December 11, 2023 at 06:51

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

The purpose of this program is to enumerate abstract order types (sometimes also called generalized configurations or pseudoconfigurations) and their duals, the pseudoline arrangements (PSLAs).

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

We consider the nondegenerate (*simple*) case only: no three points on a line, and no three curves through a point. We abbreviate *abstract order type* by AOT and *oriented abstract order type* by OAOT. (An *oriented* abstract order type can be distinguished from its mirror image.) As a baseline, we consider everything *oriented*, i.e., the mirror object can be isomorphic or not. In the end, we also check for mirror symmetry, and we can choose to report only one orientation of two mirror types.

1.1 Pseudoline arrangements and abstract order types

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.

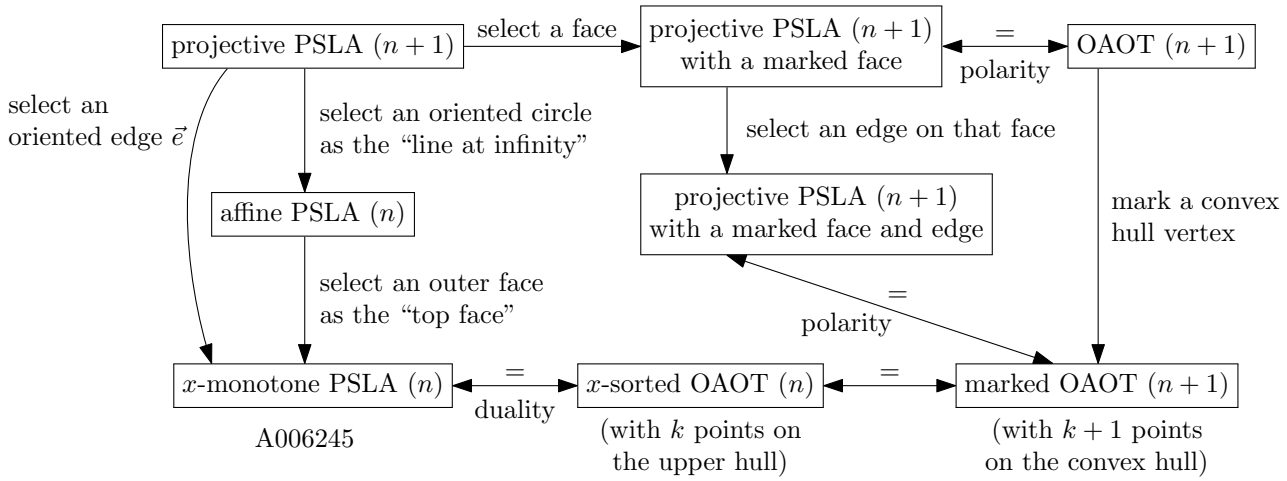


Figure 1: Relations between different concepts. There are different paths from the top left to the bottom right, which apply specialization or geometric reinterpretation in different order.

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 $n = 16$. For example, to get the 158,830 AOTs with 9 points, we go through all 1,232,944 xPSLAs with 8 pseudolines, and select a subset by a lexicographic comparison, see Sections **??** and **??**.

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

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

2 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.

```
3  #define MAXN 15     /* 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 72 ⟩
     return 0;
#endif
#if enumAOT
     ⟨ Initialize statistics and open reporting file 50 ⟩;
     ⟨ Start the generation 15 ⟩;
     ⟨ Report statistics 52 ⟩;
#endif
     return 0;
   }
```

2.1 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, see Section **??**.)

```
4  #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 ≡ 1. */
   #define profile 1     /* gather statistics and profiling information */
```

¶ Type definitions.

```
5  ⟨ Types and data structures 5 ⟩ ≡
     typedef enum { false, true } boolean;
```

See also chunks 11, 62, and 68

This code is used in chunk 3.

¶ Standard libraries

6 ⟨ Include standard libaries 6 ⟩ ≡
```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```
See also chunk 70.

This code is used in chunk 3.

2.2 Auxiliary macro for for-loops

I don't want to write x three times.

7 **#define for_int_from_to**$(x, first, last)$ **for** (**int** $x \leftarrow first;\ x \le last;\ x{+}{+}$)
format *for_int_from_to for*

2.3 Command-line arguments

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

⟨ Global variables 8 ⟩ ≡
```
int n_max, split_level ← 0;
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 12, 18, 37, 48, 49, and 66

This code is used in chunk 3.

