1. An Aztec diamond bijection. This little program is meant to verify an algorithmic bijection, proposed by Frédéric Bosio, between on one hand families of certain lattice paths across a $n \times n$ square, and on the other hand triangular arrays of side length n of Boolean values (giving $\frac{n(n+1)}{2}$ bits in all). It also produces PostScript output for configurations obtained by this procedure, possibly generated at random

To fix ideas geometrically, we use swapped Cartesian coordinates: the first coordinate increases upwards and the second to the right. Therefore the first coordinate determines the "row" of a point, and the second coordinate its "column", as in matrices, but columns grow upwards. When producing PostScript output, we shall take care to inverse the order of coordinates.

⟨ Type definitions 2 ⟩
⟨ Function definitions 3 ⟩

2. A class for path families. Our lattice paths will come in families of n, path P_i for $0 \le i < n$ going from $(0,i)$ on the vertical axis to $(i,0)$, on the horizontal axis, with steps that either increase the first coordinate, or decrease the second, or both. Path P_0 has no steps, but occupies the point $(0,0)$; all paths in our family are supposed to be disjoint at the end of the construction. Path P_i is determined by i bits in $B[i]$ and $i+1$ natural numbers in $D[i]$: the former describe whether the step between columns j and $j+1$ is horizontal (value 0, **false**) or diagonal (value 1, **true**) for $0 \le j < i$, while the latter count the number of vertical steps in column j, $0 \le j \le i$. We provide $step(i,j)$ as a read-only way to refer to $B[i][j]$ viewed as value in $\{0,1\}$; it is the level decrease in path P_i going from column j to $j+1$. The sum of all values in $B[i]$ and $D[i]$, which are associated to P_i, should be i (this is tested among other things by the *valid* method), so that along that path the second coordinate decreases for i to 0.

We provide basic manipulators *untangle*, that (under specific conditions) modifies P_i and P_{i+1} in a special way that will ensure they become disjoint, and that the original paths can be recovered from the modified one, and its inverse operation *cliffify* (so called because it moves any vertical steps in column k from P_{i+1} to P_i, which when iterated down to $k = i$ will make P_k end with a sheer vertical drop). The latter requires and modifies the values of two additional quantities that it would otherwise need to laboriously calculate each time each time, namely the levels h_0 and h_1 where the paths P_i and P_{i+1} enter into column k.

To produce output from a **family** we provide the methods *flex_points* for display as a path family, which lists all points where paths start, end, or change direction, and *Aztec_tiling* for display as a tiling of the Aztec diamond, which lists the orientations of all dominoes listed according to their black squares.

```
#include <vector>
#include <iostream>
```

⟨ Type definitions 2 ⟩ ≡

```
typedef std::vector<unsigned> vec;
typedef std::vector<bool> bitvec;
typedef std::vector<bitvec> triangle;
class family
{ unsigned n;
  triangle B;   // B[i][j] gives direction of P_i when leaving column j
  std::vector<vec> D;   // D[i][j] counts vertical steps of P_i in column j
public:
  family(unsigned nn);   // nn determines max_path()
  family(const triangle &tri);   // set from triangle tri of bits

  unsigned int n_paths() const { return n; }
  unsigned int max_path() const { return n − 1; }
  unsigned int step (unsigned int i, unsigned int j) const
  { assert(i > j); return unsigned(B[i][j]); }
  bool operator≠ (const family &y) const;
  bool valid() const;
  bool disjoint (unsigned i) const;   // whether paths P_i and P_{i+1} are disjoint

  bool untangle (unsigned i, unsigned k);   // paths P_i, P_{i+1} up to column k; anything changed?
  void cliffify (unsigned i, unsigned k, unsigned &h_0, unsigned &h_1);   // inverse

  std::vector<std::vector<std::pair<unsigned, unsigned>>> flex_points() const;
  std::vector<vec> Aztec_tiling() const;   // convert to (n − 1) × n matrix of domino orientations
};
```

See also section 13.

This code is used in section 1.

3. When constructing a **family** value, the vectors $B[i]$ and $D[i]$ for path P_i are dimensioned to allow column indices form 0 to i, exclusive for B and inclusive for D. In practice one will always have $D[i][0] = 0$ so that we could do with one entry less for each vector $D[i]$, but it is not worth the complications this causes to do so. More generally each path that reaches column j will decrease its second coordinate by at most j up to that column inclusive, so that all paths are Schröder paths. Externally we use as value to parameterise this class the *size* of the square across which the paths run (of which the end points of P_{n-1} span a diagonal, and which is the value of *max_path*( )), that is $nn = n - 1$, and it is therefore that number that is passed to the constructor. The number nn is also the order of the corresponding Aztec diamond.

⟨ Function definitions 3 ⟩ ≡

```
family::family(unsigned nn) : n(nn + 1), B(n), D(n)
{ for (unsigned i ⇐ 0; i < n; i++)
    { B[i].resize(i, true); D[i].resize(i + 1, 0); }
}
```

See also sections 4, 5, 6, 7, 8, 9, 12, 14, 15, 16, 17, 18, 25, and 26.

This code is used in section 1.

