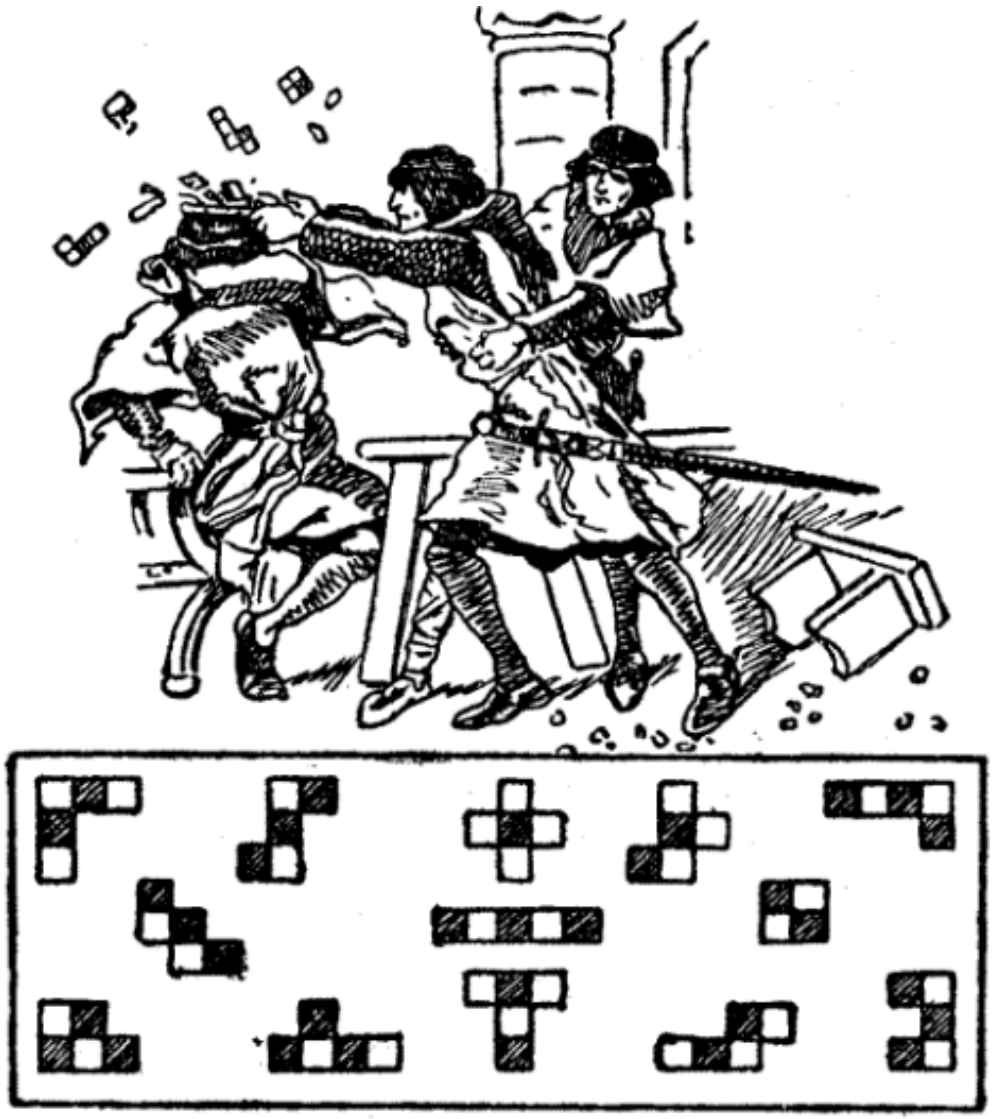
Reduced Row Echelon Form for Matrices
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

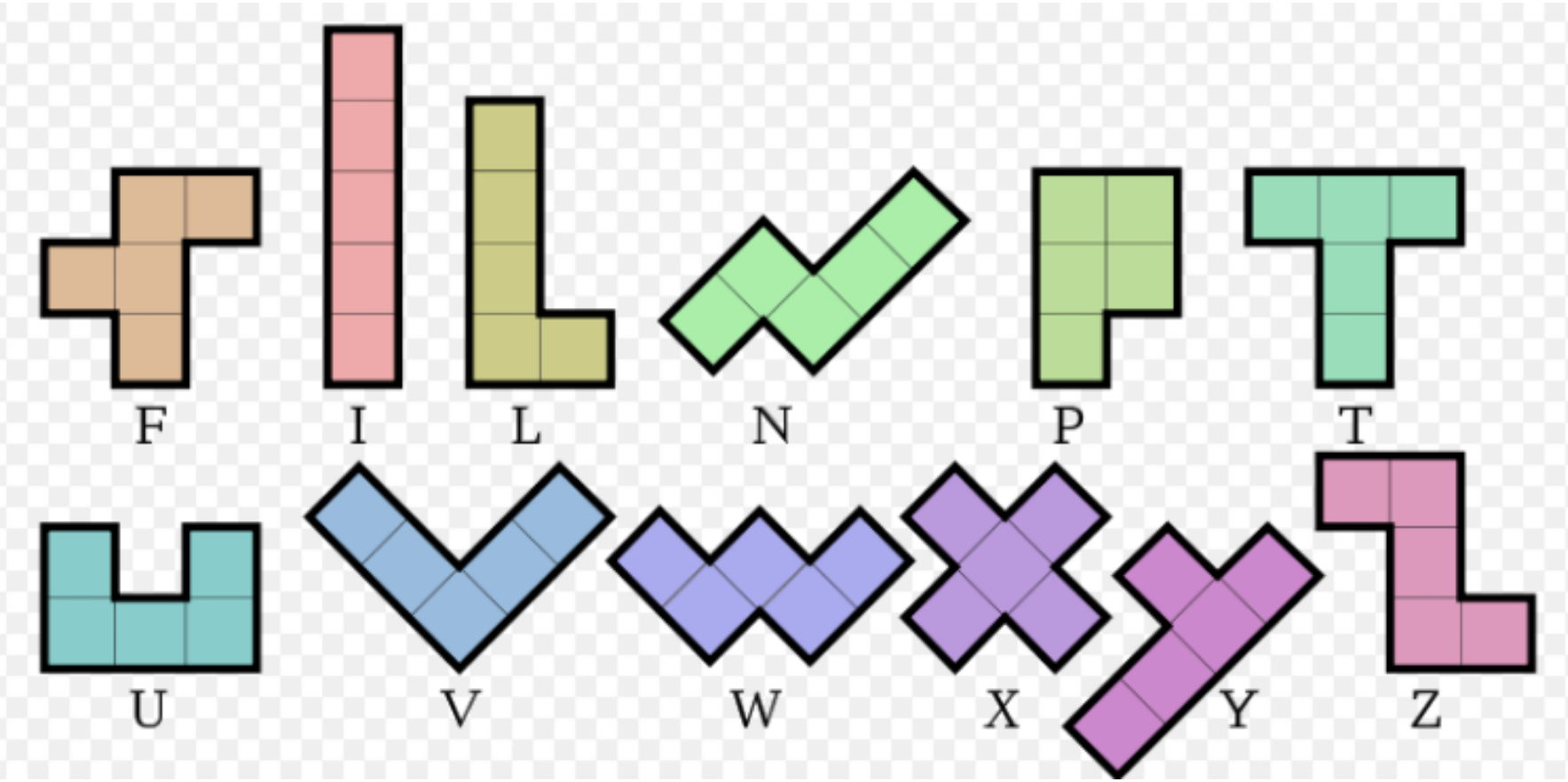
https://people.sc.fsu.edu/~jburkardt/classes/python_2025/tiling/tiling.pdf

