n-vector $\boldsymbol{a}_i$, a *challenge vector*, and sends it to the human. The human sends back a single bit β_i, which is supposed to be the dot-product of $\boldsymbol{a}_i$ and the password $\hat{\boldsymbol{x}}$, and Carole checks whether $\beta_i = \boldsymbol{a}_i \cdot \hat{\boldsymbol{x}}$. If the human passes enough trials, Carole concludes that the human knows the password, and allows the human to log in.

Example 2.9.17: The password is $\hat{\boldsymbol{x}} = 10111$. Harry initiates log-in. In response, Carole selects the challenge vector $\boldsymbol{a}_1 = 01011$ and sends it to Harry. Harry computes the dot-product $\boldsymbol{a}_1 \cdot \hat{\boldsymbol{x}}$:

$$\begin{array}{cccccccccccc} & 0 & & 1 & & 0 & & 1 & & 1 & & \\ \bullet & 1 & & 0 & & 1 & & 1 & & 1 & & \\ \hline & 0 & + & 0 & + & 0 & + & 1 & + & 1 & = & 0 \end{array}$$

and responds by sending the resulting bit $\beta_1 = 0$ back to Carole.

Next, Carole sends the challenge vector $\boldsymbol{a}_2 = 11110$ to Harry. Harry computes the dot-product $\boldsymbol{a}_2 \cdot \hat{\boldsymbol{x}}$:

$$\begin{array}{cccccccccccc} & 1 & & 1 & & 1 & & 1 & & 0 & & \\ \bullet & 1 & & 0 & & 1 & & 1 & & 1 & & \\ \hline & 1 & + & 0 & + & 1 & + & 1 & + & 0 & = & 1 \end{array}$$

and responds by sending the resulting bit $\beta_2 = 1$ back to Carole.

This continues for a certain number k of trials. Carole lets Harry log in if $\beta_1 = \boldsymbol{a}_1 \cdot \hat{\boldsymbol{x}}, \beta_2 = \boldsymbol{a}_2 \cdot \hat{\boldsymbol{x}}, \ldots, \beta_k = \boldsymbol{a}_k \cdot \hat{\boldsymbol{x}}$.

2.9.7 Attacking the simple authentication scheme

We consider how Eve might attack this scheme. Suppose she eavesdrops on m trials in which Harry correctly responds. She learns a sequence of challenge vectors $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_m$ and the corresponding response bits $\beta_1, \beta_2, \ldots, \beta_m$. *What do these tell Eve about the password?*

Since the password is unknown to Eve, she represents it by a vector-valued variable $\boldsymbol{x}$. Since Eve knows that Harry correctly computed the response bits, she knows that the following linear equations are true:

$$\begin{aligned} \boldsymbol{a}_1 \cdot \boldsymbol{x} &= \beta_1 \\ \boldsymbol{a}_2 \cdot \boldsymbol{x} &= \beta_2 \\ &\vdots \\ \boldsymbol{a}_m \cdot \boldsymbol{x} &= \beta_m \end{aligned} \tag{2.4}$$

Perhaps Eve can compute the password by using an algorithm for Computational Problem 2.9.12, solving a linear system! Well, perhaps she can find *some* solution to the system of equations but is it the correct one? We need to consider Question 2.9.11: does the linear system have a unique solution?

Perhaps uniqueness is too much to hope for. Eve would likely be satisfied if the number of solutions were not too large, as long as she could compute them all and then try them out one by one. Thus we are interested in the following Question and Computational Problem:

Question 2.9.18: *Number of solutions to a linear system over $GF(2)$*
How many solutions are there to a given linear system over $GF(2)$?

Computational Problem 2.9.19: *Computing all solutions to a linear system over $GF(2)$*
Find all solutions to a given linear system over $GF(2)$.

However, Eve has another avenue of attack. Perhaps even without precisely identifying the password, she can use her knowledge of Harry's response bits to derive the answers to future challenges! For which future challenge vectors $\boldsymbol{a}$ can the dot-products with $\boldsymbol{x}$ be computed from the m equations? Stated more generally:

Question 2.9.20: Does a system of linear equations imply any other linear equations? If so, what other linear equations?

We next study properties of dot-product, one of which helps address this Question.

2.9.8 Algebraic properties of the dot-product

In this section we introduce some simple but powerful algebraic properties of the dot-product. These hold regardless of the choice of field (e.g. $\mathbb{R}$ or $GF(2)$).

Commutativity When you take a dot-product of two vectors, the order of the two does not matter:

Proposition 2.9.21 (Commutativity of dot-product): $\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{v} \cdot \boldsymbol{u}$

Commutativity of the dot-product follows from the fact that scalar-scalar multiplication is commutative:

Proof

$$\begin{aligned}
[u_1, u_2, \ldots, u_n] \cdot [v_1, v_2, \ldots, v_n] &= u_1 v_1 + u_2 v_2 + \cdots + u_n v_n \\
&= v_1 u_1 + v_2 u_2 + \cdots + v_n u_n \\
&= [v_1, v_2, \ldots, v_n] \cdot [u_1, u_2, \ldots, u_n]
\end{aligned}$$

□

Homogeneity The next property relates dot-product to scalar-vector multiplication: multiplying one of the vectors in the dot-product is equivalent to multiplying the value of the dot-product.