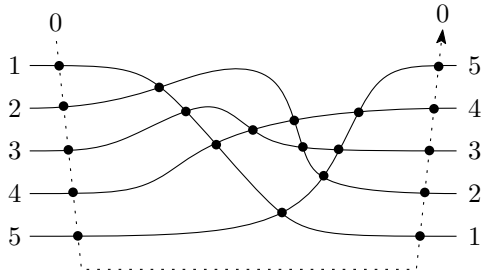
9 ¶⟨ Parse the command line 9 ⟩ ≡
```
if (argc < 2) n_max ← 7;
else {
  if (argv[1][0] ≡ '-') {      /* first argument "--help" gives help message. */
    PRINT_INSTRUCTIONS;
    exit(0);
  }
  n_max ← atoi(argv[1]);
}
printf("Enumeration up to n = %d pseudolines, %d points.\n", n_max, n_max + 1);
if (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 20 ⟩
    }
    else {
      PRINT_INSTRUCTIONS;
      exit(1);
```

```
    }
  }
}
if (argc ≥ 3 + argshift) {
  split_level ← atoi(argv[2 + argshift]);
  if (split_level ≡ 0) {
    if (argv[3 + argshift][0] ≠ '-') fileprefix ← argv[3 + argshift];
    snprintf(fname, sizeof (fname) − 1, "%s-%d.txt", fileprefix, n_max);
    parts ← 1;
  }
  else {
    if (exclude_file_name ≠ 0) {
      printf("The -exclude option with a positive splitlevel %d is not im\
          plemented. Aborting.\n", split_level);
      exit(1);
    }
    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 3.

3 Representations of pseudoline arrangements

Here is an x-monotone pseudoline arrangement with $n = 5$ pseudolines, together with a primitive graphic representation of a different pseudoline arrangement as produced by the function *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.

3.1 The P-matrix (local sequences matrix) and its inverse

Here is a representation of the right example as a two-dimensional array, indicating for each pseudoline i the sequence P_i of crossings with the other lines. These sequences are called the *local sequences*. We will refer to the whole matrix as the P-matrix representation of a PSLA.

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

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 each row. 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.

The binary matrix B is discussed in Section **??**. It is defined in terms of the P-matrix by the rule $B_i[j] = 1$ if $P_i[j] < i$.

11 ⟨ Types and data structures 5 ⟩ +≡
```
typedef int P_matrix [MAXN + 1] [MAXN +1];
```

3.2 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.

12
```
#define SUCC(i, j) succ[i][j]      /* access macros */
#define PRED(i, j) pred[i][j]
#define LINK(j, k1, k2)
        {     /* make crossing with k1 and k2 adjacent on line j */
          SUCC(j, k1) ← k2;
          PRED(j, k2) ← k1;
        }
```
⟨ Global variables 8 ⟩ +≡
```
int succ[MAXN +1][MAXN +1];
int pred[MAXN +1][MAXN +1];
```

4 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 $1, 2, \ldots, n$.

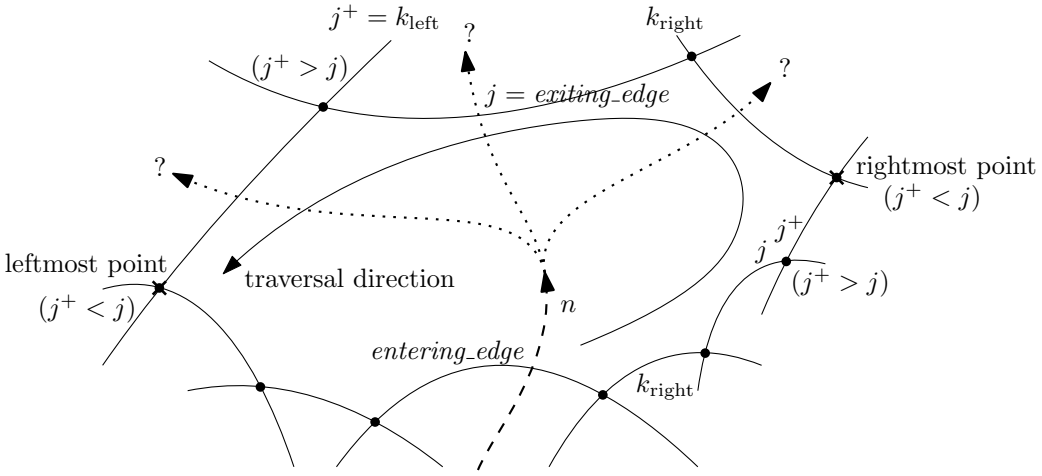


Figure 2: Threading line n through a face

14 ¶⟨ Core subroutine for recursive generation 14 ⟩ ≡
```
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 17 ⟩
      ⟨ Indicate Progress 16 ⟩;
      boolean is_excluded ← false;
      ⟨ Check for exclusion and set the flag is_excluded 19 ⟩
      if (is_excluded) return;
      ⟨ Gather statistics about the AOT, collect output 51 ⟩
      ⟨ Further processing of the AOT 56 ⟩
      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 47 ⟩
          }
          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 3.

