidates. 48⟩ ≡

```
  {
    if (numcandidates_forward > 0) *rotation_period ← candidate[0];
    else *rotation_period ← hullsize;
    *is_symmetric ← (numcandidates > numcandidates_forward);
    if (*is_symmetric) {
      int symmetric_shift ← candidate[numcandidates_forward];
          /* There is a mirror symmetry that maps 0 to this hull vertex. */
      *has_fixpoint ← ((*rotation_period) % 2 ≡ 1) ∨ (symmetric_shift % 2 ≡ 0);
    }
  }
```

This code is used in chunk 46.

10 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 fixpoint on the hull (3 possibilities).

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_FIXPOINT` 1
#define `MIRROR_WITHOUT_FIXPOINT` 2

⟨ Global variables 7 ⟩ +≡
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 ∗/

50 ¶⟨ Initialize statistics and open reporting file 50 ⟩ ≡
countPSLA[1] ← *countPSLA*[2] ← 1;
countO[3] ← *countU*[3] ← *PSLAcount*[2] ← *xPSLAcount*[2] ← 1;
/∗ All other counters are automatically initialized to 0. ∗/
if (*strlen*(*fname*)) {
reportfile ← *fopen*(*fname*, `"w"`);
}

This code is used in chunk 2*.

51 ¶* ⟨ Gather statistics about the AOT, collect output 51 ⟩ ≡ /∗ Determine the extreme points: ∗/
small_int *hulledges*[**MAXN** +1];
small_int *hullsize* ← *upper_hull_PSLA*(*n*, *hulledges*);
small_int *rotation_period*;
boolean *has_fixpoint*;
boolean *is_symmetric*;
int *n_points* ← $n + 1$; /∗ number of points of the AOT ∗/
boolean *lex_smallest* ← *is_lex_smallest_PSLA*(*n*, *hulledges*, *hullsize*, &*rotation_period*, &*is_symmetric*, &*has_fixpoint*);
if (*lex_smallest*) {
countU[*n_points*]++;
if (*is_symmetric*) {
countO[*n_points*]++;
PSLAcount[*n*] += *rotation_period*;
if (*has_fixpoint*) *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][¬is_symmetric ? NO_MIRROR : has_fixpoint ?
        MIRROR_WITH_FIXPOINT : MIRROR_WITHOUT_FIXPOINT][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 13.

¶* written to a file so that a subsequent program can conveniently read and process it.

52 ⟨Report statistics 52⟩ ≡

```
  printf("%34s%69s\n", "#PSLA␣visited␣by␣the␣program", "#PSLA␣computed␣from␣AOT");
  for_int_from_to (n, 3, n_max + 1) {
    long long symmetric ← 2 * countU[n] − countO[n];
    printf("n=%2d", n);
    if (split_level ≠ 0 ∧ n > split_level) printf("*,"); else printf(",␣");
    printf("#PSLA=%11Ld", countPSLA[n]);
#if 1
    printf(",␣#AOT=%10Ld,␣#OAOT=%10Ld,␣#symm.␣AOT=%7Ld,␣", countU[n], countO[n], symmetric);
    printf("#PSLA=%11Ld,␣#xPSLA=%10Ld", PSLAcount[n], xPSLAcount[n]);
#endif
    printf("\n");
  }
  if (split_level ≠ 0) printf("*␣Lines␣with␣\"*\"␣give␣results␣from␣partial␣enumeration.\n");
#if profile
  printf("Total␣tests␣is_lex_min␣(after␣screening)␣=␣%Ld,␣total␣comparisons␣=␣%Ld,␣averag\
      e␣=%6.3f\n", numTests, numComparisons, numComparisons/(double) numTests);
#endif
  printf("passed␣%Ld,␣saved␣%Ld␣out␣of␣%Ld␣=␣%.2f%%\n", cpass, csaved, cpass + csaved,
      100 * csaved/(double)(cpass + csaved));
  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);
  }
```

This code is used in chunk 2*.