Proposition 2.9.22 (Homogeneity of dot-product): $(\alpha\,\boldsymbol{u})\cdot\boldsymbol{v} = \alpha\,(\boldsymbol{u}\cdot\boldsymbol{v})$

Problem 2.9.23: Prove Proposition 2.9.22.

Problem 2.9.24: Show that $(\alpha\,\boldsymbol{u})\cdot(\alpha\,\boldsymbol{v}) = \alpha\,(\boldsymbol{u}\cdot\boldsymbol{v})$ is *not* always true by giving a counterexample.

Distributivity The final property relates dot-product to vector addition.

Proposition 2.9.25 (Dot-product distributes over vector addition): $(\boldsymbol{u}+\boldsymbol{v})\cdot\boldsymbol{w} = \boldsymbol{u}\cdot\boldsymbol{w} + \boldsymbol{v}\cdot\boldsymbol{w}$

Proof

Write $\boldsymbol{u} = [u_1,\ldots,u_n], \boldsymbol{v} = [v_1,\ldots,v_n]$ and $\boldsymbol{w} = [w_1,\ldots,w_n]$.

$$\begin{aligned}
(\boldsymbol{u}+\boldsymbol{v})\cdot\boldsymbol{w} &= ([u_1,\ldots,u_n]+[v_1,\ldots,v_n])\cdot[w_1,\ldots,w_n]\\
&= [u_1+v_1,\ldots,u_n+v_n]\cdot[w_1,\ldots,w_n]\\
&= (u_1+v_1)w_1+\cdots+(u_n+v_n)w_n\\
&= u_1w_1+v_1w_1+\cdots+u_nw_n+v_nw_n\\
&= (u_1w_1+\cdots+u_nw_n)+(v_1w_1+\cdots+v_nw_n)\\
&= [u_1,\ldots,u_n]\cdot[w_1,\ldots,w_n]+[v_1,\ldots,v_n]\cdot[w_1,\ldots,w_n]
\end{aligned}$$

□

Problem 2.9.26: Show by giving a counterexample that $(\boldsymbol{u}+\boldsymbol{v})\cdot(\boldsymbol{w}+\boldsymbol{x}) = \boldsymbol{u}\cdot\boldsymbol{w}+\boldsymbol{v}\cdot\boldsymbol{x}$ is *not* true.

Example 2.9.27: We first give an example of the distributive property for vectors over the

reals: $[27, 37, 47] \cdot [2, 1, 1] = [20, 30, 40] \cdot [2, 1, 1] + [7, 7, 7] \cdot [2, 1, 1]$:

$$\begin{array}{rccccccc} & 20 & & 30 & & 40 & & \\ \bullet & 2 & & 1 & & 1 & & \\ \hline & 20 \cdot 2 & + & 30 \cdot 1 & + & 40 \cdot 1 & = & 110 \end{array}$$

$$\begin{array}{rccccc} & 7 & & 7 & & 7 & \\ \bullet & 2 & & 1 & & 1 & \\ \hline & 7 \cdot 2 & + & 7 \cdot 1 & + & 7 \cdot 1 & = 28 \end{array}$$

$$\begin{array}{rccccccc} & 27 & & 37 & & 47 & & \\ \bullet & 2 & & 1 & & 1 & & \\ \hline & 27 \cdot 2 & + & 37 \cdot 1 & + & 47 \cdot 1 & = & 138 \end{array}$$

2.9.9 Attacking the simple authentication scheme, revisited

I asked in Section 2.9.7 whether Eve can use her knowledge of Harry's responses to some challenges to derive the answers to others. We address that question by using the distributive property for vectors over $GF(2)$.

Example 2.9.28: This example builds on Example 2.9.17 (Page 122). Carole had previously sent Harry the challenge vectors 01011 and 11110, and Eve had observed that the response bits were 0 and 1. Suppose Eve subsequently tries to log in as Harry, and Carole happens to send her as a challenge vector the sum of 01011 and 11110. Eve can use the distributive property to compute the dot-product of this sum with the password $\boldsymbol{x}$ even though she does not know the password:

$$\begin{array}{rcccc} (01011 + 11110) \cdot \boldsymbol{x} & = & 01011 \cdot \boldsymbol{x} & + & 11110 \cdot \boldsymbol{x} \\ & = & 0 & + & 1 \\ & = & & 1 & \end{array}$$

Since you know the password, you can verify that this is indeed the correct response to the challenge vector.

This idea can be taken further. For example, suppose Carole sends a challenge vector that is the sum of three previously observed challenge vectors. Eve can compute the response bit (the dot-product with the password) as the sum of the responses to the three previous challenge vectors.