The Broken Chessboard (Dudeney, 1919)



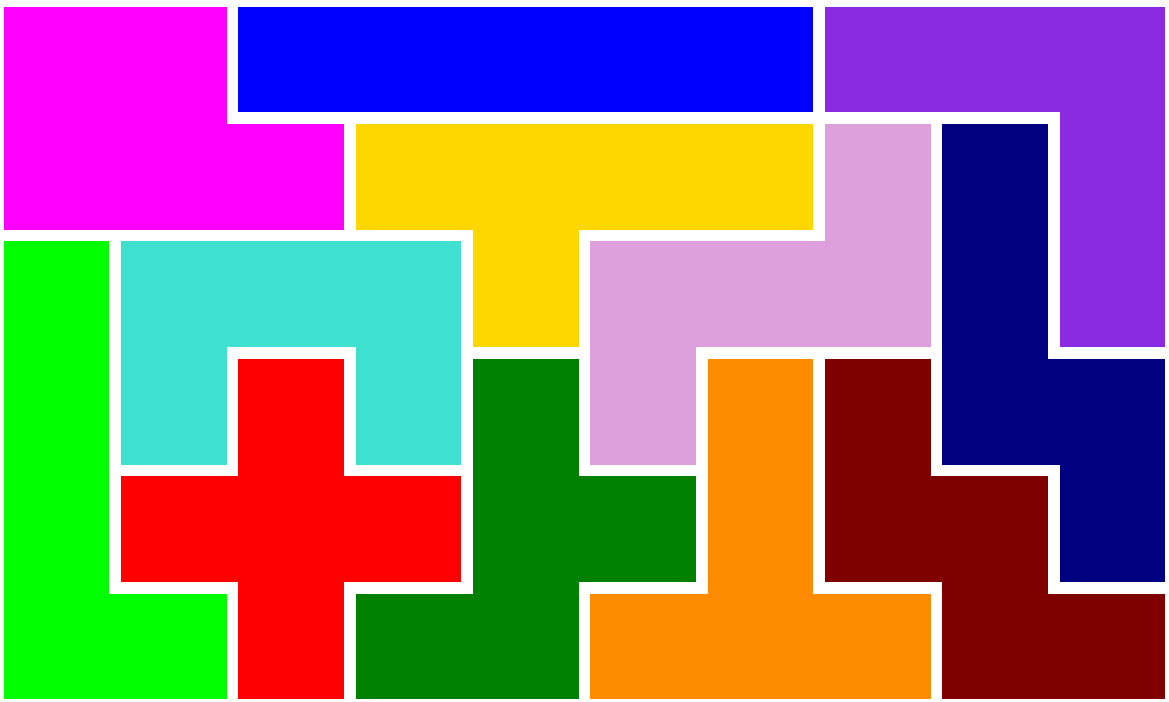
Pentominoes & Polyominoes (Golomb, 1965)

Can we tile a 6x10 rectangle with the 12 pentominos?

