

Polyominoes and Related Families

Jaime Rangel-Mondragón

Polyominoes are generalizations of dominoes constructed by joining congruent squares side by side. By describing a polyomino as a list of Gaussian integers, we generate all different polyominoes of a given size. This method is extended to the generation of the families of polyiamonds, polyhexes, and polykites. We also give a method to tile rectangles using polyominoes and explore the fractal family of rep-tiles.

■ Introduction

In 1953, when delivering a talk at the Harvard Mathematics Club, mathematician Solomon Golomb [1] defined a new class of geometric figures he named polyominoes. Polyominoes are generalizations of dominoes [2] and have been extremely popular since their use in the game of Tetris [3]. They have also enjoyed a prominent place in the recreational mathematics literature since Martin Gardner further popularized them in 1957.

A polyomino is any connected figure that can be constructed by joining congruent squares side by side. A polyomino formed by n squares is referred to as an n -omino. In the first part of this work, we generate all n -ominoes and extend our method to the corresponding generation of polyiamonds, polyhexes, and polykites [4]. In the second part, we will tessellate rectangles using polyominoes and introduce the family of *rep-tiles*.

Some sections of this work are ordered so that we can compare performance issues among different computer configurations and also calculate timings for a 2003-vintage model 2.4 GHz personal computer using *Mathematica* 5.

```
In[1]:= Off[General::"spell", General::"spell1"]
SetOptions[Graphics, AspectRatio -> Automatic];
<< "Graphics`Colors`";
```

■ The Naive Approach

In this section we generate n -ominoes using a straightforward approach. We consider all possible 0-1 $n \times n$ matrices and select those that are orthogonally connected, that is, that have no isolated blocks of ones. To check this property

we read the first one and change its sign. We then mark those adjacent to it and repeatedly spread this changing of signs to those adjacent to them until no further changes occur. At the end, the new matrix cannot contain a one if the original matrix represented a connected shape.

```
In[4]:= connectedQ[n0mino_] := Module[{n, m, i, j, h, k, s = n0mino},
  {n, m} = Dimensions[s];
  {i, j} = First[Position[s, 1]];
  s[[i, j]] = -1;
  FixedPoint[
    (Do[If[s[[h, k]] == 1,
      If[ ((k > 1) && (s[[h, k-1]] < 0)) || ((k < m) && (s[[h, k+1]] < 0)) ||
        ((h > 1) && (s[[h-1, k]] < 0)) || ((h < n) && (s[[h+1, k]] < 0)),
        s[[h, k]] = -1]], {k, m}, {h, n}];
    s) &, s];
  Position[s, 1] == {}]

In[5]:= connectedQ[ $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ ]
```

Out[5]= False

The following function provides a canonical form for our matrices. It pushes the entire configuration up and to the left so that neither the first row nor the first column are all zero.

```
In[6]:= standard[m_] := Module[{p = Position[m, 1], h, ans, n = Length[m]},
  ans = Table[0, {n}, {n}];
  h = {Min[First /@ p], Min[Last /@ p]};
  p = (# - h + {1, 1}) & /@ p;
  Map[(ans[[First[#], Last[#]]] = 1) &, p];
  ans]

In[7]:= standard[ $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ ] // MatrixForm
```

Out[7]//MatrixForm=

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Our strategy is to generate all 0-1 $n \times n$ matrices (there are 2^{n^2} of them), select those having n ones that are connected, convert them to canonical form, and, finally, remove repetitions from this list. For example, for $n = 4$, out of 65536 possible 4×4 matrices we get only 1820 that have four ones, and, of those, 113 are connected. Only 19 are left when we select those with a different canonical form.

```

In[8]:= Timing[
  n = 4;
  u = Select[Range[2n2], (Plus @@ IntegerDigits[#, 2] == n) &];
  v = Select[
    Map[Partition[IntegerDigits[#, 2, n2], n] &, u], connectedQ[#] &];
  w = Union[standard /@ v];
  {MatrixForm /@ w, 2n2, Length[u], Length[v], Length[w]}]
Out[8]= {1.156 Second,
  { {
     $\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$ 
     $\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$ 
     $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$ 
     $\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$ 
     $\begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \}, 65536, 1820, 113, 19 \}$ 

```

Although this is the simplest way, it takes nearly an hour to complete for $n = 5$, which forces us to look for another approach. If, instead of selecting those binary numbers having n digits that are 1 out of all 2^{n^2} possible, we generate them directly (there are $\binom{n}{2}$ of them), we will save a substantial amount of time. The function `bin[n, b]` computes all lists of length b having n ones.

```

In[9]:= bin[n_, l_] := {} /; n > l
  bin[0, l_] := {Table[0, {l}]}
  bin[1, 1] := {{1}}
  bin[n_, l_] := bin[n, l] = Join[Map[Join[{0}, #] &, bin[n, l - 1]],
    Map[Join[{1}, #] &, bin[n - 1, l - 1]]]

```

```
In[13]:= bin[3, 5]
Out[13]= {{0, 0, 1, 1, 1}, {0, 1, 0, 1, 1}, {0, 1, 1, 0, 1},
          {0, 1, 1, 1, 0}, {1, 0, 0, 1, 1}, {1, 0, 1, 0, 1}, {1, 0, 1, 1, 0},
          {1, 1, 0, 0, 1}, {1, 1, 0, 1, 0}, {1, 1, 1, 0, 0}}
```

We get the 63 canonical forms for the case $n = 5$ more quickly.

```
In[14]:= Timing[
  n = 5;
  v = Select[Map[Partition[#, n] &, bin[n, n2]], connectedQ[#] &];
  w = Union[standard /@ v];
  Length[w]
Out[14]= {19.719 Second, 63}
```

However, some of these patterns are still equivalent under rotations and reflections, so more processing is needed. Although we have substantially reduced the computing time, we will not pursue this approach any further because there is a faster and more general alternative.

■ Polyominoes

The straightforward approach presented in the previous section has many disadvantages. The representation of polyominoes as 0-1 matrices is wasteful because they are very sparse. In this section we consider a polyomino to be a list of unit squares, where a square is specified by the Cartesian coordinates of its bottom-left vertex; further, these coordinates will be encapsulated as complex numbers. So, a polyomino will be a list of Gaussian integers; that is, complex numbers having integral real and imaginary parts. We thus define the following type.

```
In[15]:= polyominoQ[p_] := And@@((IntegerQ[Re[#]] && IntegerQ[Im[#]]) & /@ p)
```

In this definition we cannot use the pattern `_Complex..` because only numbers with nonzero imaginary parts have head `Complex`.

To determine whether two figures are equivalent, we consider the action of the dihedral group $D_4 = \{e, r, r^2, r^3, f, fr, fr^2, fr^3\}$, where r rotates the figure by 90° and f turns it over. Given a polyomino, we then have at most eight different equivalent polyominoes.

If we were using a matrix `m` to describe the positions of the squares forming a polyomino, we could perform a 90° rotation with `Reverse[Transpose[m]]` and a reflection with `Transpose[m]`.

The following functions act on a given polyomino `p`; `rot` and `ref` correspond to the transformations `r` and `f`. The function `cyclic` computes the list of the rotations of `p` and is used in the construction of D_4 produced by the function `dihedral`. (In this case `cyclic` is equal to `NestList[rot, p, 3]`.) The function `canonical` computes a standard representation of `p` by removing repetitions and

sorting. The functions `liC` and `polC` convert a list of complex numbers into a list of points to be drawn as lines or as polygons. The arguments of the functions `cyclic` and `allPieces` are not restricted to polyominoes as they will also be used for the other families.

```
In[16]:= rot[p_?polyominoQ] :=  $i$  p
ref[p_?polyominoQ] := (# - 2 Re[#]) & /@ p

cyclic[p_] := Module[{i = p, ans = {p}},
  While[(i = rot[i]) ≠ p, AppendTo[ans, i]]; ans]

dihedral[p_?polyominoQ] := Flatten[#, ref[#]] & /@ cyclic[p], 1]

canonical[p_?polyominoQ] :=
  Union[#, - (Min[Re[p]] + Min[Im[p]]  $i$ )] & /@ p]

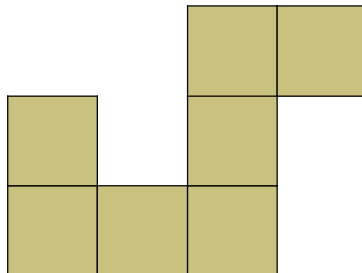
allPieces[p_] := Union[canonical /@ dihedral[p]]

liC[z_] := Line[{Re[#], Im[#]} & /@ z]
polC[z_] := Polygon[{Re[#], Im[#]} & /@ z]

draw[p_?polyominoQ, pr_ : All] :=
  Graphics[{{DarkKhaki, polC[#, # + 1, # + 1 +  $i$ , # +  $i$ ]},
    liC[#, # + 1, # + 1 +  $i$ , # +  $i$ , #]} & /@ p, PlotRange → pr]

In[25]:= polyomino = {0, 1, 2,  $i$ , 2 +  $i$ , 2 + 2  $i$ , 3 + 2  $i$ };
pol = draw[polyomino];
Show[pol]
```