Indeed, the following math shows that Eve can compute the right response to the sum of any number of previous challenges for which she has the right response:

$$\begin{array}{ll} \text{if} & \boldsymbol{a}_1 \cdot \boldsymbol{x} = \beta_1 \\ \text{and} & \boldsymbol{a}_2 \cdot \boldsymbol{x} = \beta_2 \\ \vdots & \vdots \\ \text{and} & \boldsymbol{a}_k \cdot \boldsymbol{x} = \beta_k \\ \text{then} & (\boldsymbol{a}_1 + \boldsymbol{a}_2 + \cdots + \boldsymbol{a}_k) \cdot \boldsymbol{x} = (\beta_1 + \beta_2 + \cdots + \beta_k) \end{array}$$

Problem 2.9.29: Eve knows the following challenges and responses:

challenge	response
110011	0
101010	0
111011	1
001100	1

Show how she can derive the right responses to the challenges 011101 and 000100.

Imagine that Eve has observed hundreds of challenges $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ and responses $\beta_1, \ldots, \beta_n$, and that she now wants to respond to the challenge $\boldsymbol{a}$. She must try to find a subset of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ whose sum equals $\boldsymbol{a}$.

Question 2.9.20 asks: Does a system of linear equations imply any other linear equations? The example suggests a partial answer:

$$\begin{array}{ll} \text{if} & \boldsymbol{a}_1 \cdot \boldsymbol{x} = \beta_1 \\ \text{and} & \boldsymbol{a}_2 \cdot \boldsymbol{x} = \beta_2 \\ \vdots & \vdots \\ \text{and} & \boldsymbol{a}_k \cdot \boldsymbol{x} = \beta_k \\ \text{then} & (\boldsymbol{a}_1 + \boldsymbol{a}_2 + \cdots + \boldsymbol{a}_k) \cdot \boldsymbol{x} = (\beta_1 + \beta_2 + \cdots + \beta_k) \end{array}$$

Therefore, from observing challenge vectors and the response bits, Eve can derive the response to any challenge vector that is the sum of *any subset* of previously observed challenge vectors.

That presumes, of course, that she can *recognize* that the new challenge vector can be expressed as such a sum, and determine which sum! This is precisely Computational Problem 2.8.7. We are starting to see the power of computational problems in linear algebra; the same computational problem arises in addressing solving a puzzle and attacking an authentication scheme! Of course, there are many other settings in which this problem arises.

2.10 Our implementation of `Vec`

In Section 2.7, we gave the definition of a rudimentary Python class for representing vectors, and we developed some procedures for manipulating this representation.

2.10.1 Syntax for manipulating `Vecs`

We will expand our class definition of `Vec` to provide some notational conveniences:

operation	syntax
vector addition	`u+v`
vector negation	`-v`
vector subtraction	`u-v`
scalar-vector multiplication	`alpha*v`
division of a vector by a scalar	`v/alpha`
dot-product	`u*v`
getting value of an entry	`v[d]`
setting value of an entry	`v[d] = ...`
testing vector equality	`u == v`
pretty-printing a vector	`print(v)`
copying a vector	`v.copy()`

In addition, if an expression has as a result a `Vec` instance, the value of the expression will be presented not as an obscure Python incantation

```
>>> v
<__main__.Vec object at 0x10058cad0>
```

but as an expression whose value is a vector:

```
>>> v
Vec({'A', 'B', 'C'},{'A': 1.0})
```

2.10.2 The implementation

In Problem 2.14.10, you will implement `Vec`. However, since this book is not about the intricacies of defining classes in Python, you need not write the class definition; it will be provided for you. All you need to do is fill in the missing bodies of some procedures, most of which you wrote in Section 2.7.

2.10.3 Using `Vec`s

You will write the bodies of named procedures such as `setitem(v, d, val)` and `add(u,v)` and `scalar_mul(v, alpha)`. However, in actually using Vecs in other code, you must use operators instead of named procedures, e.g.

```
>>> v['a'] = 1.0
```

instead of

```
>>> setitem(v, 'a', 1.0)
```

and

```
>>> b = b - (b*v)*v
```

instead of

```
>>> b = add(b, neg(scalar_mul(v, dot(b,v))))
```

In fact, in code outside the `vec` module that uses `Vec`, you will import just `Vec` from the `vec` module:

```
from vec import Vec
```

so the named procedures will not be imported into the namespace. Those named procedures in the `vec` module are intended to be used *only* inside the `vec` module itself.

2.10.4 Printing `Vecs`

The class `Vec` defines a procedure that turns an instance into a string for the purpose of printing:

```
>>> print(v)

 A B C
------
 1 0 0
```

The procedure for pretty-printing a vector `v` must select some order on the domain `v.D`. Ours uses `sorted(v.D, key=hash)`, which agrees with numerical order on numbers and with alphabetical order on strings, and which does something reasonable on tuples.

2.10.5 Copying `Vecs`

The `Vec` class defines a `.copy()` method. This method, called on an instance of `Vec`, returns a new instance that is equal to the old instance. It shares the domain `.D` with the old instance. but has a new function `.f` that is initially equal to that of the old instance.