¶ Start with 2 pseudolines.

15 ⟨ Start the generation 15 ⟩ ≡

```
  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 3.

16 ¶⟨ Indicate Progress 16 ⟩ ≡

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

This code is used in chunk 14.

17 ¶⟨ Update counters 17 ⟩ ≡

```
  countPSLA[n]++;      /* update global counter ("accession number") */
  localCountPSLA[n]++;      /* update local counter */
```

This code is used in chunk 14.

5 Handling the exclude-file

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 checked.

The array *excluded_code*[3... *excluded_length*] always 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*.

18 ⟨ Global variables 8 ⟩ +≡

```
  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];
```

19 ¶⟨ Check for exclusion and set the flag *is_excluded* 19 ⟩ ≡
```
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 21 ⟩
    ⟨ Determine the matched length matched_length 22 ⟩
  }
}
```
See also chunk 55.
This code is used in chunk 14.

20 ¶⟨ Open the exclude-file and read first line 20 ⟩ ≡
```
exclude_file ← fopen(exclude_file_name, "r");
⟨ Get the next excluded decimal code from the exclude-file 21 ⟩
matched_length ← 2;
```
This code is used in chunk 9.

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

21 ⟨ Get the next excluded decimal code from the exclude-file 21 ⟩ ≡
```
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 19 and 20

¶ (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.)

22 ⟨ Determine the matched length *matched_length* 22 ⟩ ≡
```
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 19.

6 Conversion between different representations

¶

6.1 Convert from linked list to P-matrix

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

25 ⟨ Subroutines 25 ⟩ ≡
```
void convert_to_P_matrix(P_matrix *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 26, 28, 29, 31, 32, 33, 36, 38, 39, 41, 63, 65, 67, 69, 71, and 73

This code is used in chunk 3.

6.2 Convert from linked list to inverse P-matrix

The inverse P-matrix 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 7.

26 ⟨ Subroutines 25 ⟩ +≡

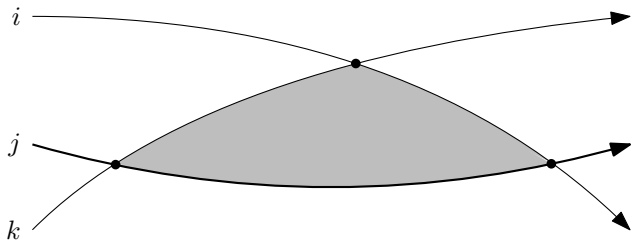
```
  void convert_to_inverse_P_matrix(P_matrix ∗P̄, int n)
  {
    int j ← 1;
    for_int_from_to (i, 0, n) {
      for_int_from_to (p, 0, n − 1) {
        (∗P̄)[i][j] ← p;
        j ← SUCC(i, j);
      }
      j ← 0;      /∗ j starts at 0 except for the very first line. ∗/
    }
  }
```

7 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.



