```
>>> print(solution)

 putty gnome slinky hoop shooter
--------------------------------
   133   240    150   55      90
```

Does this vector solve the equation? We can test it by computing the *residual vector* (often called the *residual*):

```
>>> residual = b - solution*M
```

If the solution were exact, the residual would be the zero vector. An easy way to see if the residual is *almost* the zero vector is to calculate the sum of squares of its entries, which is just its dot-product with itself:

```
>>> residual * residual
1.819555009546577e-25
```

About 10^{-25}, so zero for our purposes!

However, we cannot yet truly be confident we have penetrated the secrets of JunkCo. Perhaps the solution we have computed is not the only solution to the equation! More on this topic later.

Example 4.5.16: Continuing with Example 4.5.12 (Page 197), we use `solve(A,b)` to solve 5×5 *Lights Out* starting from a state in which only the middle light is on:

```
>>> s = Vec(b.D, {(2,2):one})
>>> sol = solve(B, s)
```

You can check that this is indeed a solution:

```
>>> B*sol == s
True
```

Here there is no issue of accuracy since elements of $GF(2)$ are represented precisely. Moreover, for this problem we don't care if there are multiple solutions to the equation. This solution tells us one collection of buttons to press:

```
>>> [(i,j) for (i,j) in sol.D if sol[i,j] == one]
[(4,0),(2,2),(4,1),(3,2),(0,4),(1,4),(2,3),(1,0),(0,1),(2,0),(0,2)]
```

4.6 Matrix-vector multiplication in terms of dot-products

We will also define matrix-vector product in terms of dot-products.

4.6.1 Definitions

Definition 4.6.1 (*Dot-Product* Definition of Matrix-Vector Multiplication): If M is an $R \times C$ matrix and $\boldsymbol{u}$ is a C-vector then $M * \boldsymbol{u}$ is the R-vector $\boldsymbol{v}$ such that $\boldsymbol{v}[r]$ is the dot-product of row r of M with $\boldsymbol{u}$.

Example 4.6.2: Consider the matrix-vector product

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 10 & 0 \end{bmatrix} * [3, -1]$$

The product is a 3-vector. The first entry is the dot-product of the first row, $[1, 2]$, with $[3, -1]$, which is $1 \cdot 3 + 2 \cdot (-1) = 1$. The second entry is the dot-product of the second row, $[3, 4]$, with $[3, -1]$, which is $3 \cdot 3 + 4 \cdot (-1) = 5$. The third entry is $10 \cdot 3 + 0 \cdot (-1) = 30$. Thus the product is $[1, 5, 30]$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 10 & 0 \end{bmatrix} * [3, -1] \quad = \quad [\quad [1,2] \cdot [3,-1], \quad [3,4] \cdot [3,-1], \quad [10,0] \cdot [3,-1] \quad] \quad = \quad [1, 5, 30]$$

Vector-matrix multiplication is defined in terms of dot-products with the columns.

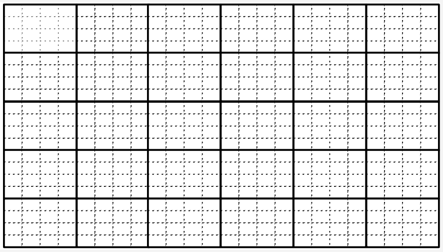
Definition 4.6.3 (*Dot-Product* Definition of Vector-Matrix Multiplication): If M is an $R \times C$ matrix and $\boldsymbol{u}$ is a R-vector then $\boldsymbol{u} * M$ is the C-vector $\boldsymbol{v}$ such that $\boldsymbol{v}[c]$ is the dot-product of $\boldsymbol{u}$ with column c of M.

4.6.2 Example applications

Example 4.6.4: You are given a high-resolution image. You would like a lower-resolution version to put on your web page so the page will load more quickly.

>>>
134 234 212
172 223
134 157
185 132
211

You therefore seek to *downsample* the image.



