[2] The Vector

The Vector: William Rowan Hamilton

By age 5, Latin, Greek, and Hebrew
By age 10, twelve languages including Persian, Arabic, Hindustani and Sanskrit.

William Rowan Hamilton, the inventor of the theory of quaternions...

And here there dawned on me the notion that we must admit, in some sense, a fourth dimension of space for the purpose of calculating with triples ... An electric circuit seemed to close, and a spark flashed forth.

The Vector: William Rowan Hamilton

By age 5, Latin, Greek, and Hebrew
By age 10, twelve languages including Persian, Arabic, Hindustani and Sanskrit.

William Rowan Hamilton, the inventor of the theory of quaternions...

$$i^2 = j^2 = k^2 = ijk = -1$$

And here there dawned on me the notion that we must admit, in some sense, a fourth dimension of space for the purpose of calculating with triples ... An electric circuit seemed to close, and a spark flashed forth.

The Vector: William Rowan Hamilton

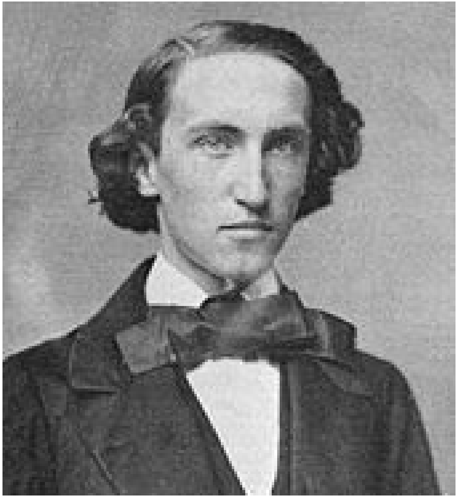
By age 5, Latin, Greek, and Hebrew
By age 10, twelve languages including Persian, Arabic, Hindustani and Sanskrit.

William Rowan Hamilton, the inventor of the theory of quaternions... and the plaque on Brougham Bridge, Dublin, commemorating Hamilton's act of vandalism.

$$i^2 = j^2 = k^2 = ijk = -1$$

And here there dawned on me the notion that we must admit, in some sense, a fourth dimension of space for the purpose of calculating with triples ... An electric circuit seemed to close, and a spark flashed forth.

The Vector: Josiah Willard Gibbs



Started at Yale at 15
Got Ph.D. at Yale at 24
(1st engineering doctorate in US)
Tutored at Yale
Spent three years in Europe
Returned to be professor at Yale

Developed *vector analysis* as an alternative to quaternions.

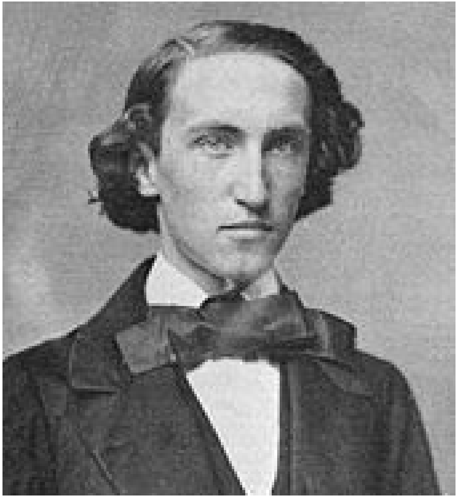
His unpublished notes were passed around for twenty years.

"Professor Willard Gibbs must be ranked as one of the retarders of ... progress in virtue of his pamphlet on *Vector Analysis*; a sort of hermaphrodite monster." (Peter Guthrie Tait, a partisan of quaternions)

The Vector: Josiah Willard Gibbs

Started at Yale at 15
Got Ph.D. at Yale at 24
(1st engineering doctorate in US)
Tutored at Yale
Spent three years in Europe
Returned to be professor at Yale

Developed *vector analysis* as an alternative to quaternions.



His unpublished notes were passed around for twenty years.

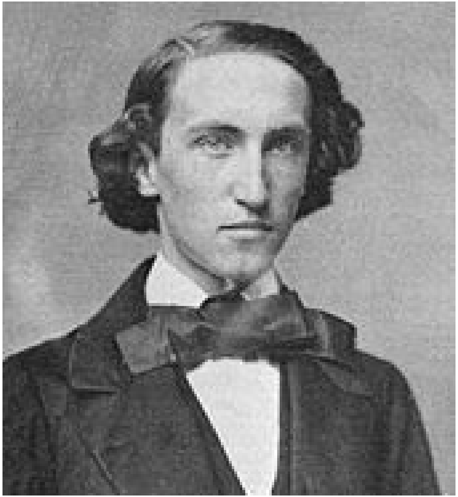
"Professor Willard Gibbs must be ranked as one of the retarders of ... progress in virtue of his pamphlet on *Vector Analysis*; a sort of hermaphrodite monster."
(Peter Guthrie Tait, a partisan of quaternions)

The Vector: Josiah Willard Gibbs

Started at Yale at 15
Got Ph.D. at Yale at 24
(1st engineering doctorate in US)
Tutored at Yale
Spent three years in Europe
Returned to be professor at Yale

Developed *vector analysis* as an alternative to quaternions.

His unpublished notes were passed around for twenty years.



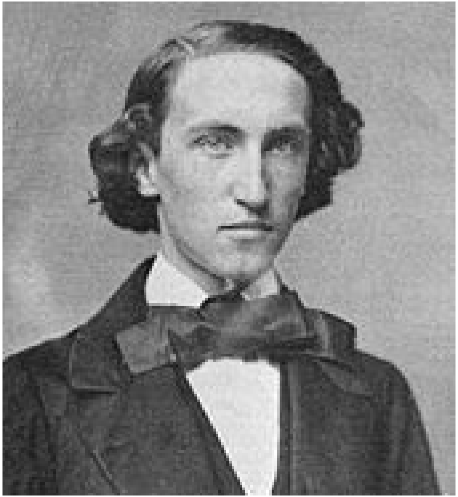
"Professor Willard Gibbs must be ranked as one of the retarders of ... progress in virtue of his pamphlet on *Vector Analysis*; a sort of hermaphrodite monster."
(Peter Guthrie Tait, a partisan of quaternions)

The Vector: Josiah Willard Gibbs

Started at Yale at 15
Got Ph.D. at Yale at 24
(1st engineering doctorate in US)
Tutored at Yale
Spent three years in Europe
Returned to be professor at Yale

Developed *vector analysis* as an alternative to quaternions.

His unpublished notes were passed around for twenty years.



"Professor Willard Gibbs must be ranked as one of the retarders of ... progress in virtue of his pamphlet on *Vector Analysis*; a sort of hermaphrodite monster."
(Peter Guthrie Tait, a partisan of quaternions)

What is a vector?

- This is a 4-vector over $\mathbb{R}$:

$$[3.14159, 2.718281828, -1.0, 2.0]$$

- We will often use Python's lists to represent vectors.
- Set of all 4-vectors over $\mathbb{R}$ is written $\mathbb{R}^4$.
- This notation might remind you of the notation $\mathbb{R}^D$: the set of functions from D to $\mathbb{R}$.

What is a vector?

- This is a 4-vector over $\mathbb{R}$:

$$[3.14159, 2.718281828, -1.0, 2.0]$$

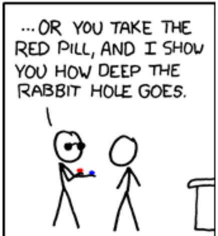
- We will often use Python's lists to represent vectors.
- Set of all 4-vectors over $\mathbb{R}$ is written $\mathbb{R}^4$.
- This notation might remind you of the notation $\mathbb{R}^D$: the set of functions from D to $\mathbb{R}$.

What is a vector?

- This is a 4-vector over $\mathbb{R}$:

$$[3.14159, 2.718281828, -1.0, 2.0]$$

- We will often use Python's lists to represent vectors.
- Set of all 4-vectors over $\mathbb{R}$ is written $\mathbb{R}^4$.
- This notation might remind you of the notation $\mathbb{R}^D$: the set of functions from D to $\mathbb{R}$.

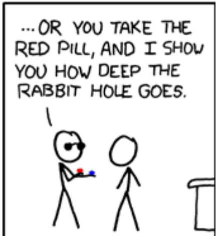


What is a vector?

- This is a 4-vector over $\mathbb{R}$:

$$[3.14159, 2.718281828, -1.0, 2.0]$$

- We will often use Python's lists to represent vectors.
- Set of all 4-vectors over $\mathbb{R}$ is written $\mathbb{R}^4$.
- This notation might remind you of the notation $\mathbb{R}^D$: the set of functions from D to $\mathbb{R}$.



Vectors are functions

Think of our 4-vector $[3.14159, 2.718281828, -1.0, 2.0]$ as the function

$$0 \mapsto 3.14159, \quad 1 \mapsto 2.718281828, \quad 2 \mapsto -1.0, \quad 3 \mapsto 2.0$$

$\mathbb{F}^d$ is notation for set of functions from $\{0, 1, 2, \ldots, d-1\}$ to $\mathbb{F}$.

Example: $GF(2)^5$ is set of 5-element bit sequences, e.g. [0,0,0,0,0], [0,0,0,0,1], ...

Let WORDS = set of all English words
In information retrieval, a document is represented ("bag of words" model) by a function $f : WORDS \longrightarrow \mathbb{R}$ specifying, for each word, how many times it appears in the document.
We would refer to such a function as a WORDS-vector over $\mathbb{R}$

Definition: For a field $\mathbb{F}$ and a set D, a *D-vector over* $\mathbb{F}$ is a function from D to $\mathbb{F}$.
The set of such functions is written $\mathbb{F}^D$

For example, $\mathbb{R}^{WORDS}$

Vectors are functions

Think of our 4-vector $[3.14159, 2.718281828, -1.0, 2.0]$ as the function

$$0 \mapsto 3.14159, \quad 1 \mapsto 2.718281828, \quad 2 \mapsto -1.0, \quad 3 \mapsto 2.0$$

$\mathbb{F}^d$ is notation for set of functions from $\{0, 1, 2, \ldots, d-1\}$ to $\mathbb{F}$.

Example: $GF(2)^5$ is set of 5-element bit sequences, e.g. [0,0,0,0,0], [0,0,0,0,1], ...

Let WORDS = set of all English words
In information retrieval, a document is represented ("bag of words" model) by a function $f : WORDS \longrightarrow \mathbb{R}$ specifying, for each word, how many times it appears in the document.
We would refer to such a function as a WORDS-vector over $\mathbb{R}$

Definition: For a field $\mathbb{F}$ and a set D, a *D-vector over $\mathbb{F}$* is a function from D to $\mathbb{F}$.
The set of such functions is written $\mathbb{F}^D$

For example, $\mathbb{R}^{WORDS}$

Vectors are functions

Think of our 4-vector $[3.14159, 2.718281828, -1.0, 2.0]$ as the function

$$0 \mapsto 3.14159, \quad 1 \mapsto 2.718281828, \quad 2 \mapsto -1.0, \quad 3 \mapsto 2.0$$

$\mathbb{F}^d$ is notation for set of functions from $\{0, 1, 2, \ldots, d-1\}$ to $\mathbb{F}$.

Example: $GF(2)^5$ is set of 5-element bit sequences, e.g. [0,0,0,0,0], [0,0,0,0,1], ...

Let WORDS = set of all English words

In information retrieval, a document is represented ("bag of words" model) by a function $f : WORDS \longrightarrow \mathbb{R}$ specifying, for each word, how many times it appears in the document.

We would refer to such a function as a WORDS-vector over $\mathbb{R}$

Definition: For a field $\mathbb{F}$ and a set D, a *D-vector over* $\mathbb{F}$ is a function from D to $\mathbb{F}$.
The set of such functions is written $\mathbb{F}^D$

For example, $\mathbb{R}^{WORDS}$

Vectors are functions

Think of our 4-vector $[3.14159, 2.718281828, -1.0, 2.0]$ as the function

$$0 \mapsto 3.14159, \quad 1 \mapsto 2.718281828, \quad 2 \mapsto -1.0, \quad 3 \mapsto 2.0$$

$\mathbb{F}^d$ is notation for set of functions from $\{0, 1, 2, \ldots, d-1\}$ to $\mathbb{F}$.

Example: $GF(2)^5$ is set of 5-element bit sequences, e.g. [0,0,0,0,0], [0,0,0,0,1], ...

Let WORDS $=$ set of all English words
In information retrieval, a document is represented ("bag of words" model) by a function $f : WORDS \longrightarrow \mathbb{R}$ specifying, for each word, how many times it appears in the document.
We would refer to such a function as a WORDS-vector over $\mathbb{R}$

Definition: For a field $\mathbb{F}$ and a set D, a *D-vector over $\mathbb{F}$* is a function from D to $\mathbb{F}$.
The set of such functions is written $\mathbb{F}^D$

For example, $\mathbb{R}^{WORDS}$

Vectors are functions

Think of our 4-vector $[3.14159, 2.718281828, -1.0, 2.0]$ as the function

$$0 \mapsto 3.14159, \quad 1 \mapsto 2.718281828, \quad 2 \mapsto -1.0, \quad 3 \mapsto 2.0$$

$\mathbb{F}^d$ is notation for set of functions from $\{0, 1, 2, \ldots, d-1\}$ to $\mathbb{F}$.

Example: $GF(2)^5$ is set of 5-element bit sequences, e.g. [0,0,0,0,0], [0,0,0,0,1], ...

Let WORDS $=$ set of all English words
In information retrieval, a document is represented ("bag of words" model) by a function $f : WORDS \longrightarrow \mathbb{R}$ specifying, for each word, how many times it appears in the document.
We would refer to such a function as a WORDS-vector over $\mathbb{R}$

Definition: For a field $\mathbb{F}$ and a set D, a *D-vector over $\mathbb{F}$* is a function from D to $\mathbb{F}$.
The set of such functions is written $\mathbb{F}^D$

For example, $\mathbb{R}^{WORDS}$

Vectors are functions

Think of our 4-vector $[3.14159, 2.718281828, -1.0, 2.0]$ as the function

$$0 \mapsto 3.14159, \quad 1 \mapsto 2.718281828, \quad 2 \mapsto -1.0, \quad 3 \mapsto 2.0$$

$\mathbb{F}^d$ is notation for set of functions from $\{0, 1, 2, \ldots, d-1\}$ to $\mathbb{F}$.

Example: $GF(2)^5$ is set of 5-element bit sequences, e.g. [0,0,0,0,0], [0,0,0,0,1], ...

Let WORDS = set of all English words
In information retrieval, a document is represented ("bag of words" model) by a function $f : WORDS \longrightarrow \mathbb{R}$ specifying, for each word, how many times it appears in the document.
We would refer to such a function as a WORDS-vector over $\mathbb{R}$

Definition: For a field $\mathbb{F}$ and a set D, a *D-vector over* $\mathbb{F}$ is a function from D to $\mathbb{F}$.
The set of such functions is written $\mathbb{F}^D$

For example, $\mathbb{R}^{WORDS}$

Vectors are functions

Think of our 4-vector $[3.14159, 2.718281828, -1.0, 2.0]$ as the function

$$0 \mapsto 3.14159, \quad 1 \mapsto 2.718281828, \quad 2 \mapsto -1.0, \quad 3 \mapsto 2.0$$

$\mathbb{F}^d$ is notation for set of functions from $\{0, 1, 2, \ldots, d-1\}$ to $\mathbb{F}$.

Example: $GF(2)^5$ is set of 5-element bit sequences, e.g. [0,0,0,0,0], [0,0,0,0,1], ...

Let WORDS = set of all English words
In information retrieval, a document is represented ("bag of words" model) by a function $f : WORDS \longrightarrow \mathbb{R}$ specifying, for each word, how many times it appears in the document.
We would refer to such a function as a WORDS-vector over $\mathbb{R}$

Definition: For a field $\mathbb{F}$ and a set D, a *D-vector over* $\mathbb{F}$ is a function from D to $\mathbb{F}$.
The set of such functions is written $\mathbb{F}^D$

For example, $\mathbb{R}^{WORDS}$

Representation of vectors using Python dictionaries

We often use Python's dictionaries to represent such functions, e.g.

`{0:3.14159, 1:2.718281828, 2:-1.0, 3:2.0}`

What about representing a WORDS-vector over $\mathbb{R}$?

For any single document, most words are *not* represented. They should be mapped to zero.

Our convention for representing vectors by dictionaries: we are allowed to omit key-value pairs when value is zero.

Example: "The rain in Spain falls mainly on the plain" would be represented by the dictionary

```
{'on': 1, 'Spain': 1, 'in': 1, 'plain': 1, 'the': 2,
 'mainly': 1, 'rain': 1, 'falls': 1}
```

Representation of vectors using Python dictionaries

We often use Python's dictionaries to represent such functions, e.g.

```
{0:3.14159, 1:2.718281828, 2:-1.0, 3:2.0}
```

What about representing a WORDS-vector over $\mathbb{R}$?

For any single document, most words are *not* represented. They should be mapped to zero.

Our convention for representing vectors by dictionaries: we are allowed to omit key-value pairs when value is zero.

Example: "The rain in Spain falls mainly on the plain" would be represented by the dictionary

```
{'on': 1, 'Spain': 1, 'in': 1, 'plain': 1, 'the': 2,
 'mainly': 1, 'rain': 1, 'falls': 1}
```

Representation of vectors using Python dictionaries

We often use Python's dictionaries to represent such functions, e.g.

```
{0:3.14159, 1:2.718281828, 2:-1.0, 3:2.0}
```

What about representing a WORDS-vector over $\mathbb{R}$?

For any single document, most words are *not* represented. They should be mapped to zero.

Our convention for representing vectors by dictionaries: we are allowed to omit key-value pairs when value is zero.

Example: "The rain in Spain falls mainly on the plain" would be represented by the dictionary

```
{'on': 1, 'Spain': 1, 'in': 1, 'plain': 1, 'the': 2,
 'mainly': 1, 'rain': 1, 'falls': 1}
```

Representation of vectors using Python dictionaries

We often use Python's dictionaries to represent such functions, e.g.

`{0:3.14159, 1:2.718281828, 2:-1.0, 3:2.0}`

What about representing a WORDS-vector over $\mathbb{R}$?

For any single document, most words are *not* represented. They should be mapped to zero.

Our convention for representing vectors by dictionaries: we are allowed to omit key-value pairs when value is zero.

Example: "The rain in Spain falls mainly on the plain" would be represented by the dictionary

```
{'on': 1, 'Spain': 1, 'in': 1, 'plain': 1, 'the': 2,
 'mainly': 1, 'rain': 1, 'falls': 1}
```

Representation of vectors using Python dictionaries

We often use Python's dictionaries to represent such functions, e.g.

```
{0:3.14159, 1:2.718281828, 2:-1.0, 3:2.0}
```

What about representing a WORDS-vector over $\mathbb{R}$?

For any single document, most words are *not* represented. They should be mapped to zero.