4. When constructing from a triangular array of Boolean values, with array i having size i for $0 \leq i < n$, these values are used to determine the path from $(0, i)$ in column 0 until reaching column i, taking horizontal steps and diagonal steps only; for each **false** value the step will be horizontal, and diagonal for a **true** value. A number of vertical steps equal to the number of horizontal steps remains at the end in column i.

```
#include <algorithm>
```

⟨ Function definitions 3 ⟩ +≡

```
family::family(const triangle &tri) : n(tri.size()), B(tri), D(n)
{ for (unsigned i ⇐ 0; i < n; ++i)
  { D[i].resize(i + 1); std::fill(D[i].begin(), D[i].end() − 1, 0);    // set off-diagonal entries of D to 0
    unsigned s ⇐ i;
    for (unsigned j ⇐ 0; j < i; ++j)
      s −⇐ step(i, j);    // count horizontal steps (total minus diagonal ones)
    D[i][i] ⇐ s;    // number of vertical steps at end of P_i matches that of horizontal ones
  }
}
```

5. The method *disjoint* tells whether a pair of successive paths P_i, P_{i+1}, have no points in common. The method *valid* tells whether the whole family is valid, which means all paths are disjoint, and end at level 0.

⟨ Function definitions 3 ⟩ +≡

```
bool family::disjoint (unsigned i) const
{ unsigned s ⇐ 0, t ⇐ D[i + 1][0];    // in fact t = 0, but morally it is D[i + 1][0]
  for (unsigned j ⇐ 0; j < i and t ≤ s; ++j)
    { t +⇐ step(i + 1, j) + D[i + 1][j + 1]; s +⇐ D[i][j] + step(i, j); }
  return t ≤ s;    // if at any point we have t > s, we return false immediately
}
bool family::valid() const
{ for (unsigned i ⇐ 0; i < n; ++i)
  { if (i < n − 1 and not disjoint(i)) return false;
    unsigned s ⇐ D[i][0];
    for (unsigned j ⇐ 0; j < i; ++j) s +⇐ step(i, j) + D[i][j + 1];
    if (s ≠ i) return false;
  }
  return true;
}
```

6. The method *untangle* below is the first element used to define the bijection: it modifies a pair of successive paths P_i, P_{i+1}, such that afterwards *disjoint*(i) holds, provided P_i and P_{i+1} were already disjoint beyond column k (this explains the name *untangle*, but it must satisfy much more than this, in particular the method should be invertible). It is not intended to be used on paths of general form: it is assumed that no vertical steps are present in these paths in columns $j < k$, and for P_{i+1} none in column k either. Moreover, it is assumed that P_i has *enough* vertical steps in column k, namely at least the value of *depth* as computed at the end of the function below. Only columns up to column k are considered or altered in this method. After the method has operated some vertical steps of path P_i in column k will be transfered to path P_{i+1}, while remaining in column k; their number is the final value of *depth*, whence the mentioned condition is necessary to ensure that P_i remains a valid path.

If one considers the paths up to the point where they first enter column k, they only have horizontal and diagonal steps, and by taking the difference between the two paths, we classify the situation in three types based on the value of $\mathit{step}(i+1,j) - \mathit{step}(i,j) \in \{-1, 0, +1\}$. When this value is equal to $+1$ the paths approach, or if already crossed increase their crossing: path i is horizontal and P_{i+1} diagonal. When this difference is 0, the paths evolve in parallel: either both are horizontal or both are diagonal. And when the value is -1 the paths move apart or if currently crossed decrease their amount of crossing: P_i is diagonal and P_{i+1} horizontal. Define the "current crossing" in column j to be the sum of these differences $\mathit{step}(i+1,j') - \mathit{step}(i,j')$ for $0 \le j' < j$. The variable *depth* records the maximal crossing level seen so far. Calling *untangle*(i,k) will interchange, only for those steps where *depth* increases (necessarily a situation where $\mathit{step}(i+1,j) = 1$ and $\mathit{step}(i,j) = 0$), the directions of the steps in both paths, so P_i becomes diagonal and P_{i+1} becomes horizontal at these places.

#include `<cassert>`

⟨ Function definitions 3 ⟩ +≡

```
bool family::untangle (unsigned i, unsigned k)
{ assert(k ≤ i);
  signed int cur ⇐ 0; unsigned int depth ⇐ 0;
  for (unsigned j ⇐ 0; j < k; ++j)
  { cur +⇐ step(i + 1, j) − step(i, j);
    if (cur > signed(depth))
    { depth ⇐ cur;    // increase depth whenever cur rises above it
      B[i][j] ⇐ true; B[i + 1][j] ⇐ false;    // P_i gets a diagonal step, P_{i+1} a horizontal step
    }
  }
  assert(depth ≤ D[i][k]);    // we did not remove more vertical steps than were present in P_i
  assert(D[i + 1][k] = 0);    // and none were present in P_{i+1}
  D[i][k] −⇐ depth; D[i + 1][k] ⇐ depth;    // transfer depth vertical steps to P_{i+1}
  return depth > 0;
}
```

7. The method *cliffify* is the inverse operation of *untangle*, whenever one of them is called with the necessary conditions satisfied (this implies in particular the conditions for the other operation will be satisfied whenever one operation has finished). Like for the call $untangle(i,k)$, when calling $cliffify(i,k,h_0,h_1)$ it is assumed that any vertical steps that occur in paths P_i or P_{i+1} are in columns $j \geq k$ (but now vertical steps in column k can appear both in P_i and in P_{i+1}). Moreover P_i and P_{i+1} are assumed to be disjoint initially, up to column k inclusive. The call will not inspect of modify anything in columns $j > k$; after the call the vertical steps in column k appear only in P_i (while their total number does not change), but the paths P_i and in P_{i+1} may intersect in any of the columns $j \leq k$ (and they *will* intersect if any modification was made, which happens if and only if there were any vertical steps of P_{i+1} in column k).