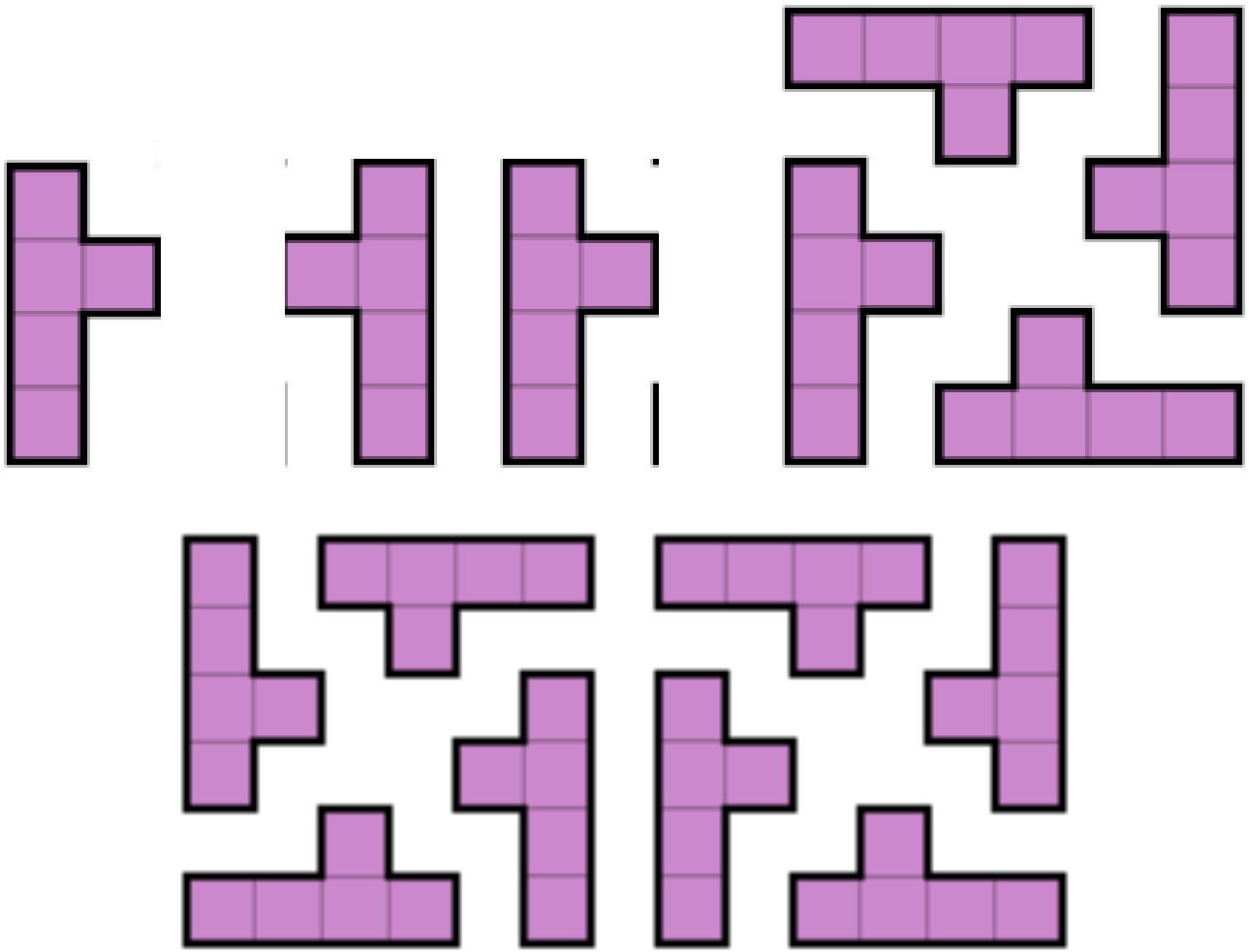


OK, But Is It Math?

"One can guess that there are several tilings of a 6×10 *rectangle using the twelve pentominoes. However, one might not predict just how many there are. An exhaustive computer search has found that there are 2339 such tilings. These questions make nice puzzles, but are* **not the kind of interesting mathematical problem** *that we are looking for."*
"Tilings" - Federico Ardila, Richard Stanley



Tile Variations: Reflections, Rotations, All Orientations



Exact Cover

Knuth: *"Tiling is a version of the exact cover problem."*

Add selected columns of a 0/1 matrix to form a column of 1's:

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

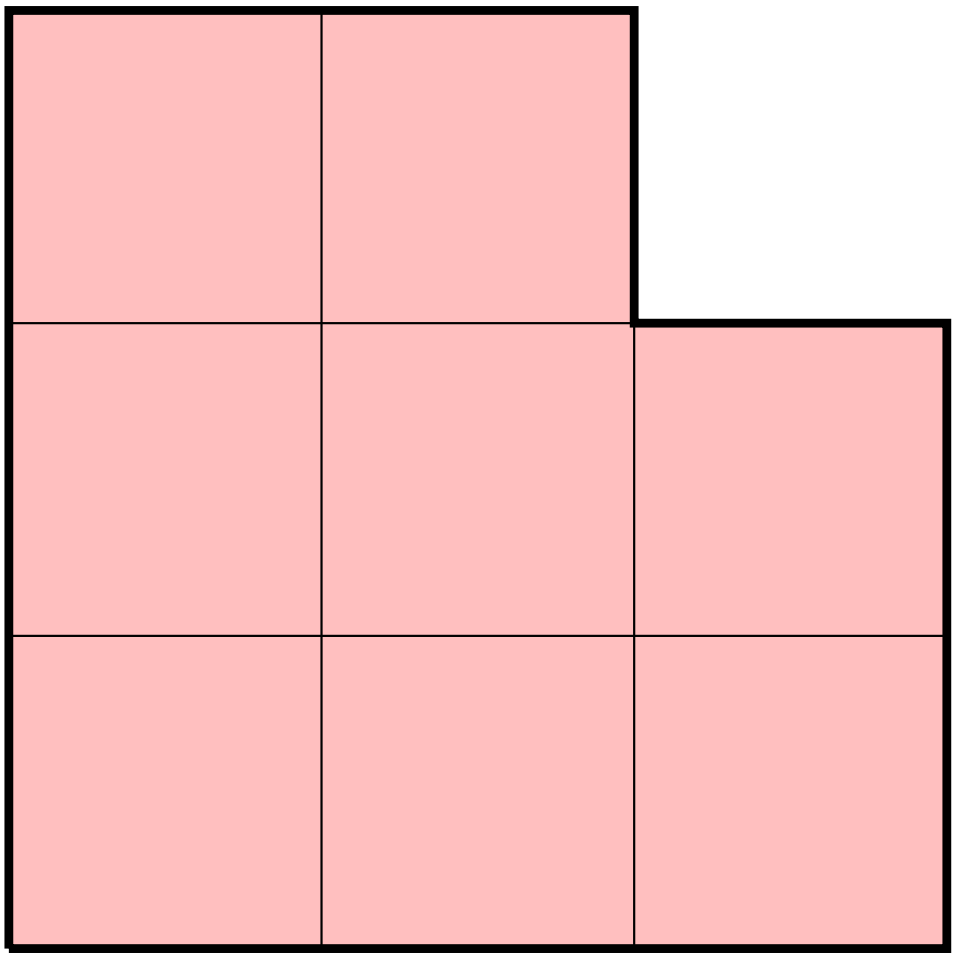
Exact Cover as a Linear System

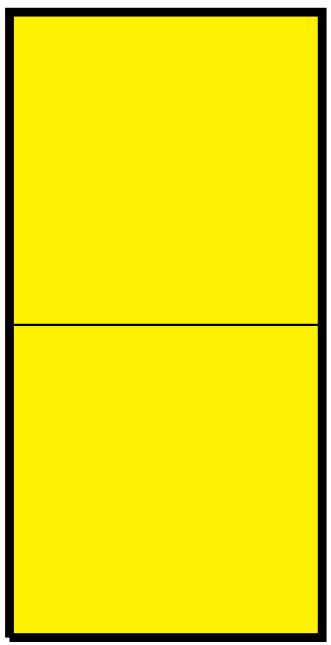
This is equivalent to
*Find x so that A*x=b where b is a vector of 1's.*

Now this is linear algebra!

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

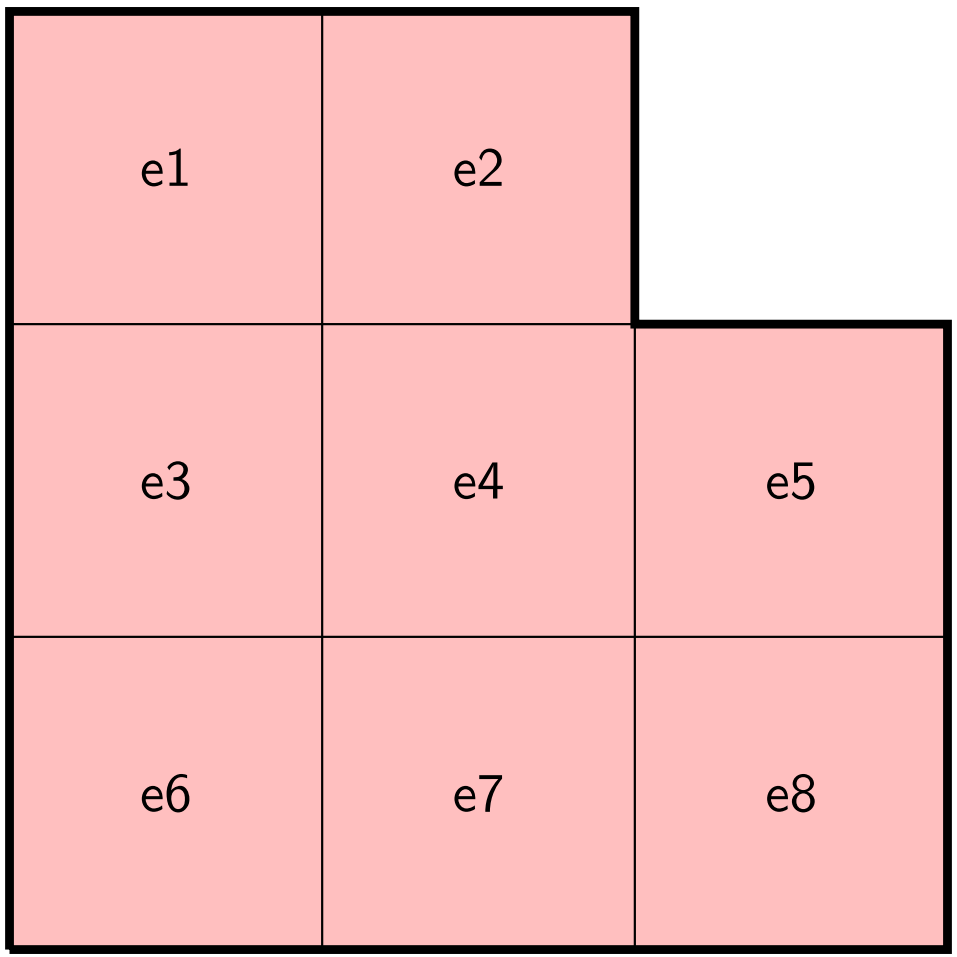
Tile the Reid Polygon (pink) with 4 Dominoes (yellow)