Our convention for representing vectors by dictionaries: we are allowed to omit key-value pairs when value is zero.

Example: "The rain in Spain falls mainly on the plain" would be represented by the dictionary

```
{'on': 1, 'Spain': 1, 'in': 1, 'plain': 1, 'the': 2,
 'mainly': 1, 'rain': 1, 'falls': 1}
```

Sparsity

A vector most of whose values are zero is called a *sparse* vector.
If no more than k of the entries are nonzero, we say the vector is *k-sparse*.

A k-sparse vector can be represented using space proportional to k.

Example: when we represent a corpus of documents by WORD-vectors, the storage required is proportional to the total number of words in all documents.

Most signals acquired via physical sensors (images, sound, ...) are not exactly sparse.

Later we study *lossy compression*: making them sparse while preserving perceptual similarity.

Sparsity

A vector most of whose values are zero is called a *sparse* vector.
If no more than k of the entries are nonzero, we say the vector is k-*sparse*.

A k-sparse vector can be represented using space proportional to k.

Example: when we represent a corpus of documents by WORD-vectors, the storage required is proportional to the total number of words in all documents.

Most signals acquired via physical sensors (images, sound, ...) are not exactly sparse.

Later we study *lossy compression*: making them sparse while preserving perceptual similarity.

Sparsity

A vector most of whose values are zero is called a *sparse* vector.
If no more than k of the entries are nonzero, we say the vector is *k-sparse*.

A k-sparse vector can be represented using space proportional to k.

Example: when we represent a corpus of documents by WORD-vectors, the storage required is proportional to the total number of words in all documents.

Most signals acquired via physical sensors (images, sound, ...) are not exactly sparse.

Later we study *lossy compression*: making them sparse while preserving perceptual similarity.

Sparsity

A vector most of whose values are zero is called a *sparse* vector.
If no more than k of the entries are nonzero, we say the vector is *k-sparse*.

A k-sparse vector can be represented using space proportional to k.

Example: when we represent a corpus of documents by WORD-vectors, the storage required is proportional to the total number of words in all documents.

Most signals acquired via physical sensors (images, sound, ...) are not exactly sparse.

Later we study *lossy compression*: making them sparse while preserving perceptual similarity.

Sparsity

A vector most of whose values are zero is called a *sparse* vector.
If no more than k of the entries are nonzero, we say the vector is *k-sparse*.

A k-sparse vector can be represented using space proportional to k.

Example: when we represent a corpus of documents by WORD-vectors, the storage required is proportional to the total number of words in all documents.

Most signals acquired via physical sensors (images, sound, ...) are not exactly sparse.

Later we study *lossy compression*: making them sparse while preserving perceptual similarity.

Sparsity

A vector most of whose values are zero is called a *sparse* vector.
If no more than k of the entries are nonzero, we say the vector is *k-sparse*.

A k-sparse vector can be represented using space proportional to k.

Example: when we represent a corpus of documents by WORD-vectors, the storage required is proportional to the total number of words in all documents.

Most signals acquired via physical sensors (images, sound, ...) are not exactly sparse.

Later we study *lossy compression*: making them sparse while preserving perceptual similarity.

What can we represent with a vector?

- Document (for information retrieval)
- Binary string (for cryptography/information theory)
- Collection of attributes
  - Senate voting record
  - demographic record of a consumer
  - characteristics of cancer cells
- State of a system
  - Population distribution in the world
  - number of copies of a virus in a computer network
  - state of a pseudorandom generator
  - state of *Lights Out*
- Probability distribution, e.g. {`1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6`}

What can we represent with a vector?

- Document (for information retrieval)
- Binary string (for cryptography/information theory)
- Collection of attributes
  - Senate voting record
  - demographic record of a consumer
  - characteristics of cancer cells
- State of a system
  - Population distribution in the world
  - number of copies of a virus in a computer network
  - state of a pseudorandom generator
  - state of *Lights Out*
- Probability distribution, e.g. {`1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6`}

What can we represent with a vector?

- Document (for information retrieval)
- Binary string (for cryptography/information theory)
- Collection of attributes
  - Senate voting record
  - demographic record of a consumer
  - characteristics of cancer cells
- State of a system
  - Population distribution in the world
  - number of copies of a virus in a computer network
  - state of a pseudorandom generator
  - state of *Lights Out*
- Probability distribution, e.g. {`1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6`}

What can we represent with a vector?

- Document (for information retrieval)
- Binary string (for cryptography/information theory)
- Collection of attributes
  - Senate voting record
  - demographic record of a consumer
  - characteristics of cancer cells
- State of a system
  - Population distribution in the world
  - number of copies of a virus in a computer network
  - state of a pseudorandom generator
  - state of *Lights Out*
- Probability distribution, e.g. {`1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6`}

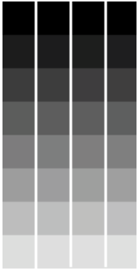
What can we represent with a vector?

- Document (for information retrieval)
- Binary string (for cryptography/information theory)
- Collection of attributes
  - Senate voting record
  - demographic record of a consumer
  - characteristics of cancer cells
- State of a system
  - Population distribution in the world
  - number of copies of a virus in a computer network
  - state of a pseudorandom generator
  - state of *Lights Out*
- Probability distribution, e.g. $\{$`1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6`$\}$

What can we represent with a vector?

- Image

```
{(0,0): 0,    (0,1): 0,    (0,2): 0,    (0,3): 0,
 (1,0): 32,   (1,1): 32,   (1,2): 32,   (1,3): 32,
 (2,0): 64,   (2,1): 64,   (2,2): 64,   (2,3): 64,
 (3,0): 96,   (3,1): 96,   (3,2): 96,   (3,3): 96,
 (4,0): 128,  (4,1): 128,  (4,2): 128,  (4,3): 128,
 (5,0): 160,  (5,1): 160,  (5,2): 160,  (5,3): 160,
 (6,0): 192,  (6,1): 192,  (6,2): 192,  (6,3): 192,
 (7,0): 224,  (7,1): 224,  (7,2): 224,  (7,3): 224 }
```



What can we represent with a vector?

- Points
  - Can interpret the 2-vector $[x, y]$ as a point in the plane.
  - Can interpret 3-vectors as points in space, and so on.

What can we represent with a vector?

- Points
  - Can interpret the 2-vector $[x, y]$ as a point in the plane.
  - Can interpret 3-vectors as points in space, and so on.

What can we represent with a vector?

- Points
  - Can interpret the 2-vector $[x, y]$ as a point in the plane.
  - Can interpret 3-vectors as points in space, and so on.

Vector addition: Translation and vector addition

With complex numbers, translation achieved by adding a complex number, e.g. $f(z) = z + (1 + 2\mathbf{i})$

Let's do the same thing with vectors...

Definition of vector addition:

$$[u_1, u_2, \ldots, u_n] + [v_1, v_2, \ldots, v_n] = [u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n]$$

For 2-vectors represented in Python as 2-element lists, addition procedure is

```
def add2(v,w): return [v[0]+w[0], v[1]+w[1]]
```

Vector addition: Translation and vector addition

With complex numbers, translation achieved by adding a complex number, e.g. $f(z) = z + (1 + 2\mathbf{i})$

Let's do the same thing with vectors...

Definition of vector addition:

$$[u_1, u_2, \ldots, u_n] + [v_1, v_2, \ldots, v_n] = [u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n]$$

For 2-vectors represented in Python as 2-element lists, addition procedure is

```
def add2(v,w): return [v[0]+w[0], v[1]+w[1]]
```

Vector addition: Translation and vector addition

With complex numbers, translation achieved by adding a complex number, e.g. $f(z) = z + (1 + 2\mathbf{i})$

Let's do the same thing with vectors...

Definition of vector addition:

$$[u_1, u_2, \ldots, u_n] + [v_1, v_2, \ldots, v_n] = [u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n]$$

For 2-vectors represented in Python as 2-element lists, addition procedure is

```
def add2(v,w): return [v[0]+w[0], v[1]+w[1]]
```

Vector addition: Translation and vector addition

Quiz: Suppose we represent n-vectors by n-element lists. Write a procedure `addn(v, w)` to compute the sum of two vectors so represented.

Vector addition: Translation and vector addition

Quiz: Suppose we represent *n*-vectors by *n*-element lists. Write a procedure `addn(v, w)` to compute the sum of two vectors so represented.

Answer:

```
def addn(v, w): return [v[i]+w[i] for i in range(len(v))]
```

Vector addition: The zero vector

The D-vector whose entries are all zero is the *zero vector*, written $\mathbf{0}_D$ or just $\mathbf{0}$

$$\mathbf{v} + \mathbf{0} = \mathbf{v}$$

Vector addition: The zero vector

The D-vector whose entries are all zero is the *zero vector*, written $\mathbf{0}_D$ or just $\mathbf{0}$

$$\mathbf{v} + \mathbf{0} = \mathbf{v}$$

Vector addition: Vector addition is associative and commutative

- *Associativity*

$$(\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{x} + (\mathbf{y} + \mathbf{z})$$

- *Commutativity*

$$\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$$

Vector addition: Vector addition is associative and commutative

- *Associativity*

$$(\mathbf{x}+\mathbf{y})+\mathbf{z}=\mathbf{x}+(\mathbf{y}+\mathbf{z})$$

- *Commutativity*

$$\mathbf{x}+\mathbf{y}=\mathbf{y}+\mathbf{x}$$

Vector addition: Vectors as arrows

Like complex numbers in the plane, n-vectors over $\mathbb{R}$ can be visualized as *arrows* in $\mathbb{R}^n$.

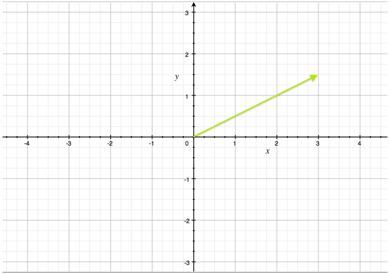
The 2-vector $[3, 1.5]$ can be represented by an arrow with its tail at the origin and its head at $(3, 1.5)$.

or, equivalently, by an arrow whose tail is at $(-2, -1)$ and whose head is at $(1, 0.5)$.

Vector addition: Vectors as arrows

Like complex numbers in the plane, n-vectors over $\mathbb{R}$ can be visualized as *arrows* in $\mathbb{R}^n$.

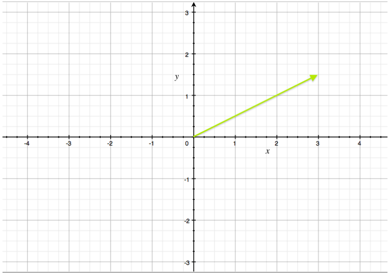
The 2-vector $[3, 1.5]$ can be represented by an arrow with its tail at the origin and its head at $(3, 1.5)$.



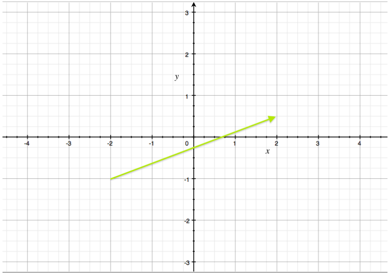
or, equivalently, by an arrow whose tail is at $(-2, -1)$ and whose head is at $(1, 0.5)$.

Vector addition: Vectors as arrows

Like complex numbers in the plane, n-vectors over $\mathbb{R}$ can be visualized as *arrows* in $\mathbb{R}^n$.



The 2-vector $[3, 1.5]$ can be represented by an arrow with its tail at the origin and its head at $(3, 1.5)$.



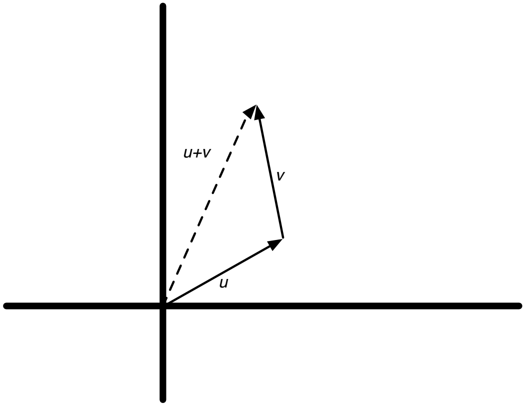
or, equivalently, by an arrow whose tail is at $(-2, -1)$ and whose head is at $(1, 0.5)$.

Vector addition: Vectors as arrows

Like complex numbers, addition of vectors over $\mathbb{R}$ can be visualized using arrows.

To add **u** and **v**:

- place tail of **v**'s arrow on head of **u**'s arrow;
- draw a new arrow from tail of **u** to head of **v**.

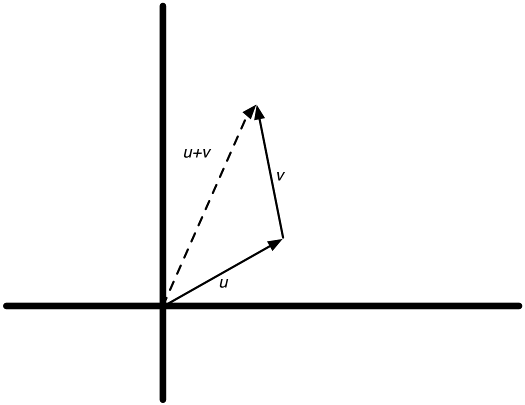


Vector addition: Vectors as arrows

Like complex numbers, addition of vectors over $\mathbb{R}$ can be visualized using arrows.

To add $\mathbf{u}$ and $\mathbf{v}$:

- place tail of $\mathbf{v}$'s arrow on head of $\mathbf{u}$'s arrow;
- draw a new arrow from tail of $\mathbf{u}$ to head of $\mathbf{v}$.

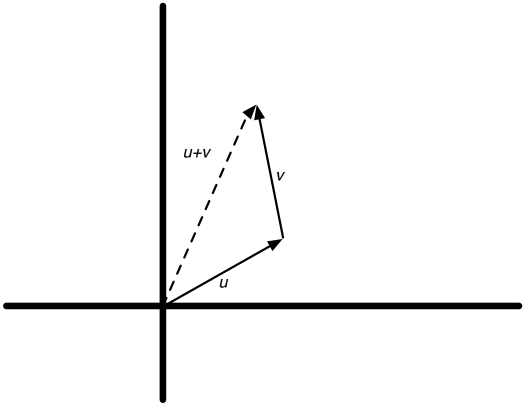


Vector addition: Vectors as arrows

Like complex numbers, addition of vectors over $\mathbb{R}$ can be visualized using arrows.

To add **u** and **v**:

- place tail of **v**'s arrow on head of **u**'s arrow;
- draw a new arrow from tail of **u** to head of **v**.



Scalar-vector multiplication

With complex numbers, *scaling* was multiplication by a real number $f(z) = r\,z$

For vectors,

- we refer to field elements as *scalars*;
- we use them to scale vectors:

$$\alpha\,\mathbf{v}$$

Greek letters (e.g. α, β, γ) denote scalars.

Scalar-vector multiplication

With complex numbers, *scaling* was multiplication by a real number $f(z) = r\,z$

For vectors,

- we refer to field elements as *scalars*;
- we use them to scale vectors:

$$\alpha\,\mathbf{v}$$

Greek letters (e.g. α, β, γ) denote scalars.

Scalar-vector multiplication

With complex numbers, *scaling* was multiplication by a real number $f(z) = r\,z$

For vectors,

- we refer to field elements as *scalars*;
- we use them to scale vectors:

$$\alpha\,\mathbf{v}$$

  Greek letters (e.g. α, β, γ) denote scalars.

Scalar-vector multiplication

Definition: Multiplying a vector **v** by a scalar α is defined as multiplying each entry of **v** by α:

$$\alpha\,[v_1, v_2, \ldots, v_n] = [\alpha\, v_1, \alpha\, v_2, \ldots, \alpha\, v_n]$$

Example: $2\,[5, 4, 10] = [2 \cdot 5, 2 \cdot 4, 2 \cdot 10] = [10, 8, 20]$

Scalar-vector multiplication

Definition: Multiplying a vector **v** by a scalar α is defined as multiplying each entry of **v** by α:

$$\alpha\,[v_1, v_2, \ldots, v_n] = [\alpha\, v_1, \alpha\, v_2, \ldots, \alpha\, v_n]$$

Example: $2\,[5, 4, 10] = [2 \cdot 5, 2 \cdot 4, 2 \cdot 10] = [10, 8, 20]$

Scalar-vector multiplication

Quiz: Suppose we represent n-vectors by n-element lists. Write a procedure `scalar_vector_mult(alpha, v)` that multiplies the vector `v` by the scalar `alpha`.

Scalar-vector multiplication

Quiz: Suppose we represent n-vectors by n-element lists. Write a procedure `scalar_vector_mult(alpha, v)` that multiplies the vector `v` by the scalar `alpha`.

Answer:

```
def scalar_vector_mult(alpha, v):
```

Scalar-vector multiplication

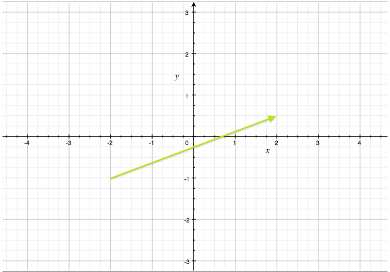
Quiz: Suppose we represent n-vectors by n-element lists. Write a procedure `scalar_vector_mult(alpha, v)` that multiplies the vector `v` by the scalar `alpha`.

Answer:

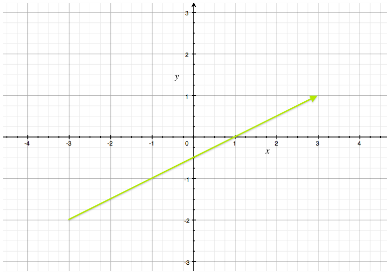
```
def scalar_vector_mult(alpha, v): return [alpha*x for x in v]
```

Scalar-vector multiplication: Scaling arrows

An arrow representing the vector $[3, 1.5]$ is this:

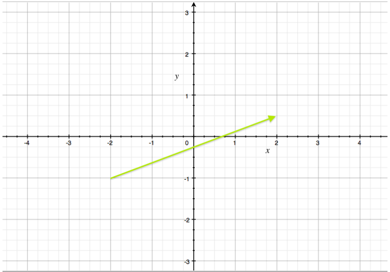


and an arrow representing two times this vector is this:

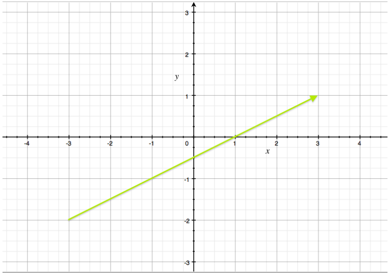


Scalar-vector multiplication: Scaling arrows

An arrow representing the vector $[3, 1.5]$ is this:



and an arrow representing two times this vector is this:



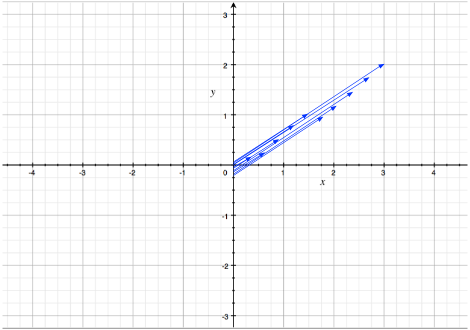
Scalar-vector multiplication: Associativity of scalar-vector multiplication

Associativity: $\alpha(\beta \mathbf{v}) = (\alpha\beta)\mathbf{v}$

Scalar-vector multiplication: Line segments through the origin

Consider scalar multiples of $\mathbf{v} = [3, 2]$:
$\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$

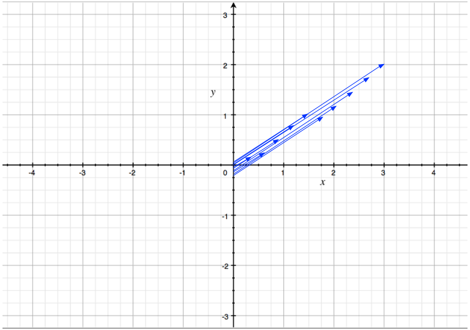
For each value of α in this set,
$\alpha\,\mathbf{v}$ is shorter than $\mathbf{v}$ but in same direction.