From In[25]:=



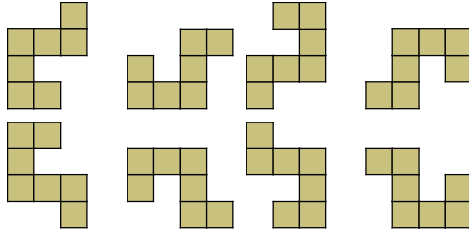
The function `draw` generates the graphical object with an optional second argument specifying the plot range. Note also the change in the representation of a polyomino; the function `canonical` places the polyomino in the first quadrant touching both axes. Here is an example.

```
In[28]:= canonical[polyomino]
Out[28]= {0,  $i$ , 1, 2, 2 +  $i$ , 2 + 2  $i$ , 3 + 2  $i$ }
```

The function `allpieces` computes all eight equivalent figures (or less for some polyominoes).

```
In[29]:= pol = draw[#, {{0, 4.1}, {-0.1, 4}}] & /@ allPieces[polyomino];
Show[GraphicsArray[Partition[pol, 4]]]
```

From In[29]:=

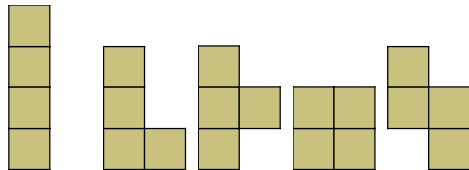


The following incremental method generates all n -ominoes. Having generated the set of all $(n - 1)$ -ominoes, we take each of its members and append a square to each of its squares in all four possible directions. Once we have this extended set containing all the n -ominoes, we obtain their canonical representation and proceed to eliminate the redundant ones.

```
In[31]:= polyominoes[1] := {{0}}
polyominoes[n_] := polyominoes[n] = Module[{f, fig, ans = {}},
  fig = Map[f = #; Map[{f, # + 1, f, # + i, f, # - 1, f, # - i} &, f] &,
    polyominoes[n - 1]];
  fig = Partition[Flatten[fig], n];
  f = Select[Union[canonical /@ fig], Length[#] == n &];
  While[f != {},
    ans = {ans, First[f]};
    f = Complement[f, allPieces[First[f]]];
  Partition[Flatten[ans], n]]
```

```
In[33]:= pol = draw[#, {{0, 2.1}, {-0.1, 4}}] & /@ polyominoes[4];
Show[GraphicsArray[pol]]
```

From In[33]:=



We can now compare the time taken to generate all pentominoes with the time it took in the previous section. Because *Mathematica* uses dynamic programming to retain the generated n -ominoes, the time it takes to recompute old values will only amount to how long it takes to retrieve them and hence would be negligible. So for a fair comparison, we have to start anew so that all previous values are cleared. Let us restart the *Mathematica* kernel and evaluate the following after all initialization cells are evaluated.

```
In[1]:= Timing[polyominoes[5];]
```

```
Out[1]:= {0.047 Second, Null}
```

We can now obtain the number of free n -ominoes (free meaning equivalent under rotation and reflection) ([5, 6] seq. A000105).

```
In[2]:= Timing[Table[Length[polyominoes[n]], {n, 9}]]
```

```
Out[2]= {13.625 Second, {1, 1, 2, 5, 12, 35, 108, 369, 1285}}
```

As an illustration, let us generate all 108 heptominoes, because it is not until $n = 7$ that a hole appears. According to our definition, a polyomino can have one or many interior holes. Can you spot the one having a hole among the following heptominoes? (That is the reason we color the squares; otherwise, it would be impossible to distinguish it.)

```
In[3]:= pol = draw[#, {{-0.5, 5}, {-0.5, 7}}] & /@ polyominoes[7];
Show[GraphicsArray[Partition[pol, 12]]]
```

From In[3]:=



We can also obtain the one-sided (no reflections allowed) or “chiral” polyominoes by running the following cell, which redefines function `allPieces`, rerunning the previous function `polyominoes`, and computing the table as was done before ([5, 6] seq. A000988). At the end of the generation of this table, we have to recover the original definition of `allPieces` because it is needed in the next section. The easiest way is to restart the *Mathematica* kernel, evaluate the initialization cells, and continue with the cells in the next section.

```
In[1]:= allPieces[p_?polyominoQ] := Union[canonical /@ cyclic[p]]
```

```
In[2]:= Timing[Table[Length[polyominoes[n]], {n, 8}]]
```

```
Out[2]= {3.688 Second, {1, 1, 2, 7, 18, 60, 196, 704}}
```

■ Polyiamonds

Polyiamonds are figures built from congruent unit-side equilateral triangles in the same way that polyominoes are built from squares. Any triangle forming a polyiamond will be described by a pair $\{A, t\}$, wherein A codifies the complex coordinates $a + b e^{\frac{\pi i}{3}}$ of the leftmost vertex (anchor) of the triangle (we are assuming one of its sides rests horizontally) and t is its type taken from the set $\{1, -1\}$, corresponding to whether the apex is pointing up or down. The underlying grid on which to place the triangles is generated by all integral linear combinations of the numbers 1 and $e^{\frac{\pi i}{3}}$. Any triangle $\{A, t\}$ has three neighbors adjacent to it, namely, $\{A, -t\}$, $\{A + tv, -t\}$, and $\{A + t(v - u), -t\}$. As with polyominoes, the canonical representation of a polyiamond moves the piece so that its leftmost vertex touches the origin.

To construct the corresponding equivalent versions of a polyiamond, only rotations of multiples of 60° are allowed (a rotation of 60° implies the type of a triangle is changed) so that point (a, b) gets rotated to point $(-b, a + b)$. The reflection of a triangle is achieved simply by sending each of its vertices (a, b) to $(-a - b, b)$. (In this representation a reflection gives the reverse negative of a rotation!) The type of a polyiamond is tested as follows.

```
In[3]:= polyiamondQ[{{_Integer, _Integer}, 1 | -1} ..] := True
        polyiamondQ[_] := False
```

Here are the polyiamond functions.

```
In[5]:= rot[p_?polyiamondQ] :=
        {{-#[[1,2]], Plus@@First[#]} + If[Last[#] == 1, {-1, 1}, {0, 0}],
         -Last[#]} & /@ p
ref[p_?polyiamondQ] := {{-Plus@@First[#], #[[1,2]]}, Last[#]} & /@ p

dihedral[p_?polyiamondQ] := Flatten[#, ref[#]} & /@ cyclic[p, 1]

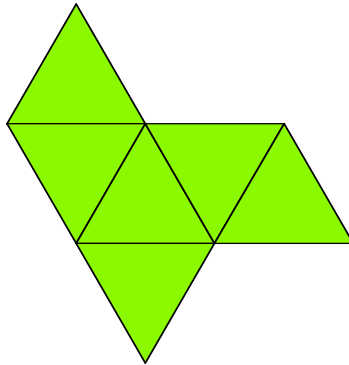
canonical[p_?polyiamondQ] :=
        Sort[Map[{First[#] - {Min[First[First[#]]} & /@ p],
                Min[Last[First[#]} & /@ p], Last[#]} &, p]]

draw[p_?polyiamondQ, pr_ : All] := Module[{a, b, v, t, u = e^{\frac{\pi i}{3}}},
        Graphics[{{a, b}, t} = #; v = a + b u;
                {Chartreuse, polC[v + {0, If[t == 1, u, 1 - u], 1]}},
                liC[v + {0, If[t == 1, u, 1 - u], 1, 0]}] & /@ p, PlotRange -> pr]
```

Here is an example of a polyiamond and its conversion to canonical form.


```
In[10]:= polyiamond = {{ {0, 0}, 1}, { {0, 0}, -1},
  { {1, -1}, 1}, { {1, -1}, -1}, { {1, 0}, -1}, { {2, -1}, 1}};
pol = draw[polyiamond];
Show[pol]
```

From In[10]:=

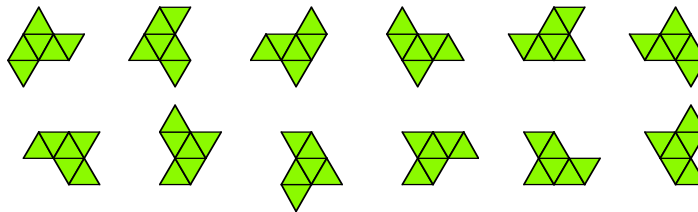


```
In[13]:= canonical[polyiamond]
Out[13]= {{ {0, 1}, -1}, { {0, 1}, 1}, { {1, 0}, -1},
  { {1, 0}, 1}, { {1, 1}, -1}, { {2, 0}, 1}}
```

Here we obtain the polyiamonds equivalent to the previous one.

```
In[14]:= pol = draw[#, {{-0.1, 3.5}, {-1, 2.7}}] & /@ allPieces[polyiamond];
Show[GraphicsArray[Partition[pol, 6]]]
```

From In[14]:=



Finally, we generate all nonisomorphic polyiamonds. We make use of the fact that any triangle $\{a, b, 1\}$ has neighbors $\{a, b, -1\}$, $\{a, b + 1, -1\}$, and $\{a - 1, b + 1, -1\}$, and that triangle $\{a, b, -1\}$ has neighbors $\{a, b, 1\}$, $\{a, b - 1, 1\}$, and $\{a + 1, b - 1, 1\}$.