27 **#define** *getOrientation_explicit*$(\bar{P}, i, j, k)$
$(i < j \wedge j < k \;?\; \bar{P}[i][j] > \bar{P}[i][k] : i < k \wedge k < j \;?\; \bar{P}[i][j] > \bar{P}[i][k] : j < i \wedge i < k \;?\; \bar{P}[i][j] < \bar{P}[i][k] :$
$j < k \wedge k < i \;?\; \bar{P}[i][j] > \bar{P}[i][k] : k < j \wedge j < i \;?\; \bar{P}[i][j] > \bar{P}[i][k] : k < i \wedge i < j \;?\; \bar{P}[i][j] < \bar{P}[i][k] : 0)$
#define *getOrientation*$(\bar{P}, i, j, k)$ $((i < j) \oplus (j < k) \oplus (\bar{P}[j][i] > \bar{P}[j][k]))$

8 Compute the convex hull points of an AOT 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.)

28 ⟨ Subroutines 25 ⟩ +≡

```
int upper_hull_PSLA(int n, int *hulledges)
{
  hulledges[0] ← 0;
  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. */
}
```

9 Unique identifiers, 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.

The enumeration tree has only one node on levels 1 and 2. Thus we start the fingerprint at level 3.

(In addition, the PSLAs of each size n are numbered by the *global counter countPSLA*. This can be used as an "accession number" to identify a PSLA, provided that the PSLAs of size n are enumerated in full.)

29 ⟨ Subroutines 25 ⟩ +≡

```
unsigned localCountPSLA[MAXN +3];      /* another global variable */
void print_id(int n)
{
  printf("%d", localCountPSLA[3]);
  for_int_from_to (i, 4, n) printf(".%d", localCountPSLA[i]);
}
```

10 Output

10.1 Prettyprinting of a wiring diagram

Fill a buffer of lines columnwise from left to right.

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

⟨ Subroutines 25 ⟩ +≡

```
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)); }
```

10.2 Fingerprints

A concise description of a PSLA consists of the P-matrix entries, prefixed by the letter `P` and with the rows separated by `!` symbols. The procedure *print_pseudolines_compact* prints a more compact version where the redundant parts, which are the same in all P-matrices or which can be easily inferred from the reamining information, is omitted.

32 ⟨ Subroutines 25 ⟩ +≡

```
void print_pseudolines_short(P_matrix ∗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(P_matrix *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]));
  }
}
```

10.3 A more compact fingerprint

A PSLA is uniquely determined by the $n \times n$ binary matrix B, which is defined in terms of the P-matrix by the rule $B_i[j] = 1$ if $P_i[j] < i$. An example is shown in Section **??**. The fact that this is enough can be seen from the fact that this information is sufficient for drawing the wiring-diagram. It has been shown by Stefan Felsner, On the number of arrangements of pseudolines, *Discrete & Computational Geometry* **18** (1997), 257–267, see also Felsner, *Geometric Graphs and Arrangements*, Vieweg, 2004, Chapter 6. (The so-called *replace matrices* from that paper would offer even more savings.)

The first column of B 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 borders from the original $n \times n$ array.

We code 6 bits into on of 64 ASCII symbols, using the 52 small and capital letters, the 10 digits, and the 2 symbols `+` and `-`. (Care must be taken when sorting such keys with the UNIX `sort` utility, because, depending on the *locale* settings, the sorting program may conflate uppercase and lowercase letters.)

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

33
```
#define FINGERPRINT_LENGTH 30    /*enough for 13 × 13 bits plus terminating null */
⟨ Subroutines 25 ⟩ +≡
  char fingerprint[FINGERPRINT_LENGTH];    /* global variable */
  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(P_matrix *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 ⟩ ≡

```
{
  P_matrix P;
  convert_to_P_matrix(&P, n);
  compute_fingerprint(&P, n);
  printf("%s:", fingerprint);      /* terminated by a colon */
}
```

This code is used in chunk 57.

11 Abstract order types

11.1 Compute the P-matrix for a different starting edge

For reference we show how to compute the matrix from another *starting edge*. The starting edge is specified by the line *line0* on which it lies, its right endvertex *right_vertex*, and a a *direction*. The edge lies between *left_vertex* $\equiv$ PRED(*line0*, *right_vertex*) and *right_vertex*, (which fulfills *right_vertex* $\equiv$ SUCC(*line0*, *left_vertex*)). The direction is in the direction of the *succ*-pointers if *reversed* $\equiv$ *false* and in the direction of the *pred*-pointers if *reversed* $\equiv$ *true*. The P-matrix is filled row-wise from right to left.

The main application of this procedure is when we try out different convex hull vertices as pivot points, generating all PSLAs that can represent a given AOT, see Section **??**. (However, we will never use the procedure *compute_new_P_matrix* directly, we use a version that computes several such P-matrices in parallel, entry by entry.)

36 ⟨ Subroutines 25 ⟩ +≡