¶ Problem-specific processing can be added here.

After computing the inverse PSLA matrix, one can perform a few tests on the order type, using orientation queries.

The following test program compares the orientation queries against an explicitly computed "large Λ-matrix".

53 ⟨ Further processing of the AOT 53 ⟩ ≡

```
#if generatelist
    /* List all PSLAs plus their IDs, as preparation for generating exclude-files of nonrealizable AOTs */
  if (n ≡ n_max ∧ lex_smallest) {
    ⟨ Print PSLA-fingerprint 34 ⟩print_id(n);
    printf("\n");
  }
#endif
#if 0
  if (n ≡ n_max ∧ countPSLA[n] ≡ 50) {      /* print “some” example */
    PSLA PP, invPP;
    convert_to_PS_array(&PP, n);
    convert_to_inverse_PS_array(&invPP, n);
    print_pseudolines_short(&PP, n);
    printf("inverse␣");
    print_pseudolines_short(&invPP, n + 1);
    print_wiring_diagram(n);
  }
#endif
#if 0      /* estimate size of possibly subproblems for d&c Ansatz */
#define MID 5
  if (n ≡ 2 ∗ MID − 2) {
    PSLA P;
    convert_to_PS_array(&P, n);
    for_int_from_to (i, 2, MID − 1) {
      boolean show ← true;
      for_int_from_to (j, 1, n − 1) {
        int x ← P[i][j];
        if (x ≡ MID ∨ x ≡ 1) break;
        printf("%c", TO_CHAR(x));
      }
      printf("!");
    }
    for_int_from_to (i, MID + 1, n) {
      boolean show ← false;
      for_int_from_to (j, 1, n − 1) {
        int x ← P[i][j];
        if (show) printf("%c", TO_CHAR(x));
        if (x ≡ MID) show ← true;
        if (x ≡ 1) break;
      }
      printf(i < n ? "!" : "␣");
    }
    for_int_from_to (j, 1, n − 1) {
      int x ← P[1][j];
      if (x ≡ MID) break;
      printf("%c", TO_CHAR(x));
    }
    printf("!");
    for_int_from_to (j, 1, n − 1) {
      int x ← P[MID][j];
      if (x ≡ 1) break;
      printf("%c", TO_CHAR(x));
    }
    printf("\n");
  }
#endif
#if 0
  PSLA P̄;      /* the orientation test is computed from this array. */
```

```
  convert_to_inverse_PS_array(&P̄, n);
  small_matrix S;
  convert_to_small_lambda_matrix(&S, n_points);
  large_matrix L;
  convert_small_to_large(&S, &L, n_points);
  ⟨Compare orientation tests 54⟩
#endif
```

This code is used in chunk 13.

54 ¶⟨Compare orientation tests 54⟩ ≡

```
{
  int n ← n_points;
  for_int_from_to (i, 0, n − 1)
    for_int_from_to (j, 0, n − 1)
      if (i ≠ j)
        for_int_from_to (k, 0, n − 1)
          if (k ≠ j ∧ k ≠ i)
            if (getOrientation(P̄, i, j, k) ≠ L[i][j][k]) {
              printf("[%d,%d,%d]=%d!=%d\n", i, j, k, getOrientation(P̄, i, j, k), L[i][j][k]);
              exit(1);
            }
  ;
}
```

This code is used in chunk 53.

11 Data structures for abstract order types

¶ λ-matrices.

In this program, entries Λ_{ijk} of the large matrix are only ever accessed for $i < j < k$. For more general access, we provide the macro *get_entry_large*. It would be possible to save space by a more elaborate indexing function into a one-dimensional array.

natural labeling around the *pivot* point, which is assumed to lie on the convex hull.

56 **#define** *entry_small*(A, i, j) $(A)[i][j]$

¶ More type definitions.