Ordinarily you won't need to copy `Vecs`. The scalar-vector multiplication and vector addition operations return new instances of `Vec` and do not mutate their inputs.

2.10.6 From list to `Vec`

The `Vec` class is a useful way of representing vectors, but it is not the only such representation. As mentioned in Section 2.1, we will sometimes represent vectors by lists. A list L can be viewed as a function from $\{0, 1, 2, \ldots, \text{len}(L)-1\}$, so it is possible to convert from a list-based representation to a dictionary-based representation.

Quiz 2.10.1: Write a procedure `list2vec(L)` with the following spec:

- *input:* a list L of field elements
- *output:* an instance $\boldsymbol{v}$ of `Vec` with domain $\{0, 1, 2, \ldots, \text{len}(L) - 1\}$ such that $\boldsymbol{v}[i] = L[i]$ for each integer i in the domain

Answer

```
def list2vec(L):
```

```
  return Vec(set(range(len(L))), {k:x for k,x in enumerate(L)})
```
or
```
def list2vec(L):
  return Vec(set(range(len(L))), {k:L[k] for k in range(len(L))})
```

This procedure facilitates quickly creating small `Vec` examples. The procedure definition is included in the provided file `vecutil.py`.

2.11 Solving a triangular system of linear equations

As a step towards Computational Problem 2.9.12 (Solving a linear system), we describe an algorithm for solving a system if the system has a special form.

2.11.1 Upper-triangular systems

A *upper-triangular system of linear equations* has the form

$$\begin{array}{rrrrcrrcccc}
[& a_{11}, & a_{12}, & a_{13}, & a_{14}, & \cdots & a_{1,n-1}, & a_{1,n} &] & \cdot & \boldsymbol{x} = \beta_1 \\
[& 0, & a_{22}, & a_{23}, & a_{24}, & \cdots & a_{2,n-1}, & a_{2,n} &] & \cdot & \boldsymbol{x} = \beta_2 \\
[& 0, & 0, & a_{33}, & a_{34}, & \cdots & a_{3,n-1}, & a_{3,n} &] & \cdot & \boldsymbol{x} = \beta_3 \\
 & & & & & \vdots & & & & & \\
[& 0, & 0, & 0, & 0, & \cdots & a_{n-1,n-1}, & a_{n-1,n} &] & \cdot & \boldsymbol{x} = \beta_{n-1} \\
[& 0, & 0, & 0, & 0, & \cdots & 0, & a_{n,n} &] & \cdot & \boldsymbol{x} = \beta_n
\end{array}$$

That is,

- the first vector need not have any zeroes,
- the second vector has a zero in the first position,
- the third vector has zeroes in the first and second positions,
- the fourth vector has zeroes in the first, second, and third positions,

  $\vdots$

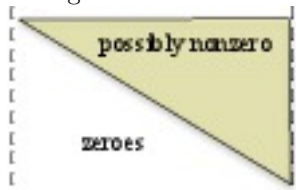
- the $n-1^{st}$ vector is all zeroes except possibly for the $n-1^{st}$ and n^{th} entries, and
- the n^{th} vector is all zeroes except possibly for the n^{th} entry.

Example 2.11.1: Here's an example using 4-vectors:

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

The right-hand sides are -8, 3, -4, and 6.

The origin of the term *upper-triangular system* should be apparent by considering the positions of the nonzero entries: they form a triangle:



Writing $\boldsymbol{x} = [x_1, x_2, x_3, x_4]$ and using the definition of dot-product, we can rewrite this system as four ordinary equations in the (scalar) unknowns x_1, x_2, x_3, x_4:

$$\begin{array}{rcrcrcrcr} 1x_1 & + & 0.5x_2 & - & 2x_3 & + & 4x_4 & = & -8 \\ & & 3x_2 & + & 3x_3 & + & 2x_4 & = & 3 \\ & & & & 1x_3 & + & 5x_4 & = & -4 \\ & & & & & & 2x_4 & = & 6 \end{array}$$

2.11.2 Backward substitution

This suggests a solution strategy. First, solve for x_4 using the fourth equation. Plug the resulting value for x_4 into the third equation, and solve for x_3. Plug the values for x_3 and x_4 into the second equation and solve for x_2. Plug the values for x_2, x_3, and x_4 into the first equation and solve for x_1. In each iteration, only one variable needs to be solved for.

Thus the above system is solved as follows:

$$\begin{array}{rrcrcrcr} & 2x_4 & = & 6 & & & & \\ \text{so} & x_4 & & & = & 6/2 & = & 3 \\ & & & & & & & \\ & 1x_3 & = & -4 - 5x_4 & = & -4 - 5(3) & = & -19 \\ \text{so} & x_3 & & & = & -19/1 & = & -19 \\ & & & & & & & \\ & 3x_2 & = & 3 - 3x_3 - 2x_4 & = & 3 - 2(3) - 3(-19) & = & 54 \\ \text{so} & x_2 & & & = & 54/3 & = & 18 \\ & & & & & & & \\ & 1x_1 & = & -8 - 0.5x_2 + 2x_3 - 4x_4 & = & -8 - 4(3) + 2(-19) - 0.5(18) & = & -67 \\ \text{so} & x_1 & & & = & -67/1 & = & -67 \end{array}$$

The algorithm I have illustrated is called *backward substitution* ("backward" because it starts with the last equation and works its way towards the first).