Scalar-vector multiplication: Line segments through the origin

Consider scalar multiples of $\mathbf{v} = [3, 2]$:
$\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$

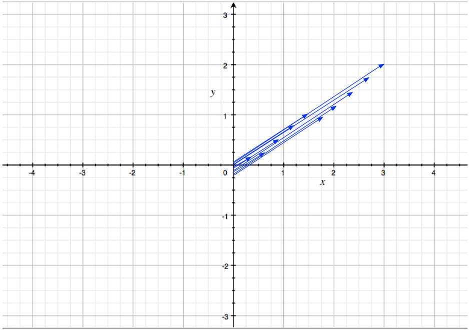
For each value of α in this set,
$\alpha\, \mathbf{v}$ is shorter than $\mathbf{v}$ but in same direction.



Scalar-vector multiplication: Line segments through the origin

Consider scalar multiples of $\mathbf{v} = [3, 2]$:
$\{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$

For each value of α in this set,
$\alpha\,\mathbf{v}$ is shorter than $\mathbf{v}$ but in same direction.



Scalar-vector multiplication: Line segments through the origin

Conclusion: The set of points

$$\{\alpha\,\mathbf{v} \ : \ \alpha \in \mathbb{R}, 0 \leq \alpha \leq 1\}$$

forms the line segment between the origin and $\mathbf{v}$

Scalar-vector multiplication: Lines through the origin

What if we let α range over all real numbers?

- Scalars bigger than 1 give rise to somewhat larger copies
- Negative scalars give rise to vectors pointing in the opposite direction

Scalar-vector multiplication: Lines through the origin

What if we let α range over all real numbers?

- Scalars bigger than 1 give rise to somewhat larger copies
- Negative scalars give rise to vectors pointing in the opposite direction

Scalar-vector multiplication: Lines through the origin

What if we let α range over all real numbers?

- Scalars bigger than 1 give rise to somewhat larger copies
- Negative scalars give rise to vectors pointing in the opposite direction

Scalar-vector multiplication: Lines through the origin

What if we let α range over all real numbers?

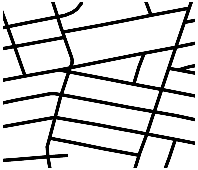
- Scalars bigger than 1 give rise to somewhat larger copies
- Negative scalars give rise to vectors pointing in the opposite direction

The set of points

$$\{\alpha\, \mathbf{v} \,:\, \alpha \in \mathbb{R}\}$$

forms the line through the origin and $\mathbf{v}$

Combining vector addition and scalar multiplication



We want to describe the set of points forming an arbitrary line segment (not necessarily through the origin).

Idea: Use the idea of translation.

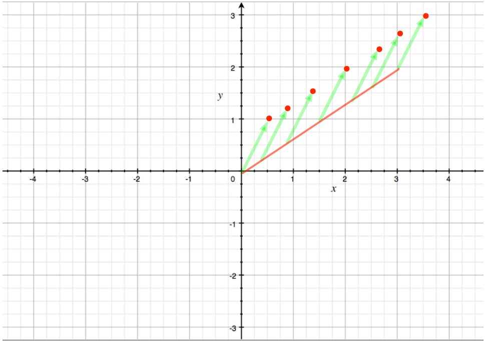
Start with line segment from $[0, 0]$ to $[3, 2]$:

$$\{\alpha\,[3, 2] \;:\; 0 \leq \alpha \leq 1\}$$

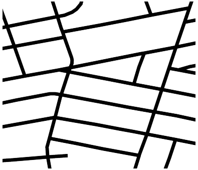
Translate it by adding $[0.5, 1]$ to every point:

$$\{[0.5, 1] + \alpha\,[3, 2] \;:\; 0 \leq \alpha \leq 1\}$$

Get line segment from $[0, 0]+[0.5, 1]$ to $[3, 2]+[0.5, 1]$



Combining vector addition and scalar multiplication



We want to describe the set of points forming an arbitrary line segment (not necessarily through the origin).

Idea: Use the idea of translation.

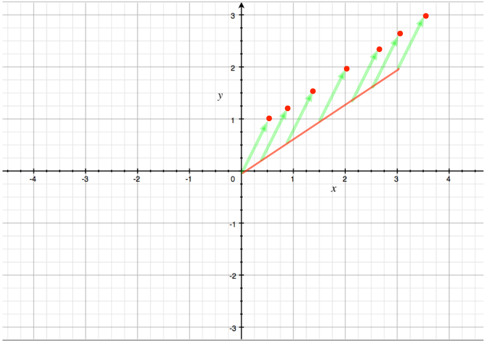
Start with line segment from $[0,0]$ to $[3,2]$:

$$\{\alpha\,[3,2] \;:\; 0 \leq \alpha \leq 1\}$$

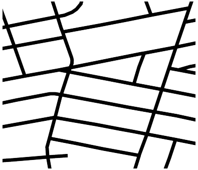
Translate it by adding $[0.5,1]$ to every point:

$$\{[0.5,1] + \alpha\,[3,2] \;:\; 0 \leq \alpha \leq 1\}$$

Get line segment from $[0,0]+[0.5,1]$ to $[3,2]+[0.5,1]$



Combining vector addition and scalar multiplication



We want to describe the set of points forming an arbitrary line segment (not necessarily through the origin).

Idea: Use the idea of translation.

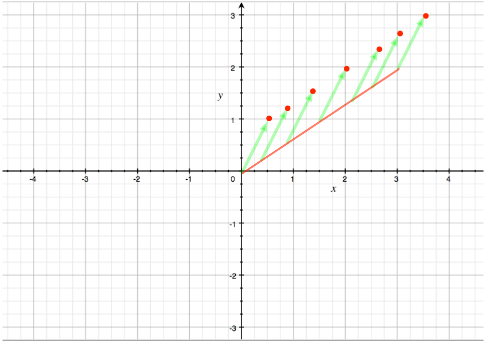
Start with line segment from $[0,0]$ to $[3,2]$:

$$\{\alpha\,[3,2]\ :\ 0 \le \alpha \le 1\}$$

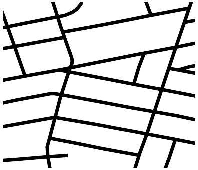
Translate it by adding $[0.5,1]$ to every point:

$$\{[0.5,1] + \alpha\,[3,2]\ :\ 0 \le \alpha \le 1\}$$

Get line segment from $[0,0]+[0.5,1]$ to $[3,2]+[0.5,1]$



Combining vector addition and scalar multiplication

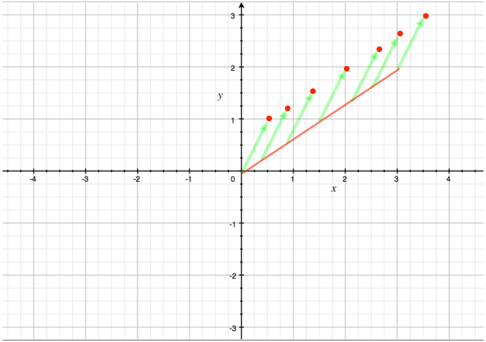


We want to describe the set of points forming an arbitrary line segment (not necessarily through the origin).

Idea: Use the idea of translation.

Start with line segment from $[0,0]$ to $[3,2]$:

$$\{\alpha\,[3,2]\ :\ 0 \leq \alpha \leq 1\}$$

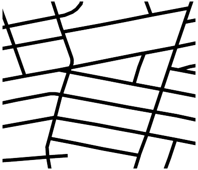


Translate it by adding $[0.5,1]$ to every point:

$$\{[0.5,1] + \alpha\,[3,2]\ :\ 0 \leq \alpha \leq 1\}$$

Get line segment from $[0,0]+[0.5,1]$ to $[3,2]+[0.5,1]$

Combining vector addition and scalar multiplication



We want to describe the set of points forming an arbitrary line segment (not necessarily through the origin).

Idea: Use the idea of translation.

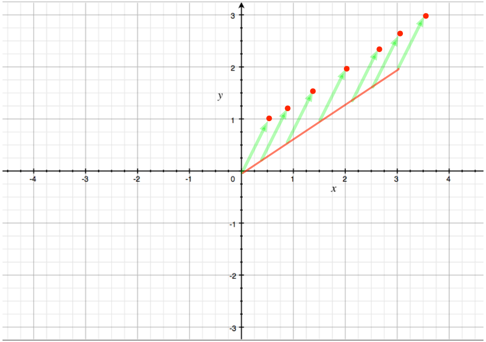
Start with line segment from $[0, 0]$ to $[3, 2]$:

$$\{\alpha\,[3, 2]\ :\ 0 \le \alpha \le 1\}$$

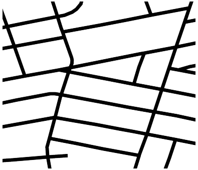
Translate it by adding $[0.5, 1]$ to every point:

$$\{[0.5, 1] + \alpha\,[3, 2]\ :\ 0 \le \alpha \le 1\}$$

Get line segment from $[0, 0]+[0.5, 1]$ to $[3, 2]+[0.5, 1]$



Combining vector addition and scalar multiplication

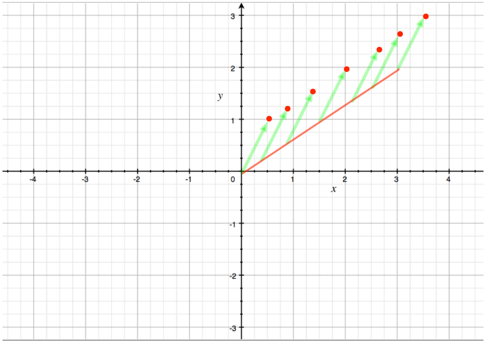


We want to describe the set of points forming an arbitrary line segment (not necessarily through the origin).

Idea: Use the idea of translation.

Start with line segment from $[0, 0]$ to $[3, 2]$:

$$\{\alpha\,[3, 2] \;:\; 0 \leq \alpha \leq 1\}$$



Translate it by adding $[0.5, 1]$ to every point:

$$\{[0.5, 1] + \alpha\,[3, 2] \;:\; 0 \leq \alpha \leq 1\}$$

Get line segment from $[0, 0]+[0.5, 1]$ to $[3, 2]+[0.5, 1]$

Combining vector addition and scalar multiplication: Distributive laws for scalar-vector multiplication and vector addition

Scalar-vector multiplication distributes over vector addition:

$$\alpha(\mathbf{u} + \mathbf{v}) = \alpha\mathbf{u} + \alpha\mathbf{v}$$

Example:

- On the one hand,

$$2\,([1,2,3] + [3,4,4]) = 2\,[4,6,7] = [8,12,14]$$

- On the other hand,

$$2\,([1,2,3] + [3,4,4]) = 2\,[1,2,3] + 2\,[3,4,4] = [2,4,6] + [6,8,8] = [8,12,14]$$

Combining vector addition and scalar multiplication: Distributive laws for scalar-vector multiplication and vector addition

Scalar-vector multiplication distributes over vector addition:

$$\alpha(\mathbf{u}+\mathbf{v}) = \alpha\mathbf{u} + \alpha\mathbf{v}$$

Example:

- On the one hand,

$$2\,([1,2,3]+[3,4,4]) = 2\,[4,6,7] = [8,12,14]$$

- On the other hand,

$$2\,([1,2,3]+[3,4,4]) = 2\,[1,2,3] + 2\,[3,4,4] = [2,4,6]+[6,8,8] = [8,12,14]$$

Combining vector addition and scalar multiplication: Distributive laws for scalar-vector multiplication and vector addition

Scalar-vector multiplication distributes over vector addition:

$$\alpha(\mathbf{u} + \mathbf{v}) = \alpha\mathbf{u} + \alpha\mathbf{v}$$

Example:

- On the one hand,

$$2\left([1,2,3] + [3,4,4]\right) = 2\,[4,6,7] = [8,12,14]$$

- On the other hand,

$$2\left([1,2,3] + [3,4,4]\right) = 2\,[1,2,3] + 2\,[3,4,4] = [2,4,6] + [6,8,8] = [8,12,14]$$

Combining vector addition and scalar multiplication: First look at convex combinations

Set of points making up the the $[0.5, 1]$-to-$[3.5, 3]$ segment:

$$\{\alpha\,[3,2] + [0.5,1] \;:\; \alpha \in \mathbb{R}, 0 \leq \alpha \leq 1\}$$

Not symmetric with respect to endpoints ☹

Combining vector addition and scalar multiplication: First look at convex combinations

Set of points making up the the $[0.5, 1]$-to-$[3.5, 3]$ segment:

$$\{\alpha\,[3,2] + [0.5,1] \;:\; \alpha \in \mathbb{R}, 0 \leq \alpha \leq 1\}$$

Not symmetric with respect to endpoints ☹

Use distributivity:

$$\alpha\,[3,2] + [0.5,1] \;=\; \alpha\,([3.5,3] - [0.5,1]) + [0.5,1]$$

Combining vector addition and scalar multiplication: First look at convex combinations

Set of points making up the the $[0.5, 1]$-to-$[3.5, 3]$ segment:

$$\{\alpha\,[3, 2] + [0.5, 1] \;:\; \alpha \in \mathbb{R}, 0 \leq \alpha \leq 1\}$$

Not symmetric with respect to endpoints ☹

Use distributivity:

$$\begin{aligned}
\alpha\,[3, 2] + [0.5, 1] &= \alpha\,([3.5, 3] - [0.5, 1]) + [0.5, 1] \\
&= \alpha\,[3.5, 3] - \alpha\,[0.5, 1] + [0.5, 1]
\end{aligned}$$

Combining vector addition and scalar multiplication: First look at convex combinations

Set of points making up the the $[0.5, 1]$-to-$[3.5, 3]$ segment:

$$\{\alpha\,[3, 2] + [0.5, 1] \ : \ \alpha \in \mathbb{R}, 0 \leq \alpha \leq 1\}$$

Not symmetric with respect to endpoints ☹

Use distributivity:

$$\begin{aligned}
\alpha\,[3, 2] + [0.5, 1] &= \alpha\,([3.5, 3] - [0.5, 1]) + [0.5, 1] \\
&= \alpha\,[3.5, 3] - \alpha\,[0.5, 1] + [0.5, 1] \\
&= \alpha\,[3.5, 3] + (1 - \alpha)\,[0.5, 1]
\end{aligned}$$

Combining vector addition and scalar multiplication: First look at convex combinations

Set of points making up the the $[0.5, 1]$-to-$[3.5, 3]$ segment:

$$\{\alpha\,[3, 2] + [0.5, 1] \;:\; \alpha \in \mathbb{R}, 0 \leq \alpha \leq 1\}$$

Not symmetric with respect to endpoints ☹

Use distributivity:

$$\begin{aligned}
\alpha\,[3, 2] + [0.5, 1] &= \alpha\,([3.5, 3] - [0.5, 1]) + [0.5, 1] \\
&= \alpha\,[3.5, 3] - \alpha\,[0.5, 1] + [0.5, 1] \\
&= \alpha\,[3.5, 3] + (1 - \alpha)\,[0.5, 1] \\
&= \alpha\,[3.5, 3] + \beta\,[0.5, 1]
\end{aligned}$$

where $\beta = 1 - \alpha$

Combining vector addition and scalar multiplication: First look at convex combinations

Set of points making up the the [0.5, 1]-to-[3.5, 3] segment:

$$\{\alpha\,[3,2] + [0.5,1] \;:\; \alpha \in \mathbb{R}, 0 \leq \alpha \leq 1\}$$

Not symmetric with respect to endpoints ☹

Use distributivity:

$$\begin{aligned}
\alpha\,[3,2] + [0.5,1] &= \alpha\,([3.5,3] - [0.5,1]) + [0.5,1] \\
&= \alpha\,[3.5,3] - \alpha\,[0.5,1] + [0.5,1] \\
&= \alpha\,[3.5,3] + (1-\alpha)\,[0.5,1] \\
&= \alpha\,[3.5,3] + \beta\,[0.5,1]
\end{aligned}$$

where $\beta = 1 - \alpha$

New formulation:

$$\{\alpha\,[3.5,3] + \beta\,[0.5,1] \;:\; \alpha, \beta \in \mathbb{R}, \alpha, \beta \geq 0, \alpha + \beta = 1\}$$

Combining vector addition and scalar multiplication: First look at convex combinations

Set of points making up the the [0.5, 1]-to-[3.5, 3] segment:

$$\{\alpha\,[3,2] + [0.5,1] \;:\; \alpha \in \mathbb{R}, 0 \leq \alpha \leq 1\}$$

Not symmetric with respect to endpoints ☹

Use distributivity:

$$\begin{aligned} \alpha\,[3,2] + [0.5,1] &= \alpha\,([3.5,3] - [0.5,1]) + [0.5,1] \\ &= \alpha\,[3.5,3] - \alpha\,[0.5,1] + [0.5,1] \\ &= \alpha\,[3.5,3] + (1-\alpha)\,[0.5,1] \\ &= \alpha\,[3.5,3] + \beta\,[0.5,1] \end{aligned}$$

where $\beta = 1 - \alpha$
New formulation:

$$\{\alpha\,[3.5,3] + \beta\,[0.5,1] \;:\; \alpha, \beta \in \mathbb{R}, \alpha, \beta \geq 0, \alpha + \beta = 1\}$$

Symmetric with respect to endpoints ☺

Combining vector addition and scalar multiplication: First look at convex combinations

New formulation:

$$\{\alpha\,[3.5, 3] + \beta\,[0.5, 1] \;:\; \alpha, \beta \in \mathbb{R}, \alpha, \beta \geq 0, \alpha + \beta = 1\}$$

Symmetric with respect to endpoints ☺

An expression of the form

$$\alpha\,\mathbf{u} + \beta\,\mathbf{v}$$

where $0 \leq \alpha \leq 1, 0 \leq \beta \leq 1$, and $\alpha + \beta = 1$ is called a *convex combination* of **u** and **v**

> The **u**-to-**v** line segment consists of the set of convex combinations of **u** and **v**.

