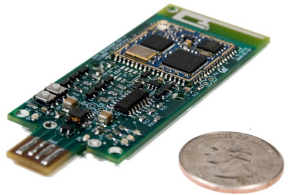
Dot-product: Linear equations

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 $D = \{radio, sensor, memory, CPU\}$

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

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

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

Total energy consumed (in Joules): $\textbf{duration} \cdot \textbf{rate}$

Dot-product: Linear equations

Limitation: We can only measure *total energy consumed by 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,

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

- 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 $\mathbf{a}$ 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: 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:

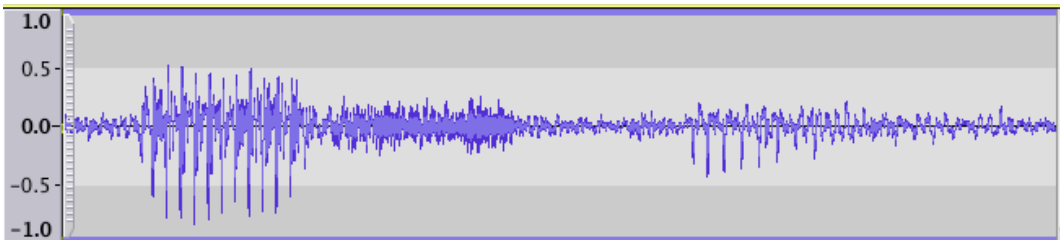
$$[+1, +1, 0, -1]$$

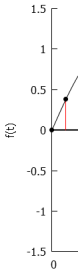
$+1 =$ *In favor*, $0 =$ *not voting*, -1 $=$ *against*

- 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 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								

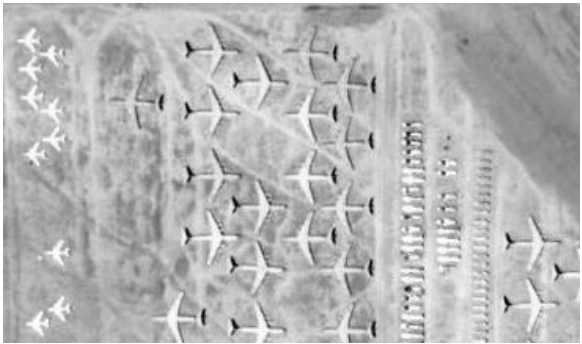
5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3
					2	7	4	-3	0	-1	-6	4	5	-8	-9							

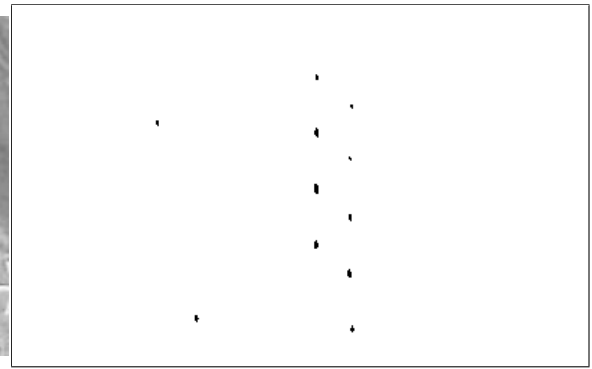
5	-6	9	-9	-5	-9	-5	5	-8	-5	-9	9	8	-5	-9	6	-2	-4	-9	-1	-1	-9	-3

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: finding something in an image





Search for 

Dot-product: Vectors over $GF(2)$

Consider the dot-product of 11111 and 10101:

	1		1		1		1		1	
•	1		0		1		0		1	
	1	+	0	+	1	+	0	+	1	= 1
	1		1		1		1		1	
•	0		0		1		0		1	
	0	+	0	+	1	+	0	+	1	= 0

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 $\mathbf{a}_1 \cdot \hat{\mathbf{x}}$:

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

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

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

(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

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
```

- 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`

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)
```

Start from stencil file `vec.py`

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

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.

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)})
```

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.