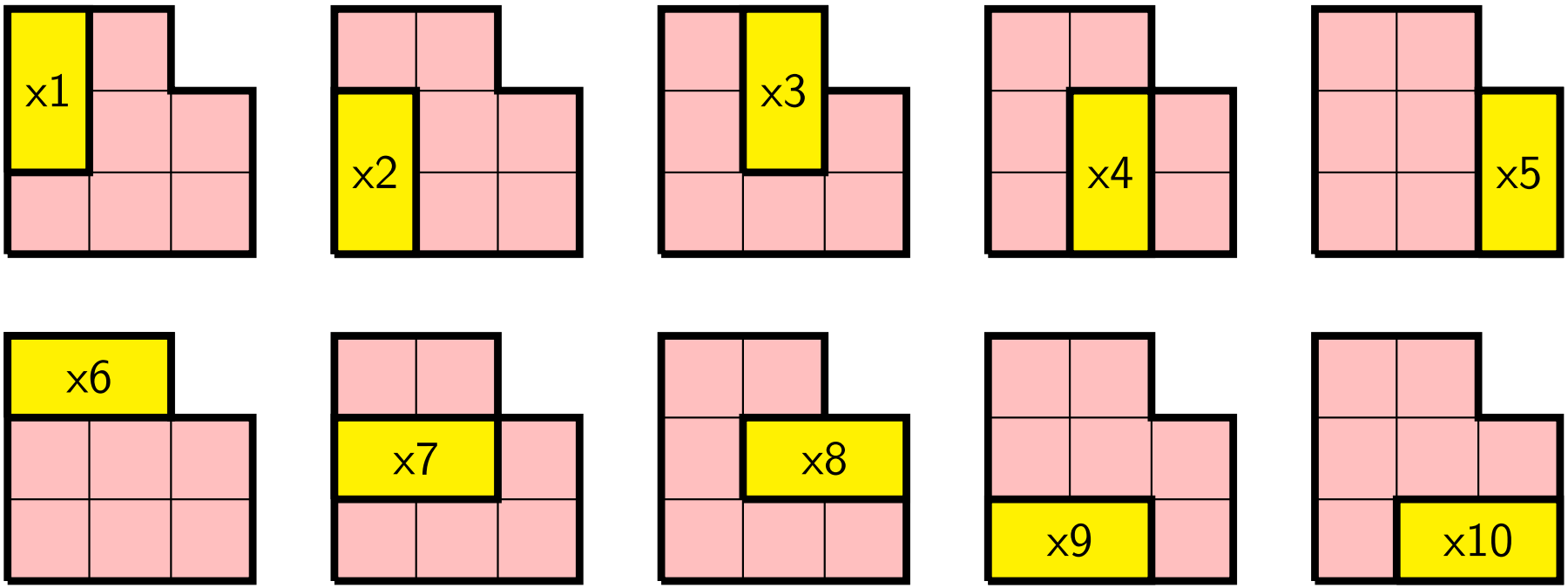
The Reid Equations: $e_1 : e_8$

The number of tiles above each square e_i must be exactly 1!



The Reid Variables $x_1 : x_{10}$

Each variable x_j is a particular orientation and position of a domino.



We start to see how the equations and variables combine:

$$x_1 + x_6 = 1 \text{ Cell 1 must be covered once}$$
$$x_3 + x_6 = 1 \text{ Cell 2 must be covered once}$$
$$x_1 + x_2 + x_7 = 1 \text{ Cell 3 must be covered once}$$
$$\ldots \quad \ldots$$

The Reid Equations: 8 Equations, 10 Unknowns

Eight equations (squares in Reid polygon) for 10 variables (position and orientation of a domino)

$$
\begin{array}{lrrrrrrrrrrcc}
 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & & b \\
e_1: & x_1 & & & & & +x_6 & & & & & = & 1 \\
e_2: & & & +x_3 & & & +x_6 & & & & & = & 1 \\
e_3: & x_1 & +x_2 & & & & & +x_7 & & & & = & 1 \\
e_4: & & & x_3 & +x_4 & & & +x_7 & +x_8 & & & = & 1 \\
e_5: & & & & & x_5 & & & +x_8 & & & = & 1 \\
e_6: & & x_2 & & & & & & & +x_9 & & = & 1 \\
e_7: & & & & x_4 & & & & & +x_9 & +x_{10} & = & 1 \\
e_8: & & & & & x_5 & & & & & +x_{10} & = & 1
\end{array}
$$

Python set up of linear system

```
  A = np.array ( [ \
     [1,0,0,0,0,1,0,0,0,0], \
     [0,0,1,0,0,1,0,0,0,0], \
     [1,1,0,0,0,0,1,0,0,0], \
     [0,0,1,1,0,0,1,1,0,0], \
     [0,0,0,0,1,0,0,1,0,0], \
     [0,1,0,0,0,0,0,0,1,0], \
     [0,0,0,1,0,0,0,0,1,1], \
     [0,0,0,0,1,0,0,0,0,1] ] )
#
#  We need right hand side to be a column vector.
#
  b = np.array (  [ [1],[1],[1],[1],[1],[1],[1],[1]] )
#
#  Append b to A, and get RREF.
#
  Ab = np.hstack ( [ A, b ] )

  RREF = rref_compute ( Ab )
```

Reduced Row Echelon Form of Reid Linear System

Notice that variables 7, 9 and 10 are free!

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}		b
e_1 :	1						1		-1		$=$	0
e_2 :		1							1		$=$	1
e_3 :			1				1		-1		$=$	0
e_4 :				1					1	1	$=$	1
e_5 :					1					1	$=$	1
e_6 :						1	-1		1		$=$	0
e_7 :								1		-1	$=$	0
e_8 :											$=$	0

Equation 8 disappears because once we have covered the first 7 cells, cell 8 is guaranteed to be covered.

Drop Zero Row, Add Degrees of Freedom

We add placeholder equations for variables 7, 9 and 10. Each of these free variables might be 0 (don't use) or 1 (use).

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}		b
e_1 :	1						1		-1		=	0
e_2 :		1							1		=	1
e_3 :			1				1		-1		=	0
e_4 :				1					1	1	=	1
e_5 :					1					1	=	1
e_6 :						1	-1		1		=	0
f_1 :							1				=	0?/1?
e_7 :								1		1	=	0
f_2 :									1		=	0?/1?
f_3 :										1	=	0?/1?

The RREF tells us which variables are free. If we choose arbitrary values for them, the RREF lets us solve the complete system.

Python modify RREF system

```
  m = RREF.shape[0]
  np1 = RREF.shape[1]
  n = np1 - 1

  C = np.eye ( n )
  b_RREF = np.zeros ( n )
  f = np.arange ( n )
#
#  Active rows of RREF replace corresponding rows of C.
#  f records the "free" variables.
#
  for i in range ( 0, m ):
    for j in range ( 0, n ):
      if ( RREF[i,j] == 1.0 ):
        C[j,:] = RREF[i,0:n]
        b_RREF[j] = RREF[i,-1]
        for k in range ( len ( f ) ):
          if ( f[k] == j ):
            f = np.delete ( f, k )
            break

        break
```

4 Acceptable Solutions (only 0 and 1 values)

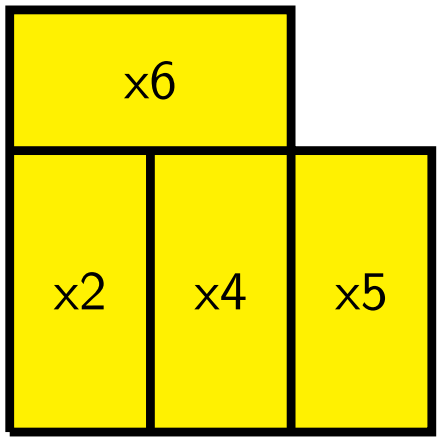
Each column is a solution, specifying which variables x_j to use in the tiling. Some of these are not legal solutions, since they involve stacking 2 tiles on top of each other, or using negative tiles!