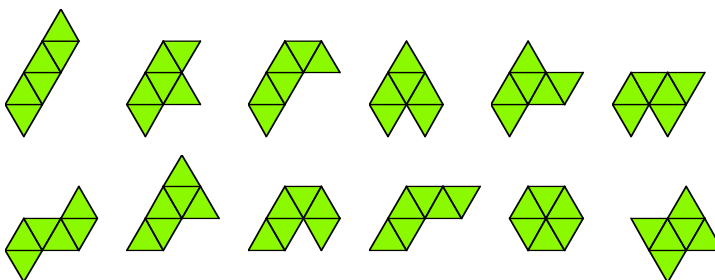
```

In[16]:= polyiamonds[1] := {{{{0, 0}, 1}}}
polyiamonds[n_] :=
polyiamonds[n] = Module[{f, A, t, x, y, z, fig, ans = {}},
  fig = Map[{f = #; Map[{A, t} = #; {f, {A, -t}, f, {A + {0, t}, -t},
    f, {A + {-t, t}, -t}}] &, f]} &, polyiamonds[n - 1]];
  fig = Partition[Partition[Flatten[fig], 3] /.
    {x_Integer, y_, z_} -> {{x, y}, z}, n];
  f = Union[canonical /@ Select[Union /@ fig, Length[#] == n &]];
  While[f != {},
    ans = {ans, First[f]};
    f = Complement[f, allPieces[First[f]]];
    Partition[Partition[Flatten[ans], 3] /.
      {x_Integer, y_, z_} -> {{x, y}, z}, n]]

In[18]:= pol = draw[#, {{0, 3}, {-1, 2.6}}] & /@ polyiamonds[6];
Show[GraphicsArray[Partition[pol, 6]]]

```

From In[18]:=



The number of different polyiamonds is given by the following table ([5, 6], seq. A000577).

```

In[20]:= Timing[Table[Length[polyiamonds[n]], {n, 8}]]
Out[20]= {0.969 Second, {1, 1, 1, 3, 4, 12, 24, 66}}

```

■ Polyhexes

Polyhexes are figures built from congruent unit-length hexagons. We represent a hexagon forming a polyhex simply by a pair (a, b) corresponding to the complex coordinates of its left-bottom corner $a + b e^{\frac{\pi i}{3}}$. For this reason, all the corresponding functions turn out to be straightforward adaptations of their polyiamond counterparts.

```

In[21]:= polyhexeQ[{{_Integer, _Integer} ..}] := True
polyhexeQ[_] := False

```

```

In[23]:= rot[p_?polyhexeQ] := {-Last[#], Plus@@#} & /@ p
ref[p_?polyhexeQ] := {-Plus@@#, Last[#]} & /@ p

dihedral[p_?polyhexeQ] := Flatten[ {#, ref[#]} & /@ cyclic[p], 1]

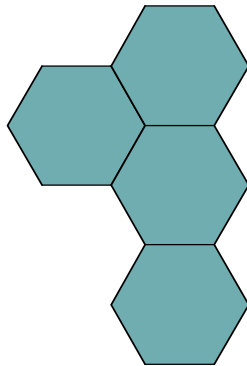
canonical[p_?polyhexeQ] :=
Sort[Map[ {# - {Min[First/@p], Min[Last/@p]} } &, p]]

draw[p_?polyhexeQ, pr_ : All] := Module[{a, b, v, t, u = e $\frac{\pi i}{3}$ },
Graphics[{ {a, b} = #; v = a + b u; {CadetBlue,
polC[{v, v + 1, v + 1 + u, v + 2 u, v + 2 u - 1, v + u - 1}]},
liC[{v, v + 1, v + 1 + u, v + 2 u, v + 2 u - 1, v + u - 1, v]}] & /@
p, PlotRange -> pr]

In[28]:= polyhex = {{0, 0}, {1, 1}, {2, -1}, {3, -3}};
pol = draw[polyhex];
Show[pol]

```

From In[28]:=



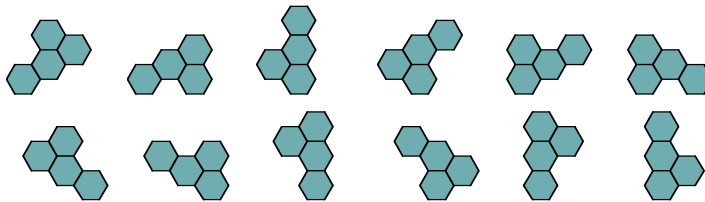
```

In[31]:= canonical[polyhex]
Out[31]= {{0, 3}, {1, 4}, {2, 2}, {3, 0}}

In[32]:= pol = draw[#, {{-0.5, 6}, {-0.2, 5.5}}] & /@ allPieces[polyhex];
Show[GraphicsArray[Partition[pol, 6]]]

```

From In[32]:=



The neighbors of a hexagon (a, b) are $(a - 1, b - 1)$, $(a + 1, b + 2)$, $(a + 2, b - 1)$, $(a + 1, b + 1)$, $(a - 1, b + 2)$, and $(a - 2, b + 1)$. Thus, we can simulate a hexagonal cellular space by a rectangular cellular space in which the neighbors are these

nonadjacent ones. The following function generates all polyhexes of a given order.

```

In[34]:= polyhexes[1] := {{0, 0}}
polyhexes[n_] := polyhexes[n] = Module[{f, a, b, fig, ans = {}},
  fig =
    Map[{f = #; Map[({a, b} = #; {f, {a - 1, b - 1}, f, {a + 1, b - 2}, f,
      {a + 2, b - 1}, f, {a + 1, b + 1}, f, {a - 1, b + 2},
      f, {a - 2, b + 1}}) &, f]} &, polyhexes[n - 1]];
  fig = Partition[Partition[Flatten[fig], 2], n];
  f = Union[canonical /@ Select[Union /@ fig, Length[#] == n &]];
  While[f != {},
    ans = {ans, First[f]};
    f = Complement[f, allPieces[First[f]]];
    Partition[Partition[Flatten[ans], 2], n]]

In[36]:= pol = draw[#, {{-0.5, 6.2}, {-0.5, 5.5}}] & /@ polyhexes[4];
Show[GraphicsArray[pol]]

```

From In[36]:=



The number of different polyhexes is given by the following table ([5, 6], seq. A000228).

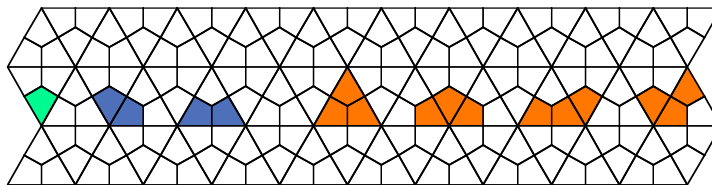
```

In[38]:= Timing[Table[Length[polyhexes[n]], {n, 8}]]
Out[38]= {18.891 Second, {1, 1, 3, 7, 22, 82, 333, 1448}}

```

■ Polykites

Polykites are figures built from the quadrilaterals forming the following lattice. On top of this lattice, we have shown the arrangements corresponding to all the possible shapes for the 1-polykite, 2-polykites, and 3-polykites using different colors. Larger-order polykites offer a lovely resemblance of faces, birds, and animal shapes in a way similar to tangrams. Their different balance and aesthetic properties motivated the author to include their generation as a proper sibling of polyominoes; however, the other Penrose piece, the Dart, being nonconvex, resisted all attempts at generating polydarts.