Each pixel of the low-res image (represented as a solid rectangle) corresponds to a little grid of pixels of the high-res image (represented as dotted rectangles). The intensity value of a pixel of the low-res image is the *average* of the intensity values of the corresponding pixels of the high-res image.

Let's represent the high-res image as a vector $\boldsymbol{u}$. We saw in Quiz 2.9.3 that averaging can be expressed as a dot-product. In downsampling, for each pixel of the low-res image to be created, the intensity is computed as the average of a subset of the entries of $\boldsymbol{u}$; this, too, can be expressed as a dot-product. Computing the low-res image thus requires one dot-product for each pixel of that image.

Employing the dot-product definition of matrix-vector multiplication, we can construct a matrix M whose rows are the vectors that must be dotted with $\boldsymbol{u}$. The column-labels of M are the pixel coordinates of the high-res image. The row-labels of M are the pixel coordinates of the low-res image. We write $\boldsymbol{v} = M * \boldsymbol{u}$ where $\boldsymbol{v}$ is a vector representing the low-res image.

Suppose the high-res image has dimensions 3000×2000 and our goal is to create a low-res image with dimensions 750×500. The high-res image is represented by a vector $\boldsymbol{u}$ whose domain is $\{0, 1, \ldots, 2999\} \times \{0, 1, \ldots, 1999\}$ and the low-res image is represented by a vector $\boldsymbol{v}$ whose domain is $\{0, 1, \ldots, 749\} \times \{0, 1, \ldots, 499\}$.

The matrix M has column-label set $\{0, 1, \ldots, 2999\} \times \{0, 1, \ldots, 1999\}$ and row-label set $\{0, 1, \ldots, 749\} \times \{0, 1, \ldots, 499\}$. For each low-res pixel coordinate pair (i, j), the corresponding row of M is the vector that is all zeroes except for the 4×4 grid of high-res pixel coordinates

$$(4i, 4j), (4i, 4j+1), (4i, 4j+2), (4i, 4j+3), (4i+1, 4j), (4i+1, 4j+1), \ldots, (4i+3, 4j+3)$$

where the values are $\frac{1}{16}$.

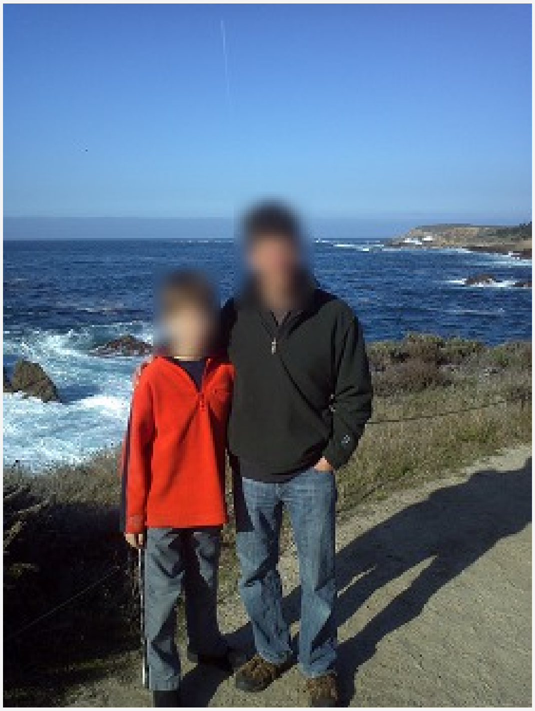
Here is the Python code to construct the matrix M.

```
D_high = {(i,j) for i in range(3000) for j in range(2000)}
D_low ={(i,j) for i in range(750) for j in range(500)}
M = Mat((D_low, D_high),
    {((i,j), (4*i+m, 4*j+n)):1./16 for m in range(4) for n in range(4)
                                 for i in range(750) for j in range(500)})
```

However, you would never actually want to create this matrix! I provide the code just for illustration.