Quiz 2.11.2: Using the above technique, solve the following system by hand:

$$\begin{array}{rcrcrcl} 2x_1 & + & 3x_2 & - & 4x_3 & = & 10 \\ & & 1x_2 & + & 2x_3 & = & 3 \\ & & & & 5x_3 & = & 15 \end{array}$$

Answer

$$x_3 = 15/5 = 3$$

$$x_2 = 3 - 2x_3 = -3$$

$$x_1 = (10 + 4x_3 - 3x_2)/2 = (10 + 12 + 9)/2 = 31/2$$

Exercise 2.11.3: Solve the following system:

$$\begin{array}{rcrcrcl} 1x_1 & - & 3x_2 & - & 2x_3 & = & 7 \\ & & 2x_2 & + & 4x_3 & = & 4 \\ & & & & -10x_3 & = & 12 \end{array}$$

2.11.3 First implementation of backward substitution

There is a convenient way to express this algorithm in terms of vectors and dot-products. The procedure initializes the solution vector `x` to the all-zeroes vector. The procedure will populate `x` entry by entry, starting at the last entry. By the beginning of the entry in which x_i will be populated, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will have already been populated and the other entries are zero, so the procedure can use a dot-product to calculate the part of the expression that involves variables whose values are already known:

$$\text{entry } a_{ii} \cdot \text{ value of } x_i = \beta_i - (\text{expression involving known variables})$$

so

$$\text{value of } x_i = \frac{\beta_i - (\text{expression involving known variables})}{a_{ii}}$$

Using this idea, let's write a procedure `triangular_solve_n(rowlist, b)` with the following spec:

- *input:* for some integer n, a triangular system consisting of a list `rowlist` of n-vectors, and a length-n list $\boldsymbol{b}$ of numbers
- *output:* a vector $\hat{\boldsymbol{x}}$ such that, for $i = 0, 1, \ldots, n-1$, the dot-product of `rowlist`$[i]$ with $\hat{\boldsymbol{x}}$ equals $\boldsymbol{b}[i]$

The `n` in the name indicates that this procedure requires each of the vectors in `rowlist` to have domain $\{0, 1, 2, \ldots, n-1\}$. (We will later write a procedure without this requirement.)

Here is the code:

```
def triangular_solve_n(rowlist, b):
    D = rowlist[0].D
    n = len(D)
    assert D == set(range(n))
    x = zero_vec(D)
    for i in reversed(range(n)):
        x[i] = (b[i] - rowlist[i] * x)/rowlist[i][i]
    return x
```

Exercise 2.11.4: Enter `triangular_solve_n` into Python and try it out on the example system above.

2.11.4 When does the algorithm work?

The *backward substitution* algorithm does not work on all upper triangular systems of equations. If `rowlist[i][i]` is zero for some `i`, the algorithm will fail. We must therefore require when using this algorithm that these entries are not zero. Thus the spec given above is incomplete.

If these entries are nonzero so the algorithm *does* succeed, it will have found the *only* solution to the system of linear equations. The proof is by induction; it is based on the observation that the value assigned to a variable in each iteration is the *only* possible value for that variable that is consistent with the values assigned to variables in previous iterations.

Proposition 2.11.5: For a triangular system specified by a length-n list `rowlist` of n-vectors and an n-vector $\boldsymbol{b}$, if `rowlist`$[i][i] \neq 0$ for $i = 0, 1, \ldots, n-1$ then the solution found by `triangular_solve_n(rowlist, b)` is the *only* solution to the system.

On the other hand,

Proposition 2.11.6: For a length-n list `rowlist` of n-vector, if `rowlist`$[i][i] = 0$ for some integer i then there is a vector $\boldsymbol{b}$ for which the triangular system has no solution.

Proof

Let k be the largest integer less than n such that `rowlist`$[k][k] = 0$. Define $\boldsymbol{b}$ to be a vector whose entries are all zero except for entry k which is nonzero. The algorithm iterates $i = n-1, n-2, \ldots, k+1$. In each of these iterations, the value of `x` before the iteration is the zero vector, and `b[i]` is zero, so `x[i]` is assigned zero. In each of these iterations, the value assigned is is the only possible value consistent with the values assigned to variables in previous iterations.

