Orthogonalization

[9] Orthogonalization
Finding the closest point in a plane

**Goal:** Given a point $\mathbf{b}$ and a plane, find the point in the plane closest to $\mathbf{b}$. 
Finding the closest point in a plane

**Goal:** Given a point \( \mathbf{b} \) and a plane, find the point in the plane closest to \( \mathbf{b} \).

By translation, we can assume the plane includes the origin.

The plane is a vector space \( \mathcal{V} \). Let \( \{ \mathbf{v}_1, \mathbf{v}_2 \} \) be a basis for \( \mathcal{V} \).

**Goal:** Given a point \( \mathbf{b} \), find the point in \( \text{Span} \{ \mathbf{v}_1, \mathbf{v}_2 \} \) closest to \( \mathbf{b} \).

**Example:**

\[
\mathbf{v}_1 = [8, -2, 2] \quad \text{and} \quad \mathbf{v}_2 = [4, 2, 4]
\]

\[
\mathbf{b} = [5, -5, 2]
\]

point in plane closest to \( \mathbf{b} \): \([6, -3, 0]\).
**Closest-point problem in in higher dimensions**

**Goal:** An algorithm that, given a vector $b$ and vectors $v_1, \ldots, v_n$, finds the vector in \( \text{Span} \{ v_1, \ldots, v_n \} \) that is closest to $b$.

**Special case:** We can use the algorithm to determine whether $b$ lies in \( \text{Span} \{ v_1, \ldots, v_n \} \):
If the vector in \( \text{Span} \{ v_1, \ldots, v_n \} \) closest to $b$ is $b$ itself then clearly $b$ is in the span; if not, then $b$ is not in the span.

Let $A = \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix}$.

Using the linear-combinations interpretation of matrix-vector multiplication, a vector in \( \text{Span} \{ v_1, \ldots, v_n \} \) can be written $Ax$.

Thus testing if $b$ is in \( \text{Span} \{ v_1, \ldots, v_n \} \) is equivalent to testing if the equation $Ax = b$ has a solution.

**More generally:**
Even if $Ax = b$ has no solution, we can use the algorithm to find the point in \( \{ Ax : x \in \mathbb{R}^n \} \) closest to $b$.

**Moreover:** We hope to extend the algorithm to also find the best solution $x$. 
Closest point and coefficients

Not enough to find the point \( p \) in \( \text{Span}\ \{v_1, \ldots, v_n\} \) closest to \( b \)....

We need an algorithm to find the representation of \( p \) in terms of \( v_1, \ldots, v_n \).

**Goal:** find the coefficients \( x_1, \ldots, x_n \) so that \( x_1 v_1 + \cdots + x_n v_n \) is the vector in \( \text{Span}\ \{v_1, \ldots, v_n\} \) closest to \( b \).

**Equivalent:** Find the vector \( x \) that minimizes

\[
\| b \| - \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} \begin{bmatrix} x \end{bmatrix}
\]

**Equivalent:** Find the vector \( x \) that minimizes

\[
\| b \| - \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} \begin{bmatrix} x \end{bmatrix}^2
\]

**Equivalent:** Find the vector \( x \) that minimizes

\[
\| b \| - \begin{bmatrix} a_1 & \vdots & a_m \end{bmatrix} \begin{bmatrix} x \end{bmatrix}^2
\]

**Equivalent:** Find the vector \( x \) that minimizes

\[
(b[1] - a_1 \cdot x)^2 + \cdots + (b[m] - a_m \cdot x)^2
\]

This last problem was addressed using gradient descent in Machine Learning lab.
Closest point and least squares

Find the vector $x$ that minimizes

$$\left\| \begin{bmatrix} b \\ \vdots \\ v_n \end{bmatrix} - \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} \begin{bmatrix} x \end{bmatrix} \right\|^2$$

**Equivalent:** Find the vector $x$ that minimizes

$$(b[1] - a_1 \cdot x)^2 + \cdots + (b[m] - a_m \cdot x)^2$$

This problem is called *least squares* ("méthode des moindres carrés", due to Adrien-Marie Legendre but often attributed to Gauss)

**Equivalent:** Given a matrix equation $Ax = b$ that might have no solution, find the best solution available in the sense that the norm of the error $b - Ax$ is as small as possible.

- There is an algorithm based on Gaussian elimination.
- We will develop an algorithm based on orthogonality (used in solver)

Much faster and more reliable than gradient descent.
High-dimensional projection onto/orthogonal to

For any vector $b$ and any vector $a$, define vectors $b||a$ and $b\perp a$ so that

$$b = b||a + b\perp a$$

and there is a scalar $\sigma \in \mathbb{R}$ such that

$$b||a = \sigma a$$

and

$$b\perp a$$ is orthogonal to $a$

**Definition:** For a vector $b$ and a vector space $V$, we define the projection of $b$ onto $V$ (written $b||V$) and the projection of $b$ orthogonal to $V$ (written $b\perp V$) so that

$$b = b||V + b\perp V$$

and $b||V$ is in $V$, and $b\perp V$ is orthogonal to every vector in $V$. 
**High-Dimensional Fire Engine Lemma**

**Definition:** For a vector $b$ and a vector space $\mathcal{V}$, we define the projection of $b$ onto $\mathcal{V}$ (written $b_{\parallel}\mathcal{V}$) and the projection of $b$ orthogonal to $\mathcal{V}$ (written $b_{\perp}\mathcal{V}$) so that

$$b = b_{\parallel}\mathcal{V} + b_{\perp}\mathcal{V}$$

and $b_{\parallel}\mathcal{V}$ is in $\mathcal{V}$, and $b_{\perp}\mathcal{V}$ is orthogonal to every vector in $\mathcal{V}$.