Example 4.6.5: You are given an image and a set of pixel-coordinate pairs forming regions in the image, and you wish to produce a version of the image in which the regions are blurry.

Perhaps the regions are faces, and you want to blur them to protect the subjects' privacy. Once again, the transformation can be formulated as matrix-vector multiplication $M * \boldsymbol{v}$. (Once again, there is no reason you would actually want to construct the matrix explicitly, but the existence of such a matrix is useful in quickly computing the transformation, as we will discuss in Chapter 10.)



This time, the input image and output image have the same dimensions. For each pixel that needs to be blurred, the intensity is computed as an average of the intensities of many nearby pixels. Once again, we use the fact that average can be computed as a dot-product and that matrix-vector multiplication can be interpreted as carrying out many dot-products, one for each row of the matrix.

Averaging treats all nearby pixels equally. This tends to produce undesirable visual artifacts and is not a faithful analogue of the kind of blur we see with our eyes. A *Gaussian* blur more heavily weights very nearby pixels; the weights go down (according to a specific formula) with distance from the center.

Whether blurring is done using simple averaging or weighted averaging, the transformation is an example of a *linear filter*, as mentioned in Section 2.9.3.

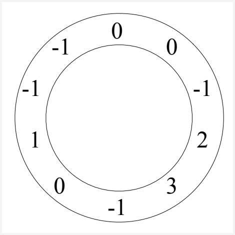
Example 4.6.6: As in Section 2.9.3, searching for an audio clip within an audio segment can be formulated as finding many dot-products, one for each of the possible locations of the audio clip or subimage. It is convenient to formulate finding these dot-products as a matrix-vector product.

Supppose we are trying to find the sequence $[0, 1, -1]$ in the longer sequence

$$[0, 0, -1, 2, 3, -1, 0, 1, -1, -1]$$

We need to compute one dot-product for each of the possible positions of the short sequence within the long sequence. The long sequence has ten entries, so there are ten possible positions for the short sequence, hence ten dot-products to compute.

You might think a couple of these positions are not allowed since these positions do not leave enough room for matching all the entries of the short sequence. However, we adopt a *wrap-around* convention: we look for the short sequence starting at the end of the long sequence, and wrapping around to the beginning. It is exactly as if the long sequence were written on a circular strip.



We formulate computing the ten dot-products as a product of a ten-row matrix with the ten-element long sequence:

$$\begin{bmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * [0, 0, -1, 2, 3, -1, 0, 1, -1, -1]$$

The product is the vector $[1, -3, -1, 4, -1, -1, 2, 0, -1, 0]$. The second-biggest dot-product, 2, indeed occurs at the best-matching position, though the biggest dot-product, 5, occurs at a not-so-great match.

Why adopt the wrap-around convention? It allows us to use a remarkable algorithm to compute the matrix-vector product much more quickly than would seem possible. The *Fast Fourier Transform* (FFT) algorithm, described in Chapter 10, makes use of the fact that the matrix has a special form.

4.6.3 Formulating a system of linear equations as a matrix-vector equation

In Section 2.9.2, we defined a linear equation as an equation of the form $\boldsymbol{a} \cdot \boldsymbol{x} = \beta$, and we defined a system of linear equations as a collection of such equations:

$$\begin{aligned} \boldsymbol{a}_1 \cdot \boldsymbol{x} &= \beta_1 \\ \boldsymbol{a}_2 \cdot \boldsymbol{x} &= \beta_2 \\ &\vdots \\ \boldsymbol{a}_m \cdot \boldsymbol{x} &= \beta_m \end{aligned}$$

Using the dot-product definition of matrix-vector multiplication, we can rewrite this system of equations as a single matrix-vector equation. Let A be the matrix whose rows are $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_m$.