```
void compute_new_P_matrix(P_matrix *P, int n, int line0, int right_vertex, boolean reversed)
{
  int Sequence[MAXN +1];
    /* Sequence[p] gives the p-th crossing (in the SUCC-direction) on line start_line. */
  int new_label[MAXN +1];      /* new_label[j] gives the label that is use for the line with the original
      label j. */      /* Sequence and new_label are inverse permutations of each other. */
  new_label[line0] ← 0;
  int i ← right_vertex;
  for_int_from_to (p, 1, n) {
    new_label[i] ← p;
    Sequence[p] ← i;
    i ← SUCC(line0, i);
  }
  for_int_from_to (q, 0, n − 1) (*P)[0][q] ← q + 1;      /* row 0 is always the same */
  for_int_from_to (p, 1, n) {      /* compute row P_p of P-matrix */
    int pos ← reversed ? n + 1 − p : p;
    (*P)[p][0] ← 0;
    int i ← Sequence[pos];
    int j ← line0;
    for_int_from_to (q, 1, n − 1) {      /* Compute P_{p,n−q} */
      j ← reversed ? SUCC(i, j) : PRED(i, j);
      (*P)[p][n − q] ← new_label[j];
    }
  }
}
```

11.2 Lexicographically smallest P-matrix representation

In order to generate every AOT only once, we check whether the the current P-matrix P the smallest among all P-matrices P' that represent the same AOT, except that the AOT is rotated or reflected.

We have to try all convex hull points as pivot points, and for each pivot point we have to choose two directions (reflected and unreflected). 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!)

In the lexicographic comparison between PSLAs, we consider the elements of the P-matrix row-wise *from right to left*, i.e., in the order $P_{1n}, P_{1,n-1}, \ldots, P_{11}; P_{2n}, P_{2,n-1}, \ldots, P_{21}; \ldots$, assuming that the entries in each row are numbered from 1 to n (unlike in the C program). In comparison with the more natural left-to-right order, this gives, experimentally, a quicker way to eliminate tentative P-matrices than the left-to-right order.

37 ⟨ Global variables 8 ⟩ +≡
```
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 the index that is use for the (original) line j. */
int candidate[2*(MAXN +1)];      /* list of candidates, gives the 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];
```

¶ The label arrays are not computed for those candidates that are excluded by the comparison of the *P_1_n_forward* values (unless the flag *compute_all* is set).

38 ⟨ Subroutines 25 ⟩ +≡
```
void prepare_label_arrays(int n, int *hulledges, int hullsize, boolean compute_all)
{
  for_int_from_to (r, 0, hullsize − 1)
    if (compute_all ∨ 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);
      }
    }
}
```

11.3 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.

If the test returns *true*, the procedure also sets some output parameters that characterize the symmetry of the AOT: These output parameters — *rotation_period*, *has_mirror_symmetry*, and *has_fixed_vertex* — are determined on the way as an easy side result.

has_fixed_vertex 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.

39 ⟨ Subroutines 25 ⟩ +≡
```
void compute_lex_smallest_P_matrix(P_matrix *P, int n, int *hulledges, int hullsize)
{
  for_int_from_to (q, 0, n − 1) (*P)[0][q] ← q + 1;      /* row 0 */
  prepare_label_arrays(n, hulledges, hullsize, true);
  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 P-matrix */
    (*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 40 ⟩
    }
    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 40 ⟩
    }
    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.

40 ⟨ Process candidate c, keep in list and advance *new_candidates* if equal; reset *new_candidates* if better value than *current_min* 40 ⟩ ≡

```
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 39.

11.4 Test if the current PSLA gives the lex-smallest P-matrix corresponding to the same AOT

This is a variation of the procedure *compute_lex_smallest_P_matrix*. The output parameters *rotation_period*, *has_mirror_symmetry*, *has_fixed_vertex*, which characterize the symmetry of the AOT, are determined on the way, as an easy side result.

As a speed-up, there is a fast screening procedure that tries to eliminate a few candidates in advance.

41 ⟨ Subroutines 25 ⟩ +≡