The idea for defining *cliffify* is simple: the final value of *depth* in the call $untangle(i,k)$ is saved as the number of vertical steps of P_{i+1} in column k, and if we can retrace the values of *depth* for all previous columns, then it will be easy to restore the paths P_i and P_{i+1} to their state before applying $untangle(i,k)$. Every time *depth* was increased in $untangle(i,k)$, say at column j, the positive quantity $\sum_{j'<j}(step(i+1,j')-step(i,j'))$ rises when replacing j by $j+1$, and *it will never descend to the value it had at j when further increasing j*. This means that when scanning in the reverse direction, and assuming we have computed in *cur* and *depth* the values of this summation respectively of the variable *depth* as they were in the forward direction at $j+1$, we can easily spot when *depth* needs decreasing when advancing (downwards) to j, namely if $cur = depth$ and $step(i+1,j) - step(i,j) = -1$. If we decrease *depth* only in that case, and change *cur* by adding $step(i+1,j) - step(i,j)$ in all cases, then we shall find the correct values of *cur* and *depth* for each column, and moreover $cur \geq depth$ after every iteration. Note that contrary to *depth*, the variable *cur* does not retrace the values of the variable of the same name in *untangle*; in particular we can take *cur* to be **unsigned** here, but not in *untangle*. Since *cur* becomes 0 at $j = 0$, one is bound to find *depth* decreasing all the way down to 0 at some point. It is clear that nothing will change to the paths anymore once $depth = 0$, so we return from this method immediately when that happens.

⟨Function definitions 3⟩ +≡
```
void family::cliffify (unsigned i, unsigned k, unsigned &h0, unsigned &h1)
{ assert(k ≤ i);
  unsigned depth ⇐ D[i + 1][k];    // pick up final value of depth
  unsigned cur ⇐ h1 − h0 − 1;
  assert(h1 > h0 and depth ≤ cur);    // Pi and Pi+1 are initially disjoint in column k
  if (depth = 0) return;    // nothing needs to be done in this case
  D[i + 1][k] ⇐ 0; D[i][k] +⇐ depth;    // transfer depth vertical steps to Pi
  h1 −⇐ depth; h0 +⇐ depth;    // and adapt the entry points into column k correspondingly
  for (unsigned j ⇐ k; j−− > 0; )
  { cur +⇐ step(i + 1, j) − step(i, j);
    if (cur < depth)
    { depth ⇐ cur;    // decrease depth when cur first descends below it
      B[i][j] ⇐ false; B[i + 1][j] ⇐ true;    // Pi gets a horizontal step, Pi+1 a diagonal step
      if (depth = 0) return;    // and if it reaches 0, nothing is left to do
    }
  }
  assert(cur = 0);    // shouldn't be reached, since depth also descended to 0, causing return
}
```

8. The function *to_disjoint* takes a configuration as obtained after constructing from a **triangle** of bits, and transforms it into a disjoint family of paths; the main purpose of our program is to test that this is indeed the case, and that the original configuration can be recovered from it. Since it is known from a determinant evaluation that there are exactly $2^{\binom{n}{2}}$ disjoint families of paths of the kind we consider, this one-sided verification shows that we do indeed have a bijection.

For integrating path P_k into the partial family $\{P_{k+1},\ldots,P_{n-1}\}$, already made disjoint, it suffices to call $untangle(k,k)$, $untangle(k+1,k)$, …, $untangle(n-1,k)$. All of these calls stop at column k, and the call $untangle(i,k)$ may move some vertical steps in column k from path P_i to path P_{i+1}. The result of this call is certainly to make P_i and P_{i+1} disjoint; the one point to prove is that it cannot make P_{i-1} and P_i (which were disjoint at that point) intersecting again, which we shall do below.

The method *to_cliffs* is a straightforward inverse of *to_disjoint*. While in *to_disjoint* any further action in the inner loop serves to "repair" the effect of the previous action, so that we can stop once nothing happens, this is not the case with *to_cliffs*, where on the contrary action increases during the inner loop.

```
⟨ Function definitions 3 ⟩ +≡
  void to_disjoint (family &f)
  { unsigned n ⇐ f.n_paths( );
    for (unsigned k ⇐ n − 1; k−− > 0; )      // treat columns from the last down to first
      for (unsigned i ⇐ k; i < n − 1; ++i)
      { if (not f.untangle(i, k))
            break;     // separate P_i and P_{i+1} in column k; stop if nothing changed
        if (i > k) assert(f.disjoint(i − 1));      // no ricochet of P_i into P_{i−1}
      }
    assert(f.valid( ));
  }
  void to_cliffs (family &f)
  { assert(f.valid( ));
    unsigned n ⇐ f.n_paths( ); vec h(n);
    for (unsigned k ⇐ 0; k < n − 1; ++k)
      for (unsigned i ⇐ n; i−− > k; )
      { h[i] ⇐ k = 0 ? i : h[i] − f.step(i, k − 1);      // there should be no vertical steps in column k − 1
        if (i < n − 1) f.cliffify(i, k, h[i], h[i + 1]);      // move vertical steps in column k from P_{i+1} to P_i
      }
  }
```

9. Finally we shall need to test for inequality of families, which is easy.

```
⟨ Function definitions 3 ⟩ +≡
  bool family::operator≠ (const family &y) const
  { return B ≠ y.B or D ≠ y.D; }
```

10. A proof that disjointness is achieved. It is fairly obvious that calling *untangle*(i) achieves disjointness of P_i and P_{i+1} in all columns $j < k$, since in that method *cur* measures, at iteration j, the amount by which this condition is violated initially, and both paths are moved away from each other in that column by the value of *depth* $\geq$ *cur* at that iteration (so the violation is *doubly* corrected). However the bijectivity (and well-definedness) of the construction requires that after calling *untangle*(k,k), *untangle*$(k+1, k)$, …, *untangle*$(n-1,k)$ in the inner loop of *to_disjoint* above (or an initial part of those, the last of which returns **false** to indicate that it changed nothing), the paths $P_k, \ldots, P_{n-1}$ are disjoint. This proof has two parts, one that establishes a somewhat sharpened version of the disjointness property that the call *untangle*(i, k) obtains, and a second part that uses this to show that after the pair of calls *untangle*(i,k); *untangle*$(i+1, k)$, the paths P_i and P_{i+1} are (still) disjoint up to column k inclusive.