57 ⟨Types and data structures 4⟩ +≡

```
  typedef uint_fast8_t XXsmall_matrix_entry;   /* suffices up to n = 255 + 1 */
  typedef int_fast8_t XXsmall_int;   /* suffices for n */
  typedef boolean large_matrix_entry;
  typedef unsigned small_matrix_entry;
  typedef int small_int;   /* simpler and maybe even faster? */
  typedef small_matrix_entry small_matrix [MAXN +1][MAXN +1];
  typedef large_matrix_entry large_matrix [MAXN +1][MAXN +1][MAXN +1];
```

¶ Generating the Λ-matrix. Only for testing purposes. Assumes natural ordering. Assumes general position. Works by plucking points from the convex hull one by one.

58 ⟨Subroutines 23⟩ +≡

```
  void copy_small(small_matrix *A, small_matrix *B, small_int n)
  {
    for (small_int i ← 0; i < n; i++)
      for (small_int j ← 0; j < n; j++) entry_small(*B, i, j) ← entry_small(*A, i, j);
  }
  void convert_small_to_large(small_matrix *A, large_matrix *B, small_int n)
  {
    small_matrix Temp;
```

```
    copy_small(A, &Temp, n);      /* the small matrix Temp will be destroyed */
    for (small_int k ← 0; k < n; k++)
      for (small_int i ← k + 1; i < n; i++)
        for (small_int j ← i + 1; j < n; j++)      /* k < i < j */
        {
          if (entry_small(Temp, i, k) < entry_small(Temp, j, k)) {
            entry_small(Temp, i, j)−−;
            (*B)[k][i][j] ← (*B)[i][j][k] ← (*B)[j][k][i] ← true;
            (*B)[k][j][i] ← (*B)[i][k][j] ← (*B)[j][i][k] ← false;
          }
          else {
            entry_small(Temp, j, i)−−;
            (*B)[k][i][j] ← (*B)[i][j][k] ← (*B)[j][k][i] ← false;
            (*B)[k][j][i] ← (*B)[i][k][j] ← (*B)[j][i][k] ← true;
          }
        }
  }
```

12 Auxiliary routines and conversion to other formats

¶* Input: PSLA with n lines $1 \,..\, n$ plus line 0 "at ∞". Output: small λ-matrix B for AOT on $n+1$ points. Line at ∞ corresponds to point 0 on the convex hull.

60 ⟨Subroutines 23⟩ +≡

```
  void convert_to_small_lambda_matrix(small_matrix *B, int n)
  {
    for_int_from_to (i, 0, n) {
      (*B)[i][i] ← 0;
    }
    for_int_from_to (i, 1, n) {
      int level ← i − 1;      /* number of lines above the crossing */
      (*B)[0][i] ← level;
      (*B)[i][0] ← n − 1 − level;
      int j ← SUCC(i, 0);
      while (j ≠ 0) {
        if (i < j) {
          (*B)[i][j] ← level;
          level++;
        }
        else {
          level−−;
          (*B)[i][j] ← n − 1 − level;
        }
        j ← SUCC(i, j);
      }
    }
  }
```

12.1 Extension: Compute crossing-number for each AOT

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. With a bound of 50 we should be on the safe side. $n = 11$ point has at most 24 halving-lines

61

```
#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. consecutive tracks p and $p+1$. */

Use the formular from Convex quadrilaterals and k-sets, DOI:10.1090/conm/342/06138

The program computes the number of crossings *num_crossings_on_level*$[p]$ at each level p except for the crossings with line 0. (From this information, there is actually an easy formula to compute the crossing number of the complete graph K_n when it is drawn on this point set.)

62 **#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 23 ⟩ +≡

```
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]++;    /* 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;
}
```

13 Reading from the Order-Type Database

For simplicity, we work only with numbers in the 16-bit format. Inputs in 8-bit formats are converted.

63 ⟨ Global variables 7 ⟩ +≡

```
struct {      /* 16-bit unsigned coordinates: */
   uint16_t x, y;
} points[MAXN +1];
struct {      /* 8-bit unsigned coordinates: */
   uint8_t x, y;
} pointsmall[MAXN +1];
```

13.1 Orientation test for points

The return value of *orientation_test* is positive for counterclockwise orientation of the points i, j, k.

64 ⟨ Subroutines 23 ⟩ +≡