Combining vector addition and scalar multiplication: First look at convex combinations

New formulation:

$$\{\alpha\,[3.5, 3] + \beta\,[0.5, 1] \;:\; \alpha, \beta \in \mathbb{R}, \alpha, \beta \geq 0, \alpha + \beta = 1\}$$

Symmetric with respect to endpoints ☺

An expression of the form

$$\alpha\,\mathbf{u} + \beta\,\mathbf{v}$$

where $0 \leq \alpha \leq 1, 0 \leq \beta \leq 1$, and $\alpha + \beta = 1$ is called a *convex combination* of **u** and **v**

The **u**-to-**v** line segment consists of the set of convex combinations of **u** and **v**.

Combining vector addition and scalar multiplication: First look at convex combinations

New formulation:

$$\{\alpha\,[3.5, 3] + \beta\,[0.5, 1] \;:\; \alpha, \beta \in \mathbb{R}, \alpha, \beta \geq 0, \alpha + \beta = 1\}$$

Symmetric with respect to endpoints ☺

An expression of the form

$$\alpha\,\mathbf{u} + \beta\,\mathbf{v}$$

where $0 \leq \alpha \leq 1, 0 \leq \beta \leq 1$, and $\alpha + \beta = 1$ is called a *convex combination* of **u** and **v**

The **u**-to-**v** line segment consists of the set of convex combinations of **u** and **v**.

Combining vector addition and scalar multiplication: First look at convex combinations

$\mathbf{u} =$  and $\mathbf{v} =$ 

Use scalars $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{2}$:

$$\frac{1}{2} \qquad + \quad \frac{1}{2} \qquad =$$

Combining vector addition and scalar multiplication: First look at convex combinations

$\mathbf{u} =$  and $\mathbf{v} =$ 

Use scalars $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{2}$:

$$\frac{1}{2} \qquad + \quad \frac{1}{2} \qquad =$$

Combining vector addition and scalar multiplication: First look at convex combinations

$\mathbf{u} =$  and $\mathbf{v} =$ 

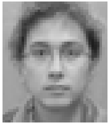
Use scalars $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{2}$:

$\frac{1}{2}$  $+$ $\frac{1}{2}$  $=$

Combining vector addition and scalar multiplication: First look at convex combinations

$\mathbf{u} =$  and $\mathbf{v} =$ 

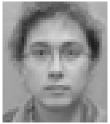
Use scalars $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{2}$:

$\frac{1}{2}$  $+$ $\frac{1}{2}$  $=$ 

Combining vector addition and scalar multiplication: First look at convex combinations

$\mathbf{u} =$  and $\mathbf{v} =$ 

Use scalars $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{2}$:

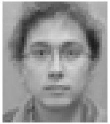
$\frac{1}{2}$  $+$ $\frac{1}{2}$  $=$ 

"Line segment" between two faces:

Combining vector addition and scalar multiplication: First look at convex combinations

$\mathbf{u} =$  and $\mathbf{v} =$ 

Use scalars $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{2}$:

$\frac{1}{2}$  $+$ $\frac{1}{2}$  $=$ 

"Line segment" between two faces:

$1\mathbf{u} + 0\mathbf{v}$

$\frac{7}{8}\mathbf{u} + \frac{1}{8}\mathbf{v}$

$\frac{6}{8}\mathbf{u} + \frac{2}{8}\mathbf{v}$

$\frac{5}{8}\mathbf{u} + \frac{3}{8}\mathbf{v}$

$\frac{4}{8}\mathbf{u} + \frac{4}{8}\mathbf{v}$

$\frac{3}{8}\mathbf{u} + \frac{5}{8}\mathbf{v}$

$\frac{2}{8}\mathbf{u} + \frac{6}{8}\mathbf{v}$

$\frac{1}{8}\mathbf{u} + \frac{7}{8}\mathbf{v}$

$1\mathbf{u} + 0\mathbf{v}$

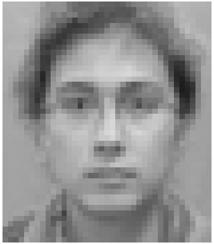
Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations



Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at convex combinations

Combining vector addition and scalar multiplication: First look at affine combinations

Infinite line through $[0.5, 1]$ and $[3.5, 3]$?

Our formulation so far ☹

$$\{[0.5, 1] + \alpha\,[3, 2]\ :\ \alpha \in \mathbb{R}\}$$

Nicer formulation ☺:

$$\{\alpha\,[3.5, 3] + \beta\,[0.5, 1]\ :\ \alpha \in \mathbb{R}, \beta \in R, \alpha + \beta = 1\}$$

An expression of the form $\alpha\,\mathbf{u} + \beta\,\mathbf{v}$ where $\alpha + \beta = 1$ is called an *affine* combination of $\mathbf{u}$ and $\mathbf{v}$.

The line through $\mathbf{u}$ and $\mathbf{v}$ consists of the set of affine combinations of $\mathbf{u}$ and $\mathbf{v}$.

Combining vector addition and scalar multiplication: First look at affine combinations

Infinite line through $[0.5, 1]$ and $[3.5, 3]$?

Our formulation so far ☹

$$\{[0.5, 1] + \alpha\,[3, 2] \;:\; \alpha \in \mathbb{R}\}$$

Nicer formulation ☺:

$$\{\alpha\,[3.5, 3] + \beta\,[0.5, 1] \;:\; \alpha \in \mathbb{R}, \beta \in R, \alpha + \beta = 1\}$$

An expression of the form $\alpha\,\mathbf{u} + \beta\,\mathbf{v}$ where $\alpha + \beta = 1$ is called an *affine* combination of $\mathbf{u}$ and $\mathbf{v}$.

The line through $\mathbf{u}$ and $\mathbf{v}$ consists of the set of affine combinations of $\mathbf{u}$ and $\mathbf{v}$.

Combining vector addition and scalar multiplication: First look at affine combinations

Infinite line through $[0.5, 1]$ and $[3.5, 3]$?
Our formulation so far ☹

$$\{[0.5, 1] + \alpha\,[3, 2] \; : \; \alpha \in \mathbb{R}\}$$

Nicer formulation ☺:

$$\{\alpha\,[3.5, 3] + \beta\,[0.5, 1] \; : \; \alpha \in \mathbb{R}, \beta \in R, \alpha + \beta = 1\}$$

An expression of the form $\alpha\,\mathbf{u} + \beta\,\mathbf{v}$ where $\alpha + \beta = 1$ is called an *affine* combination of $\mathbf{u}$ and $\mathbf{v}$.

The line through $\mathbf{u}$ and $\mathbf{v}$ consists of the set of affine combinations of $\mathbf{u}$ and $\mathbf{v}$.

Combining vector addition and scalar multiplication: First look at affine combinations

Infinite line through $[0.5, 1]$ and $[3.5, 3]$?
Our formulation so far ☹

$$\{[0.5, 1] + \alpha\,[3, 2] \;:\; \alpha \in \mathbb{R}\}$$

Nicer formulation ☺:

$$\{\alpha\,[3.5, 3] + \beta\,[0.5, 1] \;:\; \alpha \in \mathbb{R}, \beta \in R, \alpha + \beta = 1\}$$

An expression of the form $\alpha\,\mathbf{u} + \beta\,\mathbf{v}$ where $\alpha + \beta = 1$ is called an *affine* combination of $\mathbf{u}$ and $\mathbf{v}$.

The line through $\mathbf{u}$ and $\mathbf{v}$ consists of the set of affine combinations of $\mathbf{u}$ and $\mathbf{v}$.

Combining vector addition and scalar multiplication: First look at affine combinations

Infinite line through $[0.5, 1]$ and $[3.5, 3]$?
Our formulation so far ☹

$$\{[0.5, 1] + \alpha\,[3, 2] \;:\; \alpha \in \mathbb{R}\}$$

Nicer formulation ☺:

$$\{\alpha\,[3.5, 3] + \beta\,[0.5, 1] \;:\; \alpha \in \mathbb{R}, \beta \in R, \alpha + \beta = 1\}$$

An expression of the form $\alpha\,\mathbf{u} + \beta\,\mathbf{v}$ where $\alpha + \beta = 1$ is called an *affine* combination of $\mathbf{u}$ and $\mathbf{v}$.

The line through **u** and **v** consists of the set of affine combinations of **u** and **v**.

Vectors over $GF(2)$

Addition of vectors over $GF(2)$:

$$\begin{array}{cccccc} & 1 & 1 & 1 & 1 & 1 \\ + & 1 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 & 1 & 0 \end{array}$$

For brevity, in doing $GF(2)$, we often write 1101 instead of [1,1,0,1].

Example: Over $GF(2)$, what is $1101 + 0111$?
Answer: 1010

Vectors over $GF(2)$

Addition of vectors over $GF(2)$:

$$\begin{array}{cccccc} & 1 & 1 & 1 & 1 & 1 \\ + & 1 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 & 1 & 0 \end{array}$$

For brevity, in doing $GF(2)$, we often write 1101 instead of [1,1,0,1].

Example: Over $GF(2)$, what is $1101 + 0111$?
Answer: 1010

Vectors over $GF(2)$

Addition of vectors over $GF(2)$:

$$\begin{array}{cccccc} & 1 & 1 & 1 & 1 & 1 \\ + & 1 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 & 1 & 0 \end{array}$$

For brevity, in doing $GF(2)$, we often write 1101 instead of [1,1,0,1].

Example: Over $GF(2)$, what is $1101 + 0111$?

Answer: 1010

Vectors over $GF(2)$

Addition of vectors over $GF(2)$:

$$\begin{array}{cccccc} & 1 & 1 & 1 & 1 & 1 \\ + & 1 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 & 1 & 0 \end{array}$$

For brevity, in doing $GF(2)$, we often write 1101 instead of [1,1,0,1].

Example: Over $GF(2)$, what is $1101 + 0111$?
Answer: 1010

Vectors over $GF(2)$: Perfect secrecy

Represent encryption of n bits by addition of n-vectors over $GF(2)$.

Example:

Alice and Bob agree on the following 10-vector as a key:

$$\mathbf{k} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]$$

Alice wants to send this message to Bob:

$$\mathbf{p} = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

She encrypts it by adding $\mathbf{p}$ to $\mathbf{k}$:

$$\mathbf{c} = \mathbf{k} + \mathbf{p} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] + [0, 0, 0, 1, 1, 1, 0, 1, 0, 1] = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0]$$

When Bob receives $\mathbf{c}$, he decrypts it by adding $\mathbf{k}$:

$$\mathbf{c} + \mathbf{k} = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0] + [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

which is the original message.

Vectors over $GF(2)$: Perfect secrecy

Represent encryption of n bits by addition of n-vectors over $GF(2)$.

Example:

Alice and Bob agree on the following 10-vector as a key:

$$\mathbf{k} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]$$

Alice wants to send this message to Bob:

$$\mathbf{p} = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

She encrypts it by adding $\mathbf{p}$ to $\mathbf{k}$:

$$\mathbf{c} = \mathbf{k} + \mathbf{p} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] + [0, 0, 0, 1, 1, 1, 0, 1, 0, 1] = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0]$$

When Bob receives $\mathbf{c}$, he decrypts it by adding $\mathbf{k}$:

$$\mathbf{c} + \mathbf{k} = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0] + [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

which is the original message.

Vectors over $GF(2)$: Perfect secrecy

Represent encryption of n bits by addition of n-vectors over $GF(2)$.

Example:

Alice and Bob agree on the following 10-vector as a key:

$$\mathbf{k} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]$$

Alice wants to send this message to Bob:

$$\mathbf{p} = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

She encrypts it by adding $\mathbf{p}$ to $\mathbf{k}$:

$$\mathbf{c} = \mathbf{k} + \mathbf{p} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] + [0, 0, 0, 1, 1, 1, 0, 1, 0, 1] = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0]$$

When Bob receives $\mathbf{c}$, he decrypts it by adding $\mathbf{k}$:

$$\mathbf{c} + \mathbf{k} = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0] + [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

which is the original message.

Vectors over $GF(2)$: Perfect secrecy

Represent encryption of n bits by addition of n-vectors over $GF(2)$.

Example:

Alice and Bob agree on the following 10-vector as a key:

$$\mathbf{k} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]$$

Alice wants to send this message to Bob:

$$\mathbf{p} = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

She encrypts it by adding $\mathbf{p}$ to $\mathbf{k}$:

$$\mathbf{c} = \mathbf{k} + \mathbf{p} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] + [0, 0, 0, 1, 1, 1, 0, 1, 0, 1] = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0]$$

When Bob receives $\mathbf{c}$, he decrypts it by adding $\mathbf{k}$:

$$\mathbf{c} + \mathbf{k} = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0] + [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

which is the original message.

Vectors over $GF(2)$: Perfect secrecy

Represent encryption of n bits by addition of n-vectors over $GF(2)$.

Example:

Alice and Bob agree on the following 10-vector as a key:

$$\mathbf{k} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]$$

Alice wants to send this message to Bob:

$$\mathbf{p} = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

She encrypts it by adding $\mathbf{p}$ to $\mathbf{k}$:

$$\mathbf{c} = \mathbf{k} + \mathbf{p} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] + [0, 0, 0, 1, 1, 1, 0, 1, 0, 1] = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0]$$

When Bob receives $\mathbf{c}$, he decrypts it by adding $\mathbf{k}$:

$$\mathbf{c} + \mathbf{k} = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0] + [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

which is the original message.

Vectors over $GF(2)$: Perfect secrecy

Represent encryption of n bits by addition of n-vectors over $GF(2)$.

Example:

Alice and Bob agree on the following 10-vector as a key:

$$\mathbf{k} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]$$

Alice wants to send this message to Bob:

$$\mathbf{p} = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

She encrypts it by adding $\mathbf{p}$ to $\mathbf{k}$:

$$\mathbf{c} = \mathbf{k}+\mathbf{p} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]+[0, 0, 0, 1, 1, 1, 0, 1, 0, 1] = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0]$$

When Bob receives $\mathbf{c}$, he decrypts it by adding $\mathbf{k}$:

$$\mathbf{c} + \mathbf{k} = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0] + [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

which is the original message.

Vectors over $GF(2)$: Perfect secrecy

Represent encryption of n bits by addition of n-vectors over $GF(2)$.

Example:

Alice and Bob agree on the following 10-vector as a key:

$$\mathbf{k} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]$$

Alice wants to send this message to Bob:

$$\mathbf{p} = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

She encrypts it by adding $\mathbf{p}$ to $\mathbf{k}$:

$$\mathbf{c} = \mathbf{k}+\mathbf{p} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]+[0, 0, 0, 1, 1, 1, 0, 1, 0, 1] = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0]$$

When Bob receives $\mathbf{c}$, he decrypts it by adding $\mathbf{k}$:

$$\mathbf{c}+\mathbf{k} = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0] + [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

which is the original message.

Vectors over $GF(2)$: Perfect secrecy

Represent encryption of n bits by addition of n-vectors over $GF(2)$.

Example:

Alice and Bob agree on the following 10-vector as a key:

$$\mathbf{k} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1]$$

Alice wants to send this message to Bob:

$$\mathbf{p} = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

She encrypts it by adding $\mathbf{p}$ to $\mathbf{k}$:

$$\mathbf{c} = \mathbf{k} + \mathbf{p} = [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] + [0, 0, 0, 1, 1, 1, 0, 1, 0, 1] = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0]$$

When Bob receives $\mathbf{c}$, he decrypts it by adding $\mathbf{k}$:

$$\mathbf{c} + \mathbf{k} = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0] + [0, 1, 1, 0, 1, 0, 0, 0, 0, 1] = [0, 0, 0, 1, 1, 1, 0, 1, 0, 1]$$

which is the original message.

Vectors over $GF(2)$: Perfect secrecy

If the key is chosen according to the uniform distribution, encryption by addition of vectors over $GF(2)$ achieves *perfect secrecy*.

For each plaintext $\mathbf{p}$, the function that maps the key to the cyphertext

$$\mathbf{k} \mapsto \mathbf{k} + \mathbf{p}$$

is invertible

Since the key $\mathbf{k}$ has the uniform distribution,
the cyphertext $\mathbf{c}$ also has the uniform distribution.

Vectors over $GF(2)$: Perfect secrecy

If the key is chosen according to the uniform distribution, encryption by addition of vectors over $GF(2)$ achieves *perfect secrecy*.
For each plaintext $\mathbf{p}$, the function that maps the key to the cyphertext

$$\mathbf{k} \mapsto \mathbf{k} + \mathbf{p}$$

is invertible

Since the key $\mathbf{k}$ has the uniform distribution,
the cyphertext $\mathbf{c}$ also has the uniform distribution.

Vectors over $GF(2)$: Perfect secrecy

If the key is chosen according to the uniform distribution, encryption by addition of vectors over $GF(2)$ achieves *perfect secrecy*.
For each plaintext $\mathbf{p}$, the function that maps the key to the cyphertext

$$\mathbf{k} \mapsto \mathbf{k} + \mathbf{p}$$

is invertible

Since the key $\mathbf{k}$ has the uniform distribution,
the cyphertext $\mathbf{c}$ also has the uniform distribution.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- What can Alice learn without Bob?
- All she receives is a random n-vector.
- What about Bob?
- He receives the output of $f(\mathbf{x}) = \mathbf{v} - \mathbf{x}$ where the input is random and uniform.
- Since $f(\mathbf{x})$ is invertible, the output is also random and uniform.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- What can Alice learn without Bob?
- All she receives is a random n-vector.
- What about Bob?
- He receives the output of $f(\mathbf{x}) = \mathbf{v} - \mathbf{x}$ where the input is random and uniform.
- Since $f(\mathbf{x})$ is invertible, the output is also random and uniform.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- What can Alice learn without Bob?
- All she receives is a random n-vector.
- What about Bob?
- He receives the output of $f(\mathbf{x}) = \mathbf{v} - \mathbf{x}$ where the input is random and uniform.
- Since $f(\mathbf{x})$ is invertible, the output is also random and uniform.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- What can Alice learn without Bob?
- All she receives is a random n-vector.
- What about Bob?
- He receives the output of $f(\mathbf{x}) = \mathbf{v} - \mathbf{x}$ where the input is random and uniform.
- Since $f(\mathbf{x})$ is invertible, the output is also random and uniform.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- What can Alice learn without Bob?
- All she receives is a random n-vector.
- What about Bob?
- He receives the output of $f(\mathbf{x}) = \mathbf{v} - \mathbf{x}$ where the input is random and uniform.
- Since $f(\mathbf{x})$ is invertible, the output is also random and uniform.