Let numbers $i \geq k$ be fixed, and assume our **family** is in a state in which *untangle*(i,k) may be called, namely with $D[i][k] \geq \sum_{j'<j}(\mathit{step}(i+1,j') - \mathit{step}(i,j))$ for all $j \leq k$, and $D[i+1][k] = 0$. Put

$$a_j = i - \sum_{j'<j} \mathit{step}(i,j') \quad \text{and} \quad b_j = i+1-\sum_{j'<j} \mathit{step}(i+1,j') \qquad \text{for } 0 \leq j \leq k,$$

the levels at which P_i and P_{i+1} initially enter column j, and let a'_j, b'_j be the corresponding values after *untangle*(i,k) is called. Then one has

$$a'_j < b_j \leq b'_j \quad \text{and} \quad a'_j \leq a_j < b'_j \qquad \text{for } 0 \leq j \leq k.$$

This follows from the fact that the value depth_j of *depth* in *untangle*(i,k) after its possible adjustment in iteration j of the inner loop, satisfies $\mathit{depth}_j > a_j - b_j$ as well as $\mathit{depth}_j \geq 0$, and that the call *untangle*(i,k) sets $a'_j = a_j - \mathit{depth}_j$ and $b'_j = b_j + \mathit{depth}_j$.

11. The inequality $a'_j < b'_j$ shows that *untangle*(i,k) has succeeded in making P_i and P_{i+1} disjoint. We wish to prove that the next call *untangle*$(i+1,k)$, which may lower P_{i+1}, cannot cause them to be intersecting again. Let $c_j = i+2-\sum_{j'<j} \mathit{step}(i+2,j')$ be the level at which P_{i+2} initially enters column j, for $0 \leq j \leq k$. The assumption that P_{i+1} and P_{i+2} are disjoint in column j at that time gives $b_j+1 \leq c_j$. Let depth_j as above denote the value of this variable in column j during the call of *untangle*(i,k), and depth'_j its corresponding value during the call of *untangle*$(i+1,k)$. One has $b'_j = b_j + \mathit{depth}_j$, and the level b''_j at which P_{i+1} enters column j after the call of *untangle*$(i+1,k)$ is given by $b''_j = b'_j - \mathit{depth}'_j$. If we can prove $\mathit{depth}'_j \leq \mathit{depth}_j$ for $0 \leq j \leq k$, then it will follow that $a'_j < b_j \leq b_j + \mathit{depth}_j - \mathit{depth}'_j = b''_j$, which means that P_i and P_{i+1} are disjoint in columns $j < k$ after the call *untangle*$(i+1,k)$. One has $\mathit{depth}_0 = 0 = \mathit{depth}'_0$, so we may assume that $j > 0$, and by induction on j that $\mathit{depth}'_{j-1} \leq \mathit{depth}_{j-1}$. Now the algorithm of *untangle*$(i+1,k)$ sets $\mathit{depth}'_j = \max(\mathit{depth}'_{j-1}, b'_j+1-c_j)$, and by the induction hypothesis one has $\mathit{depth}'_{j-1} \leq \mathit{depth}_{j-1} \leq \mathit{depth}_j$, so it remains to prove that $b'_j + 1 - c_j \leq \mathit{depth}_j$. But since $b'_j = b_j + \mathit{depth}_j$ this is equivalent to $b_j + 1 \leq c_j$, the initial disjointness of P_{i+1} and P_{i+2} mentioned above.

So P_i and P_{i+1} are disjoint in columns $j < k$ after the call *untangle*$(i+1,k)$, and will remain so after any further calls *untangle*(i',k) with $i' > i+1$. In column k the concern is slightly different due to the presence of vertical steps. The level at which path P_{i+1} *leaves* this column is unchanged, so provided that *untangle*(i,k) makes P_i and P_{i+1} disjoint in column k and that *untangle*$(i+1,k)$ leaves P_{i+1} a valid path, the paths will remain disjoint. For the first point, $a'_k < b_k$ shows that after *untangle*(i,k), path P_i enters column k below the level where P_{i+1} originally entered it, which is the level where it (still) leaves that column, by the initial condition $D[i+1][k] = 0$ (no vertical steps in P_{i+1}). The call *untangle*(i,k) makes $D[i+1][k] = \mathit{depth}_k$, and $\mathit{depth}'_k \leq \mathit{depth}_k$ shows that the initial condition for *untangle*$(i+1,k)$ (at least as many vertical steps present as are to be removed) is satisfied: P_{i+1} remains a valid path. Finally this initial condition is also met for the first call *untangle*(k,k) of the sequence, because one has $D[k][k] = a_k = a'_k + \mathit{depth}_k \geq \mathit{depth}_k$.

12. Exporting paths as sequences of vertices. For the purpose of producing graphic output, we generate lists of vertices from which the paths can be drawn. We need only starting and ending vertices, and intermediate vertices where the direction changes.