Just as we have two types of triangles forming polyiamonds, we have six types of kites forming polykites and we will describe them in a similar fashion. The

following comprises an adaptation of the functions we have previously designed. The type of a polykite is tested as follows.

```
In[39]:= polykiteQ[{{_Integer, _Integer}, 1 | 2 | 3 | 4 | 5 | 6} ..] := True
polykiteQ[_] := False
```

Here are the polykite functions.

```
In[41]:= rot[p_?polykiteQ] :=
  {{-#[[1,2]], Plus@@First[#]}, If[Last[#] == 6, 1, 1 + Last[#]]} & /@p
ref[p_?polykiteQ] := {{-Plus@@First[#], #[[1,2]]},
  Switch[Last[#], 1, 3, 2, 2, 3, 1, 4, 6, 5, 5, 6, 4]} & /@p

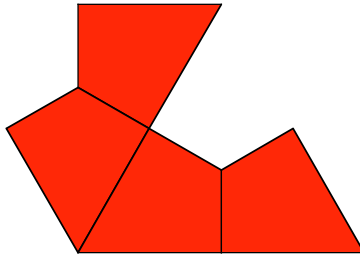
dihedral[p_?polykiteQ] := Flatten[{-#, ref[#]} & /@ cyclic[p], 1]

canonical[p_?polykiteQ] :=
  Sort[Map[{First[#] - {Min[First[First[#]] & /@p],
    Min[Last[First[#]] & /@p]}, Last[#]} &, p]]

draw[p_?polykiteQ, pr_ : All] := Module[{a, b, v, t, u = e $\frac{\pi i}{3}$ },
  Graphics[{{a, b}, t} = #; v = a + b u;
    {LightCadmiumRed, polC[v + {0, ut-1,  $\frac{2(u^t + u^{t-1})}{3}$ , ut]}]},
    liC[v + {0, ut-1,  $\frac{2(u^t + u^{t-1})}{3}$ , ut, 0}]} & /@ p, PlotRange -> pr]]

In[46]:= polykite = {{{0, 0}, 4}, {{0, -2}, 2}, {{0, -2}, 1}, {{2, -2}, 3}};
pol = draw[polykite];
Show[pol]
```

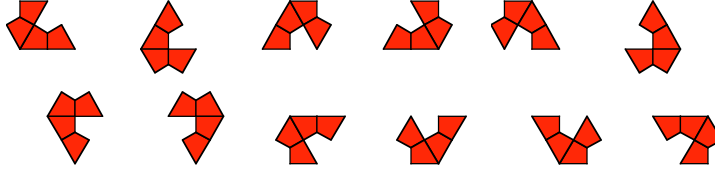
From In[46]:=



```
In[49]:= canonical[polykite]
Out[49]= {{{0, 0}, 1}, {{0, 0}, 2}, {{0, 2}, 4}, {{2, 0}, 3}}
```

```
In[50]:= pol = draw[#, {{-0.5, 3.5}, {-1, 2.8}}] & /@ allPieces[polykite];
Show[GraphicsArray[Partition[pol, 6]]]
```

From In[50]:=

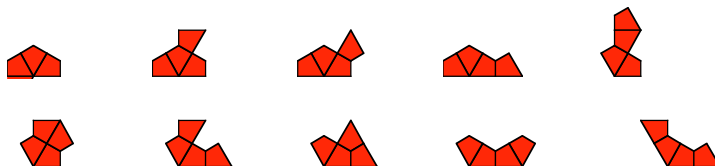


The polykite $\{A, t\}$ has neighbors depending on its anchor point and its type as shown in the following Switch instruction.

```
In[52]:= polykites[1] := {{{0, 0}, 1}}
polykites[n_] :=
polykites[n] = Module[{f, A, t, x, y, z, fig, ans = {}},
  fig = Map[{f = #; Map[{A, t} = #;
    Switch[t,
      1, {f, {A + {2, 0}, 3},
        f, {A + {0, 2}, 5}, f, {A, 2}, f, {A, 6}},
      2, {f, {A + {0, 2}, 4}, f, {A + 2{-1, 1}, 6},
        f, {A, 3}, f, {A, 1}},
      3, {f, {A + 2{-1, 1}, 5}, f, {A + {-2, 0}, 1},
        f, {A, 4}, f, {A, 2}},
      4, {f, {A + {-2, 0}, 6}, f, {A + {0, -2}, 2},
        f, {A, 5}, f, {A, 3}},
      5, {f, {A + {0, -2}, 1}, f, {A + 2{1, -1}, 3},
        f, {A, 6}, f, {A, 4}},
      6, {f, {A + {2, 0}, 4}, f, {A + 2{1, -1}, 2}, f,
        {A, 1}, f, {A, 5}}]}] &, f]] &, polykites[n - 1]];
  fig = Partition[Partition[Flatten[fig], 3] /.
    {x_Integer, y_, z_} -> {{x, y}, z}, n];
  f = Union[canonical /@ Select[Union /@ fig, Length[#] == n &]];
  While[f != {},
    ans = {ans, First[f]};
    f = Complement[f, allPieces[First[f]]];
  Partition[Partition[Flatten[ans], 3] /.
    {x_Integer, y_, z_} -> {{x, y}, z}, n]]

In[54]:= pol = draw[#, {{-1, 4}, {-0.1, 3}}] & /@ polykites[4];
Show[GraphicsArray[Partition[pol, 5]]]
```

From In[54]:=



The number of different polykites is given by the following table ([5, 6], seq. A057786).

```
In[56]:= Timing[Table[Length[polykites[n]], {n, 8}]]
Out[56]= {16.578 Second, {1, 2, 4, 10, 27, 85, 262, 873}}
```

We invite the reader to generate all six polykites that, besides having beautiful shapes, present for the first time a hollowed piece. Here is the corresponding code.

```
In[57]:= pol = draw[#, {{-1, 5}, {-1, 3}}] & /@ polykites[6];
          Show[GraphicsArray[Partition[pol, 17]]]
```

In the next part of this work, we will tile rectangles with polyominoes and introduce the family of *rep-tiles*, which presents a fascinating way of tiling the whole plane with specific polyominoes and polyiamonds.

■ Tiling Rectangles with Polyominoes

In this section, we approach the problem of tiling a rectangle by using pieces taken from a set of polyominoes. Naturally, the area of the rectangle has to be equal to the sum of the areas of the individual pieces in the set. Thus, we immediately know that it would be impossible to fill a rectangle of area 11 with dominoes or one of area 10 with triominoes, although it is not immediately apparent that we could fill a rectangle of arbitrary area (greater than 2) using dominoes and triominoes.

This section comprises three parts. The first one discusses tiling a rectangle allowing repetitions of the pieces taken from a given list. The second part will consider those tilings having no fault lines. Finally, in the last part we will tile using all pieces of a given set.

The function `tess` uses the backtrack method to construct all possible tilings of an $n \times m$ rectangle with pieces taken from a given list `poly`. The optional argument `justOneSolution` is included in case we do not want all solutions. The function `tess` uses the recursive function `tessAux`, which implements the backtrack mechanism: all possible candidates for the placement of the next piece are computed and the lexicographically smallest one is chosen. Each piece, including its rotations and reflections, is placed, and then `tessAux` is called recursively. As the method works faster if $n > m$, their values are swapped before and after computing the solutions. The function `getLines` takes a tiling and computes the endpoints of the horizontal and vertical lines forming the rightmost boundaries of the pieces of the tiling. The function `tile` displays a graphic array of the solutions of given width r .

```

In[59]:= lexic[p_] := Sort[p,
      (Im[#1] < Im[#2]) || ((Im[#1] == Im[#2]) && (Re[#1] ≤ Re[#2])) &]

tess[{n_, m_}, poly_, justOneSolution_: False] :=
Module[{avail, pieces, i, j, ans = {}, tessAux, na, ma},

  tessAux[partial_] := Module[{f, c, candidates, newp, k},
    candidates = Complement[avail, Flatten[partial]];
    If[candidates == {},
      AppendTo[ans, partial]; If[justOneSolution, Throw[1]],
      k = First[lexic[candidates]];
      Map[newp = k + # - First[#];
        If[(Complement[newp, avail] == {}) && (f =
          Flatten[{partial, newp}); Length[f] == Length[Union[f]]],
          tessAux[Append[partial, newp]]] &, pieces]]];

  {na, ma} = If[n < m, {m, n}, {n, m}];
  pieces = lexic /@ Union[Flatten[allPieces /@ poly, 1]];
  avail = Flatten[Table[i + j i, {j, 0, na - 1}, {i, 0, ma - 1}]];
  Catch[tessAux[{}]];
  If[n < m, Map[m - 1 + i # &, ans], ans]]

getLines[tiling_] := Module[{p},
  Partition[Flatten[
    Map[{p = #; Map[{If[Not[MemberQ[p, # + 1]], {# + 1, # + 1 + i}, {}],
      If[Not[MemberQ[p, # + i]], {# + i, # + 1 + i}, {}]} &,
      p] &, tiling]], 2]]

tile[{n_, m_}, poly_, r_, justOneSolution_: False] :=
Module[{t, u, g},
  t = tess[{n, m}, poly, justOneSolution];
  g = Map[Graphics[Append[{{LightBlue, Rectangle[{0, 0}, {m, n}]},
    Line[{{0, n}, {0, 0}, {m, 0}]}], liC /@ getLines[#]]] &, t];
  Show[GraphicsArray[Partition[If[Mod[Length[t], r] == 0,
    g, Join[g, Table[Graphics[Point[{0, 0}]],
      {r - Mod[Length[t], r}]]]], r]]];]