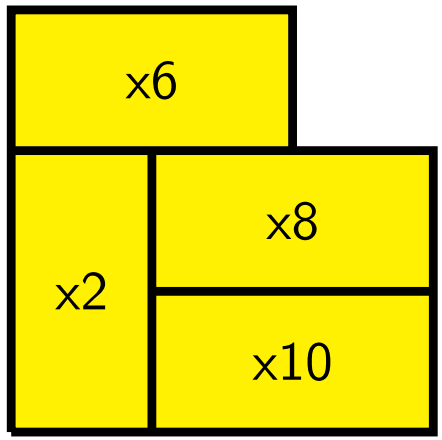
	✓	✓	✓	×	×	×	✓	×
x_1 :	0	0	1	1	−1	−1	0	0
x_2 :	1	1	0	0	1	1	0	0
x_3 :	0	0	1	1	−1	−1	0	0
x_4 :	1	0	0	−1	1	0	0	−1
x_5 :	1	0	1	0	1	0	1	0
x_6 :	1	1	0	0	2	2	1	1
x_7 :	0	0	0	0	1	1	1	1
x_8 :	0	1	0	1	0	1	0	1
x_9 :	0	0	1	1	0	0	1	1
x_{10} :	0	1	0	1	0	1	0	1

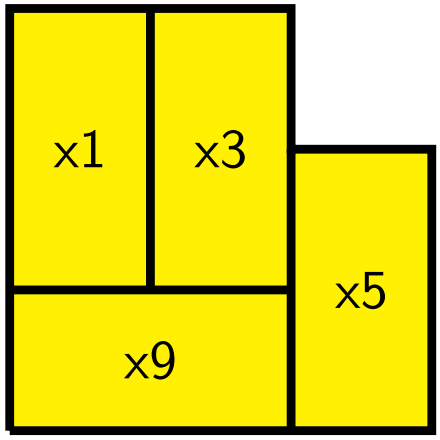
Python retrieve legal solutions (only 0 or 1)

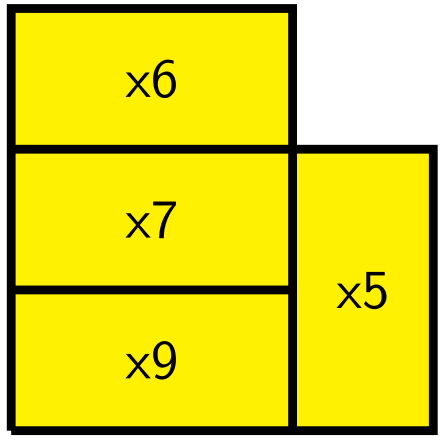
```
from itertools import combinations

for nz in range ( 0, 5 ):
  for combo in combinations ( f, nz ):
    bb = b_RREF.copy()
    for i in combo:
      bb[i] = 1
    x = np.linalg.solve ( C, bb )
    if ( np.all ( ( x == 0 ) | ( x == 1 ) ) ):
      print ( x )
```

The Reid Tilings (Labeled)







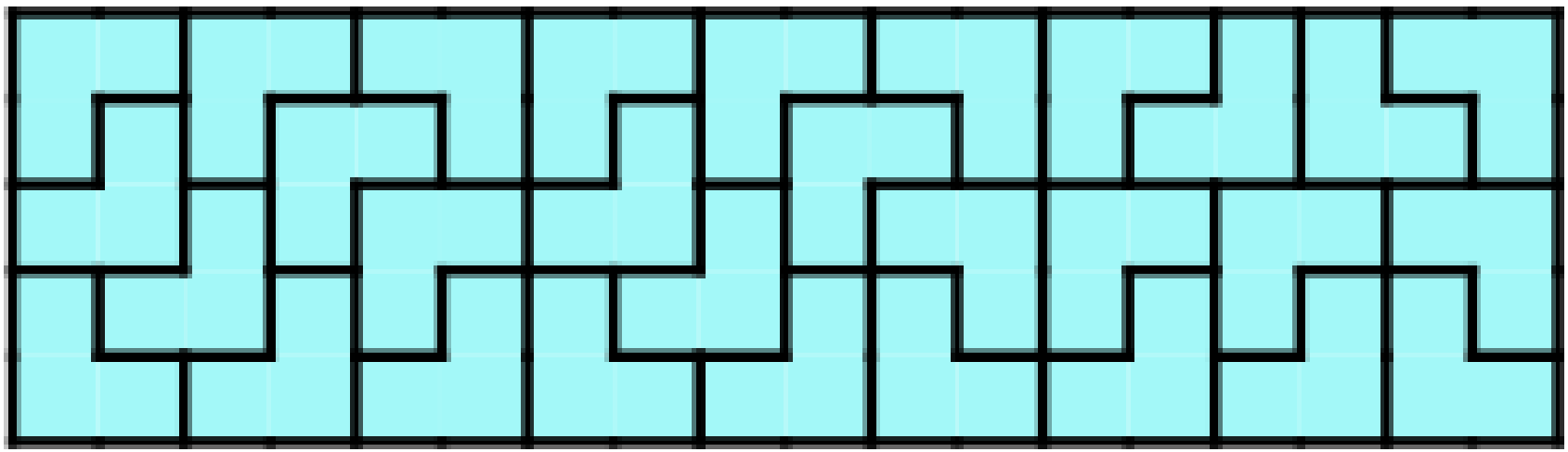


Bigger Problems Need a Better Solver

The Reid linear system A*x=b was 8 equations in 10 unknowns. It was easy to write a code to reduce A and b, via reduced row-echelon form; then to deal with the free variables, and then to eliminate solutions with unacceptable values. But for larger problems, this approach won't work.