⟨ Function definitions 3 ⟩ +≡

```
std::vector⟨std::vector⟨std::pair⟨unsigned, unsigned⟩⟩⟩ family::flex_points( ) const
{ std::vector⟨std::vector⟨std::pair⟨unsigned, unsigned⟩⟩⟩ result(n);
  for (unsigned i ⇐ 0; i < n; ++i)
  { result[i].reserve(i = 0 ? 2 : 2 * i + 1);    // maximum number needed
    result[i].push_back(std::make_pair(i, 0u));    // starting point
    unsigned level ⇐ i;
    for (unsigned j ⇐ 1; j ≤ i; ++j)
    { level −⇐ step(i, j − 1);
      if (D[i][j] > 0 or j = i or B[i][j − 1] ≠ B[i][j])    // do nothing if continuing straight on
      { result[i].push_back(std::make_pair(level, j));    // trace horizontal or diagonal segment
        if (D[i][j] > 0) result[i].push_back(std::make_pair(level −⇐ D[i][j], j));    // vertical
      }
    }
    assert(level = 0);
    if (i = 0)    // add separate ending point, so drawing will give a dot
      result[i].push_back(std::make_pair(0u, 0u));    // ending point
  }
  return result;
}
```

13. Converting to tilings of the Aztec diamond. So far everything we have done is in terms of paths. There is however a straightforward correspondence between disjoint families of paths and tilings of the Aztec diamond by dominoes. The Aztec diamond of order m (we shall take $m = n - 1$) is the union of 4 (rectangular) "triangles" of $\frac{m+1}{2}$ squares each, touching each other along their straight sides. To efficiently encode domino tilings, we view the squares as coloured in checkerboard fashion, and tell for each black square with which of its four white neighbours it is paired up. The set of black squares forms a $m \times (m+1)$ rectangular grid tilted 45°. We shall imagine the Aztec diamond itself rotated so that the rectangle has it longer side horizontal, in which case the dominoes go in diagonal directions. It turns out that the 4 possible directions of dominoes correspond in the path family setting to the following four possible statuses of a grid point (not on the arrival line) with respect to the path family: (0) no path runs through the point, (1) a path goes through the point, parting in horizontal direction, (2) a path passes parting in vertical direction, and (3) a path passes parting in diagonal direction. This explains our following definition.

$\langle$ Type definitions 2 $\rangle$ +≡
  enum { *empty*, *horizontal*, *vertical*, *diagonal* }; // possible domino directions: NW, NE, SW, SE

14. With this encoding we can convert a disjoint family into a tiling of the Aztec diamond of order $n - 1 = max\_path(\,)$ in a straightforward way. The main work needed is tracking the (vertical) level of the path in column j, which descends from i to 0 as j goes from 0 to i. The current path direction is assigned as domino orientation to *result*[*level*][j].

$\langle$ Function definitions 3 $\rangle$ +≡

```
std::vector⟨vec⟩ family::Aztec_tiling( ) const
{ assert(valid( ));   // this ensures no overwriting and level ≥ 0 below
  std::vector⟨vec⟩ result(n);
  if (n = 0) return result;   // you never know
  std::fill(result.begin( ) + 1, result.end( ), vec(n, empty));   // result[0] remains empty
  for (unsigned i ⇐ 0; i < n; ++i)
  { assert(i = n − 1 or disjoint(i));
    unsigned level ⇐ i;
    for (unsigned j ⇐ 0; j < i; ++j)
    { result[level][j] ⇐ B[i][j] ? diagonal : horizontal; level −⇐ step(i, j);
      for (unsigned k ⇐ 0; k < D[i][j + 1]; ++k) result[level −−][j + 1] ⇐ vertical;
    }
    assert(level = 0);   // double-check that P_i reached the bottom level
  }
  return result;
}
```

15. PostScript producing functions. To illustrate the family of paths constructed, we provide output in PostScript format. The following simple function provide necessary starting and ending code for pages.

⟨ Function definitions 3 ⟩ +≡
```
void ps_start_page (std::ostream &f, unsigned no)
{ f ≪ "%%Page:␣" ≪ no ≪ '␣' ≪ no ≪ "\nsave\n"; }
void ps_end_page (std::ostream &f)
{ f ≪ "\nrestore\nshowpage\n\n"; }
```

16. Drawing a path is straightforward, using a vector of coordinate pairs as provided by the *flex_points* method. It is at the point where we switch to the Cartesian coordinate ordering (column coordinate first) used in PostScript.

⟨ Function definitions 3 ⟩ +≡
```
void ps_output_path (const std::vector⟨std::pair⟨unsigned, unsigned⟩⟩ &p, std::ostream &f)
{ f ≪ "newpath\n" ≪ p[0].second ≪ '␣' ≪ p[0].first ≪ "␣moveto\n";
  for (unsigned j ⇐ 1; j < p.size( ); ++j) f ≪ p[j].second ≪ '␣' ≪ p[j].first ≪ "␣lineto\n";
  f ≪ "stroke\n";
}
```

17. We make a complete Postscript page, and do worry a bit about scale: we make the page 500 big-points wide, which is about 17.5 cm. We also allow some variation in the display format, to take into account the column k up to which the paths have been made disjoint: the paths with index less than k, which may intersect mutually and with the others, are printed in light gray. When $k = 0$, all paths will therefore print black. If in addition the optional argument *highlight* is set, then paths with index less than k are not printed at all, and path *highlight* is made red to highlight it.