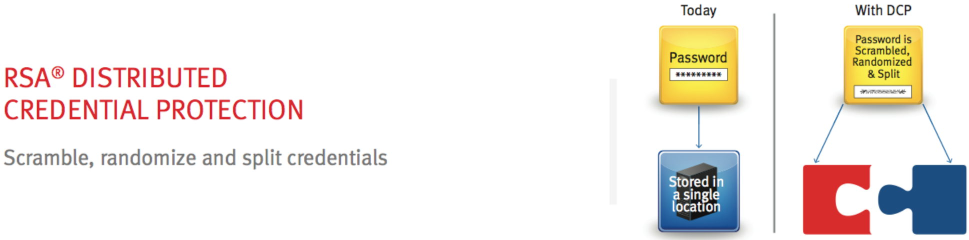
Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

Too simple to be useful, right?
RSA just introduced a product based on this idea:

- Split each password into two parts.
- Store the two parts on two separate servers.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

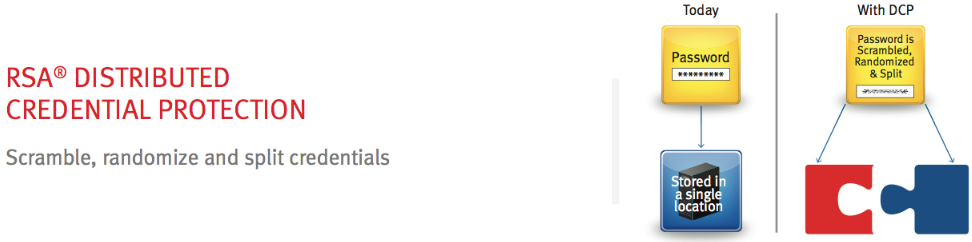
Too simple to be useful, right?
RSA just introduced a product based on this idea:



- Split each password into two parts.
- Store the two parts on two separate servers.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

Too simple to be useful, right?
RSA just introduced a product based on this idea:



- Split each password into two parts.
- Store the two parts on two separate servers.

Vectors over $GF(2)$: *Lights Out*

- *input:* Configuration of lights
- *output:* Which buttons to press in order to turn off all lights?

Computational Problem: $\boxed{\text{Solve an instance of } \textit{Lights Out}}$

Represent state using `range(5)`$\times$`range(5)`-vector over $GF(2)$.

Example state vector:

Represent each button as a vector (with ones in positions that the button toggles)

Example button vector:

Vectors over $GF(2)$: *Lights Out*

- *input:* Configuration of lights
- *output:* Which buttons to press in order to turn off all lights?

Computational Problem: $\boxed{\text{Solve an instance of } \textit{Lights Out}}$

Represent state using `range(5)`$\times$`range(5)`-vector over $GF(2)$.

Example state vector:

Represent each button as a vector (with ones in positions that the button toggles)

Example button vector:

Vectors over $GF(2)$: *Lights Out*

- *input:* Configuration of lights
- *output:* Which buttons to press in order to turn off all lights?

Computational Problem: $\boxed{\text{Solve an instance of } \textit{Lights Out}}$

Represent state using `range(5)`$\times$`range(5)`-vector over $GF(2)$.

Example state vector:

Represent each button as a vector (with ones in positions that the button toggles)

Example button vector:

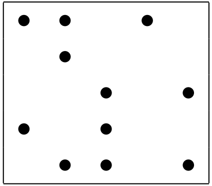
Vectors over $GF(2)$: *Lights Out*

- *input:* Configuration of lights
- *output:* Which buttons to press in order to turn off all lights?

Computational Problem: $\boxed{\text{Solve an instance of } \textit{Lights Out}}$

Represent state using `range(5)`$\times$`range(5)`-vector over $GF(2)$.

Example state vector:



Represent each button as a vector (with ones in positions that the button toggles)

Example button vector:

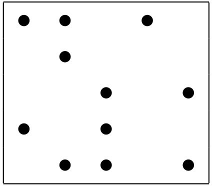
Vectors over $GF(2)$: *Lights Out*

- *input:* Configuration of lights
- *output:* Which buttons to press in order to turn off all lights?

Computational Problem: $\boxed{\text{Solve an instance of } \textit{Lights Out}}$

Represent state using `range(5)`$\times$`range(5)`-vector over $GF(2)$.

Example state vector:



Represent each button as a vector (with ones in positions that the button toggles)

Example button vector:

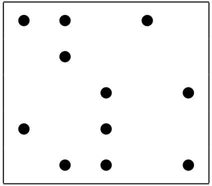
Vectors over $GF(2)$: *Lights Out*

- *input:* Configuration of lights
- *output:* Which buttons to press in order to turn off all lights?

Computational Problem: $\boxed{\text{Solve an instance of } \textit{Lights Out}}$

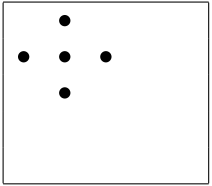
Represent state using `range(5)`$\times$`range(5)`-vector over $GF(2)$.

Example state vector:



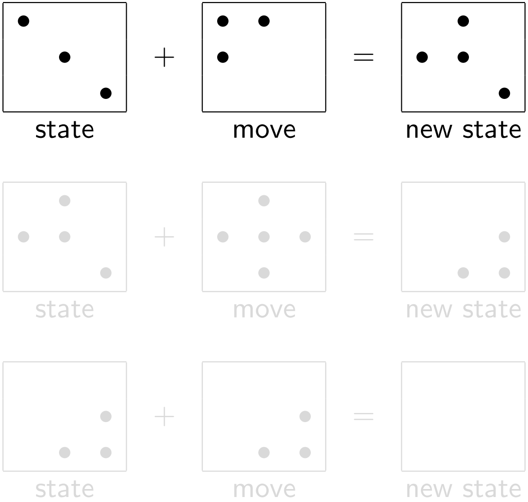
Represent each button as a vector (with ones in positions that the button toggles)

Example button vector:



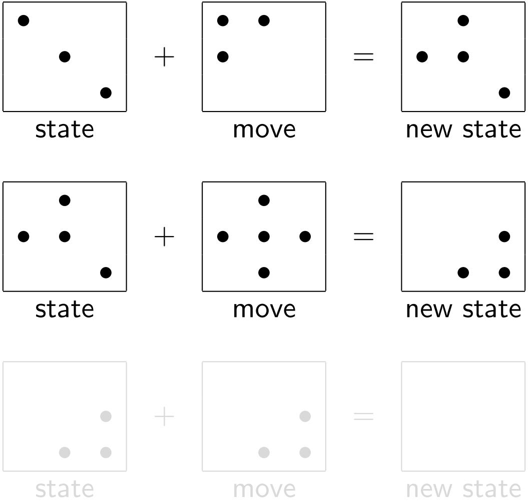
Vectors over $GF(2)$: *Lights Out*

Look at 3×3 case.



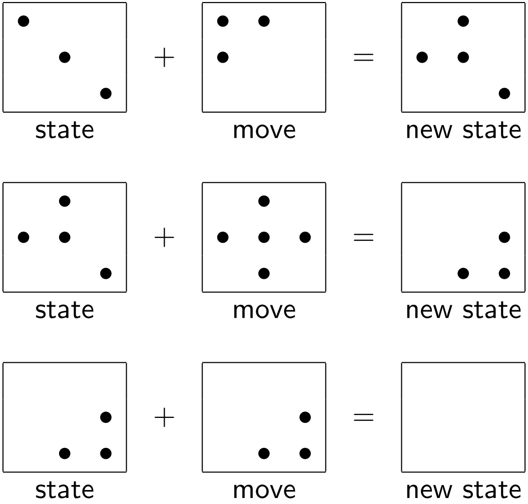
Vectors over $GF(2)$: *Lights Out*

Look at 3×3 case.



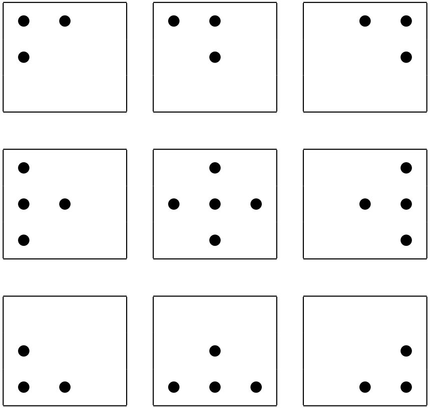
Vectors over $GF(2)$: *Lights Out*

Look at 3×3 case.



Vectors over $GF(2)$: *Lights Out*

Button vectors for 3×3:



Computational Problem: $\boxed{\text{Which sequence of button vectors sum to } \mathbf{s}\text{?}}$

Vectors over $GF(2)$: *Lights Out*

Computational Problem: $\boxed{\text{Which sequence of button vectors sum to } \mathbf{s}\text{?}}$

Observations:

- By commutative property of vector addition, order doesn't matter.
- A button vector occuring twice cancels out.

Replace Computational Problem with: $\boxed{\text{Which } \textit{set} \text{ of button vectors sum to } \mathbf{s}\text{?}}$

Vectors over $GF(2)$: *Lights Out*

Computational Problem: $\boxed{\text{Which sequence of button vectors sum to } \mathbf{s}\text{?}}$

Observations:

- By commutative property of vector addition, order doesn't matter.
- A button vector occuring twice cancels out.

Replace Computational Problem with: $\boxed{\text{Which } \textit{set} \text{ of button vectors sum to } \mathbf{s}\text{?}}$

Vectors over $GF(2)$: *Lights Out*

Computational Problem: $\boxed{\text{Which sequence of button vectors sum to } \mathbf{s}?}$

Observations:

- By commutative property of vector addition, order doesn't matter.
- A button vector occuring twice cancels out.

Replace Computational Problem with: $\boxed{\text{Which } \textit{set} \text{ of button vectors sum to } \mathbf{s}?}$

Vectors over $GF(2)$: *Lights Out*

Computational Problem: $\boxed{\text{Which sequence of button vectors sum to } \mathbf{s}?}$

Observations:

- By commutative property of vector addition, order doesn't matter.
- A button vector occuring twice cancels out.

Replace Computational Problem with: $\boxed{\text{Which } \textit{set} \text{ of button vectors sum to } \mathbf{s}?}$

Vectors over $GF(2)$: *Lights Out*

Computational Problem: $\boxed{\text{Which sequence of button vectors sum to } \mathbf{s}\text{?}}$

Observations:

- By commutative property of vector addition, order doesn't matter.
- A button vector occuring twice cancels out.

Replace Computational Problem with: $\boxed{\text{Which } \textit{set} \text{ of button vectors sum to } \mathbf{s}\text{?}}$

Vectors over $GF(2)$: *Lights Out*

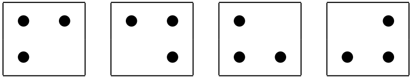
Replace our original Computational Problem with a more general one:

$\boxed{\text{Solve an instance of } \textit{Lights Out}}$ $\Rightarrow$ $\boxed{\text{Which set of button vectors sum to } \mathbf{s}\text{?}}$

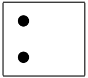
$\Rightarrow$ $\boxed{\text{Find subset of } GF(2) \text{ vectors } \mathbf{v}_1, \ldots, \mathbf{v}_n \text{ whose sum equals } \mathbf{s}}$

Vectors over $GF(2)$: *Lights Out*

Button vectors for 2×2 version:

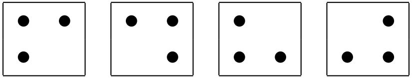


where the black dots represent ones.

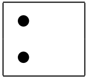
Quiz: Find the subset of the button vectors whose sum is 

Vectors over $GF(2)$: *Lights Out*

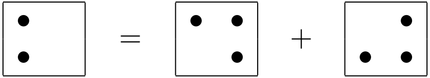
Button vectors for 2×2 version:



where the black dots represent ones.

Quiz: Find the subset of the button vectors whose sum is 

Answer:



Dot-product

Dot-product of two D-vectors is sum of product of corresponding entries:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{k \in D} \mathbf{u}[k]\ \mathbf{v}[k]$$

Example: For traditional vectors $\mathbf{u} = [u_1, \ldots, u_n]$ and $\mathbf{v} = [v_1, \ldots, v_n]$,

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n$$

Output is a scalar, not a vector

Dot-product sometimes called *scalar product*.

Dot-product

Dot-product of two D-vectors is sum of product of corresponding entries:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{k \in D} \mathbf{u}[k]\ \mathbf{v}[k]$$

Example: For traditional vectors $\mathbf{u} = [u_1, \ldots, u_n]$ and $\mathbf{v} = [v_1, \ldots, v_n]$,

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n$$

Output is a scalar, not a vector

Dot-product sometimes called *scalar product*.

Dot-product

Dot-product of two D-vectors is sum of product of corresponding entries:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{k \in D} \mathbf{u}[k]\ \mathbf{v}[k]$$

Example: For traditional vectors $\mathbf{u} = [u_1, \ldots, u_n]$ and $\mathbf{v} = [v_1, \ldots, v_n]$,

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n$$

Output is a scalar, not a vector
Dot-product sometimes called *scalar product*.

Dot-product

Example: Dot-product of $[1, 1, 1, 1, 1]$ and $[10, 20, 0, 40, -100]$:

$$
\begin{array}{rrrrrrrrrrr}
 & 1 & & 1 & & 1 & & 1 & & 1 & \\
\bullet & 10 & & 20 & & 0 & & 40 & & \text{-100} & \\
\hline
 & 10 & + & 20 & + & 0 & + & 40 & + & (\text{-100}) & = & \text{-30}
\end{array}
$$

Dot-product

Example: Dot-product of $[1, 1, 1, 1, 1]$ and $[10, 20, 0, 40, -100]$:

$$
\begin{array}{rccccccccccc}
 & 1 & & 1 & & 1 & & 1 & & 1 & & \\
\bullet & 10 & & 20 & & 0 & & 40 & & \text{-100} & & \\
\hline
 & 10 & + & 20 & + & 0 & + & 40 & + & (\text{-100}) & = & \text{-30}
\end{array}
$$

Quiz: Dot-product

Quiz: Write a procedure `list_dot(u, v)` with the following spec:

- *input:* equal-length lists `u` and `v` of field elements
- *output:* the dot-product of `u` and `v` interpreted as vectors

Hint: Use the `sum(·)` procedure together with a list comprehension.

Quiz: Dot-product

Quiz: Write a procedure `list_dot(u, v)` with the following spec:

- *input:* equal-length lists `u` and `v` of field elements
- *output:* the dot-product of `u` and `v` interpreted as vectors

Hint: Use the `sum(·)` procedure together with a list comprehension.

Answer:

```
def list_dot(u, v): return sum([u[i]*v[i] for i in range(len(u))])
```

or

```
def list_dot(u, v): return sum([a*b for (a,b) in zip(u,v)])
```

Dot-product: Total cost or benefit

Suppose D consists of four main ingredients of beer:

$$D = \{\text{malt}, \text{hops}, \text{yeast}, \text{water}\}$$

A *cost* vector maps each food to a price per unit amount:

$$cost = \{\text{hops} : \$2.50/ounce, \text{malt} : \$1.50/pound, \text{water} : \$0.06/gallon, \text{yeast} : \$.45/g\}$$

A *quantity* vector maps each food to an amount (e.g. measured in pounds).
$quantity = \{$hops:6 oz, malt:14 pounds, water:7 gallons, yeast:11 grams$\}$

The total cost is the dot-product of *cost* with *quantity*:

$$cost \cdot quantity = \$2.50 \cdot 6 + \$1.50 \cdot 14 + \$0.006 \cdot 7 + \$0.45 \cdot 11 = \$40.992$$

A *value* vector maps each food to its caloric content per pound:

$$value = \{\text{hops} : 0, \text{malt} : 960, \text{water} : 0, \text{yeast} : 3.25\}$$

The total calories represented by a pint of beer is the dot-product of *value* with *quantity*:

Dot-product: Total cost or benefit

Suppose D consists of four main ingredients of beer:

$$D = \{\text{malt}, \text{hops}, \text{yeast}, \text{water}\}$$

A *cost* vector maps each food to a price per unit amount:

$$cost = \{\text{hops} : \$2.50/ounce, \text{malt} : \$1.50/pound, \text{water} : \$0.06/gallon, \text{yeast} : \$.45/g\}$$

A *quantity* vector maps each food to an amount (e.g. measured in pounds).
$quantity = \{$hops:6 oz, malt:14 pounds, water:7 gallons, yeast:11 grams$\}$

The total cost is the dot-product of *cost* with *quantity*:

$$cost \cdot quantity = \$2.50 \cdot 6 + \$1.50 \cdot 14 + \$0.006 \cdot 7 + \$0.45 \cdot 11 = \$40.992$$

A *value* vector maps each food to its caloric content per pound:

$$value = \{\text{hops} : 0, \text{malt} : 960, \text{water} : 0, \text{yeast} : 3.25\}$$

The total calories represented by a pint of beer is the dot-product of *value* with *quantity*:

Dot-product: Total cost or benefit

Suppose D consists of four main ingredients of beer:

$$D = \{\text{malt}, \text{hops}, \text{yeast}, \text{water}\}$$

A *cost* vector maps each food to a price per unit amount:

$$cost = \{\text{hops} : \$2.50/ounce, \text{malt} : \$1.50/pound, \text{water} : \$0.06/gallon, \text{yeast} : \$.45/g\}$$

A *quantity* vector maps each food to an amount (e.g. measured in pounds).
$quantity = \{$hops:6 oz, malt:14 pounds, water:7 gallons, yeast:11 grams$\}$

The total cost is the dot-product of *cost* with *quantity*:

$$cost \cdot quantity = \$2.50 \cdot 6 + \$1.50 \cdot 14 + \$0.006 \cdot 7 + \$0.45 \cdot 11 = \$40.992$$

A *value* vector maps each food to its caloric content per pound:

$$value = \{\text{hops} : 0, \text{malt} : 960, \text{water} : 0, \text{yeast} : 3.25\}$$

The total calories represented by a pint of beer is the dot-product of *value* with *quantity*:

Dot-product: Total cost or benefit

Suppose D consists of four main ingredients of beer:

$$D = \{\text{malt}, \text{hops}, \text{yeast}, \text{water}\}$$

A *cost* vector maps each food to a price per unit amount:

$$\mathit{cost} = \{\text{hops} : \$2.50/\mathit{ounce}, \text{malt} : \$1.50/\mathit{pound}, \text{water} : \$0.06/\mathit{gallon}, \text{yeast} : \$.45/g\}$$

A *quantity* vector maps each food to an amount (e.g. measured in pounds).
$\mathit{quantity} = \{$hops:6 oz, malt:14 pounds, water:7 gallons, yeast:11 grams$\}$

The total cost is the dot-product of *cost* with *quantity*:

$$\mathit{cost} \cdot \mathit{quantity} = \$2.50 \cdot 6 + \$1.50 \cdot 14 + \$0.006 \cdot 7 + \$0.45 \cdot 11 = \$40.992$$

A *value* vector maps each food to its caloric content per pound:

$$\mathit{value} = \{\text{hops} : 0, \text{malt} : 960, \text{water} : 0, \text{yeast} : 3.25\}$$

The total calories represented by a pint of beer is the dot-product of *value* with *quantity*:

Dot-product: Total cost or benefit

Suppose D consists of four main ingredients of beer:

$$D = \{\text{malt}, \text{hops}, \text{yeast}, \text{water}\}$$

A *cost* vector maps each food to a price per unit amount:

$$cost = \{\text{hops} : \$2.50/ounce, \text{malt} : \$1.50/pound, \text{water} : \$0.06/gallon, \text{yeast} : \$.45/g\}$$

A *quantity* vector maps each food to an amount (e.g. measured in pounds).
$quantity = \{$hops:6 oz, malt:14 pounds, water:7 gallons, yeast:11 grams$\}$

The total cost is the dot-product of *cost* with *quantity*:

$$cost \cdot quantity = \$2.50 \cdot 6 + \$1.50 \cdot 14 + \$0.006 \cdot 7 + \$0.45 \cdot 11 = \$40.992$$

A *value* vector maps each food to its caloric content per pound:

$$value = \{\text{hops} : 0, \text{malt} : 960, \text{water} : 0, \text{yeast} : 3.25\}$$

The total calories represented by a pint of beer is the dot-product of *value* with *quantity*:

Dot-product: Total cost or benefit

Suppose D consists of four main ingredients of beer:

$$D = \{\text{malt}, \text{hops}, \text{yeast}, \text{water}\}$$

A *cost* vector maps each food to a price per unit amount:

$$cost = \{\text{hops} : \$2.50/ounce, \text{malt} : \$1.50/pound, \text{water} : \$0.06/gallon, \text{yeast} : \$.45/g\}$$

A *quantity* vector maps each food to an amount (e.g. measured in pounds).
$quantity = \{$hops:6 oz, malt:14 pounds, water:7 gallons, yeast:11 grams$\}$

The total cost is the dot-product of *cost* with *quantity*:

$$cost \cdot quantity = \$2.50 \cdot 6 + \$1.50 \cdot 14 + \$0.006 \cdot 7 + \$0.45 \cdot 11 = \$40.992$$

A *value* vector maps each food to its caloric content per pound:

$$value = \{\text{hops} : 0, \text{malt} : 960, \text{water} : 0, \text{yeast} : 3.25\}$$

The total calories represented by a pint of beer is the dot-product of *value* with *quantity*:

Dot-product: Linear equations

Example: A sensor node consist of hardware components, e.g.