Finally, the algorithm gets to $i = k$. The equation considered at this point is

`rowlist[k][k]*x[k]+rowlist[k][k+1]*x[k+1]+ ⋯ +rowlist[k][n-1]*x[n-1]` = *nonzero*

but the variables `x[k+1]`, `x[k+2]`, `x[n-1]` have all been forced to be zero, and `rowlist[k][k]` is zero, so the left-hand side of the equation is zero, so the equation cannot be satisfied. □

2.11.5 Backward substitution with arbitrary-domain vectors

Next we write a procedure `triangular_solve(rowlist, label_list, b)` to solve a triangular system in which the domain of the vectors in `rowlist` need not be $\{0, 1, 2, \ldots, n-1\}$. What does it mean for a system to be triangular? The argument `label_list` is a list that specifies an ordering of the domain. For the system to be triangular,

- the first vector in `rowlist` need not have any zeroes,
- the second vector has a zero in the entry labeled by the first element of `label_list`,

- the third vector has zeroes in the entries labeled by the first two elements of `label_list`,

and so on.

The spec of the procedure is:

- *input:* for some positive integer n, a list *rowlist* of n Vecs all having the same n-element domain D, a list *label_list* consisting of the elements of D, and a list b consisting of n numbers such that, for $i = 0, 1, \ldots, n-1$,
  - *rowlist*[i][label_list[j]] is zero for $j = 0, 1, 2, \ldots, i-1$ and is nonzero for $j = i$
- *output:* the Vec $\boldsymbol{x}$ such that, for $i = 0, 1, \ldots, n-1$, the dot-product of rowlist[i] and $\boldsymbol{x}$ equals $b[i]$.

The procedure involves making small changes to the procedure given in Section 2.11.3.

Here I illustrate how the procedure is used.

```
>>> label_list = ['a','b','c','d']
>>> D = set(label_list)
>>> rowlist=[Vec(D,{'a':4, 'b':-2,'c':0.5,'d':1}), Vec(D,{'b':2,'c':3,'d':3}),
              Vec(D,{'c':5, 'd':1}), Vec(D,{'d':2.})]
>>> b = [6, -4, 3, -8]
>>> triangular_solve(rowlist, label_list, b)
Vec({'d', 'b', 'c', 'a'},{'d': -4.0, 'b': 1.9, 'c': 1.4, 'a': 3.275})
```

Here is the code for `triangular_solve`. Note that it uses the procedure `zero_vec(D)`.

```
def triangular_solve(rowlist, label_list, b):
    D = rowlist[0].D
    x = zero_vec(D)
    for j in reversed(range(len(D))):
        c = label_list[j]
        row = rowlist[j]
        x[c] = (b[j] - x*row)/row[c]
    return x
```

The procedures `triangular_solve(rowlist, label_list, b)` and `triangular_solve_n(rowlist, b)` are provided in the module `triangular`.

2.12 *Lab: Comparing voting records using dot-product*

In this lab, we will represent a US senator's voting record as a vector over $\mathbb{R}$, and will use dot-products to compare voting records. For this lab, we will just use a list to represent a vector.

2.12.1 *Motivation*

These are troubled times. You might not have noticed from atop the ivory tower, but take our word for it that the current sociopolitical landscape is in a state of abject turmoil. Now

is the time for a hero. Now is the time for someone to take up the mantle of protector, of the people's shepherd. Now is the time for linear algebra.

In this lab, we will use vectors to evaluate objectively the political mindset of the senators who represent us. Each senator's voting record can be represented as a vector, where each element of that vector represents how that senator voted on a given piece of legislation. By looking at the difference between the "voting vectors" of two senators, we can dispel the fog of politics and see just where our representatives stand.

Or, rather, stood. Our data are a bit dated. On the bright side, you get to see how Obama did as a senator. In case you want to try out your code on data from more recent years, we will post more data files on `resources.codingthematrix.com`.

2.12.2 *Reading in the file*

As in the last lab, the information you need to work with is stored in a whitespace-delimited text file. The senatorial voting records for the 109th Congress can be found in `voting_record_dump109.txt`.

Each line of the file represents the voting record of a different senator. In case you've forgotten how to read in the file, you can do it like this:

```
>>> f = open('voting_record_dump109.txt')
>>> mylist = list(f)
```

You can use the `split(·)` procedure to split each line of the file into a list; the first element of the list will be the senator's name, the second will be his/her party affiliation (R or D), the third will be his/her home state, and the remaining elements of the list will be that senator's voting record on a collection of bills. A "1" represents a 'yea' vote, a "-1" a 'nay', and a "0" an abstention.

Task 2.12.1: Write a procedure `create_voting_dict(strlist)` that, given a list of strings (voting records from the source file), returns a dictionary that maps the last name of a senator to a list of numbers representing that senator's voting record. You will need to use the built-in procedure `int(·)` to convert a string representation of an integer (e.g. '1') to the actual integer (e.g. 1).