⟨ Function definitions 3 ⟩ +≡
```
void ps_output_family_as_page
      (const family &p, std::ostream &f, unsigned k, unsigned pageno, unsigned highlight ⇐
        ~0u)
{ ps_start_page(f, pageno);
  float scale ⇐ 500.0/(p.max_path( ) + 2);
  f ≪ scale ≪ '␣' ≪ scale ≪ "␣scale␣␣1␣1␣translate␣0.1␣setlinewidth\n\n";
  std::vector⟨std::vector⟨std::pair⟨unsigned, unsigned⟩⟩⟩ points ⇐ p.flex_points( );
  if (highlight = ~0u and k > 0)
  { f ≪ "gsave␣0.75␣setgray\n";      // set to light gray
    for (unsigned i ⇐ 0; i < k; ++i) ps_output_path(points[i], f);
    f ≪ "grestore\n";
  }
  for (unsigned i ⇐ k; i < points.size( ); ++i)
    if (i ≠ highlight) ps_output_path(points[i], f);
  if (k ≤ highlight and highlight < points.size( ))      // highlighted path last, so on top
  { f ≪ "gsave␣red\n";      // switch to red
    ps_output_path(points[highlight], f); f ≪ "grestore\n";
  }
  ps_end_page(f);
}
```

18. The *Aztec_tiling* method to produces a $(n-1)\times n$ matrix of domino orientations, used below to draw each domino in its correct place. The Aztec diamond is tilted, so that its black squares form a non-tilted $(n-1)\times n$ rectangular grid; it corresponds to the grid points traversed by the path family, excluding the points of arrival of the paths. More precisely for a titled black square b, we shall place such a grid point on the midpoint M_b of its northwest edge. The midpoints of the other three sides of b are not grid points, but by reflecting M in each of them give rise to grid points situated east, southeast or south of M. The orientations *horizontal*, *diagonal* and *vertical* at M describe the domino containing the segment from M_b to its reflected image, formed by b and its white neighbour in the corresponding direction; the final orientation *empty* corresponds to the domino formed by the two squares separated by the edge on which M_b itself lies.

The function below outputs the coordinates $(i-\frac14, j+\frac14)$ of the centre of the black square b (with diagonal of length 1) of which $M_b=(i,j)$ lies on the northwest edge, followed by an indication of the direction of the domino with respect to this square, which will actually name a PostScript operator defined appropriately.

⟨ Function definitions 3 ⟩ +≡
```
void ps_write_dominoes (std::ostream &f, const std::vector⟨vec⟩ &orient, unsigned no)
{ unsigned n ⇐ orient.size( ); static char compass[ ][3] ⇐ { "NW", "NE", "SW", "SE" };
  ps_start_page(f, no);
  float scale ⇐ 500.0/(n + 1);
  f ≪ scale ≪ '␣' ≪ scale ≪ "␣scale␣1␣1␣translate␣0.1␣setlinewidth\n\n";
  for (unsigned i ⇐ n; i−− > 1; )
    for (unsigned j ⇐ 0; j < n; ++j)
      f ≪ j ≪ ".25␣" ≪ i − 1 ≪ ".75␣" ≪ compass[orient[i][j]] ≪ '\n';
  ps_end_page(f);
}
```

19. The following code will be executed by the main program whenever PostScript output is

⟨ Write preamble of PostScript file to *out_stream* 19 ⟩ ≡
```
{ out_stream ≪ "%!PS-Adobe-3.0\n%%Pages:␣(atend)\n%%BoundingBox:␣0␣0␣500␣500\n"
      "%%DocumentPaperSizes:␣a4\n%%EndComments\n\n1␣setlinecap\n";
  out_stream ≪ "/yellow␣{1␣1␣0␣setrgbcolor}␣bind␣def\n"
      "/red␣{1␣0␣0␣setrgbcolor}␣bind␣def\n""/green␣{0␣1␣0␣setrgbcolor}␣bind␣def\n"
      "/blue␣{0␣0␣1␣setrgbcolor}␣bind␣def\n""/orient␣%␣angle\n"
      "{rotate␣.5␣sqrt␣dup␣scale}␣bind␣def\n";
  out_stream ≪ "/domino\n""{␣gsave␣0␣setlinewidth\n"
      "␣␣newpath␣-.5␣.5␣moveto␣2␣0␣rlineto␣0␣-1␣rlineto␣-2␣0␣rlineto␣closepath\n"
      "␣␣stroke␣grestore\n""␣␣newpath␣0␣.5␣moveto␣1␣-1␣rlineto␣load␣exec␣stroke\n"
      "␣␣newpath␣0␣-.5␣moveto␣1␣1␣rlineto␣load␣exec␣stroke\n"
      "␣␣newpath␣-.5␣0␣moveto␣2␣0␣rlineto␣load␣exec␣stroke}\n""␣␣bind␣def\n";
  out_stream ≪ "/NW␣{gsave␣translate␣135␣orient␣/yellow␣/green␣/blue␣domino"
      "␣grestore}␣bind␣def\n"
      "/NE␣{gsave␣translate␣␣45␣orient␣/blue␣/yellow␣/red␣domino"
      "␣grestore}␣bind␣def\n"
      "/SW␣{gsave␣translate␣225␣orient␣/green␣/red␣/yellow␣domino"
      "␣grestore}␣bind␣def\n""/SE␣{gsave␣translate␣315␣orient␣/red␣/blue␣/green␣domino"
      "␣grestore}␣bind␣def\n";
}
```
This code is used in section 21.

20. The following code will be executed at the end by the main program whenever PostScript output is generated. The variable *cur_page*, used to number pages, at the end contains the numbers of pages produced.

⟨ Write trailer of PostScript file to *out_stream* 20 ⟩ ≡
```
{ out_stream ≪ "\n%%Trailer\n%%Pages:␣" ≪ cur_page ≪ "\n%%EOF\n"; }
```
This code is used in section 21.