- CPU
- radio
- temperature sensor
- memory

Battery-driven and remotely located so we care about energy usage.

Suppose we know the power consumption for each hardware component.
Represent it as a D-vector with $D = \{radio, sensor, memory, CPU\}$

$$\textbf{rate} = \text{Vec}(D, \{memory : 0.06\text{W}, radio : 0.06\text{W}, sensor : 0.004\text{W}, CPU : 0.0025\text{W}\})$$

Have a test period during which we know how long each component was working.
Represent as another D vector:

$$\textbf{duration} = \text{Vec}(D, \{memory : 1.0\text{s}, radio : 0.2\text{s}, sensor : 0.5\text{s}, CPU : 1.0\text{s}\})$$

Total energy consumed (in Joules):

$$\textbf{duration} \cdot \textbf{rate}$$

Dot-product: Linear equations

Example: A sensor node consist of hardware components, e.g.

- CPU
- radio
- temperature sensor
- memory

Battery-driven and remotely located so we care about energy usage.

Suppose we know the power consumption for each hardware component.
Represent it as a D-vector with $D = \{radio, sensor, memory, CPU\}$

$$\textbf{rate} = \text{Vec}(D, \{memory : 0.06\text{W}, radio : 0.06\text{W}, sensor : 0.004\text{W}, CPU : 0.0025\text{W}\})$$

Have a test period during which we know how long each component was working.
Represent as another D vector:

$$\textbf{duration} = \text{Vec}(D, \{memory : 1.0\text{s}, radio : 0.2\text{s}, sensor : 0.5\text{s}, CPU : 1.0\text{s}\})$$

Total energy consumed (in Joules):

$$\textbf{duration} \cdot \textbf{rate}$$

Dot-product: Linear equations

Example: A sensor node consist of hardware components, e.g.

- CPU
- radio
- temperature sensor
- memory

Battery-driven and remotely located so we care about energy usage.

Suppose we know the power consumption for each hardware component.
Represent it as a D-vector with $D = \{radio, sensor, memory, CPU\}$

$$\textbf{rate} = \text{Vec}(D, \{memory : 0.06\text{W}, radio : 0.06\text{W}, sensor : 0.004\text{W}, CPU : 0.0025\text{W}\})$$

Have a test period during which we know how long each component was working.
Represent as another D vector:

$$\textbf{duration} = \text{Vec}(D, \{memory : 1.0\text{s}, radio : 0.2\text{s}, sensor : 0.5\text{s}, CPU : 1.0\text{s}\})$$

Total energy consumed (in Joules):

$$\textbf{duration} \cdot \textbf{rate}$$

Dot-product: Linear equations

Example: A sensor node consist of hardware components, e.g.

- CPU
- radio
- temperature sensor
- memory

Battery-driven and remotely located so we care about energy usage.

Suppose we know the power consumption for each hardware component.
Represent it as a D-vector with $D = \{radio, sensor, memory, CPU\}$

$$\mathbf{rate} = \text{Vec}(D, \{memory : 0.06\text{W}, radio : 0.06\text{W}, sensor : 0.004\text{W}, CPU : 0.0025\text{W}\})$$

Have a test period during which we know how long each component was working.
Represent as another D vector:

$$\mathbf{duration} = \text{Vec}(D, \{memory : 1.0\text{s}, radio : 0.2\text{s}, sensor : 0.5\text{s}, CPU : 1.0\text{s}\})$$

Total energy consumed (in Joules):

$$\mathbf{duration} \cdot \mathbf{rate}$$

Dot-product: Linear equations

Example: A sensor node consist of hardware components, e.g.

- CPU
- radio
- temperature sensor
- memory

Battery-driven and remotely located so we care about energy usage.

Suppose we know the power consumption for each hardware component.
Represent it as a D-vector with $D = \{radio, sensor, memory, CPU\}$

$$\textbf{rate} = \text{Vec}(D, \{memory : 0.06\text{W}, radio : 0.06\text{W}, sensor : 0.004\text{W}, CPU : 0.0025\text{W}\})$$

Have a test period during which we know how long each component was working.
Represent as another D vector:

$$\textbf{duration} = \text{Vec}(D, \{memory : 1.0\text{s}, radio : 0.2\text{s}, sensor : 0.5\text{s}, CPU : 1.0\text{s}\})$$

Total energy consumed (in Joules):

Dot-product: Linear equations

Example: A sensor node consist of hardware components, e.g.

- CPU
- radio
- temperature sensor
- memory

Battery-driven and remotely located so we care about energy usage.

Suppose we know the power consumption for each hardware component.
Represent it as a D-vector with $D = \{radio, sensor, memory, CPU\}$

$$\mathbf{rate} = \text{Vec}(D, \{memory : 0.06\text{W}, radio : 0.06\text{W}, sensor : 0.004\text{W}, CPU : 0.0025\text{W}\})$$

Have a test period during which we know how long each component was working.
Represent as another D vector:

$$\mathbf{duration} = \text{Vec}(D, \{memory : 1.0\text{s}, radio : 0.2\text{s}, sensor : 0.5\text{s}, CPU : 1.0\text{s}\})$$

Total energy consumed (in Joules):

$$\mathbf{duration} \cdot \mathbf{rate}$$

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\mathbf{duration}_1 = \{radio : 0.2\text{s}, sensor : 0.5\text{s}, memory : 1.0\text{s}, CPU : 1.0\text{s}\}$$

$$\mathbf{duration}_2 = \{radio : 0\text{s}, sensor : 0.1\text{s}, memory : 0.2\text{s}, CPU : 0.5\text{s}\}$$

$$\mathbf{duration}_3 = \{radio : .4\text{s}, sensor : 0\text{s}, memory : 0.2\text{s}, CPU : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\mathbf{duration}_1 = \{radio : 0.2\text{s}, sensor : 0.5\text{s}, memory : 1.0\text{s}, CPU : 1.0\text{s}\}$$

$$\mathbf{duration}_2 = \{radio : 0\text{s}, sensor : 0.1\text{s}, memory : 0.2\text{s}, CPU : 0.5\text{s}\}$$

$$\mathbf{duration}_3 = \{radio : .4\text{s}, sensor : 0\text{s}, memory : 0.2\text{s}, CPU : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\textbf{duration}_1 = \{radio : 0.2\text{s}, sensor : 0.5\text{s}, memory : 1.0\text{s}, CPU : 1.0\text{s}\}$$

$$\textbf{duration}_2 = \{radio : 0\text{s}, sensor : 0.1\text{s}, memory : 0.2\text{s}, CPU : 0.5\text{s}\}$$

$$\textbf{duration}_3 = \{radio : .4\text{s}, sensor : 0\text{s}, memory : 0.2\text{s}, CPU : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\mathbf{duration}_1 = \{\mathit{radio} : 0.2\text{s}, \mathit{sensor} : 0.5\text{s}, \mathit{memory} : 1.0\text{s}, \mathit{CPU} : 1.0\text{s}\}$$

$$\mathbf{duration}_2 = \{\mathit{radio} : 0\text{s}, \mathit{sensor} : 0.1\text{s}, \mathit{memory} : 0.2\text{s}, \mathit{CPU} : 0.5\text{s}\}$$

$$\mathbf{duration}_3 = \{\mathit{radio} : .4\text{s}, \mathit{sensor} : 0\text{s}, \mathit{memory} : 0.2\text{s}, \mathit{CPU} : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\mathbf{duration}_1 = \{radio : 0.2\text{s}, sensor : 0.5\text{s}, memory : 1.0\text{s}, CPU : 1.0\text{s}\}$$
$$\mathbf{duration}_2 = \{radio : 0\text{s}, sensor : 0.1\text{s}, memory : 0.2\text{s}, CPU : 0.5\text{s}\}$$
$$\mathbf{duration}_3 = \{radio : .4\text{s}, sensor : 0\text{s}, memory : 0.2\text{s}, CPU : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\mathbf{duration}_1 = \{radio : 0.2\text{s}, sensor : 0.5\text{s}, memory : 1.0\text{s}, CPU : 1.0\text{s}\}$$

$$\mathbf{duration}_2 = \{radio : 0\text{s}, sensor : 0.1\text{s}, memory : 0.2\text{s}, CPU : 0.5\text{s}\}$$

$$\mathbf{duration}_3 = \{radio : .4\text{s}, sensor : 0\text{s}, memory : 0.2\text{s}, CPU : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\mathbf{duration}_1 = \{radio : 0.2\text{s}, sensor : 0.5\text{s}, memory : 1.0\text{s}, CPU : 1.0\text{s}\}$$
$$\mathbf{duration}_2 = \{radio : 0\text{s}, sensor : 0.1\text{s}, memory : 0.2\text{s}, CPU : 0.5\text{s}\}$$
$$\mathbf{duration}_3 = \{radio : .4\text{s}, sensor : 0\text{s}, memory : 0.2\text{s}, CPU : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\mathbf{duration}_1 = \{radio : 0.2\text{s}, sensor : 0.5\text{s}, memory : 1.0\text{s}, CPU : 1.0\text{s}\}$$
$$\mathbf{duration}_2 = \{radio : 0\text{s}, sensor : 0.1\text{s}, memory : 0.2\text{s}, CPU : 0.5\text{s}\}$$
$$\mathbf{duration}_3 = \{radio : .4\text{s}, sensor : 0\text{s}, memory : 0.2\text{s}, CPU : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component

Dot-product: Linear equations

A *linear equation* is an equation of the form

$$\mathbf{a} \cdot \mathbf{x} = \beta$$

where **a** is a vector, β is a scalar, and **x** is a vector of variables.

In sensor-node problem, we have linear equations of the form

$$\textbf{duration}_i \cdot \textbf{rate} = \beta_i$$

where **rate** is a vector of variables.

Questions:

- Can we find numbers for the entries of **rate** such that the equations hold?
- If we do, does this guarantee that we have correctly calculated the current draw for each component?

Dot-product: Linear equations

A *linear equation* is an equation of the form

$$\mathbf{a} \cdot \mathbf{x} = \beta$$

where **a** is a vector, β is a scalar, and **x** is a vector of variables.

In sensor-node problem, we have linear equations of the form

$$\mathbf{duration}_i \cdot \mathbf{rate} = \beta_i$$

where **rate** is a vector of variables.

Questions:

- Can we find numbers for the entries of **rate** such that the equations hold?
- If we do, does this guarantee that we have correctly calculated the current draw for each component?

Dot-product: Linear equations

A *linear equation* is an equation of the form

$$\mathbf{a} \cdot \mathbf{x} = \beta$$

where $\mathbf{a}$ is a vector, β is a scalar, and $\mathbf{x}$ is a vector of variables.

In sensor-node problem, we have linear equations of the form

$$\mathbf{duration}_i \cdot \mathbf{rate} = \beta_i$$

where **rate** is a vector of variables.

Questions:

- Can we find numbers for the entries of **rate** such that the equations hold?
- If we do, does this guarantee that we have correctly calculated the current draw for each component?

Dot-product: Linear equations

More general questions:

- Is there an algorithm for solving a *system of linear equations*?

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

- How can we know whether there is only one solution?
- What if our data are slightly inaccurate?

These questions motivate much of what is coming in future weeks.

Dot-product: Linear equations

More general questions:

- Is there an algorithm for solving a *system of linear equations*?

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

- How can we know whether there is only one solution?
- What if our data are slightly inaccurate?

These questions motivate much of what is coming in future weeks.

Dot-product: Linear equations

More general questions:

- Is there an algorithm for solving a *system of linear equations*?

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

- How can we know whether there is only one solution?
- What if our data are slightly inaccurate?

These questions motivate much of what is coming in future weeks.

Dot-product: Linear equations

More general questions:

- Is there an algorithm for solving a *system of linear equations*?

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

- How can we know whether there is only one solution?
- What if our data are slightly inaccurate?

These questions motivate much of what is coming in future weeks.

Dot-product: Linear equations

More general questions:

- Is there an algorithm for solving a *system of linear equations*?

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

- How can we know whether there is only one solution?
- What if our data are slightly inaccurate?

These questions motivate much of what is coming in future weeks.

Dot-product: Measuring similarity: Comparing voting records

Can use dot-product to measure similarity between vectors.

Upcoming lab:

- Represent each senator's voting record as a vector:

$$[+1, +1, 0, -1]$$

$$+1 = \textit{In favor},\ 0 = \textit{not voting},\ \text{-1} = \textit{against}$$

- Dot-product $[+1, +1, 0, -1] \cdot [-1, -1, -1, +1]$
  - very positive if the two senators tend to agree,
  - very negative if two voting records tend to disagree.

Dot-product: Measuring similarity: Comparing voting records

Can use dot-product to measure similarity between vectors.

Upcoming lab:

- Represent each senator's voting record as a vector:

$$[+1, +1, 0, -1]$$

$+1$ = *In favor*, 0 = *not voting*, -1 = *against*

- Dot-product $[+1, +1, 0, -1] \cdot [-1, -1, -1, +1]$
  - very positive if the two senators tend to agree,
  - very negative if two voting records tend to disagree.

Dot-product: Measuring similarity: Comparing voting records

Can use dot-product to measure similarity between vectors.
Upcoming lab:

- Represent each senator's voting record as a vector:

$$[+1, +1, 0, -1]$$

$+1$ = *In favor*, 0 = *not voting*, -1 = *against*

- Dot-product $[+1, +1, 0, -1] \cdot [-1, -1, -1, +1]$
  - very positive if the two senators tend to agree,
  - very negative if two voting records tend to disagree.

Dot-product: Measuring similarity: Comparing voting records

Can use dot-product to measure similarity between vectors.
Upcoming lab:

- Represent each senator's voting record as a vector:

$$[+1, +1, 0, -1]$$

$$+1 = \textit{In favor},\ 0 = \textit{not voting},\ \text{-1} = \textit{against}$$

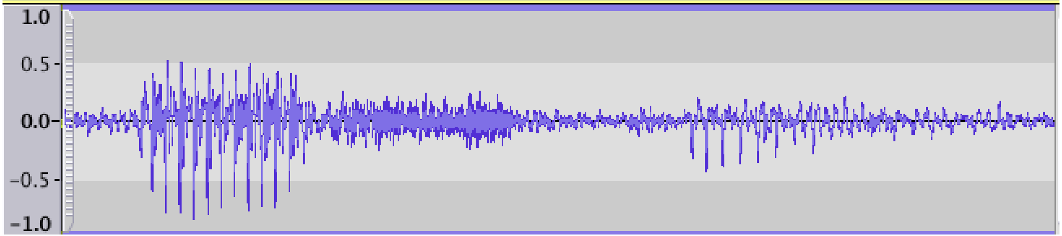
- Dot-product $[+1, +1, 0, -1] \cdot [-1, -1, -1, +1]$
  - very positive if the two senators tend to agree,
  - very negative if two voting records tend to disagree.

Dot-product: Measuring similarity: Comparing audio segments

Want to search for a short audio clip (the *needle*) in a longer audio segment (the *haystack*).

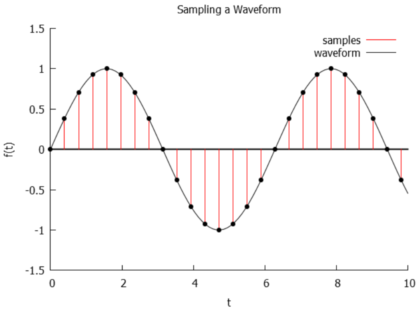
Dot-product: Measuring similarity: Comparing audio segments

Want to search for a short audio clip (the *needle*) in a longer audio segment (the *haystack*).



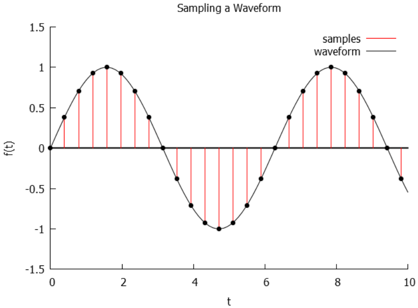
Dot-product: Measuring similarity: Comparing audio segments

Want to search for a short audio clip (the *needle*) in a longer audio segment (the *haystack*).



Dot-product: Measuring similarity: Comparing audio segments

Want to search for a short audio clip (the *needle*) in a longer audio segment (the *haystack*).