```
⟨ Screening procedures 45 ⟩
```

```
boolean is_lex_smallest_P_matrix(int n, int *hulledges, int hullsize, int *rotation_period, boolean
        *has_mirror_symmetry, boolean *has_fixed_vertex)
{
  if (¬screen(n, hulledges, hullsize)) return false;
#if profile
  numTests++;
#endif
  prepare_label_arrays(n, hulledges, hullsize, false);
  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 P-matrix */
    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 in every matrix, row P_p is a permutation of the same elements. If all
        elements except the last one agree, then the last one must also agree.. */
      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 42 ⟩
      }
      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 42 ⟩
      }
      numcandidates ← new_candidates;
      if (numcandidates ≡ 0) {      /* early return */
        *rotation_period ← hullsize;
        *has_mirror_symmetry ← false;
        return true;
      }
    }
  }
  ⟨ Determine the result parameters rotation_period, has_mirror_symmetry, has_fixed_vertex, from the set of
      remaining candidates. 43 ⟩
  return true;
}
```

¶ The current candidate is *successful* if its value $P_{p,n-q}$ agrees with the *target_value*, the value of $P_{p,n-q}$ in the matrix P^0.

42 ⟨ Process candidate c, keep in list and advance *new_candidates* if successful; return *false* if better value than *target_value* is found 42 ⟩ ≡

```
#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 41.

43 ¶⟨ Determine the result parameters *rotation_period*, *has_mirror_symmetry*, *has_fixed_vertex*, from the set of remaining candidates. 43 ⟩ ≡

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

This code is used in chunk 41.

12 Fast screening of candidates

Suppose we don't have correct labels, but we only known line 0 and line 1. We can still determine the upper right corner P_{1n} of the P-matrix, as follows (see Figure 3b).

We find $i' \leftarrow \texttt{PRED}(0,1)$; This is line n. The $\texttt{PRED}(i',0)$ would represent the last intersection on line n, which is the value of P_{1n} that we want, except that we don't have the correct label. We can recover this label by walking along line 0, using the SUCC labels, until we hit line i'.

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

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. $Q(0,1)$.

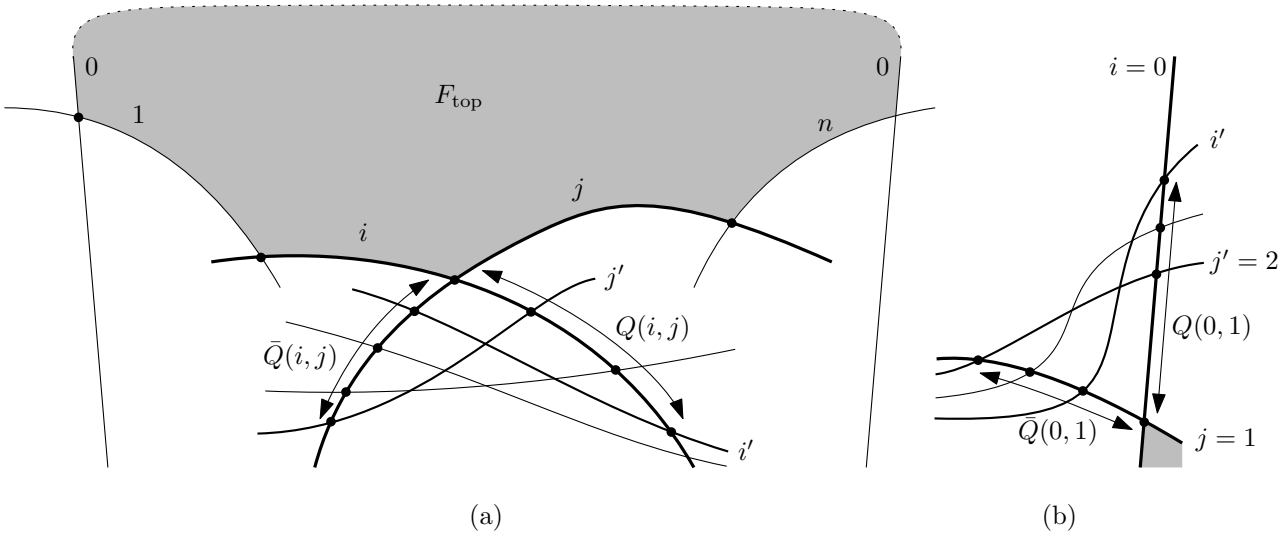


Figure 3: (a) An example with $Q(i,j)=4$ and $\bar{Q}(i,j)=5$; (b) an example with $Q(0,1)=\bar{Q}(0,1)=4$

¶ 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.