21. The main program. The main program reads the command line to find out what needs to be done.

```
#include <fstream>
#include <sstream>
#include <ctime>
  int main (int argc, char **argv)
  { bool do_family ⇐ true, do_dominoes ⇐ false, exhaustive ⇐ false, do_ps ⇐ true, seed_set ⇐ false;
    int c; unsigned int order, snapshots ⇐ 0, column_detail ⇐ ~0, random_seed ⇐ 0;
    std::string file_name_base; ⟨ Process options 24 ⟩
    std::ofstream out_file;
    triangle tri(order + 1);
    for (unsigned i ⇐ 0; i ≤ order; ++i) tri[i].resize(i, true);
    if (not exhaustive)
    { std::srand(seed_set ? random_seed : std::time(⊙));   // intialise random generator
      randomize(tri);
    }
    if (not file_name_base.empty())
    { std::ostringstream os; os ≪ file_name_base ≪ '-' ≪ order;
      if (snapshots > 0) os ≪ '-' ≪ snapshots;
      os ≪ ".ps"; out_file.open(os.str().c_str(), std::ios_base::out);
      if (not out_file.is_open())
        { std::cerr ≪ "Cannot␣open␣file␣" ≪ os.str() ≪ "␣for␣output.\n"; exit(2); }
    }
    std::ostream &out_stream(out_file.is_open() ? out_file : std::cout);
    if (do_ps) ⟨ Write preamble of PostScript file to out_stream 19 ⟩
    unsigned cur_page ⇐ 0;
    if (exhaustive)
    { unsigned long long count ⇐ 0;
      do
      { std::cout ≪ '\r' ≪ count++; ⟨ Perform conversion of tri to disjoint family and back 22 ⟩
      } while (next(tri));
    }
    else ⟨ Perform conversion of tri to disjoint family and back 22 ⟩
    if (do_ps) ⟨ Write trailer of PostScript file to out_stream 20 ⟩
    if (out_file.is_open())
    { out_file.close(); std::cout ≪ "\nSucces!\n";   // If we get here, things went well
    }
    return 0;   // success
  }
```

22. The test initialises p from *tri*, converts to disjoint form and test for validity, converts back again and tests against the initial configuration. In case *snapshots* and/or *column_detail* have been set, we shall need to interrupt our algorithm at various points to produce output, which we shall detail below.

```
⟨ Perform conversion of tri to disjoint family and back 22 ⟩ ≡
  { family p(tri);    // encode tri as probably intersecting family
    family q ⇐ p;    // keep a copy for comparison
    if (snapshots > 0 or column_detail ≠ ∼0u) ⟨ Perform untangling of p with intermediate output at
          snapshots different steps, and maybe at column column_detail 23 ⟩
    else to_disjoint(p);    // transform into disjoint family
    if (not p.valid()) { std::cout ≪ "␣Mapping␣problem␣found" ≪ std::endl; return 1; }
    if (do_ps)
    { if (do_family) ps_output_family_as_page(p, out_stream, 0, ++cur_page);
      if (do_dominoes) ps_write_dominoes(out_stream, p.Aztec_tiling(), ++cur_page);
    }
    to_cliffs(p);    // decode it again
    if (p ≠ q) { std::cout ≪ "␣Bijectivity␣problem␣found" ≪ std::endl; return 2; }
  }
```

This code is used in section 21.

23. If we want to be able to show intermediate stages of the untangling process, then we cannot call *to_disjoint*. Instead we call the method *untangle* a number of times, interrupting the output loop for output *snapshots* times, including at the very beginning. We must deal with the possibility that $snapshots = 0$ though (in case *column_detail* has been set); since we cannot divide into 0 chunk, we set a boolean and then $snapshots \Leftarrow 1$ to handle this case.

```
⟨ Perform untangling of p with intermediate output at snapshots different steps, and maybe at column
      column_detail 23 ⟩ ≡
  { bool chunk_output ⇐ snapshots > 0;
    if (not chunk_output) snapshots ⇐ 1;
    unsigned k ⇐ order;
    for (unsigned group ⇐ snapshots; group−− > 0; )    // group counts chunks, backwards
    { if (chunk_output) ps_output_family_as_page(p, out_stream, k, ++cur_page);    // partial output
      unsigned next_stop ⇐ (group ∗ order + snapshots − 1)/snapshots;    // round up
      while (k−− > next_stop)    // decrease until next stop
      { unsigned i;    // declare outside next loop for final test
        for (i ⇐ k; i < order; ++i)
        { if (k = column_detail)
              ps_output_family_as_page(p, out_stream, k, ++cur_page, i);    // partial output
          if (not p.untangle(i, k))
              break;    // separate P_i and P_{i+1} in column k; stop if nothing changed
        }
        if (k = column_detail and i = order)    // then last untangle did change
          ps_output_family_as_page(p, out_stream, k, ++cur_page, i);    // last one
      }
      ++k;    // compensate decrement after final test in while loop
    }
  }
```

This code is used in section 22.