- The row-reduced echelon form (RREF) is very sensitive to roundoff.
- Tiling regions can have hundreds of cells (equations/rows $= M$).
- Tiling problems can have tens or hundreds of tiles $= T$.
- A tile may have roughly M configurations, not even counting rotations and reflections (variables/columns $N \approx T * M$).
- The linear system may have many degrees of freedom D.
- The number of possible solutions we will need to check rises like 2^D.
- To solve interesting problems, need accurate, efficient integer linear programming solver;

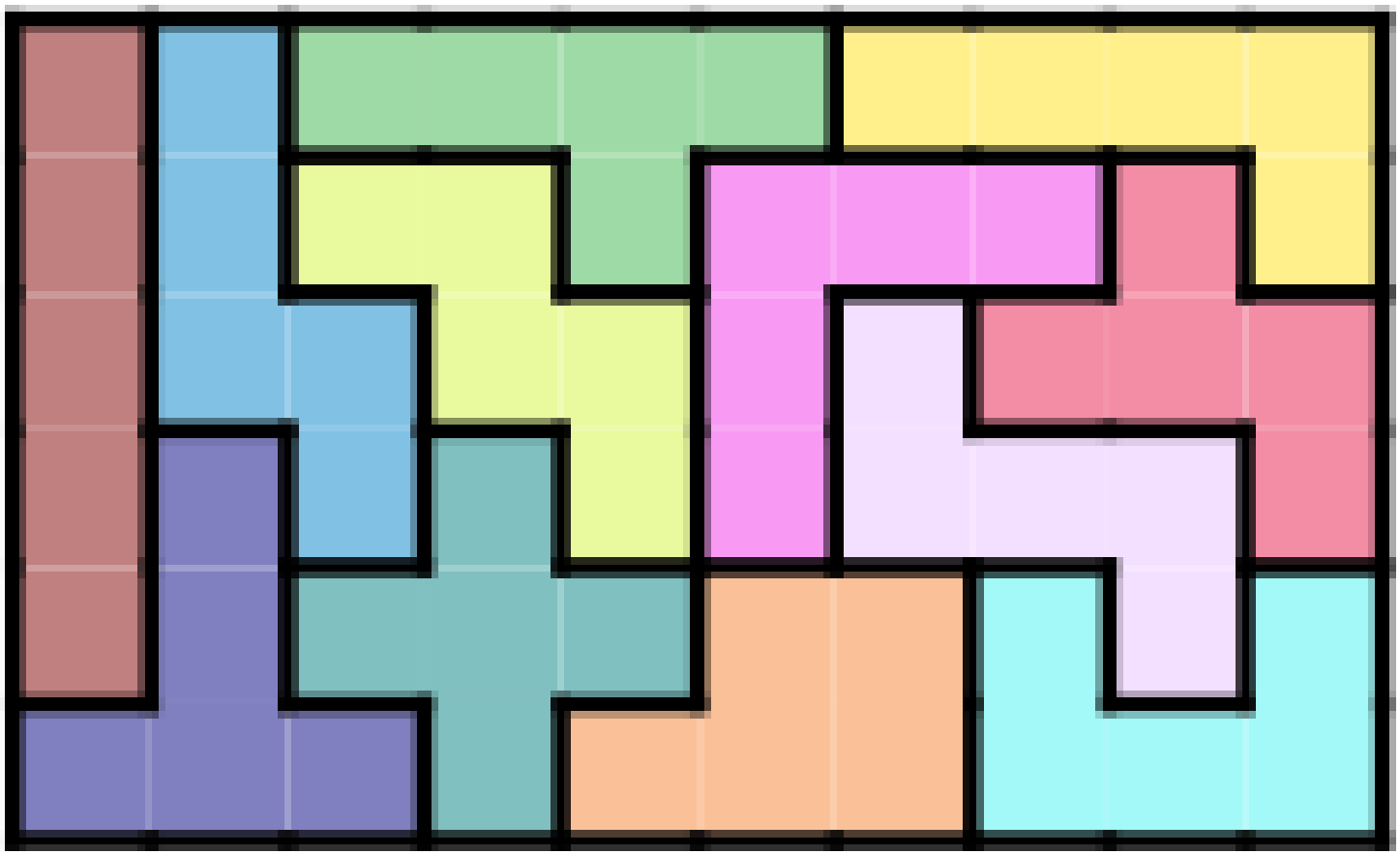
Tiling by Multiple Copies of a Single Polyomino



1 trimino, 4 orientations, 90+1 equations, 272 variables

1,168,512 solutions computed by CPLEX in 3.8 minutes.