2.12.3 *Two ways to use dot-product to compare vectors*

Suppose $\boldsymbol{u}$ and $\boldsymbol{v}$ are two vectors. Let's take the simple case (relevant to the current lab) in which the entries are all 1, 0, or -1. Recall that the dot-product of $\boldsymbol{u}$ and $\boldsymbol{v}$ is defined as

$$\boldsymbol{u} \cdot \boldsymbol{v} = \sum_k \boldsymbol{u}[k]\boldsymbol{v}[k]$$

Consider the k^{th} entry. If both $\boldsymbol{u}[k]$ and $\boldsymbol{v}[k]$ are 1, the corresponding term in the sum is 1. If both $\boldsymbol{u}[k]$ and $\boldsymbol{v}[k]$ are -1, the corresponding term in the sum is also 1. Thus a term in the

sum that is 1 indicates agreement. If, on the other hand, $\boldsymbol{u}[k]$ and $\boldsymbol{v}[k]$ have different signs, the corresponding term is -1. Thus a term in the sum that is -1 indicates disagreement. (If one or both of $\boldsymbol{u}[k]$ and $\boldsymbol{v}[k]$ are zero then the term is zero, reflecting the fact that those entries provide no evidence of either agreement or disagreement.) The dot-product of $\boldsymbol{u}$ and $\boldsymbol{v}$ therefore is a measure of how much $\boldsymbol{u}$ and $\boldsymbol{v}$ are in agreement.

2.12.4 *Policy comparison*

We would like to determine just how like-minded two given senators are. We will use the dot-product of vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ to judge how often two senators are in agreement.

Task 2.12.2: Write a procedure `policy_compare(sen_a, sen_b, voting_dict)` that, given two names of senators and a dictionary mapping senator names to lists representing voting records, returns the dot-product representing the degree of similarity between two senators' voting policies.

Task 2.12.3: Write a procedure `most_similar(sen, voting_dict)` that, given the name of a senator and a dictionary mapping senator names to lists representing voting records, returns the name of the senator whose political mindset is most like the input senator (excluding, of course, the input senator him/herself).

Task 2.12.4: Write a very similar procedure `least_similar(sen, voting_dict)` that returns the name of the senator whose voting record agrees the least with the senator whose name is `sen`.

Task 2.12.5: Use these procedures to figure out which senator is most like Rhode Island legend Lincoln Chafee. Then use these procedures to see who disagrees most with Pennsylvania's Rick Santorum. Give their names.

Task 2.12.6: How similar are the voting records of the two senators from your favorite state?

2.12.5 *Not your average Democrat*

Task 2.12.7: Write a procedure `find_average_similarity(sen, sen_set, voting_dict)` that, given the name `sen` of a senator, compares that senator's voting record to the voting records of all senators whose names are in `sen_set`, computing a dot-product for each, and then returns the average dot-product.

Use your procedure to compute which senator has the greatest average similarity with the set of Democrats (you can extract this set from the input file).

In the last task, you had to compare each senator's record to the voting record of each Democrat senator. If you were doing the same computation with, say, the movie preferences of all Netflix subscribers, it would take far too long to be practical.

Next we see that there is a computational shortcut, based on an algebraic property of the dot-product: the *distributive* property:

$$(\boldsymbol{v}_1 + \boldsymbol{v}_2) \cdot \boldsymbol{x} = \boldsymbol{v}_1 \cdot \boldsymbol{x} + \boldsymbol{v}_2 \cdot \boldsymbol{x}$$

Task 2.12.8: Write a procedure `find_average_record(sen_set, voting_dict)` that, given a set of names of senators, finds the average voting record. That is, perform vector addition on the lists representing their voting records, and then divide the sum by the number of vectors. The result should be a vector.

Use this procedure to compute the average voting record for the set of Democrats, and assign the result to the variable `average_Democrat_record`. Next find which senator's voting record is most similar to the average Democrat voting record. Did you get the same result as in Task 2.12.7? Can you explain?

2.12.6 *Bitter Rivals*

Task 2.12.9: Write a procedure `bitter_rivals(voting_dict)` to find which two senators disagree the most.

This task again requires comparing each pair of voting records. Can this be done faster than the obvious way? There is a slightly more efficient algorithm, using *fast matrix multiplication*. We will study matrix multiplication later, although we won't cover the theoretically fast algorithms.

2.12.7 *Open-ended study*

You have just coded a set of simple yet powerful tools for sifting the truth from the sordid flour of contemporary politics. Use your new abilities to answer at least one of the following questions (or make up one of your own):

- Who/which is the most Republican/Democratic senator/state?
- Is John McCain really a maverick?

- Is Barack Obama really an extremist?
- Which two senators are the most bitter rivals?
- Which senator has the most political opponents? (Assume two senators are opponents if their dot-product is very negative, i.e. is less than some negative threshold.)

2.13 Review Questions

- What is vector addition?
- What is the geometric interpretation of vector addition?
- What is scalar-vector multiplication?
- What is the distributive property that involves scalar-vector multiplication but not vector addition?
- What is the distributive property that involves both scalar-vector multiplication and vector addition?
- How is scalar-vector multiplication used to represent the line through the origin and a given point?
- How are scalar-vector multiplication and vector addition used to represent the line through a pair of given points?
- What is dot-product?
- What is the *homogeneity* property that relates dot-product to scalar-vector multiplication?
- What is the distributive property property that relates dot-product to vector addition?
- What is a linear equation (expressed using dot-product)?
- What is a linear system?
- What is an upper-triangular linear system?
- How can one solve an upper-triangular linear system?