24. Option processing. There is one obligatory argument, the rank n Aztec diamond, which should follow any options. To do an exhaustive test for generating all tilings of, the option `-x` may be specified; otherwise the program will run a random sample and produce PostScript output. The option `-X` does an exhaustive test *and* produces PostScript output for all families. If the option `-t` is given, any display of a disjoint family will be followed by the corresponding tiling of the Aztec diamond, while `-n` will suppress output of the path family. The option `-s` with (obligatory) numeric value m will produce m intermediate snapshots of not yet disjoint families during the construction; the same result can also be obtained by specifying "n/m" in place of n. An option `-c` with numeric value k will produce a detailed sequence of all stages of processing column k. A random seed to be used can be fixed by giving it as argument to a `-r` flag. An output file may be specified as final argument (if not, any PostScript output will go to *cout*).

```
#include <unistd.h>
#include <cctype>
⟨ Process options 24 ⟩ ≡
  { opterr ⇐ 0;    // clear error status
    while ((c ⇐ getopt(argc, argv, "xXtns:c:r:")) ≠ −1)
      switch (c)
      {
      case 'x': do_ps ⇐ false;    // fall through
      case 'X': exhaustive ⇐ true; break;
      case 't': do_dominoes ⇐ true; break;
      case 'n': do_family ⇐ false; break;
      case 's': case 'c': case 'r':
        { unsigned int &dst ⇐ ∗(c = 's' ? &snapshots : c = 'c' ? &column_detail : &random_seed);
          std::stringstream arg(optarg); arg ≫ dst;
          if (arg.fail())
          { std::cerr ≪ "Argument␣'" ≪ optarg ≪ "'␣of␣-" ≪ char(c)
                ≪ (std::isdigit(optarg[0]) ? "␣too␣large\n" : "␣not␣numeric\n"); std::exit(1);
          }
        }
        break;
      case '?':
        if (optopt = 's' or optopt = 'c' or optopt = 'r')
          { std::cerr ≪ "Option␣-" ≪ char(optopt) ≪ "␣requires␣an␣argument\n"; std::exit(1); }
        else { std::cerr ≪ "Unknown␣option␣-" ≪ char(optopt) ≪ ".\n"; std::exit(1); }
      default: std::abort();
      }
    if (optind ≥ argc) { std::cerr ≪ "No␣order␣of␣Aztec␣diamond␣specified\n"; std::exit(1); }
    std::stringstream arg(argv[optind++]); arg ≫ order;
    if (arg.fail())
      { std::cerr ≪ "Specified␣order␣'" ≪ arg ≪ "'␣not␣numeric\n"; std::exit(1); }
    char c; arg ≫ c;
    if (c = '/')
    { arg ≫ snapshots;
      if (arg.fail())
        { std::cerr ≪ "After␣/,␣'" ≪ arg ≪ "'␣not␣numeric\n"; std::exit(1); }
    }
    if (do_ps and optind < argc) { std::stringstream arg(argv[optind++]); arg ≫ file_name_base; }
    if (optind ≠ argc)
      { std::cerr ≪ "Found␣" ≪ argc − optind ≪ "␣trailing␣arguments\n"; std::exit(1); }
  }
```

This code is used in section 21.

25. Generating triangles of bits. To systematically loop over possible **triangle** values, the function *next* will advance to the next value, and return **true** if this was possible. In order to give a nicer enumeration of the triangles, we shall traverse by weakly increasing number of **false** values (initially all bits were set to **true**). To do this we search first for the first **false** value, then while setting it and immediately following **false** bits to **true**, search for the first **true** value. If it exists, we set it to **false**, as well a number of initial bits one less than the number of **false** bits set to true, so that the number of **false** values remains unchanged. In case we don't find any **false** values in the first place, or no **true** values in the second place, we increase the number of false values if possible, and set this number of initial bit to **false**. Finally if every bit was found to be **false**, we return **false** to indicate the end of the iteration.

⟨ Function definitions 3 ⟩ +≡

```
bool next (triangle &tri)
{ bool target ⇐ false; unsigned count ⇐ 0;   // counts false values changed to true, minus 1
  for (unsigned i ⇐ 1; i < tri.size( ); ++i)
  { bitvec &trii ⇐ tri[i];
    for (unsigned j ⇐ 0; j < i; ++j)
      if (not trii[j]) { trii[j] ⇐ target ⇐ true; ++count; }   // change false to true and count
      else if (target)   // then we found our true
        { trii[j] ⇐ false; −−count; goto phase_2; }
  }
      // if we come here we did not find false followed by true
  if (2 ∗ count = tri.size( ) ∗ (tri.size( ) − 1))   // then all bits were false
    return false;   // so we've reached the end
  else   // we found (and set) count bits false, but no following true
    ++count;   // this will increment the number of false bits by 1
phase_2:
  if (count = 0) return true;   // we just shifted one false one place further
  for (unsigned i ⇐ 1; i < tri.size( ); ++i)
  { bitvec &trii ⇐ tri[i];
    for (unsigned j ⇐ 0; j < i; ++j)
    { trii[j] ⇐ false;
      if (−−count = 0) return true;   // we finished setting count bits to false
    }
  }
  assert(false);   // we should never come here
  return true;   // but the compiler may not understand that
}
```

26. We also want to be able to generate random values in a triangle. We get random bits by calling *std::rand* and taking bit number 5, hoping it will be a bit more random than bit number 0.

#include `<cstdlib>`

⟨ Function definitions 3 ⟩ +≡

```
void randomize (triangle &tri)
{ for (unsigned i ⇐ 1; i < tri.size( ); ++i)
  { bitvec &trii ⇐ tri[i];
    for (unsigned j ⇐ 0; j < i; ++j) trii[j] ⇐ (std::rand( ) & #20) = 0;
  }
}
```

27. Index.

⟨ Function definitions 3, 4, 5, 6, 7, 8, 9, 12, 14, 15, 16, 17, 18, 25, 26 ⟩ Used in section 1.
⟨ Perform conversion of *tri* to disjoint family and back 22 ⟩ Used in section 21.
⟨ Perform untangling of *p* with intermediate output at *snapshots* different steps, and maybe at column *column_detail* 23 ⟩ Used in section 22.
⟨ Process options 24 ⟩ Used in section 21.
⟨ Type definitions 2, 13 ⟩ Used in section 1.
⟨ Write preamble of PostScript file to *out_stream* 19 ⟩ Used in section 21.
⟨ Write trailer of PostScript file to *out_stream* 20 ⟩ Used in section 21.

AZTEC