Number of solutions to a linear system

We just proved:

If $\mathbf{u}_1$ is a solution to a linear system then

$$\{\text{solutions to linear system}\} = \{\mathbf{u}_1 + \mathbf{v} : \mathbf{v} \in \mathcal{V}\}$$

where $\mathcal{V} = \{\text{solutions to corresponding homogeneous linear system}\}$

Implications:

Previously we asked: *How can we tell if a linear system has only one solution?*

Now we know: If a linear system has a solution $\mathbf{u}_1$ then that solution is unique if the only solution to the corresponding homogeneous linear system is $\mathbf{0}$.

Previously we asked: *How can we find the number of solutions to a linear system over GF(2)?*

Now we know: Number of solutions either is zero or is equal to the number of solutions to the corresponding *homogeneous* linear system.

Number of solutions: checksum function

MD5 checksums and sizes of the released files:

```
3c63a6d97333f4da35976b6a0755eb67  12732276  Python-3.2.2.tgz
9d763097a13a59ff53428c9e4d098a05  10743647  Python-3.2.2.tar.bz2
3720ce9460597e49264bbb63b48b946d   8923224  Python-3.2.2.tar.xz
f6001a9b2be57ecfbefa865e50698cdf  19519332  python-3.2.2-macosx10.3.dmg
8fe82d14dbb2e96a84fd6fa1985b6f73  16226426  python-3.2.2-macosx10.6.dmg
cccb03e14146f7ef82907cf12bf5883c  18241506  python-3.2.2-pdb.zip
72d11475c986182bcb0e5c91acec45bc  19940424  python-3.2.2.amd64-pdb.zip
ddeb3e3fb93ab5a900adb6f04edab21e  18542592  python-3.2.2.amd64.msi
8afb1b01e8fab738e7b234eb4fe3955c  18034688  python-3.2.2.msi
```

A *checksum function* maps long files to short sequences.

Idea:

- Web page shows the checksum of each file to be downloaded.
- Download the file and run the checksum function on it.
- If result does not match checksum on web page, you know the file has been corrupted.
- If random corruption occurs, how likely are you to detect it?

Impractical but instructive checksum function:

- *input:* an n-vector $\mathbf{x}$ over $GF(2)$
- *output:* $[\mathbf{a}_1 \cdot \mathbf{x}, \mathbf{a}_2 \cdot \mathbf{x}, \ \ldots, \ \mathbf{a}_{64} \cdot \mathbf{x}]$

Number of solutions: checksum function

Our checksum function:

- *input:* an n-vector $\mathbf{x}$ over $GF(2)$
- *output:* $[\mathbf{a}_1 \cdot \mathbf{x}, \mathbf{a}_2 \cdot \mathbf{x}, \ \ldots, \ \mathbf{a}_{64} \cdot \mathbf{x}]$

where $\mathbf{a}_1, \mathbf{a}_2, \ \ldots, \ \mathbf{a}_{64}$ are sixty-four n-vectors.

Suppose $\mathbf{p}$ is the original file, and it is randomly corrupted during download.

What is the probability that the corruption is undetected?

The checksum of the original file is $[\beta_1, \ldots, \beta_{64}] = [\mathbf{a}_1 \cdot \mathbf{p}, \ldots, \mathbf{a}_{64} \cdot \mathbf{p}]$.

Suppose corrupted version is $\mathbf{p} + \mathbf{e}$.

Then checksum of corrupted file matches checksum of original if and only if

$$\begin{array}{rcl} \mathbf{a}_1 \cdot (\mathbf{p}+\mathbf{e}) & = & \beta_1 \\ & \vdots & \\ \mathbf{a}_{64} \cdot (\mathbf{p}+\mathbf{e}) & = & \beta_{64} \end{array} \quad \text{iff} \quad \begin{array}{rcl} \mathbf{a}_1 \cdot \mathbf{p} - \mathbf{a}_1 \cdot (\mathbf{p}+\mathbf{e}) & = & 0 \\ & \vdots & \\ \mathbf{a}_{64} \cdot \mathbf{p} - \mathbf{a}_{64} \cdot (\mathbf{p}+\mathbf{e}) & = & 0 \end{array} \quad \text{iff} \quad \begin{array}{rcl} \mathbf{a}_1 \cdot \mathbf{e} & = & 0 \\ & \vdots & \\ \mathbf{a}_{64} \cdot \mathbf{e} & = & 0 \end{array}$$

iff $\mathbf{e}$ is a solution to the homogeneous linear system $\mathbf{a}_1 \cdot \mathbf{x} = 0, \ \ldots \ \mathbf{a}_{64} \cdot \mathbf{x} = 0$.