Let $\boldsymbol{b}$ be the vector $[\beta_1, \beta_2, \ldots, \beta_m]$. Then the system of linear equations is equivalent to the matrix-vector equation $A * \boldsymbol{x} = \boldsymbol{b}$.

Example 4.6.7: Recall that in Example 2.9.7 (Page 113) we studied current consumption of hardware components in sensor nodes. Define `D = {'radio', 'sensor', 'memory', 'CPU'}`. Our goal was to compute a `D`-vector that, for each hardware component, gives the current drawn by that component.

We have five test periods. For $i = 0, 1, 2, 3, 4$, there is a vector $\boldsymbol{duration}_i$ giving the amount of time each hardware component is on during test period i.

```
>>> D = {'radio', 'sensor', 'memory', 'CPU'}
>>> v0 = Vec(D, {'radio':.1, 'CPU':.3})
>>> v1 = Vec(D, {'sensor':.2, 'CPU':.4})
>>> v2 = Vec(D, {'memory':.3, 'CPU':.1})
>>> v3 = Vec(D, {'memory':.5, 'CPU':.4})
>>> v4 = Vec(D, {'radio':.2, 'CPU':.5})
```

We are trying to compute a `D`-vector `rate` such that
`v0*rate = 140`, `v1*rate = 170`, `v2*rate = 60`, `v3*rate = 170`, and `v4*rate = 250`
We can formulate this system of equations as a matrix-vector equation:

$$\begin{bmatrix} \texttt{v0} \\ \hline \texttt{v1} \\ \hline \texttt{v2} \\ \hline \texttt{v3} \\ \hline \texttt{v4} \end{bmatrix} * [x_0, x_1, x_2, x_3, x_4] = [140, 170, 60, 170, 250]$$

To carry out the computation in Python, we construct the vector

```
>>> b = Vec({0, 1, 2, 3, 4},{0: 140.0, 1: 170.0, 2: 60.0, 3: 170.0, 4: 250.0})
```

and construct a matrix A whose rows are `v0`, `v1`, `v2`, `v3`, and `v4`:

```
>>> A = rowdict2mat([v0,v1,v2,v3,v4])
```

Next we solve the matrix-vector equation `A*x=b`:

```
>>> rate = solve(A, b)
```

obtaining the vector
```
        Vec(D, {'radio':500, 'sensor':250, 'memory':100, 'CPU':300})
```

Now that we recognize that systems of linear equations can be formulated as matrix-vector equations, we can reformulate problems and questions involving linear equations as problems involving matrix-vector equations:

- *Solving a linear system* (Computational Problem 2.9.12) becomes *solving a matrix equation* (Computational Problem 4.5.13).

- The question *how many solutions are there to a linear system over* $GF(2)$ (Question 2.9.18), which came up in connection with attacking the authentication scheme (Section 2.9.7), becomes the question *how many solutions are there to a matrix-vector equation over* $GF(2)$.
- Computational Problem 2.9.19, *computing all solutions to a linear system over* $GF(2)$, becomes *computing all solutions to a matrix-vector equation over* $GF(2)$.

4.6.4 Triangular systems and triangular matrices

In Section 2.11, we described an algorithm to solve a triangular system of linear equations. We have just seen that a system of linear equations can be formulated as a matrix-vector equation. Let's see what happens when we start with a triangular system.

Example 4.6.8: Reformulating the triangular system of Example 2.11.1 (Page 130) as a matrix-vector equation, we obtain

$$\begin{bmatrix} 1 & 0.5 & -2 & 4 \\ 0 & 3 & 3 & 2 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 2 \end{bmatrix} * \boldsymbol{x} = [-8, 3, -4, 6]$$

Because we started with a triangular system, the resulting matrix has a special form: the first entry of the second row is zero, the first and second entries of the third row are zero, and the first and second and third entries of the fourth row are zero. Since the nonzero entries form a triangle, the matrix itself is called a *triangular matrix*.