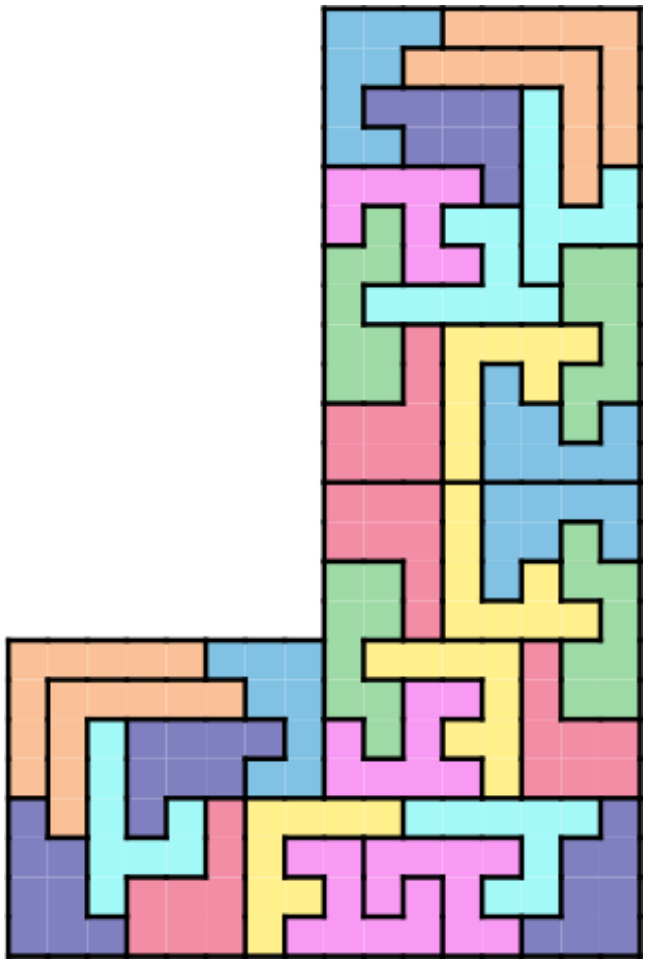
Tiling by One Copy of Each Pentomino



12 pentominoes, 1/2/4/8 orients, 60+12+1 equations, 2056 variables

9,356 solutions computed by CPLEX in 7.3 minutes.

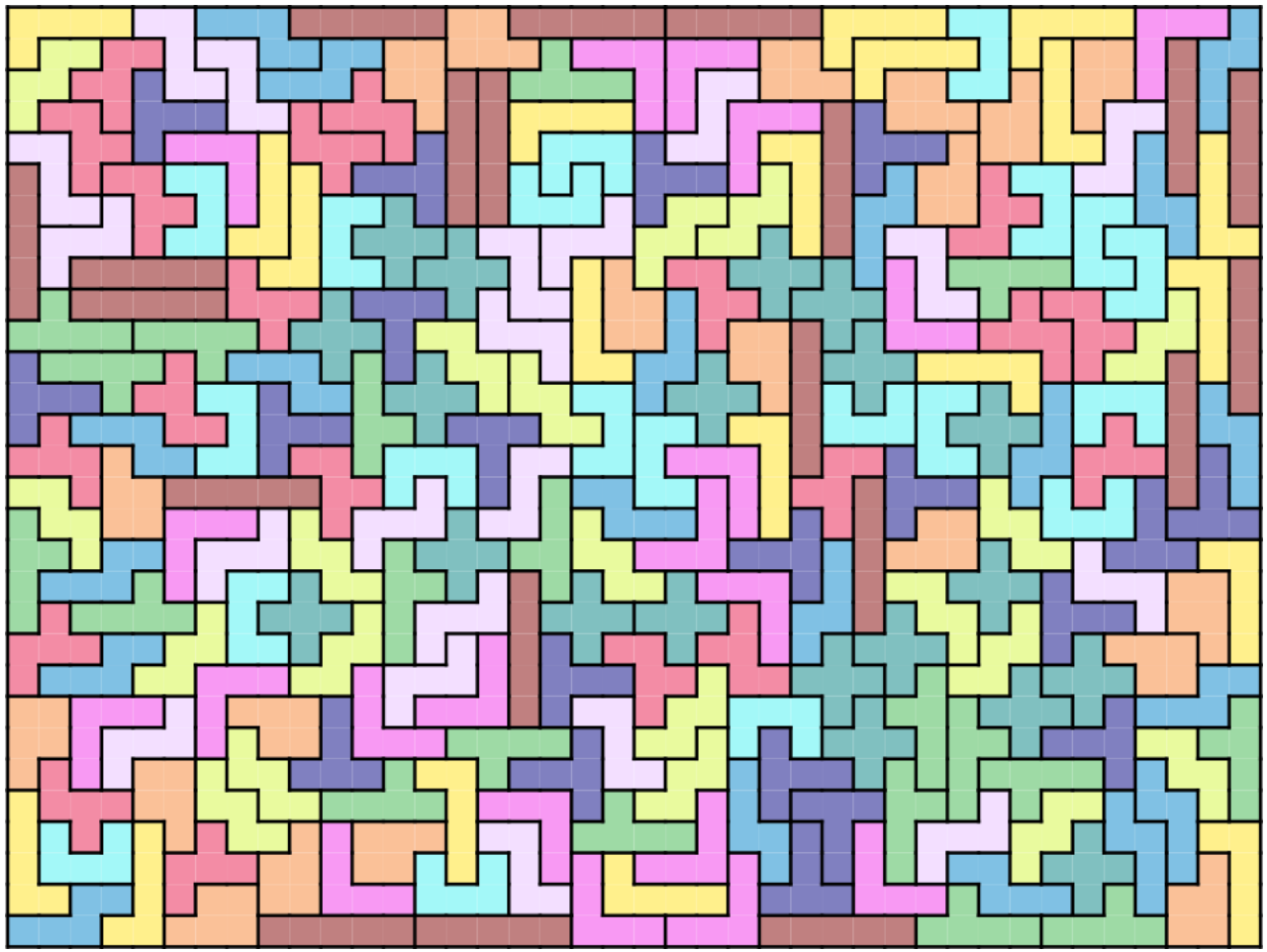
Tiling a Nonrectangular Region



8 octominoes, 4 copies each, 265 equations, 9,878 variables

1 solution computed by CPLEX in 13 minutes.

A Large Problem



12 pentominoes, 20 copies each, 1,213 equations, 67,396 variables

8 solutions computed by CPLEX in 9.5 minutes.

Conclusion: Rebuilding with Linear Algebra

To solve a tiling problem, we look for an underlying grid of cells that define both the region and the tiles. *This isn't always possible!*

- Equations: Each region cell must be covered, just once.
- Equations: Each tile must be used, just once.
- Variables: Each rotated, reflected, translated tile remaining in region
- Equations + Variables: **underdetermined linear system** $Ax = b$.
- **Reduced Row Echelon Form** lets us analyze the system.
- **Linear Programming Software** solves big systems.
- We seek **binary** vectors x whose entries are only 0 or 1.
- There may be no such solutions at all.
- If there are **free variables**, we may have multiple solutions.

Any solution x tells us exactly how to use the pieces so we can put a broken object back together...