```
large_int orientation_test(int i, int j, int k)
{
   large_int a ← points[j].x − (large_int) points[i].x;      /* range −65535 . . 65535 */
   large_int b ← points[j].y − (large_int) points[i].y;
   large_int c ← points[k].x − (large_int) points[i].x;
   large_int d ← points[k].y − (large_int) points[i].y;
   return a ∗ d − b ∗ c;
}
```

¶* Intermediate results can be almost 2^{32} in absolute value, and they have signs. The final value is the signed area of the parallelogram spanned by 3 points. Thus it can also be almost 2^{32} in absolute value. 32 bits are not enough to be safe. We use 64 bits.

65 ⟨ Types and data structures 4 ⟩ +≡

```
typedef int_least64_t large_int;      /* for intermediate calculations */
```

13.2 Turn point set with coordinates into PSLA

We insert the lines one by one into the arrangement. This is simular to the insertion of line n in the recursive enumeration procedure. The difference is that we don't try all possibilities for the edge through which line n exits, but we choose the correct edge the by orientation test. By the zone theorem, the insertion of line n takes $O(n)$ time.

We have n points. The first point (point 0) is on the convex hull and the other points are sorted around this point. We get a PSLA with $n - 1$ pseudolines.

66 ⟨ Subroutines 23 ⟩ +≡

```
void insert_line(int n);
void PSLA_from_points(int n)
{
   LINK(1, 0, 2);
   LINK(1, 2, 0);
   LINK(2, 0, 1);
   LINK(2, 1, 0);
   LINK(0, 1, 2);
      /* LINK(0, 2, 3) and LINK(0, 3, 1) will be established shortly in the first recursive call. */
   for_int_from_to (i, 3, n − 1) insert_line(i);
}
void insert_line(int n)
{
   LINK(0, n − 1, n);
   LINK(0, n, 1);
   int entering_edge ← 0, j ← 0, j⁺ ← 0;
   int k_left, k_right;
```

```
    while (1) {
      while (j+ > j) {      /* find right vertex of the cell */
        int j+_old ← j+;
        j+ ← SUCC(j+, j);
        j ← j+_old;
      }
      if (j+ ≡ 0) {      /* F is unbounded */
        if (j ≡ n − 1) {      /* F is the top face. */
          LINK(n, entering_edge, 0);      /* complete insertion of line n */
          return;
        }
        j+ ← j + 1;      /* jump to the upper ray of F */
        j ← 0;
      }      /* Now the crossing j×j+ is the rightmost vertex of the face F. j+ is on the upper side, and if F
               is bounded, j is on the lower side. */
      do {      /* scan the upper edges of F from right to left and find the correct one to cross. */
        k_right ← j;
        j ← j+;
        k_left ← j+ ← PRED(j, k_right);
      } while (j+ > j ∧ orientation_test(j, k_left, n) > 0);
      LINK(j, k_left, n);      /* insert crossing with n on line j */
      LINK(j, n, k_right);
      LINK(n, entering_edge, j);
      entering_edge ← j;
      j+ ← k_right;
    }
  }
```

13.3 Do the actual reading

We have to figure out the filenames and the format of the stored numbers. We assume that the order types with up to 10 points are stored in the current directory in with the original file names `otypes10.b16`, `otypes09.b16`, `otypes08.b08`, etc., and the order types with 11 points are stored in a subdirectory `Ordertypes` with names `Ordertypes/ord11_00.b16` ... `Ordertypes/ord11_93.b16`.

67 ⟨Include standard libaries 5⟩ +≡