```

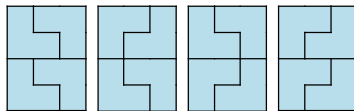
For instance, here is how to get all possible ways of tiling a 4×3 rectangle with the L-triomino.

```

In[63]:= tile[{4, 3}, {{0, i, 1}}, 4]

```

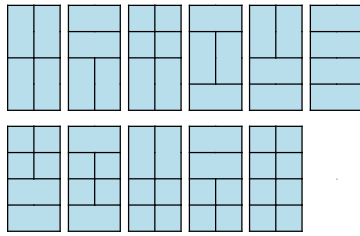
From In[63]:=



Even nonpolyomino pieces are allowed. Consider, for instance, the “number 8” piece $\{0, 1 + i\}$, that is, two squares touching at a vertex, together with a domino $\{0, 1\}$ in the tiling of a 4×2 rectangle. Naturally, two number 8 pieces have to always be together forming a square in any tiling.


```
In[64]:= tile[{4, 2}, {{0, 1 + i}, {0, 1}}, 6]
```

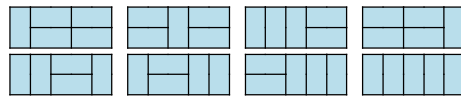
From In[64]:=



Here are all possible tilings of a 2×5 rectangle by dominoes.

```
In[65]:= tile[{2, 5}, polyominoes[2], 4]
```

From In[65]:=



In general, the number of tilings of $2 \times n$ rectangles by dominoes is well known to be the Fibonacci numbers.

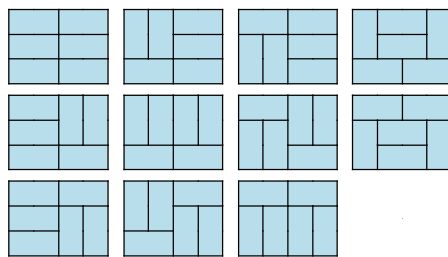
```
In[66]:= Timing[Table[Length[tess[{2, n}, polyominoes[2]]], {n, 15}]]
```

```
Out[66]= {2.547 Second,
          {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987}}
```

Here are all possible tilings of a 3×4 rectangle by dominoes and the number of tilings of $3 \times n$ rectangles, $n \leq 15$.

```
In[67]:= tile[{3, 4}, polyominoes[2], 4]
```

From In[67]:=



```
In[68]:= Timing[Table[Length[tess[{3, n}, polyominoes[2]]], {n, 15}]]
```

```
Out[68]= {50.141 Second, {0, 3, 0, 11, 0, 41, 0, 153, 0, 571, 0, 2131, 0, 7953, 0}}
```

Here is a generating function for these numbers [7].

```
In[69]:= Series[ $\frac{3 - x}{1 - 4x + x^2}$ , {x, 0, 6}]
```

```
Out[69]= 3 + 11 x + 41 x^2 + 153 x^3 + 571 x^4 + 2131 x^5 + 7953 x^6 + 0 [x]^7
```

Or, more explicitly [7].

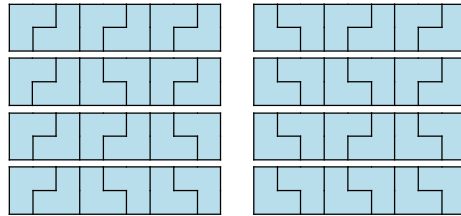
$$\text{In}[70]:= \left[\frac{(2 + \sqrt{3})^{\text{Range}[10]}}{3 - \sqrt{3}} \right]$$

Out[70]= {3, 11, 41, 153, 571, 2131, 7953, 29681, 110771, 413403}

Let us now explore the tiling of a 2×12 rectangle with the L-triomino.

In[71]:= `tile[{2, 9}, {Last[polyominoes[3]]}, 2]`

From In[71]:=



The number of such tilings is somewhat expected.

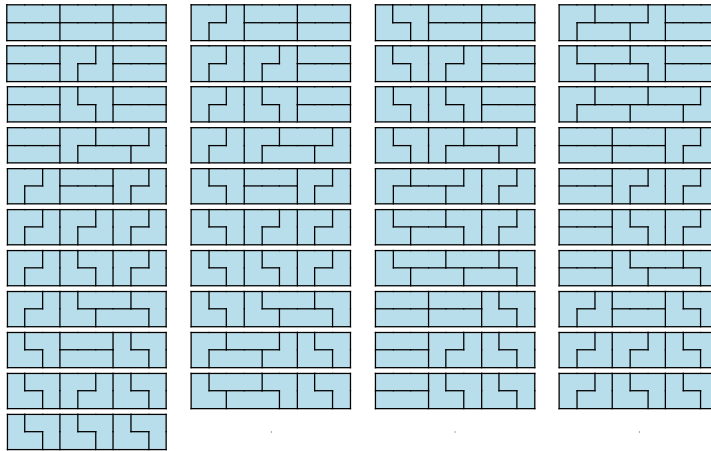
In[72]:= `Timing[Table[Length[tess[{n, 2}, {Last[polyominoes[3]]}], {n, 15}]]`

Out[72]= {0.312 Second, {0, 0, 2, 0, 0, 4, 0, 0, 8, 0, 0, 16, 0, 0, 32}}

Using the complete set of triominoes, we obtain the following ([5, 6], seq. A001835).

In[73]:= `tile[{2, 9}, polyominoes[3], 4]`

From In[73]:=



In[74]:= `Timing[Table[Length[tess[{2, n}, polyominoes[3]]], {n, 15}]]`

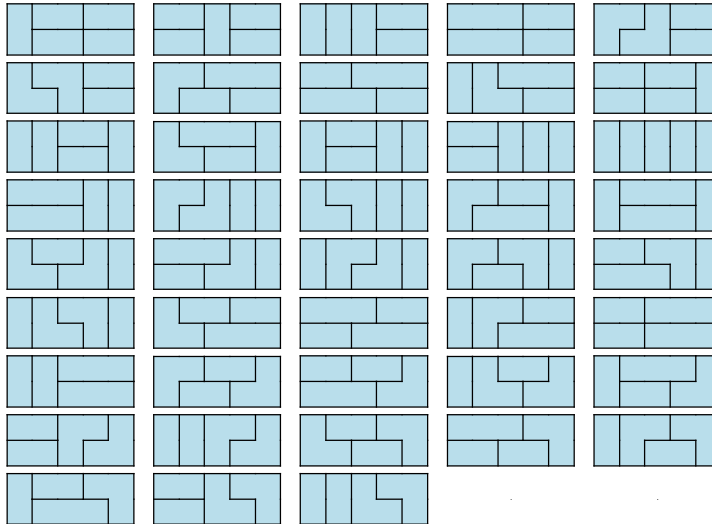
Out[74]= {2.235 Second, {0, 0, 3, 0, 0, 11, 0, 0, 41, 0, 0, 153, 0, 0, 571}}

Compare this sequence with the number of tilings of a $3 \times n$ rectangle by dominoes.

We now consider both sets of dominoes and triominoes together.

```
In[75]:= tile[{2, 5}, Join[polyominoes[2], polyominoes[3]], 5]
```

From In[75]:=



```
In[76]:= Timing[Table[Length[
      tess[{2, n}, Join[polyominoes[2], polyominoes[3]]], {n, 10}]]
```

```
Out[76]= {12.141 Second, {1, 2, 6, 17, 43, 108, 280, 727, 1875, 4832}}
```

Let us now compute the number of ways of tiling an $n \times m$ rectangle with dominoes. We know that nm has to be even; the symmetry of the problem allows us to proceed as follows.

```
In[77]:= u = Table[0, {6}, {6}];
Do[If[Mod[n m, 2] == 0,
      u[[n, m]] = u[[m, n]] = Length[tess[{n, m}, polyominoes[2]]],
      {n, 6}, {m, n, 6}];
MatrixForm[u]
```

Out[79]//MatrixForm=

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 2 & 3 & 5 & 8 & 13 \\ 0 & 3 & 0 & 11 & 0 & 41 \\ 1 & 5 & 11 & 36 & 95 & 281 \\ 0 & 8 & 0 & 95 & 0 & 1183 \\ 1 & 13 & 41 & 281 & 1183 & 6728 \end{pmatrix}$$

