start = Vec([x, y, 1])

matrix = Matrix(...)

next = start * matrix // * is matrix multiplication

Now, if your next is supposed to look something like Vec([a * x + x`, b * y + y`, 1]), we can work our way backwards to figure out the matrix. First, look at just the x component. We`re going to effectively take the dot product of our start vector and the topmost row of our matrix, yielding a * x + x`.

If we write it out more explicitly, we want a * x + 0 * y + x` * 1. Hopefully that makes it a bit more easy to see that the vector we want to dot start with is Vec([a, 0, x`]). We can repeat this for the remaining two rows of the matrix, and obtain the following matrix:

```
matrix = Matrix(
 [[a, 0, x'],
  [0, b, y'],
  [0, 0, 1]])
```

Double check that this makes sense and seems reasonable to you. If we take our start vector and multiply it with this matrix, we'll get the translated vector next as Vec([a * x + x', b * y + y', 1]).

Now for the real beauty of this- the matrix itself doesn't care at all about what our inputs are, its completely independent. So, we can repeatedly apply this matrix over and over again to step forward through more scaling and translations.

next_next_next = start * matrix * matrix * matrix

Knowing this, we can actually compute many steps ahead really quickly, using some mathematical tricks. Multiplying but the matrix n times is the same as multiplying by matrix raised to the nth power. And fortunately, we have an efficient method for computing a matrix to a power- its called exponentiation by squaring (actually applies to regular numbers as well, but here we're concerned with multiplying matrices, and the logic still applies). In a nutshell, rather than multiplying the number or matrix over and over again n times, we square it and multiply intermediate values by the original number / matrix at the right times, to very rapidly approach the desired power (in log(n) multiplications).