Number of solutions: checksum function

Suppose corrupted version is $\mathbf{p} + \mathbf{e}$. Then checksum of corrupted file matches checksum of original if and only if $\mathbf{e}$ is a solution to homogeneous linear system

$$\begin{aligned} \mathbf{a}_1 \cdot \mathbf{x} &= 0 \\ &\vdots \\ \mathbf{a}_{64} \cdot \mathbf{x} &= 0 \end{aligned}$$

If $\mathbf{e}$ is chosen according to the uniform distribution,

$$\begin{aligned} &\text{Probability}\,(\mathbf{p} + \mathbf{e} \text{ has same checksum as } \mathbf{p}) \\ &= \text{Probability}\,(\mathbf{e} \text{ is a solution to homogeneous linear system}) \\ &= \frac{\text{number of solutions to homogeneous linear system}}{\text{number of } n\text{-vectors}} \\ &= \frac{\text{number of solutions to homogeneous linear system}}{2^n} \end{aligned}$$

Question: How to find out number of solutions to a homogeneous linear system over $GF(2)$?

Geometry of sets of vectors: convex hull

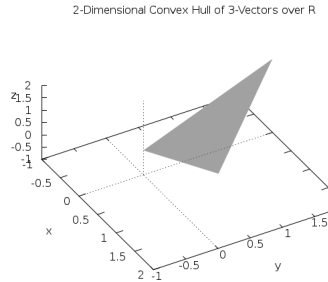
Earlier, we saw: The **u**-to-**v** line segment is

$$\{\alpha\,\mathbf{u} + \beta\,\mathbf{v} \;:\; \alpha \in \mathbb{R}, \beta \in \mathbb{R}, \alpha \geq 0, \beta \geq 0, \alpha + \beta = 1\}$$

Definition: For vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ over $\mathbb{R}$, a linear combination

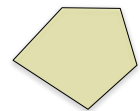
$$\alpha_1\,\mathbf{v}_1 + \cdots + \alpha_n\,\mathbf{v}_n$$

is a *convex combination* if the coefficients are all nonnegative and they sum to 1.



- Convex hull of a single vector is a point.
- Convex hull of two vectors is a line segment.
- Convex hull of three vectors is a triangle

Convex hull of more vectors? Could be higher-dimensional... but not necessarily.



For example, a convex polygon is the convex hull of its vertices

Activity: Vec

You wrote the procedures in `vec.py`:
`add(u,v)`, `scalar_mul(alpha, v)`, `neg(v)`, `dot(u,v)`
Try writing these

- without using `setitem` or `v[k] = ...`
- without doing *any* mutation
- without assigning more than once to any variable (aside from comprehensions)

Two kinds of functions

Focus on two kinds of functions:

- dot-product functions
- linear-combination functions

Dot-product function:

- A function is specified by some C-vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m$
- Input is a C-vector $\mathbf{x}$
- Output is $[\mathbf{a}_1 \cdot \mathbf{x}, \ldots, \mathbf{a}_m \cdot \mathbf{x}]$

Linear-combination function:

- A function is specified by some R-vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$
- Input is a list of n scalars $[\alpha_1, \ldots, \alpha_n]$
- Output is $\alpha_1 \mathbf{v}_1 + \cdots + \alpha_n \mathbf{v}_n$

Example applications of dot-product function

- Cost/benefit
- $C = \{\text{malt}, \text{hops}, \text{yeast}, \text{water}\}$

  cost vector

  $\mathbf{a}_1 = \{\text{hops} : \$2.50/\textit{ounce}, \text{malt} : \$1.50/\textit{pound}, \text{water} : \$0.06/\textit{gallon}, \text{yeast} : \$.45/g\}$

  calorie vector $\mathbf{a}_2 = \{\text{hops} : 0, \text{malt} : 960, \text{water} : 0, \text{yeast} : 3.25\}$

  input $\mathbf{x}$ specifies quantity of each ingredient for some recipe, e.g.

  $\mathbf{x}$ ={hops:6 oz, malt:14 pounds, water:7 gallons, yeast:11 grams}
- Consumption of resources $C = \{\textit{radio}, \textit{sensor}, \textit{memory}, \textit{CPU}\}$

  $\mathbf{a}_1$ is a vector specifying how long each hardware component is working during test period 1

⋮

  $\mathbf{a}_m$ is a vector specifying how long each hardware component is working during test period m

  $\mathbf{x}$ specifies how much energy each component consumes per second, e.g.