The elements of the diagonal correspond to [5, 6], seq. A004003. Here is the corresponding result for triominoes [8].

```
In[80]:= u = Table[0, {6}, {6}];
Do[If[Mod[n m, 3] == 0,
    u[[n,m]] = u[[m,n]] = Length[tess[{n, m}, polyominoes[3]]],
    {n, 6}, {m, n, 6}];
MatrixForm[u]
```

```
Out[82]//MatrixForm=

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 3 & 0 & 0 & 11 \\ 1 & 3 & 10 & 23 & 62 & 170 \\ 0 & 0 & 23 & 0 & 0 & 939 \\ 0 & 0 & 62 & 0 & 0 & 8342 \\ 1 & 11 & 170 & 939 & 8342 & 80092 \end{pmatrix}$$

```

The contributions to this result by the individual triominoes are small.

```
In[83]:= u = Table[0, {6}, {6}];
Do[If[Mod[n m, 3] == 0,
    u[[n,m]] = u[[m,n]] = Length[tess[{n, m}, {First[polyominoes[3]]}],
    {n, 6}, {m, n, 6}];
MatrixForm[u]
```

```
Out[85]//MatrixForm=

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 2 & 3 & 4 & 6 \\ 0 & 0 & 3 & 0 & 0 & 13 \\ 0 & 0 & 4 & 0 & 0 & 22 \\ 1 & 1 & 6 & 13 & 22 & 64 \end{pmatrix}$$

```

```
In[86]:= u = Table[0, {6}, {6}];
Do[If[Mod[n m, 3] == 0,
    u[[n,m]] = u[[m,n]] = Length[tess[{n, m}, {Last[polyominoes[3]]}],
    {n, 6}, {m, n, 6}];
MatrixForm[u]
```

```
Out[88]//MatrixForm=

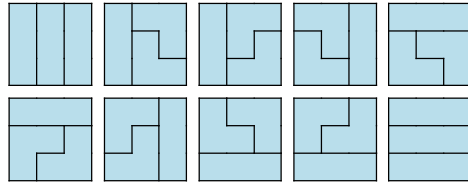
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 4 \\ 0 & 2 & 0 & 4 & 0 & 8 \\ 0 & 0 & 4 & 0 & 0 & 18 \\ 0 & 0 & 0 & 0 & 0 & 72 \\ 0 & 4 & 8 & 18 & 72 & 162 \end{pmatrix}$$

```

Although there are no 3×3 tilings by the L-triomino and only two by the straight triomino, together they manage to tile in 10 different ways.

```
In[89]:= tile[{3, 3}, polyominoes[3], 5]
```

From In[89]:=



In general, an $n \times m$ ($m, n \geq 2$) rectangle can be tiled with the L-triomino whenever m is divisible by 3, except when one is 3 and the other is odd [9]. Here are some more general results corresponding to other cases.

- An L-tetromino tiles an $n \times m$ rectangle iff $n, m > 1$ and 8 divides $n m$.
- The T-tetromino tiles an $n \times m$ rectangle iff 4 divides both n and m .
- If a rectangle can be tiled with k L-tetrominoes, then k is even.
- An $n \times m$ rectangle can be tiled by the straight p -omino iff p divides n or p divides m .

The next two examples show the time needed to get one solution versus all solutions.

```
In[90]:= Timing[tile[{6, 11}, {{0, i, 2 i, 1 + 2 i, 3 i, 1 + 3 i}}, 4]]
```

```
Out[90]= {20.359 Second, Null}
```

```
In[91]:= Timing[tile[{6, 11}, {{0, i, 2 i, 1 + 2 i, 3 i, 1 + 3 i}}, 1, True]]
```

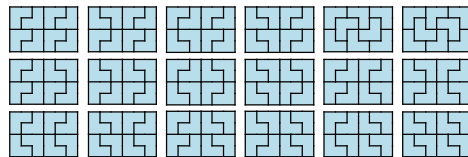
```
Out[91]= {1.219 Second, Null}
```

□ Fault-Free Tilings

As the number of different solutions grows, we would like to consider only those that have not appeared in smaller instances of the problem. For example, the tilings of a 4×6 rectangle by the L-triomino include all those appearing when tiling two separate 2×6 rectangles.

```
In[92]:= tile[{4, 6}, {{0, 1, i}}, 6]
```

From In[92]:=



In tiling a rectangle, we might generate fault lines. A fault line is any horizontal or vertical line that divides a tiling so that it can be regarded as the union of the tilings of two subrectangles [10]. The function `tileNF` computes all fault-free tilings of an $n \times m$ rectangle by a given set of pieces `poly`. As before, the graphic

array of the solutions is considered to be of a given width r . It generates the whole set of tilings and then selects the fault-free tilings, with the help of the predicate `noFaultLineQ`. This last function computes the length of the horizontal and vertical lines present in the tiling and verifies that no row or column has m or n elements, respectively.

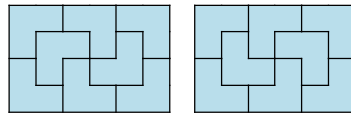
```
In[93]:= noFaultLineQ[{n_, m_}, tiling_] := Module[{l, h, v, c},
  l = getLines[tiling];
  h = Table[Length[
    Select[l, Im[First[#]] == Im[Last[#]] == c &]], {c, n - 1}];
  v = Table[Length[Select[l, Re[First[#]] == Re[Last[#]] == c &]],
    {c, m - 1}];
  Not[MemberQ[h, m] || MemberQ[v, n]]

tileNF[{n_, m_}, poly_, r_] := Module[{t, u, g},
  t = Select[tess[{n, m}, poly], noFaultLineQ[{n, m}, #] &];
  g = Map[Graphics[Append[{{LightBlue, Rectangle[{0, 0}, {m, n}]},
    Line[{{0, n}, {0, 0}, {m, 0}]}], liC/@getLines[#]]] &, t];
  Show[GraphicsArray[Partition[If[Mod[Length[t], r] == 0,
    g, Join[g, Table[Graphics[Point[{0, 0}]],
    {r - Mod[Length[t], r}]]]], r]]];]
```

We use this predicate to select the tilings without fault lines appearing in the previous example.

```
In[95]:= tileNF[{4, 6}, {{0, 1, i}}, 2]
```

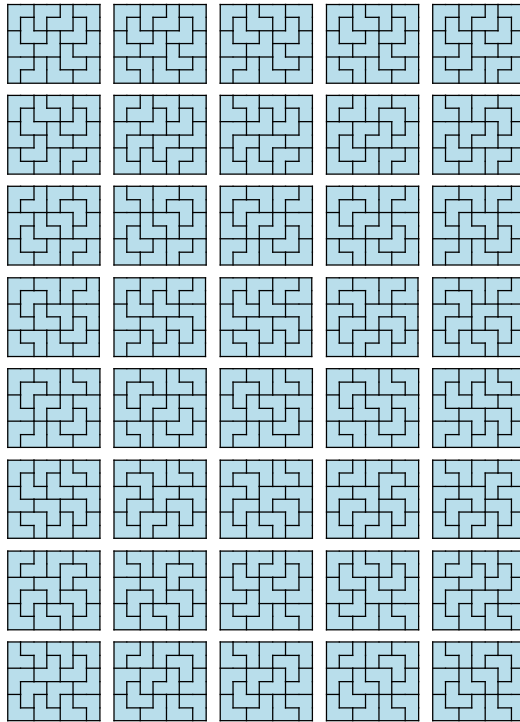
From In[95]:=



A more abundant example shows all fault-free tilings of a 6×7 rectangle using the L-triomino.

```
In[96]:= tileNF[{6, 7}, {Last[polyominoes[3]]}, 5]
```

```
From In[96]:=
```



Do tilings with dominoes always have fault lines? Not always.