Definition 4.6.9: An $n \times n$ *upper-triangular* matrix A is a matrix with the property that $A_{ij} = 0$ for $i > j$.

Note that the entries forming the triangle can be be zero or nonzero.

The definition applies to traditional matrices. To generalize to our matrices with arbitrary row- and column-label sets, we specify orderings of the label-sets.

Definition 4.6.10: Let R and C be finite sets. Let L_R be a list of the elements of R, and let L_C be a list of the elements of C. An $R \times C$ matrix A is *triangular* with respect to L_R and L_C if

$$A[L_R[i], L_C[j]] = 0$$

for $j > i$.

Example 4.6.11: The {a, b c}× {@, #, ?} matrix

	@	#	?
a	0	2	3
b	10	20	30
c	0	35	0

is triangular with respect to [a, b c] and [@, ?, #]. We can see this by reordering the rows and columns according to the list orders:

	@	?	#
b	10	30	20
a	0	3	2
c	0	0	35

To facilitate viewing a matrix with reordered rows and columns, the class `Mat` will provide a pretty-printing method that takes two arguments, the lists L_R and L_C:

```
>>> A = Mat(({'a','b','c'}, {'#',  '@', '?'}),
...         {('a','#'):2, ('a','?'):3,
...         ('b','@'):10, ('b','#'):20, ('b','?'):30,
...         ('c','#'):35})
>>>
>>> print(A)

        #  ?  @
      ----------
 a  |   2  3  0
 b  |  20 30 10
 c  |  35  0  0

>>> A.pp(['b','a','c'], ['@','?','#'])

        @  ?  #
      ----------
 b  |  10 30 20
 a  |   0  3  2
 c  |   0  0 35
```

Problem 4.6.12: (For the student with knowledge of graph algorithms) Design an algorithm that, for a given matrix, finds a list of a row-labels and a list of column-labels with respect to which the matrix is triangular (or report that no such lists exist).

4.6.5 Algebraic properties of matrix-vector multiplication

We use the dot-product interpretation of matrix-vector multiplication to derive two crucial properties. We will use the first property in the next section, in characterizing the solutions to a matrix-vector equation and in error-correcting codes.

Proposition 4.6.13: Let M be an $R \times C$ matrix.

- For any C-vector $\boldsymbol{v}$ and any scalar α,

$$M * (\alpha\, \boldsymbol{v}) = \alpha\, (M * \boldsymbol{v}) \tag{4.3}$$

- For any C-vectors $\boldsymbol{u}$ and $\boldsymbol{v}$,

$$M * (\boldsymbol{u} + \boldsymbol{v}) = M * \boldsymbol{u} + M * \boldsymbol{v} \tag{4.4}$$

Proof

To show Equation 4.3 holds, we need only show that, for each $r \in R$, entry r of the left-hand side equals entry r of the right-hand side. By the dot-product interpretation of matrix-vector multiplication,

- entry r of the left-hand side equals the dot-product of row r of M with $\alpha\boldsymbol{v}$, and
- entry r of the right-hand side equals α times the dot-product of row r of M with $\boldsymbol{v}$.

These two quantities are equal by the homogeneity of dot-product, Proposition 2.9.22.

The proof of Equation 4.4 is similar; we leave it as an exercise. $\square$

Problem 4.6.14: Prove Equation 4.4.

4.7 Null space

4.7.1 Homogeneous linear systems and matrix equations

In Section 3.6, we introduced homogeneous linear systems, i.e. systems of linear equations in which all right-hand side values were zero. Such a system can of course be formulated as a matrix-vector equation $A * \boldsymbol{x} = \boldsymbol{0}$ where the right-hand side is a zero vector.

Definition 4.7.1: The *null space* of a matrix A is the set $\{\boldsymbol{v} \;:\; A * \boldsymbol{v} = \boldsymbol{0}\}$. It is written Null A.

Since Null A is the solution set of a homogeneous linear system, it is a vector space (Section 3.4.1).