45 ⟨ Screening procedures 45 ⟩ ≡

```
boolean screen(int n, int *hulledges, 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;
}
```

This code is used in chunk 41.

12.1 More aggressive 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 $1 \times \texttt{PRED}(1,0)$, 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 $1 \times \texttt{PRED}(1,0)$, then n becomes the new $i' = \texttt{PRED}(1,0) = Q(0,1) = P_{1n}$, 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.

47 ¶⟨ Screen one level below level *n_max* 47 ⟩ ≡

```
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 14.

¶ We maintain statistics about the effectiveness of this test:

48 ⟨ Global variables 8 ⟩ +≡
```
long long unsigned cpass, csaved;
```

13 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).

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];
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 3.

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;
    }
    classcount[n_points][hullsize][rotation_period][¬has_mirror_symmetry ? NO_MIRROR : has_fixed_vertex ?
        MIRROR_WITH_FIXED_VERTEX : MIRROR_WITHOUT_FIXED_VERTEX]++;
  }
#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.

¶ First some basic statistics are written in tabular form to the terminal:

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));
```

See also chunk 53.

This code is used in chunk 3.

¶ 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  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)
            if (classcount[n][k][p][t])
              fprintf(reportfile, "%c %d %d %d %d  %Ld\n", c, n, k, p, t, classcount[n][k][p][t]);
    }
    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);
  }
```

14 Special problem-specific extensions

Program extensions for special purposes can be added here: The following data are available:
lex_smallest ...
The AOT has $n+1$ points; its convex hull has *hullsize* vertices and is stored in the array *hulledges*. ....
lex_smallest
in the *succ* and *pred* arrays
P-matrix is / is not available.

14.1 Further exlusion criteria

If some PSLAs and their subtrees should not be considered, they can be filtered here, by setting *is_excluded* to *false*.

55 ⟨ Check for exclusion and set the flag *is_excluded* 19 ⟩ +≡ /* Currently no further exclusion tests. */

14.2 Further processing of AOTs

Problem-specific processing can be added here.

56 ⟨ Further processing of the AOT 56 ⟩ ≡ /* Currently no further processing of the AOT. */
See also chunks 57, 58, and 60
This code is used in chunk 14.

¶ After computing the inverse P-matrix, one can perform a few tests on the order type, using orientation queries.
There are a couple of test programs. ...

57 ⟨ Further processing of the AOT 56 ⟩ +≡
```
#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
```

¶ The following test program compares the orientation queries against an explicitly computed three-dimensional Λ-matrix (see Section **??**).

58 ⟨ Further processing of the AOT 56 ⟩ +≡
```
#if 0
  P_matrix P̄;    /* the orientation test is computed from this array. */
  convert_to_inverse_P_matrix(&P̄, n);
  small_lambda_matrix S;
  convert_to_small_lambda_matrix(&S, n_points);
  large_Lambda_matrix L;
  convert_small_to_large(&S, &L, n_points);
  ⟨ Compare orientation tests 59 ⟩
#endif
```

59 ¶⟨ Compare orientation tests 59 ⟩ ≡
```
  {
    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);
              }
    ;
  }
#if 0
  if (n ≡ n_max ∧ countPSLA[n] ≡ 50) {      /* print "some" example */
    P_matrix PP, invPP;
    convert_to_P_matrix(&PP, n);
    convert_to_inverse_P_matrix(&invPP, n);
    print_pseudolines_short(&PP, n);
    printf("inverse␣");
    print_pseudolines_short(&invPP, n + 1);
    print_wiring_diagram(n);
  }
#endif
```

This code is used in chunk 58.

¶ Various further test programs.

60 ⟨ Further processing of the AOT 56 ⟩ +≡

```
#if 0     /* estimate size of possibly subproblems for d&c Ansatz */
#define MID 5
  if (n ≡ 2 * MID − 2) {
    P_matrix P;
    convert_to_P_matrix(&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
```

15 Other representations of abstract order types: λ-matrices and and Λ-matrices

¶ More type definitions.

62 ⟨ Types and data structures 5 ⟩ +≡