- To compare two equal-length sequences of samples, use dot-product: $\sum_{i=1}^{n} \mathbf{u}[i]\ \mathbf{v}[i]$.
- Term i in this sum is positive if $\mathbf{u}[i]$ and $\mathbf{v}[i]$ have the same sign, and negative if they have opposite signs.
- The greater the agreement, the greater the value of the dot-product.

Dot-product: Measuring similarity: Comparing audio segments

Back to needle-in-a-haystack:

If you suspect you know where the needle is...

5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3
										2	7	4	-3	0	-1	-6	4	5	-8	-9		

Dot-product: Measuring similarity: Comparing audio segments

If you don't have any idea where to find the needle, compute lots of dot-products!

5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3
2	7	4	-3	0	-1	-6	4	5	-8	-9												

5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3
	2	7	4	-3	0	-1	-6	4	5	-8	-9											

5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3
		2	7	4	-3	0	-1	-6	4	5	-8	-9										

5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3
			2	7	4	-3	0	-1	-6	4	5	-8	-9									

5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3
				2	7	4	-3	0	-1	-6	4	5	-8	-9								

5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3
					2	7	4	-3	0	-1	-6	4	5	-8	-9							

5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3

Dot-product: Measuring similarity: Comparing audio segments

Seems like a lot of dot-products—-too much computation—but there is a shortcut...

The *Fast Fourier Transform*.

Dot-product: Measuring similarity: Comparing audio segments

Seems like a lot of dot-products—-too much computation—but there is a shortcut...
The *Fast Fourier Transform*.

Dot-product: Vectors over $GF(2)$

Consider the dot-product of 11111 and 10101:

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

Dot-product: Vectors over $GF(2)$

Consider the dot-product of 11111 and 10101:

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

Dot-product: Simple authentication scheme

- Usual way of logging into a computer with a password is subject to hacking by an eavesdropper.
- **Alternative:** Challenge-response system
  - Computer asks a question about the password.
  - Human sends the answer.
  - Repeat a few times before human is considered authenticated.

  Potentially safe against an eavesdropper since probably next time will involve different questions.
- Simple challenge-response scheme based on dot-product of vectors over $GF(2)$:
  - Password is an n-vector $\hat{\mathbf{x}}$.
  - Computer sends random n-vector $\mathbf{a}$
  - Human sends back $\mathbf{a} \cdot \hat{\mathbf{x}}$.

Dot-product: Simple authentication scheme

- Usual way of logging into a computer with a password is subject to hacking by an eavesdropper.
- **Alternative:** Challenge-response system
  - Computer asks a question about the password.
  - Human sends the answer.
  - Repeat a few times before human is considered authenticated.

  Potentially safe against an eavesdropper since probably next time will involve different questions.
- Simple challenge-response scheme based on dot-product of vectors over $GF(2)$:
  - Password is an n-vector $\hat{\mathbf{x}}$.
  - Computer sends random n-vector $\mathbf{a}$
  - Human sends back $\mathbf{a} \cdot \hat{\mathbf{x}}$.

Dot-product: Simple authentication scheme

- Usual way of logging into a computer with a password is subject to hacking by an eavesdropper.
- **Alternative:** Challenge-response system
  - Computer asks a question about the password.
  - Human sends the answer.
  - Repeat a few times before human is considered authenticated.

  Potentially safe against an eavesdropper since probably next time will involve different questions.
- Simple challenge-response scheme based on dot-product of vectors over $GF(2)$:
  - Password is an n-vector $\hat{\mathbf{x}}$.
  - Computer sends random n-vector $\mathbf{a}$
  - Human sends back $\mathbf{a} \cdot \hat{\mathbf{x}}$.

Dot-product: Simple authentication scheme

- Usual way of logging into a computer with a password is subject to hacking by an eavesdropper.
- **Alternative:** Challenge-response system
  - Computer asks a question about the password.
  - Human sends the answer.
  - Repeat a few times before human is considered authenticated.

  Potentially safe against an eavesdropper since probably next time will involve different questions.
- Simple challenge-response scheme based on dot-product of vectors over $GF(2)$:
  - Password is an n-vector $\hat{\mathbf{x}}$.
  - Computer sends random n-vector $\mathbf{a}$
  - Human sends back $\mathbf{a} \cdot \hat{\mathbf{x}}$.

Dot-product: Simple authentication scheme

- Usual way of logging into a computer with a password is subject to hacking by an eavesdropper.
- **Alternative:** Challenge-response system
  - Computer asks a question about the password.
  - Human sends the answer.
  - Repeat a few times before human is considered authenticated.

  Potentially safe against an eavesdropper since probably next time will involve different questions.
- Simple challenge-response scheme based on dot-product of vectors over $GF(2)$:
  - Password is an n-vector $\hat{\mathbf{x}}$.
  - Computer sends random n-vector $\mathbf{a}$
  - Human sends back $\mathbf{a} \cdot \hat{\mathbf{x}}$.

Dot-product: Simple authentication scheme

- Usual way of logging into a computer with a password is subject to hacking by an eavesdropper.
- **Alternative:** Challenge-response system
  - Computer asks a question about the password.
  - Human sends the answer.
  - Repeat a few times before human is considered authenticated.

  Potentially safe against an eavesdropper since probably next time will involve different questions.
- Simple challenge-response scheme based on dot-product of vectors over $GF(2)$:
  - Password is an n-vector $\hat{\mathbf{x}}$.
  - Computer sends random n-vector $\mathbf{a}$
  - Human sends back $\mathbf{a} \cdot \hat{\mathbf{x}}$.

Dot-product: Simple authentication scheme

- Usual way of logging into a computer with a password is subject to hacking by an eavesdropper.
- **Alternative:** Challenge-response system
  - Computer asks a question about the password.
  - Human sends the answer.
  - Repeat a few times before human is considered authenticated.

  Potentially safe against an eavesdropper since probably next time will involve different questions.
- Simple challenge-response scheme based on dot-product of vectors over $GF(2)$:
  - Password is an n-vector $\hat{\mathbf{x}}$.
  - Computer sends random n-vector $\mathbf{a}$
  - Human sends back $\mathbf{a} \cdot \hat{\mathbf{x}}$.

Dot-product: Simple authentication scheme

- **Example:** Password is $\hat{\mathbf{x}} = 10111$.
- Computer sends $\mathbf{a}_1 = 01011$ to Human.
- Human computes dot-product $\mathbf{a}_1 \cdot \hat{\mathbf{x}}$:

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

and sends $\beta_1 = 0$ to Computer.

Dot-product: Attacking simple authentication scheme

How can an eavesdropper Eve cheat?

- She observes a sequence of challenge vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ and the corresponding response bits $\beta_1, \beta_2, \ldots, \beta_m$.
- Can she find the password?

She knows the password must satisfy the linear equations

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

Questions:

- How many solutions?
- How to compute them?

Answers will come later.

Dot-product: Attacking simple authentication scheme

How can an eavesdropper Eve cheat?

- She observes a sequence of challenge vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ and the corresponding response bits $\beta_1, \beta_2, \ldots, \beta_m$.
- Can she find the password?

She knows the password must satisfy the linear equations

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

Questions:

- How many solutions?
- How to compute them?

Answers will come later.

Dot-product: Attacking simple authentication scheme

How can an eavesdropper Eve cheat?

- She observes a sequence of challenge vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ and the corresponding response bits $\beta_1, \beta_2, \ldots, \beta_m$.
- Can she find the password?

She knows the password must satisfy the linear equations

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

Questions:

- How many solutions?
- How to compute them?

Answers will come later.

Dot-product: Attacking simple authentication scheme

How can an eavesdropper Eve cheat?

- She observes a sequence of challenge vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ and the corresponding response bits $\beta_1, \beta_2, \ldots, \beta_m$.
- Can she find the password?

She knows the password must satisfy the linear equations

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

Questions:

- How many solutions?
- How to compute them?

Answers will come later.

Dot-product: Attacking simple authentication scheme

How can an eavesdropper Eve cheat?

- She observes a sequence of challenge vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ and the corresponding response bits $\beta_1, \beta_2, \ldots, \beta_m$.
- Can she find the password?

She knows the password must satisfy the linear equations

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

Questions:

- How many solutions?
- How to compute them?

Answers will come later.

Dot-product: Attacking simple authentication scheme

How can an eavesdropper Eve cheat?

- She observes a sequence of challenge vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ and the corresponding response bits $\beta_1, \beta_2, \ldots, \beta_m$.
- Can she find the password?

She knows the password must satisfy the linear equations

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

Questions:

- How many solutions?
- How to compute them?

Answers will come later.

Dot-product: Attacking simple authentication scheme

Another way to cheat?

Can Eve derive a challenge for which she knows the response?

Algebraic properties of dot-product:

- **Commutativity:** $\mathbf{v} \cdot \mathbf{x} = \mathbf{x} \cdot \mathbf{v}$
- **Homogeneity:** $(\alpha\, \mathbf{u}) \cdot \mathbf{v} = \alpha\, (\mathbf{u} \cdot \mathbf{v})$
- **Distributive law:** $(\mathbf{v}_1 + \mathbf{v}_2) \cdot \mathbf{x} = \mathbf{v}_1 \cdot \mathbf{x} + \mathbf{v}_2 \cdot \mathbf{x}$

Example: Eve observes

- challenge 01011, response 0
- challenge 11110, response 1

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

For challenge $01011 + 11110$, Eve can derive right response.

Dot-product: Attacking simple authentication scheme

Another way to cheat?

Can Eve derive a challenge for which she knows the response?

Algebraic properties of dot-product:

- **Commutativity:** $\mathbf{v} \cdot \mathbf{x} = \mathbf{x} \cdot \mathbf{v}$
- **Homogeneity:** $(\alpha\, \mathbf{u}) \cdot \mathbf{v} = \alpha\, (\mathbf{u} \cdot \mathbf{v})$
- **Distributive law:** $(\mathbf{v}_1 + \mathbf{v}_2) \cdot \mathbf{x} = \mathbf{v}_1 \cdot \mathbf{x} + \mathbf{v}_2 \cdot \mathbf{x}$

Example: Eve observes

- challenge 01011, response 0
- challenge 11110, response 1

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

For challenge $01011 + 11110$, Eve can derive right response.

Dot-product: Attacking simple authentication scheme

Another way to cheat?

Can Eve derive a challenge for which she knows the response?

Algebraic properties of dot-product:

- **Commutativity:** $\mathbf{v} \cdot \mathbf{x} = \mathbf{x} \cdot \mathbf{v}$
- **Homogeneity:** $(\alpha\, \mathbf{u}) \cdot \mathbf{v} = \alpha\, (\mathbf{u} \cdot \mathbf{v})$
- **Distributive law:** $(\mathbf{v}_1 + \mathbf{v}_2) \cdot \mathbf{x} = \mathbf{v}_1 \cdot \mathbf{x} + \mathbf{v}_2 \cdot \mathbf{x}$

Example: Eve observes

- challenge 01011, response 0
- challenge 11110, response 1

$$\begin{aligned}(01011 + 11110) \cdot \mathbf{x} &= 01011 \cdot \mathbf{x} + 11110 \cdot \mathbf{x} \\ &= 0 + 1 \\ &= 1\end{aligned}$$

For challenge $01011 + 11110$, Eve can derive right response.

Dot-product: Attacking simple authentication scheme

Another way to cheat?

Can Eve derive a challenge for which she knows the response?

Algebraic properties of dot-product:

- **Commutativity:** $\mathbf{v} \cdot \mathbf{x} = \mathbf{x} \cdot \mathbf{v}$
- **Homogeneity:** $(\alpha\, \mathbf{u}) \cdot \mathbf{v} = \alpha\, (\mathbf{u} \cdot \mathbf{v})$
- **Distributive law:** $(\mathbf{v}_1 + \mathbf{v}_2) \cdot \mathbf{x} = \mathbf{v}_1 \cdot \mathbf{x} + \mathbf{v}_2 \cdot \mathbf{x}$

Example: Eve observes

- challenge 01011, response 0
- challenge 11110, response 1

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

For challenge $01011 + 11110$, Eve can derive right response.

Dot-product: Attacking simple authentication scheme

Another way to cheat?

Can Eve derive a challenge for which she knows the response?

Algebraic properties of dot-product:

- **Commutativity:** $\mathbf{v} \cdot \mathbf{x} = \mathbf{x} \cdot \mathbf{v}$
- **Homogeneity:** $(\alpha\, \mathbf{u}) \cdot \mathbf{v} = \alpha\, (\mathbf{u} \cdot \mathbf{v})$
- **Distributive law:** $(\mathbf{v}_1 + \mathbf{v}_2) \cdot \mathbf{x} = \mathbf{v}_1 \cdot \mathbf{x} + \mathbf{v}_2 \cdot \mathbf{x}$

Example: Eve observes

- challenge 01011, response 0
- challenge 11110, response 1

$$
\begin{aligned}
(01011 + 11110) \cdot \mathbf{x} &= 01011 \cdot \mathbf{x} + 11110 \cdot \mathbf{x} \\
&= 0 + 1 \\
&= 1
\end{aligned}
$$

For challenge $01011 + 11110$, Eve can derive right response.

Dot-product: Attacking simple authentication scheme

Another way to cheat?

Can Eve derive a challenge for which she knows the response?

Algebraic properties of dot-product:

- **Commutativity:** $\mathbf{v} \cdot \mathbf{x} = \mathbf{x} \cdot \mathbf{v}$
- **Homogeneity:** $(\alpha\, \mathbf{u}) \cdot \mathbf{v} = \alpha\, (\mathbf{u} \cdot \mathbf{v})$
- **Distributive law:** $(\mathbf{v}_1 + \mathbf{v}_2) \cdot \mathbf{x} = \mathbf{v}_1 \cdot \mathbf{x} + \mathbf{v}_2 \cdot \mathbf{x}$

Example: Eve observes

- challenge 01011, response 0
- challenge 11110, response 1

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

For challenge $01011 + 11110$, Eve can derive right response.

Dot-product: Attacking simple authentication scheme

More generally, if a vector satisfies equations

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

then what other equations does the vector satisfy?
Answer will come later.

Dictionary-based representations of vectors

- A vector is a function from some domain D to a field
- Can represent such a function in Python by a *dictionary*.
- It's convenient to define a Python class `Vec` with two instance variables (fields):
  - `f`, the function, represented by a Python dictionary, and
  - `D`, the domain of the function, represented by a Python set.
- We adopt the convention in which entries with value zero may be omitted from the dictionary `f`

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Dictionary-based representations of vectors

- A vector is a function from some domain D to a field
- Can represent such a function in Python by a *dictionary*.
- It's convenient to define a Python class `Vec` with two instance variables (fields):
  - `f`, the function, represented by a Python dictionary, and
  - `D`, the domain of the function, represented by a Python set.
- We adopt the convention in which entries with value zero may be omitted from the dictionary `f`

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Dictionary-based representations of vectors

- A vector is a function from some domain D to a field
- Can represent such a function in Python by a *dictionary*.
- It's convenient to define a Python class `Vec` with two instance variables (fields):
  - `f`, the function, represented by a Python dictionary, and
  - `D`, the domain of the function, represented by a Python set.
- We adopt the convention in which entries with value zero may be omitted from the dictionary `f`

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Dictionary-based representations of vectors

- A vector is a function from some domain D to a field
- Can represent such a function in Python by a *dictionary*.
- It's convenient to define a Python class `Vec` with two instance variables (fields):
  - `f`, the function, represented by a Python dictionary, and
  - `D`, the domain of the function, represented by a Python set.
- We adopt the convention in which entries with value zero may be omitted from the dictionary `f`

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Dictionary-based representations of vectors

- A vector is a function from some domain D to a field
- Can represent such a function in Python by a *dictionary*.
- It's convenient to define a Python class `Vec` with two instance variables (fields):
  - `f`, the function, represented by a Python dictionary, and
  - `D`, the domain of the function, represented by a Python set.
- We adopt the convention in which entries with value zero may be omitted from the dictionary `f`

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Dictionary-based representations of vectors

- A vector is a function from some domain D to a field
- Can represent such a function in Python by a *dictionary*.
- It's convenient to define a Python class `Vec` with two instance variables (fields):
  - `f`, the function, represented by a Python dictionary, and
  - `D`, the domain of the function, represented by a Python set.
- We adopt the convention in which entries with value zero may be omitted from the dictionary `f`

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Dictionary-based representations of vectors

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Can then create an instance:

```
>>> Vec({'A','B','C'}, {'A':1})
```

- First argument is assigned to `D` field.
- Second argument is assigned to `f` field.

Dictionary-based representations of vectors

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Can then create an instance:

```
>>> Vec({'A','B','C'}, {'A':1})
```

- First argument is assigned to D field.
- Second argument is assigned to f field.

Dictionary-based representations of vectors

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Can then create an instance:

```
>>> Vec({'A','B','C'}, {'A':1})
```

- First argument is assigned to `D` field.
- Second argument is assigned to `f` field.

Dictionary-based representations of vectors

Can assign an instance to a variable:

```
>>> v=Vec({'A','B','C'}, {'A':1.})
```

and subsequently access the two fields of `v`, e.g.:

```
>>> for d in v.D:
...  if d in v.f:
...   print(v.f[d])
...
1.0
```

Dictionary-based representations of vectors

Quiz: Write a procedure `zero_vec(D)` with the following spec:

- *input:* a set `D`
- *output:* an instance of `Vec` representing a `D`-vector all of whose entries have value zero

Dictionary-based representations of vectors

Quiz: Write a procedure `zero_vec(D)` with the following spec:

- *input:* a set `D`
- *output:* an instance of `Vec` representing a `D`-vector all of whose entries have value zero

Answer:

```
def zero_vec(D): return Vec(D, {})
```

Dictionary-based representations of vectors

Quiz: Write a procedure `zero_vec(D)` with the following spec:

- *input:* a set `D`
- *output:* an instance of `Vec` representing a `D`-vector all of whose entries have value zero

Answer:

```
def zero_vec(D): return Vec(D, {})
```

or

```
def zero_vec(D): return Vec(D, {d:0 for d in D})
```

Dictionary-based representations of vectors: Setter and getter

Setter:

```
def setitem(v, d, val):  v.f[d] = val
```

Dictionary-based representations of vectors: Setter and getter

Setter:

```
def setitem(v, d, val):  v.f[d] = val
```

- Second argument should be member of `v.D`.
- Third argument should be an element of the field.

Dictionary-based representations of vectors: Setter and getter

Setter:

```
def setitem(v, d, val):  v.f[d] = val
```

- Second argument should be member of `v.D`.
- Third argument should be an element of the field.