2.14 Problems

Vector addition practice

Problem 2.14.1: For vectors $\boldsymbol{v} = [-1, 3]$ and $\boldsymbol{u} = [0, 4]$, find the vectors $\boldsymbol{v} + \boldsymbol{u}$, $\boldsymbol{v} - \boldsymbol{u}$, and $3\boldsymbol{v} - 2\boldsymbol{u}$. Draw these vectors as arrows on the same graph..

Problem 2.14.2: Given the vectors $\boldsymbol{v} = [2, -1, 5]$ and $\boldsymbol{u} = [-1, 1, 1]$, find the vectors $\boldsymbol{v} + \boldsymbol{u}$, $\boldsymbol{v} - \boldsymbol{u}$, $2\boldsymbol{v} - \boldsymbol{u}$, and $\boldsymbol{v} + 2\boldsymbol{u}$.

Problem 2.14.3: For the vectors $\boldsymbol{v} = [0, one, one]$ and $\boldsymbol{u} = [one, one, one]$ over $GF(2)$, find $\boldsymbol{v} + \boldsymbol{u}$ and $\boldsymbol{v} + \boldsymbol{u} + \boldsymbol{u}$.

Expressing one $GF(2)$ vector as a sum of others

Problem 2.14.4: Here are six 7-vectors over $GF(2)$:

$\mathbf{a} =$	1100000	$\mathbf{d} =$	0001100
$\mathbf{b} =$	0110000	$\mathbf{e} =$	0000110
$\mathbf{c} =$	0011000	$\mathbf{f} =$	0000011

For each of the following vectors $\boldsymbol{u}$, find a subset of the above vectors whose sum is $\boldsymbol{u}$, or report that no such subset exists.

1. $\boldsymbol{u} = 0010010$
2. $\boldsymbol{u} = 0100010$

Problem 2.14.5: Here are six 7-vectors over $GF(2)$:

$\mathbf{a} =$	1110000	$\mathbf{d} =$	0001110
$\mathbf{b} =$	0111000	$\mathbf{e} =$	0000111
$\mathbf{c} =$	0011100	$\mathbf{f} =$	0000011

For each of the following vectors $\boldsymbol{u}$, find a subset of the above vectors whose sum is $\boldsymbol{u}$, or report that no such subset exists.

1. $\boldsymbol{u} = 0010010$
2. $\boldsymbol{u} = 0100010$

Finding a solution to linear equations over $GF(2)$

Problem 2.14.6: Find a vector $\boldsymbol{x} = [x_1, x_2, x_3, x_4]$ over $GF(2)$ satisfying the following linear equations:

$$\begin{aligned} 1100 \cdot \boldsymbol{x} &= 1 \\ 1010 \cdot \boldsymbol{x} &= 1 \\ 1111 \cdot \boldsymbol{x} &= 1 \end{aligned}$$

Show that $\boldsymbol{x} + 1111$ also satisfies the equations.

Formulating equations using dot-product

Problem 2.14.7: Consider the equations

$$\begin{array}{rcrcrcrcl} 2x_0 & + & 3x_1 & - & 4x_2 & + & x_3 & = & 10 \\ x_0 & - & 5x_1 & + & 2x_2 & + & 0x_3 & = & 35 \\ 4x_0 & + & x_1 & - & x_2 & - & x_3 & = & 8 \end{array}$$

Your job is not to solve these equations but to formulate them using dot-product. In particular, come up with three vectors `v1`, `v2`, and `v3` represented as lists so that the above equations are equivalent to

$$\begin{aligned} \texttt{v1} \cdot \boldsymbol{x} &= 10 \\ \texttt{v2} \cdot \boldsymbol{x} &= 35 \\ \texttt{v3} \cdot \boldsymbol{x} &= 8 \end{aligned}$$

where x is a 4-vector over $\mathbb{R}$.

Plotting lines and line segments

Problem 2.14.8: Use the `plot` module to plot

(a) a substantial portion of the line through [-1.5,2] and [3,0], and

(b) the line segment between [2,1] and [-2,2].

For each, provide the Python statements you used and the plot obtained.

Practice with dot-product

Problem 2.14.9: For each of the following pairs of vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ over $\mathbb{R}$, evaluate the expression $\boldsymbol{u} \cdot \boldsymbol{v}$:

(a) $\boldsymbol{u} = [1, 0], \boldsymbol{v} = [5, 4321]$

(b) $\boldsymbol{u} = [0, 1], \boldsymbol{v} = [12345, 6]$

(c) $\boldsymbol{u} = [-1, 3], \boldsymbol{v} = [5, 7]$

(d) $\boldsymbol{u} = [-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}], \boldsymbol{v} = [\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}]$

Writing procedures for the `Vec` class