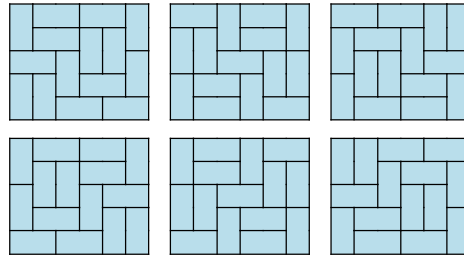
```
In[97]:= Timing[u = Table[0, {7}, {7}];
Do[If[Mod[n m, 2] == 0,
  u[[n,m]] = u[[m,n]] = Length[Select[tess[{n, m}, polyominoes[2]],
    noFaultLineQ[{n, m}, #] &]], {n, 7}, {m, n, 7}];
MatrixForm[u]]
```

```
Out[97]= {425.109 Second,
          {
            {0 1 0 0 0 0 0},
            {1 0 0 0 0 0 0},
            {0 0 0 0 0 0 0},
            {0 0 0 0 0 0 0},
            {0 0 0 0 0 6 0},
            {0 0 0 0 6 0 124},
            {0 0 0 0 0 124 0}
          }
}
```

Here are the smallest counterexamples.

```
In[98]:= tileNF[{5, 6}, polyominoes[2], 3]
```

```
From In[98]:=
```



Fault-free tilings produced by 3-ominoes are also interesting.

```
In[99]:= Timing[u = Table[0, {6}, {6}];
Do[If[Mod[n m, 3] == 0,
  u[[n,m]] = Length[Select[tess[{n, m}, polyominoes[3]],
    noFaultLineQ[{n, m}, #] &]], {n, 6}, {m, n, 6}];
MatrixForm[u]
```

```
Out[99]:= {1525.34 Second,

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 2 \\ 1 & 2 & 0 & 2 & 2 & 0 \\ 0 & 0 & 2 & 0 & 0 & 236 \\ 0 & 0 & 2 & 0 & 0 & 2060 \\ 0 & 2 & 0 & 236 & 2060 & 6312 \end{pmatrix}}$$

```

Like the tilings without restrictions from the previous section, the contributions of the individual pieces to this total are minuscule.

```
In[100]:= Timing[u = Table[0, {9}, {9}];
Do[If[Mod[n m, 3] == 0, u[[n,m]] =
  u[[m,n]] = Length[Select[tess[{n, m}, {First[polyominoes[3]}]],
    noFaultLineQ[{n, m}, #] &]], {n, 9}, {m, n, 9}];
MatrixForm[u]
```

```
Out[100]:= {1068.3 Second,

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 32 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 48 \\ 0 & 0 & 0 & 0 & 0 & 32 & 48 & 16 \end{pmatrix}}$$

```



```

In[101]:= Timing[u = Table[0, {6}, {6}];
Do[If[Mod[n m, 3] == 0, u[[n,m]] =
      u[[m,n]] = Length[Select[tess[{n, m}, {Last[polyominoes[3]]}],
      noFaultLineQ[{n, m}, #] &]], {n, 6}, {m, n, 6}];
MatrixForm[u]

```

Out[101]= {3. Second,
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 2 & 8 & 2 \end{pmatrix}$$
}

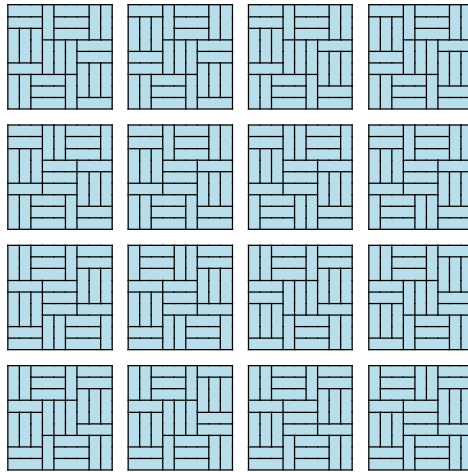
The smallest examples of fault-free tilings of squares by each polyomino produce patterns of striking beauty.

```

In[102]:= tileNF[{9, 9}, {First[polyominoes[3]]}, 4]

```

From In[102]:=

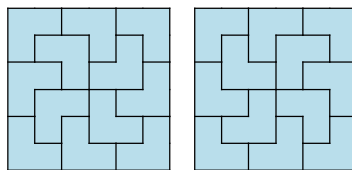


```

In[103]:= tileNF[{6, 6}, {Last[polyominoes[3]]}, 2]

```

From In[103]:=



□ Jigsaws

Let us now approach the problem of tiling an $n \times m$ rectangle using all the pieces of a list `poly` of given polyominoes once and only once. We naturally need to have `Length[Flatten[poly]] = n m`. The pieces appearing in `poly` do not need to be different and, with the inclusion of several equal ones, we can control their multiplicity. The functions `tessAll` and `jigsaw`, similar to our previous functions `tess` and `tile`, provide the necessary changes, which essentially amount to dropping a piece once used.

```
In[104]:= tessAll[{n_, m_}, poly_, justOne_: False] :=
Module[{avail, i, j, ans = {}, tessAux, na, ma},

tessAux[partial_, p_] :=
Module[{f, c, i, candidates, newp, pieces, q, k},
candidates = Complement[avail, Flatten[partial]];
If[{p == {} || (candidates == {})},
AppendTo[ans, partial]; If[justOne, Throw[1]],
k = First[lexic[candidates]];
Do[
q = lexic /@ allPieces[p[[i]];
Map[(
newp = k + # - First[#];
If[(Complement[newp, avail] == {}) && (f = Flatten[
{partial, newp}]; Length[f] == Length[Union[f]]),
tessAux[Append[partial, newp], Drop[p, {i}]]]
) &, q], {i, Length[p]}]]];

If[Length[Flatten[poly]] < n m, {},
{na, ma} = If[n < m, {m, n}, {n, m}];
avail = Flatten[Table[i + j i, {j, 0, na - 1}, {i, 0, ma - 1}]];
Catch[tessAux[{}, poly]];
If[n < m, Map[m - 1 + i # &, ans], ans]]]

jigsaw[{n_, m_}, poly_, r_, justOneSolution_: False] :=
Module[{t, u, g},
t = tessAll[{n, m}, poly, justOneSolution];
g = Map[Graphics[Append[{{LightBlue, Rectangle[{0, 0}, {m, n}]},
Line[{{0, n}, {0, 0}, {m, 0}]}], liC /@ getLines[#]]] &, t];
Show[GraphicsArray[Partition[If[Mod[Length[t], r] == 0,
g, Join[g, Table[Graphics[Point[{0, 0}]],
{r - Mod[Length[t], r]}]]], r]]];]

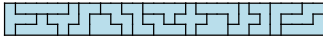
In[106]:= polyominoes[3]
Out[106]= {{0, i, 2 i}, {0, i, 1}}
```

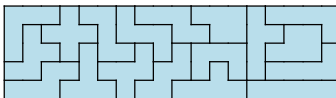
For instance, let us compute all possible tilings that also happen to be fault free using the set of pentominoes.

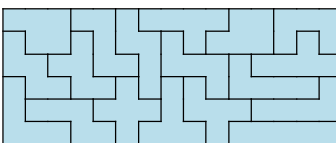
```

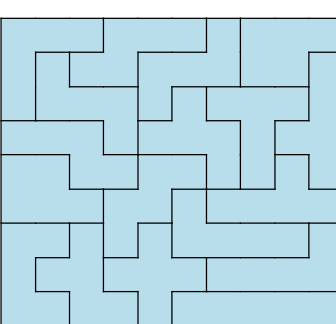
In[107]:= n = 5;
p = n Length[polyominoes[n]];
s = Select[Map[{#,  $\frac{p}{\#}$ } &, Divisors[p]], 1 < First[#] ≤ Last[#] &];
Timing[Map[jigsaw[#, polyominoes[n], 1, True] &, s];]

```

From In[107]:= 

From In[107]:= 

From In[107]:= 

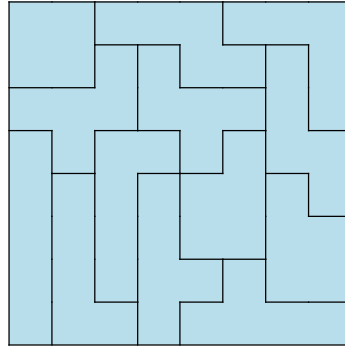
From In[107]:= 

Out[110]= {505.328 Second, Null}

We can generate a tiling of the 8×8 square with pentominoes by adding a square so that the sum of the areas is 64.

```
In[111]:= Timing[
  jigsaw[{8, 8}, Append[polyominoes[5], {0, 1, i, 1 + i}], 1, True]]
```

From In[111]:=



```
Out[111]= {209.859 Second, Null}
```

Unfortunately, the family of 35 hexominoes cannot tile a rectangle. In dealing with tilings with heptominoes, we have to consider that one of them has a hole and so we cannot tile a rectangle without a hole. For instance, in attempting to tile 12 8×8 squares (I thank the anonymous reviewer for this suggestion) in which a square has been removed from each one of them (so that their total area is 756, equal to the area covered by the 108 heptominoes), we just have to change the variable `avail` appearing in the function `tess` to construct the $8 \times 8 - 1$ squares appearing on the diagonal of a 96×96 square. This general approach takes too long and calls for another methodology, which the author hopes to report about in a future work.