Example:

```
>>> setitem(v, 'B', 2.)
```

Dictionary-based representations of vectors: Setter and getter

Quiz: Write a procedure `getitem(v, d)` with the following spec:

- *input:* an instance `v` of `Vec`, and an element `d` of the set `v.D`
- *output:* the value of entry `d` of `v`

Dictionary-based representations of vectors: Setter and getter

Quiz: Write a procedure `getitem(v, d)` with the following spec:

- *input:* an instance `v` of `Vec`, and an element `d` of the set `v.D`
- *output:* the value of entry `d` of `v`

Answer:

```
def getitem(v,d): return v.f[d] if d in v.f else 0
```

Dictionary-based representations of vectors: Setter and getter

Quiz: Write a procedure `getitem(v, d)` with the following spec:

- *input:* an instance `v` of `Vec`, and an element `d` of the set `v.D`
- *output:* the value of entry `d` of `v`

Answer:

```
def getitem(v,d): return v.f[d] if d in v.f else 0
```

Another answer:

```
def getitem(v,d):
    if d in v.f:
        return v.f[d]
    else:
        return 0
```

Dictionary-based representations of vectors: Setter and getter

Quiz: Write a procedure `getitem(v, d)` with the following spec:

- *input:* an instance `v` of `Vec`, and an element `d` of the set `v.D`
- *output:* the value of entry `d` of `v`

Answer:

```
def getitem(v,d): return v.f[d] if d in v.f else 0
```

Another answer:

```
def getitem(v,d):
    if d in v.f:
        return v.f[d]
    else:
        return 0
```

Why is `def getitem(v,d):  return v.f[d]` not enough?

Dictionary-based representations of vectors: Setter and getter

Quiz: Write a procedure `getitem(v, d)` with the following spec:

- *input:* an instance `v` of `Vec`, and an element `d` of the set `v.D`
- *output:* the value of entry `d` of `v`

Answer:

```
def getitem(v,d): return v.f[d] if d in v.f else 0
```

Another answer:

```
def getitem(v,d):
    if d in v.f:
        return v.f[d]
    else:
        return 0
```

Why is `def getitem(v,d):  return v.f[d]` not enough?
Sparsity convention

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
       labels, function):
        self.D = labels
        self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

You will code this class starting from a stencil. (See quizzes for help.)

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
        labels, function):
        self.D = labels
        self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

You will code this class starting from a stencil. (See quizzes for help.)

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
        labels, function):
         self.D = labels
         self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

You will code this class starting from a stencil. (See quizzes for help.)

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
       labels, function):
        self.D = labels
        self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

You will code this class starting from a stencil. (See quizzes for help.)

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
        labels, function):
         self.D = labels
         self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

You will code this class starting from a stencil. (See quizzes for help.)

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
       labels, function):
        self.D = labels
        self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

You will code this class starting from a stencil. (See quizzes for help.)

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
          labels, function):
        self.D = labels
        self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

You will code this class starting from a stencil. (See quizzes for help.)

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
        labels, function):
         self.D = labels
         self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

You will code this class starting from a stencil. (See quizzes for help.)

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

Using Vec

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

instead of

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

In short: Use the operators [ ], +, *, -, / when working with Vecs

Using Vec

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

instead of

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

In short: Use the operators [ ], +, *, -, / when working with Vecs

Using Vec

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

instead of

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

In short: Use the operators [ ], +, *, -, / when working with Vecs

Using Vec

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

instead of

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

In short: Use the operators [ ], +, *, -, / when working with Vecs

Using Vec

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

instead of

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

In short: Use the operators [ ], +, *, -, / when working with Vecs

Using Vec

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

instead of

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

In short: Use the operators [ ], +, *, -, / when working with Vecs

Using Vec

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

instead of

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

In short: Use the operators [ ], +, *, -, / when working with Vecs

Assertions in `Vec`

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a *documentation string*, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Assertions in `Vec`

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a *documentation string*, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Assertions in Vec

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a *documentation string*, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Assertions in `Vec`

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a *documentation string*, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Assertions in Vec

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a *documentation string*, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Assertions in `Vec`

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a *documentation string*, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Assertions in Vec

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a *documentation string*, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Assertions in `Vec`

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a *documentation string*, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Testing `Vec` with doctests

We have provided tests in the docstrings:

```
def getitem(v,k):
  """
  Return the value of entry d in v.
  >>> v = Vec({'a','b','c', 'd'},
              {'a':2,'c':1,'d':3})
  >>> v['d']
  3
  >>> v['b']
  0
  """
  pass
```

Tests show interactions with Python assuming correct implementation.

You can copy from the file and paste into your Python session.

You can also run all the tests at once from the console (outside the Python interpreter) using the following command:

```
python3 -m doctest vec.py
```

This will run the tests given in `vec.py` and will print messages about any discrepancies that arise. If your code passes the tests, nothing will be printed.

Testing `Vec` with doctests

We have provided tests in the docstrings:

```
def getitem(v,k):
  """
  Return the value of entry d in v.
  >>> v = Vec({'a','b','c', 'd'},
              {'a':2,'c':1,'d':3})
  >>> v['d']
  3
  >>> v['b']
  0
  """
  pass
```

Tests show interactions with Python assuming correct implementation.

You can copy from the file and paste into your Python session.

You can also run all the tests at once from the console (outside the Python interpreter) using the following command:

```
python3 -m doctest vec.py
```

This will run the tests given in `vec.py` and will print messages about any discrepancies that arise. If your code passes the tests, nothing will be printed.

Testing Vec with doctests

We have provided tests in the docstrings:

```
def getitem(v,k):
    """
    Return the value of entry d in v.
    >>> v = Vec({'a','b','c', 'd'},
                {'a':2,'c':1,'d':3})
    >>> v['d']
    3
    >>> v['b']
    0
    """
    pass
```

Tests show interactions with Python assuming correct implementation.

You can copy from the file and paste into your Python session.

You can also run all the tests at once from the console (outside the Python interpreter) using the following command:

```
python3 -m doctest vec.py
```

This will run the tests given in `vec.py` and will print messages about any discrepancies that arise. If your code passes the tests, nothing will be printed.

Testing Vec with doctests

We have provided tests in the docstrings:

```
def getitem(v,k):
    """
    Return the value of entry d in v.
    >>> v = Vec({'a','b','c', 'd'},
                {'a':2,'c':1,'d':3})
    >>> v['d']
    3
    >>> v['b']
    0
    """
    pass
```

Tests show interactions with Python assuming correct implementation.

You can copy from the file and paste into your Python session.

You can also run all the tests at once from the console (outside the Python interpreter) using the following command:

```
python3 -m doctest vec.py
```

This will run the tests given in `vec.py` and will print messages about any discrepancies that arise. If your code passes the tests, nothing will be printed.

Testing Vec with doctests

We have provided tests in the docstrings:

```
def getitem(v,k):
    """
    Return the value of entry d in v.
    >>> v = Vec({'a','b','c', 'd'},
                {'a':2,'c':1,'d':3})
    >>> v['d']
    3
    >>> v['b']
    0
    """
    pass
```

Tests show interactions with Python assuming correct implementation.

You can copy from the file and paste into your Python session.

You can also run all the tests at once from the console (outside the Python interpreter) using the following command:

```
python3 -m doctest vec.py
```

This will run the tests given in `vec.py` and will print messages about any discrepancies that arise. If your code passes the tests, nothing will be printed.

Testing `Vec` with doctests

We have provided tests in the docstrings:

```
def getitem(v,k):
    """
    Return the value of entry d in v.
    >>> v = Vec({'a','b','c', 'd'},
                {'a':2,'c':1,'d':3})
    >>> v['d']
    3
    >>> v['b']
    0
    """
    pass
```

Tests show interactions with Python assuming correct implementation.

You can copy from the file and paste into your Python session.

You can also run all the tests at once from the console (outside the Python interpreter) using the following command:

```
python3 -m doctest vec.py
```

This will run the tests given in `vec.py` and will print messages about any discrepancies that arise. If your code passes the tests, nothing will be printed.

list2vec

The Vec class is useful for representing vectors but is not the only useful representation.

We 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 easy to convert between list-based and dictionary-based representations.

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

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

list2vec

The Vec class is useful for representing vectors but is not the only useful representation.

We 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 easy to convert between list-based and dictionary-based representations.

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

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

`list2vec`

The `Vec` class is useful for representing vectors but is not the only useful representation.

We 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 easy to convert between list-based and dictionary-based representations.

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

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

`list2vec`

The `Vec` class is useful for representing vectors but is not the only useful representation.

We 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 easy to convert between list-based and dictionary-based representations.

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

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

list2vec

The `Vec` class is useful for representing vectors but is not the only useful representation.

We 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 easy to convert between list-based and dictionary-based representations.

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

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

`list2vec`

The `Vec` class is useful for representing vectors but is not the only useful representation.

We 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 easy to convert between list-based and dictionary-based representations.

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

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

`list2vec`

The `Vec` class is useful for representing vectors but is not the only useful representation.

We sometimes represent vectors by lists.

A list L can be viewed as a function from $\{0, 1, 2, \ldots, \mathrm{len}(L) - 1\}$, so it is easy to convert between list-based and dictionary-based representations.

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

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

The `vecutil` module

The procedures `zero_vec(D)` and `list2vec(L)` are defined in the file `vecutil.py`, which we provide.

Solving a triangular system of linear equations

How to find solution to this linear system?

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

Write $\mathbf{x} = [x_1, x_2, x_3, x_4]$.
System becomes

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

Solving a triangular system of linear equations

How to find solution to this linear system?

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

Write $\mathbf{x} = [x_1, x_2, x_3, x_4]$.
System becomes

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

Solving a triangular system of linear equations: Backward substitution

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

Solution strategy:

- Solve for x_4 using fourth equation.
- Plug value for x_4 into third equations and solve for x_3.
- Plug values for x_4 and x_3 into second equation and solve for x_2.
- Plug values for x_4, x_3, x_2 into first equation and solve for x_1.

Solving a triangular system of linear equations: Backward substitution

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

Solution strategy:

- Solve for x_4 using fourth equation.
- Plug value for x_4 into third equations and solve for x_3.
- Plug values for x_4 and x_3 into second equation and solve for x_2.
- Plug values for x_4, x_3, x_2 into first equation and solve for x_1.

Solving a triangular system of linear equations: Backward substitution

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

Solution strategy:

- Solve for x_4 using fourth equation.
- Plug value for x_4 into third equations and solve for x_3.
- Plug values for x_4 and x_3 into second equation and solve for x_2.
- Plug values for x_4, x_3, x_2 into first equation and solve for x_1.

Solving a triangular system of linear equations: Backward substitution

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

Solution strategy:

- Solve for x_4 using fourth equation.
- Plug value for x_4 into third equations and solve for x_3.
- Plug values for x_4 and x_3 into second equation and solve for x_2.
- Plug values for x_4, x_3, x_2 into first equation and solve for x_1.

Solving a triangular system of linear equations: Backward substitution

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

Solution strategy:

- Solve for x_4 using fourth equation.
- Plug value for x_4 into third equations and solve for x_3.
- Plug values for x_4 and x_3 into second equation and solve for x_2.
- Plug values for x_4, x_3, x_2 into first equation and solve for x_1.

Solving a triangular system of linear equations: Backward substitution

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

Solving a triangular system of linear equations: Backward substitution

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

$$2x_4 = 6$$

Solving a triangular system of linear equations: Backward substitution

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

$$\begin{array}{lrcl}
 & 2x_4 & = & 6 \\
\text{so} & x_4 & = & 6/2 = 3
\end{array}$$

Solving a triangular system of linear equations: Backward substitution

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

$$
\begin{array}{lrcl}
 & 2x_4 & = & 6 \\
\text{so} & x_4 & = & 6/2 = 3 \\
 & & & \\
 & 1x_3 & = & -4 - 5x_4 = -4 - 5(3) = -19
\end{array}
$$

Solving a triangular system of linear equations: Backward substitution

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

$$\begin{array}{lrcl}
 & 2x_4 & = & 6 \\
\text{so} & x_4 & = & 6/2 = 3 \\
 & & & \\
 & 1x_3 & = & -4 - 5x_4 = -4 - 5(3) = -19 \\
\text{so} & x_3 & = & -19/1 = -19
\end{array}$$

Solving a triangular system of linear equations: Backward substitution

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

$$\begin{array}{lrcl} & 2x_4 & = & 6 \\ \text{so} & x_4 & = & 6/2 = 3 \\ \\ & 1x_3 & = & -4 - 5x_4 = -4 - 5(3) = -19 \\ \text{so} & x_3 & = & -19/1 = -19 \\ \\ & 3x_2 & = & 3 - 3x_3 - 2x_4 = 3 - 2(3) - 3(-19) = 54 \end{array}$$

Solving a triangular system of linear equations: Backward substitution

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

$$
\begin{array}{lrcl}
 & 2x_4 & = & 6 \\
\text{so} & x_4 & = & 6/2 = 3 \\
 & & & \\
 & 1x_3 & = & -4 - 5x_4 = -4 - 5(3) = -19 \\
\text{so} & x_3 & = & -19/1 = -19 \\
 & & & \\
 & 3x_2 & = & 3 - 3x_3 - 2x_4 = 3 - 2(3) - 3(-19) = 54 \\
\text{so} & x_2 & = & 54/3 = 18
\end{array}
$$

Solving a triangular system of linear equations: Backward substitution

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

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

Solving a triangular system of linear equations: Backward substitution

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

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

Solving a triangular system of linear equations: Backward substitution

Quiz: Solve the following system by hand:

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

Solving a triangular system of linear equations: Backward substitution

Quiz: Solve the following system by hand:

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

Answer:

$$x_3 = 15/5 = 3$$

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

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

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector x to zero vector.
- Procedure will populate x entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0, 3, 3, 2] \cdot [x_1, x_2, x_3, x_4] = 3$
  - So far, vector $\mathrm{x} = [x_1, x_2, x_3, x_4] = [0, 0, -19, 3]$.
  - $x_2 := 3 - ([0, 3, 3, 2] \cdot \mathrm{x})$

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

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector x to zero vector.
- Procedure will populate x entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0,3,3,2] \cdot [x_1,x_2,x_3,x_4] = 3$
  - So far, vector $\mathrm{x} = [x_1,x_2,x_3,x_4] = [0,0,-19,3]$.
  - $x_2 := 3 - ([0,3,3,2] \cdot \mathrm{x})$

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

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector x to zero vector.
- Procedure will populate x entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0, 3, 3, 2] \cdot [x_1, x_2, x_3, x_4] = 3$
  - So far, vector $\mathrm{x} = [x_1, x_2, x_3, x_4] = [0, 0, -19, 3]$.
  - $x_2 := 3 - ([0, 3, 3, 2] \cdot \mathrm{x})$

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

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector x to zero vector.
- Procedure will populate x entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0, 3, 3, 2] \cdot [x_1, x_2, x_3, x_4] = 3$
  - So far, vector $\mathrm{x} = [x_1, x_2, x_3, x_4] = [0, 0, -19, 3]$.
  - $x_2 := 3 - ([0, 3, 3, 2] \cdot \mathrm{x})$

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

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector x to zero vector.
- Procedure will populate x entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0, 3, 3, 2] \cdot [x_1, x_2, x_3, x_4] = 3$
  - So far, vector $\mathrm{x} = [x_1, x_2, x_3, x_4] = [0, 0, -19, 3]$.
  - $x_2 := 3 - ([0, 3, 3, 2] \cdot \mathrm{x})$

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

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector `x` to zero vector.
- Procedure will populate `x` entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0, 3, 3, 2] \cdot [x_1, x_2, x_3, x_4] = 3$
  - So far, vector $\mathtt{x} = [x_1, x_2, x_3, x_4] = [0, 0, -19, 3]$.
  - $x_2 := 3 - ([0, 3, 3, 2] \cdot \mathtt{x})$

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

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector $\mathtt{x}$ to zero vector.
- Procedure will populate $\mathtt{x}$ entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0, 3, 3, 2] \cdot [x_1, x_2, x_3, x_4] = 3$
  - So far, vector $\mathtt{x} = [x_1, x_2, x_3, x_4] = [0, 0, -19, 3]$.
  - $x_2 := 3 - ([0, 3, 3, 2] \cdot \mathtt{x})$

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

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector x to zero vector.
- Procedure will populate x entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0, 3, 3, 2] \cdot [x_1, x_2, x_3, x_4] = 3$
  - So far, vector $\mathtt{x} = [x_1, x_2, x_3, x_4] = [0, 0, -19, 3]$.
  - $x_2 := 3 - ([0, 3, 3, 2] \cdot \mathtt{x})$

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

Solving a triangular system of linear equations: Backward substitution

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

Observations:

- If `rowlist[i][i]` is zero, procedure will raise `ZeroDivisionError`.
- If this never happens, solution found is the *only* solution to the system.

Solving a triangular system of linear equations: Backward substitution

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

Observations:

- If `rowlist[i][i]` is zero, procedure will raise `ZeroDivisionError`.
- If this never happens, solution found is the *only* solution to the system.

Solving a triangular system of linear equations: Backward substitution

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

Observations:

- If `rowlist[i][i]` is zero, procedure will raise `ZeroDivisionError`.
- If this never happens, solution found is the *only* solution to the system.

Solving a triangular system of linear equations: Backward substitution

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

Our code only works when vectors in `rowlist` have domain $D = \{0, 1, 2, \ldots, n-1\}$.

For arbitrary domains, need to specify an ordering for which system is "triangular":

```
def triangular_solve(rowlist, label_list, b):
    x = zero_vec(set(label_list))
    for r in reversed(range(len(rowlist))):
        c = label_list[r]
        x[c] = (b[r] - x*rowlist[r])/rowlist[r][c]
    return x
```

Solving a triangular system of linear equations: Backward substitution

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

Our code only works when vectors in `rowlist` have domain $D = \{0, 1, 2, \ldots, n-1\}$.

For arbitrary domains, need to specify an ordering for which system is "triangular":

```
def triangular_solve(rowlist, label_list, b):
    x = zero_vec(set(label_list))
    for r in reversed(range(len(rowlist))):
        c = label_list[r]
        x[c] = (b[r] - x*rowlist[r])/rowlist[r][c]
    return x
```

Solving a triangular system of linear equations: Backward substitution

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

Our code only works when vectors in `rowlist` have domain $D = \{0, 1, 2, \ldots, n-1\}$.

For arbitrary domains, need to specify an ordering for which system is "triangular":

```
def triangular_solve(rowlist, label_list, b):
    x = zero_vec(set(label_list))
    for r in reversed(range(len(rowlist))):
        c = label_list[r]
        x[c] = (b[r] - x*rowlist[r])/rowlist[r][c]
    return x
```

Solving a triangular system of linear equations: Backward substitution

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

Our code only works when vectors in `rowlist` have domain $D = \{0, 1, 2, \ldots, n-1\}$.

For arbitrary domains, need to specify an ordering for which system is "triangular":

```
def triangular_solve(rowlist, label_list, b):
    x = zero_vec(set(label_list))
    for r in reversed(range(len(rowlist))):
        c = label_list[r]
        x[c] = (b[r] - x*rowlist[r])/rowlist[r][c]
    return x
```