```
typedef boolean large_matrix_entry;
typedef unsigned small_matrix_entry;
typedef small_matrix_entry small_lambda_matrix [MAXN +1][MAXN +1];
typedef large_matrix_entry large_Lambda_matrix [MAXN +1][MAXN +1][MAXN +1];
```

15.1 ("Small") λ-matrices

Input: PSLA with n lines $1 \ldots n$ 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.

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

⟨ Subroutines 25 ⟩ +≡

```
void convert_to_small_lambda_matrix(small_lambda_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);
    }
  }
}
```

15.2 ("Large") Λ-matrices

The three-dimensional Λ-matrix stores the orientation of all triples.

In this program, entries Λ_{ijk} are only ever accessed for $i < j < k$. It would be possible to save space by a more elaborate indexing function into a one-dimensional array. More general access could then be provided by a macro *get_entry_large*.

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

¶ 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. The input is a λ-matrix A. The result is stored in B.

65 ⟨ Subroutines 25 ⟩ +≡

```
void copy_small(small_lambda_matrix *A, small_lambda_matrix *B, int n)
{
  for_int_from_to (i, 0, n − 1)
    for_int_from_to (j, 0, n − 1) entry_small(*B, i, j) ← entry_small(*A, i, j);
}
void convert_small_to_large(small_lambda_matrix *A, large_Lambda_matrix *B, int n)
{
  small_lambda_matrix Temp;
  copy_small(A, &Temp, n);      /* the small matrix Temp will be destroyed */
  for_int_from_to (k, 0, n − 1)
    for_int_from_to (i, k + 1, n − 1)
      for_int_from_to (j, i + 1, n − 1)      /* k < i < j */
      {
        boolean plus ← entry_small(Temp, i, k) < entry_small(Temp, j, k);
        if (plus) entry_small(Temp, i, j)−−;
        else entry_small(Temp, j, i)−−;
        (*B)[k][i][j] ← (*B)[i][j][k] ← (*B)[j][k][i] ← plus;
        (*B)[k][j][i] ← (*B)[i][k][j] ← (*B)[j][i][k] ← ¬plus;
      }
}
```

16 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.

66 ⟨ Global variables 8 ⟩ +≡

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

16.1 Orientation test for points

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

67 ⟨ Subroutines 25 ⟩ +≡

```
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.

68 ⟨ Types and data structures 5 ⟩ +≡

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

16.2 Turn point set with coordinates into PSLA

We insert the lines one by one into the arrangement. This is similar 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.

69 ⟨ Subroutines 25 ⟩ +≡

```
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;
  }
}
```

16.3 Select the order-type files to be read

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`.

70 ⟨ Include standard libaries 6 ⟩ +≡
```
#include <fcntl.h>
#include <unistd.h>
```

71 ¶⟨ Subroutines 25 ⟩ +≡
```
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. */
  }
}
```

72 ¶⟨ Read all point sets of size $n\_max + 1$ from the database and process them 72 ⟩ ≡
```
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 3.

16.4 Do the actual reading

Open and read database file and process the input points.

73 ⟨ Subroutines 25 ⟩ +≡
```
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);
    int hulledges[MAXN +1];
    int hullsize ← upper_hull_PSLA(n, hulledges);
    P_matrix P;
    compute_lex_smallest_P_matrix(&P, n, hulledges, hullsize);
    compute_fingerprint(&P, n);
    printf("%s:\n", fingerprint);
  }
  close(databasefile);
}
```

17 Things to consider

1. The `-exclude` option does not currently work with the parallelization through *splitlevel*. (This combination of input parameters is checked.)
2. Does the enumeration of PSLAs work in constant amortized time (CAT)?
3. Enumerate PSLAs for which the corresponding AOT has a given symmetry. In connection with the PSLAs without regard to symmetries, which are known up to 16 lines (17 points), this would lead to counts of AOTs with up to 17 points without much computational effort. (The current record is 13 points).
4. *Entropy encoding* of PSLAs?
5. 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.
6. The *succ* and *pred* arrays could be implemented as one-dimensional arrays. Need to check which is faster.

```
74 #define SUCC_ALTERNATE(i, j) succ[(i) ≪ 4 | (j)]      /* A shift of 4 is sufficient for MAXN +1 ≡ 16 */
```

¶

Contents

Index

List of Refinements