■ Rep-Tiles

If a polyomino can be divided into a finite number of congruent copies similar to itself, we say that it is a *rep-tile*, or more specifically, an n -reptile. The so-called P-pentomino is a 4-reptile as we show later. Rep-tiles are self-similar pieces and so can be regarded as fractals. It can be proved that they tile the plane in a nonperiodic way [11], but it was only recently discovered, as reported by Roger Penrose in his fascinating and insightful book *Shadows of the Mind*, that there are three polyominoes that tile the plane only aperiodically [12].

```

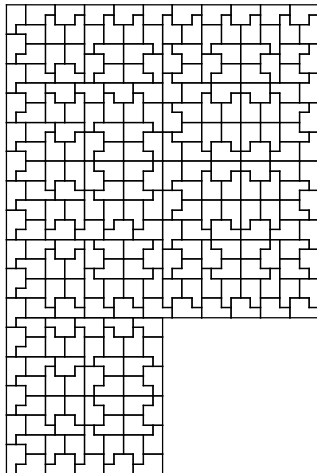
In[112]:= Ppentomino[n_] := repP[{2, 0}, {0, 0}, {0, 6}, n]

repP[a_, b_, c_, 1] :=
  {Line[{a, b, c, c + 2 (a - b),  $\frac{6a - 4b + c}{3}$ ,  $a + \frac{c - b}{3}$ , a]}]}
repP[a_, b_, c_, n_] :=
  Join[repP[ $\frac{a + c}{2}$ ,  $\frac{b + c}{2}$ , b, n - 1], repP[ $c + \frac{a - b}{2}$ , c,  $\frac{b + c}{2}$ , n - 1],
  repP[ $\frac{a + c}{2}$ ,  $\frac{3a + b + 2c}{6}$ ,  $\frac{6a - 4b + c}{3}$ , n - 1],
  repP[ $\frac{3a - 2b + 5c}{6}$ ,  $c + \frac{a - b}{2}$ , c + 2 (a - b), n - 1]]

In[115]:= Show[Graphics[Ppentomino[5]]]

```

From In[115]:=



The following L-triomino is also a 4-reptile and therefore also a 4^n -reptile for all $n > 0$.

```

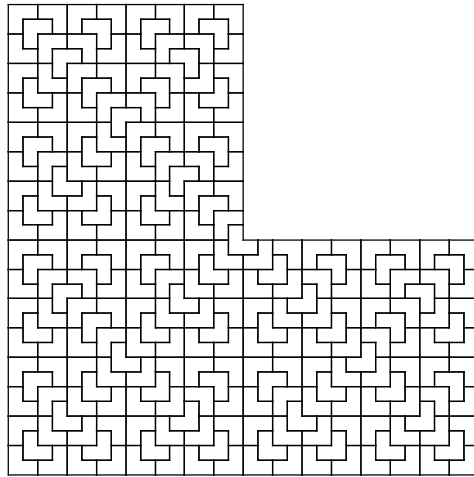
In[116]:= Ltriomino[n_] := repL[{2, 0}, {0, 0}, {0, 2}, n]

repL[a_, b_, c_, 1] := {Line[{a, b, c,  $c + \frac{a - b}{2}$ ,  $\frac{a + c}{2}$ ,  $a + \frac{c - b}{2}$ , a]}]}
repL[a_, b_, c_, n_] :=
  Join[repL[ $\frac{a + b}{2}$ , b,  $\frac{b + c}{2}$ , n - 1], repL[ $\frac{a + b}{2}$ , a,  $a + \frac{c - b}{2}$ , n - 1],
  repL[ $\frac{3a + c}{4}$ ,  $\frac{2b + a + c}{4}$ ,  $\frac{a + 3c}{4}$ , n - 1],
  repL[ $\frac{b + c}{2}$ , c,  $c + \frac{a - b}{2}$ , n - 1]]

```

```
In[119]:= Show[Graphics[Ltriomino[5]]]
```

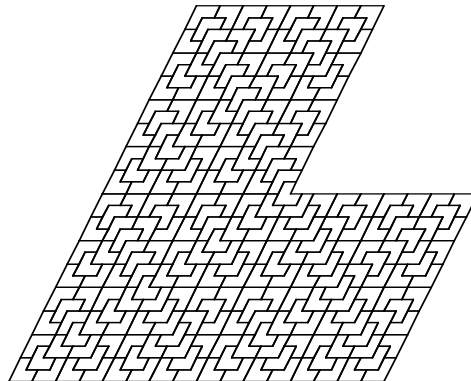
From In[119]:=



The value of c , given as an argument to the function `repL`, can be computed from those of a and b from $c = a + \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot (b - a)$. Giving it explicitly lets us distort the piece at will.

```
In[120]:= Show[Graphics[repL[{2, 0}, {0, 0}, {1, 2}, 5]]]
```

From In[120]:=



□ The Sphinx

Rep-tiles also arise in the shape of polyiamonds. The sphinx is one of the most widely known 4-reptile polyiamonds.

```

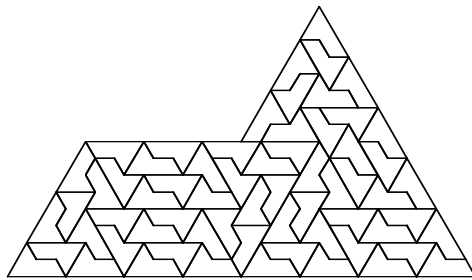
In[121]:= sphinx[n_] := sph[{0, 0}, {12, 0}, {8, 4√3}, n]

sph[a_, b_, c_, 1] := {Line[{a, b, c,  $\frac{2a+b+3c}{6}$ ,  $\frac{4a-b+3c}{6}$ , a}]}
sph[a_, b_, c_, n_] := Join[sph[ $\frac{a+b}{2}$ , a,  $\frac{4a-b+3c}{6}$ , n-1],
  sph[b,  $\frac{a+b}{2}$ ,  $\frac{a+2b+3c}{6}$ , n-1], sph[ $\frac{3b+c}{4}$ , c,  $\frac{2a+b+3c}{6}$ , n-1],
  sph[ $\frac{4a-b+3c}{6}$ ,  $\frac{a+2b+3c}{6}$ ,  $\frac{a+b}{2}$ , n-1]]

In[124]:= Show[Graphics[sphinx[4]]]

```

From In[124]:=



As indicated at the end of the previous section, more challenges are still present in the world of polyominoes. Even the seemingly simple task of finding out the number of tilings of an $n \times m$ rectangle using dominoes poses considerable difficulties (e.g., problem 7.51 in [7]). We can only guess as to the difficulty of these problems in the worlds inhabited by polyiamonds, polyhexes, and polykites. The advantages provided by the development of sophisticated languages like *Mathematica* yield a promising future for further investigations of this fascinating topic.

■ Acknowledgments

This work was completed while the author was a visiting scholar at Wolfram Research, Inc., whose assistance and enthusiastic support is gratefully acknowledged. I would also like to thank the University of Queretaro in Mexico for their continuous support.

■ References

- [1] S. W. Golomb, *Polyominoes: Puzzles, Patterns, Problems, and Packings*, 2nd ed., Princeton, NJ: Princeton University Press, 1996.
- [2] Eric W. Weisstein. "Polyomino." From *MathWorld*—A Wolfram Web Resource. <http://mathworld.wolfram.com/Polyomino.html>

- [3] "The Poly Pages." (Feb 6, 2004) www.geocities.com/alclarke0/index.htm.
- [4] M. Gardner, *Time Travel and Other Mathematical Bewilderments*, New York: W. H. Freeman & Co., 1987 pp. 175–187.
- [5] N. J. A. Sloane. "The On-Line Encyclopedia of Integer Sequences." (Sep 2, 2004) www.research.att.com/~njas/sequences.
- [6] N. J. A. Sloane and S. Plouffe, *The Encyclopedia of Integer Sequences*, San Diego: Academic Press, 1995.
- [7] R. L. Graham, O. Ptashnik, and D. E. Knuth, *Concrete Mathematics: A Foundation for Computer Science*, Reading, MA: Addison-Wesley, 1989.
- [8] T. P. Chu and R. Johnsonbaugh, "Tiling Boards with Triominoes," *Journal of Recreational Mathematics*, **18**, 1985–1986 pp. 183–193.
- [9] G. E. Martin, *Polyominoes: A Guide to Puzzles and Problems in Tiling*, Washington, DC: The Mathematical Association of America, 1996.
- [10] D. A. Klarner, "Polyominoes" *Handbook of Discrete and Computational Geometry*, Boca Raton, FL: CRC Press, 1997 pp. 225–239.
- [11] B. Brunbaum and G. C. Shephard, *Tilings and Patterns*, New York: W. H. Freeman & Co., 1987.
- [12] R. Penrose, *Shadows of the Mind: A Search for the Missing Science of Consciousness*, Oxford: Oxford University Press, 1994.

About the Author

Jaime Rangel-Mondragón earned M.Sc. and Ph.D. degrees in applied mathematics and computation from the University College of North Wales in Bangor, Great Britain. He has held research positions in the School of Computer Science at UCNW, the College of Mexico, the Center of Research and Advanced Studies, the Monterrey Institute of Technology, and the University of Querétaro in Mexico, where he is presently a member of the Computer Science faculty. As a prolific contributor to *MathSource*, his current research interests include recreational mathematics, combinatorics, the theory of computing, computational geometry, and functional languages.

Jaime Rangel-Mondragón

Facultad de Informática
Universidad Autónoma de Querétaro
Mexico
jrangel@uaq.mx