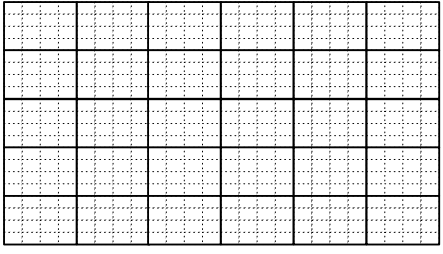
  $\mathbf{x} = \{\textit{memory} : 0.06\text{W}, \textit{radio} : 0.06\text{W}, \textit{sensor} : 0.004\text{W}, \textit{CPU} : 0.0025\text{W}\}$

  function $f(\mathbf{x}) = [\mathbf{a}_1 \cdot \mathbf{x}, \ldots, \mathbf{a}_m \cdot \mathbf{x}]$ maps energy consumption per component to total energy consumption per test period.

More example applications of dot-product functions

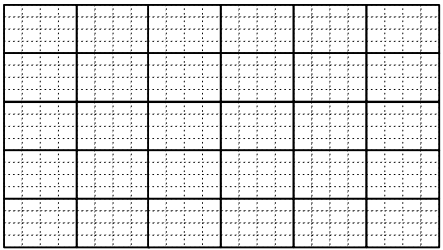
- match filter (image or audio search)
  C is set of audio sample times or pixel locations
  For each possible location of match, have a vector $\mathbf{a}_i$
  $\mathbf{x}$ is an digital audio recording or a digital image.
  $f(\mathbf{x}) = [\mathbf{a}_1, \ldots, \mathbf{a}_m]$ maps $\mathbf{x}$ to measurements of closeness of match
- Authentication
  $C = \{0, \ldots, n-1\}$
  Each $\mathbf{a}_i$ is a challenge observed by Eve
  $\mathbf{x}$ is password
  $f(\mathbf{x}) = [\mathbf{a}_1 \cdot \mathbf{x}, \ldots, \mathbf{a}_m \cdot \mathbf{x}]$ maps $\mathbf{x}$ to the list of responses Eve observed.

Applications of dot-product definition: Downsampling



- Each pixel of the low-res image corresponds to a little grid of pixels of the high-res image.
- The intensity value of a low-res pixel is the *average* of the intensity values of the corresponding high-res pixels.

Applications of dot-product functions: Downsampling



- Each pixel of the low-res image corresponds to a little grid of pixels of the high-res image.
- The intensity value of a low-res pixel is the *average* of the intensity values of the corresponding high-res pixels.

- Averaging can be expressed as dot-product.
- We want to compute a dot-product for each low-res pixel.

Applications of dot-product functions: blurring

- To blur a face, replace each pixel in face with average of pixel intensities in its neighborhood.
- Average can be expressed as dot-product.
- Gaussian blur: a kind of weighted average

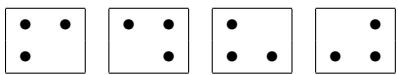
Applications of linear combinations

Resource consumption profile

For making one gnome:
$\mathbf{v}_1 = \{$`metal:0, concrete:1.3, plastic:0.2, water:0.8, electricity:0.4`$\}$
For making one hula hoop:
$\mathbf{v}_2 = \{$`metal:0, concrete:0, plastic:1.5, water:0.4, electricity:0.3`$\}$
For making one slinky:
$\mathbf{v}_3 = \{$`metal:0.25, concrete:0, plastic:0, water:0.2, electricity:0.7`$\}$
For making one silly putty:
$\mathbf{v}_4 = \{$`metal:0, concrete:0, plastic:0.3, water:0.7, electricity:0.5`$\}$
For making one salad shooter:
$\mathbf{v}_5 = \{$`metal:1.5, concrete:0, plastic:0.5, water:0.4, electricity:0.8`$\}$
input [number α_1 of gnomes, number α_2 of hula hoops, ..., number α_5 of salad shooters]
function $f([\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5] = \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2 + \alpha_3\mathbf{v}_3 + \alpha_4\mathbf{v}_4 + \alpha_5\mathbf{v}_5$ outputs the total resource consumption profile.

Applications of linear combinations

Lights Out (over $GF(2)$)

vectors $\mathbf{v}_1, \ldots, v_n$ are button vectors, e.g. 

$\mathbf{x} = [\alpha_1, \ldots, \alpha_n]$ specifies whether a button is pressed or not

$f([\alpha_1, \ldots, \alpha_n]) = \alpha_1\mathbf{v}_1 + \cdots + \alpha_n\mathbf{v}_n$ specifies what initial state this solves