```
#include <fcntl.h>
#include <unistd.h>
```

68 ¶⟨Subroutines 23⟩ +≡

```
  void swap_all_bytes(int n)
  {      /* convert numbers from little-endian to big-endian format. */
    for_int_from_to (i, 0, n − 1) {
      points[i].x ← (points[i].x ≫ 8) | (points[i].x ≪ 8);
      points[i].y ← (points[i].y ≫ 8) | (points[i].y ≪ 8);
          /* Assumes 16 bits. It is important that coordinates are UNSIGNED. */
    }
  }
```

69 ¶⟨Read all point sets of size n_max + 1 from the database and process them 69⟩ ≡

```
  int n_points ← n_max + 1;
  int bits ← n_points ≥ 9 ? 16 : 8;
  char inputfile[60];
  int record_size ← (bits/8) ∗ 2 ∗ n_points;
  printf("Reading␣order␣types␣of␣%d␣points\n", n_points);
  printf(".\n");
  printf("One␣record␣is␣%d␣bytes␣long.\n", record_size);
  boolean is_big_endian ← ( ∗ ( uint16_t ∗ ) "\0\xff" < #100/ ) ;
```

```
if (bits > 8) {
  if (is_big_endian) printf("This␣computer␣is␣big␣endian.\n");
  else printf("This␣computer␣is␣little-endian.␣No␣byte␣swaps␣are␣necessary.\n");
}
if (n_points < 11) {
  snprintf(inputfile, 60, "otypes%02d.b%02d", n_points, bits);
  read_database_file(inputfile, bits, record_size, n_points, is_big_endian);
}
else
  for_int_from_to (num_db, 0, 93) {
    snprintf(inputfile, 60, "Ordertypes/ord%02d_%02d.b16", n_points, num_db);
    read_database_file(inputfile, bits, record_size, n_points, is_big_endian);
  }
printf("%Ld␣point␣sets␣were␣read␣from␣the␣file(s).\n", read_count);
```

This code is used in chunk 2*.

¶* Open and read database file and process the input points.

70 ⟨Subroutines 23⟩ +≡

```
long long unsigned read_count ← 0;
void read_database_file(char *inputfile, int bits, int record_size, int n_points, boolean is_big_endian)
{
  printf("Reading␣from␣file␣%s\n", inputfile);
  int databasefile ← open(inputfile, O_RDONLY);
  if (databasefile ≡ −1) {
    printf("File␣could␣not␣be␣opened.\n");
    exit(1);
  }
  while (1) {
    ssize_t bytes_read;
    if (bits ≡ 16) bytes_read ← read(databasefile, &points, record_size);
    else bytes_read ← read(databasefile, &pointsmall, record_size);
    if (bytes_read ≡ 0) break;
    if (bytes_read ≠ record_size) {
      printf("Incomplete␣file.\n");
      exit(1);
    }
    read_count++;
    if (bits ≡ 16 ∧ is_big_endian) swap_all_bytes(n_points);
    if (bits ≡ 8)
      for_int_from_to (i, 0, n_points − 1) {
        points[i].x ← pointsmall[i].x;
        points[i].y ← pointsmall[i].y;
      }
    int n ← n_points − 1;
    PSLA_from_points(n_points);
    small_int hulledges[MAXN +1];
    small_int hullsize ← upper_hull_PSLA(n, hulledges);
    PSLA P;
    compute_lex_smallest_PSLA(&P, n, hulledges, hullsize);
    compute_fingerprint(&P, n);
    printf("%s:\n", fingerprint);
  }
  close(databasefile);
}
```

14 Things to consider

1. The `-exclude` option does not work with the parallelization through *splitlevel*. (This is not currently checked.)

2. Using inverse-PSLA makes *screening* slower! It is only good if combined with screening one level before! Computing *inverse_PSLA* one level before *max_n* costs almost nothing.

3. The *succ* and *pred* arrays could be implemented as one-dimensional arrays. Need to check which is faster.

71 **#define** `SUCC_ALTERNATE`(i, j) $succ[i \ll 4 \mid j]$ /* A shift of 4 is sufficient for **MAXN** $+1 \equiv 16$ */

¶*

Contents

Changed Chunks

The following chunks were changed by the change file: 2, 49, 51, 52, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 72.

Index

List of Refinements