**One-dimensional Fire Engine Lemma:** The point in $\text{Span}\{a\}$ closest to $b$ is $b_{\parallel}a$ and the distance is $\|b_{\perp}a\|$. 

**High-Dimensional Fire Engine Lemma:** The point in a vector space $\mathcal{V}$ closest to $b$ is $b_{\parallel}\mathcal{V}$ and the distance is $\|b_{\perp}\mathcal{V}\|$. 
Finding the projection of $b$ orthogonal to $\text{Span} \{a_1, \ldots, a_n\}$

**High-Dimensional Fire Engine Lemma:** Let $b$ be a vector and let $\mathcal{V}$ be a vector space. The vector in $\mathcal{V}$ closest to $b$ is $b^{\parallel\mathcal{V}}$. The distance is $\|b^{\perp\mathcal{V}}\|$.

Suppose $\mathcal{V}$ is specified by generators $v_1, \ldots, v_n$

**Goal:** An algorithm for computing $b^{\perp\mathcal{V}}$ in this case.

- **input:** vector $b$, vectors $v_1, \ldots, v_n$
- **output:** projection of $b$ orthogonal to $\mathcal{V} = \text{Span} \{v_1, \ldots, v_n\}$

We already know how to solve this when $n = 1$:

```python
def project_orthogonal_1(b, v):
    return b - project_along(b, v)
```

Let's try to generalize....
def project_orthogonal_1(b, v):
    return b - project_along(b, v)

def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b

Reviews are in....

“Short, elegant, .... and flawed”

“Beautiful—if only it worked!”

“A tragic failure.”
**project_orthogonal(b, vlist) doesn't work**

```python
def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b
```

Let \( b_i \) be value of the variable \( b \) after \( i \) iterations.

\[
\begin{align*}
    b_1 &= b_0 - \text{(projection of } [1, 1] \text{ along } [1, 0]) \\
         &= b_0 - [1, 0] \\
         &= [0, 1] \\

    b_2 &= b_1 - \text{(projection of } [0, 1] \text{ along } [\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}]) \\
         &= b_1 - [\frac{1}{2}, \frac{1}{2}] \\
         &= [-\frac{1}{2}, \frac{1}{2}] \text{ which is not orthogonal to } [1, 0]
\end{align*}
\]
project\_orthogonal(b, vlist) doesn't work

```python
def project\_orthogonal(b, vlist):
    for v in vlist:
        b = b - project\_along(b, v)
    return b
```

\[ b = [1, 1] \]
\[ vlist = \begin{bmatrix} [1, 0], \\
    [\sqrt{2} / 2, \sqrt{2} / 2] \end{bmatrix} \]

Let \( b_i \) be value of the variable \( b \) after \( i \) iterations.

\[
\begin{align*}
    b_1 &= b_0 - (\text{projection of } [1, 1] \text{ along } [1, 0]) \\
    &= b_0 - [1, 0] \\
    &= [0, 1] \\

    b_2 &= b_1 - (\text{projection of } [0, 1] \text{ along } [\sqrt{2} / 2, \sqrt{2} / 2]) \\
    &= b_1 - \begin{bmatrix} 1 / 2, 1 / 2 \end{bmatrix} \\
    &= [-1 / 2, 1 / 2] \text{ which is not orthogonal to } [1, 0]
\end{align*}
\]
project_orthogonal(b, vlist) doesn't work

```python
def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b
```

Let $b_i$ be value of the variable $b$ after $i$ iterations.

\[
\begin{align*}
    b_1 &= b_0 - \text{(projection of [1, 1] along [1, 0])} \\
         &= b_0 - [1, 0] \\
         &= [0, 1] \\

    b_2 &= b_1 - \text{(projection of [0, 1] along } \left[ \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2} \right]) \\
         &= b_1 - \left[ \frac{1}{2}, \frac{1}{2} \right] \\
         &= \left[ -\frac{1}{2}, \frac{1}{2} \right] \text{ which is not orthogonal to [1, 0]}
\end{align*}
\]
project_orthogonal(b, vlist) doesn't work

```python
def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b
```

Let $b_i$ be value of the variable $b$ after $i$ iterations.

\[
\begin{align*}
b_1 &= b_0 - (\text{projection of } [1, 1] \text{ along } [1, 0]) \\
    &= b_0 - [1, 0] \\
    &= [0, 1] \\

b_2 &= b_1 - (\text{projection of } [0, 1] \text{ along } \left[ \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2} \right]) \\
    &= b_1 - \left[ \frac{1}{2}, \frac{1}{2} \right] \\
    &= \left[ -\frac{1}{2}, \frac{1}{2} \right] \text{ which is not orthogonal to } [1, 0]
\end{align*}
\]
def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b

Let $b_i$ be value of the variable $b$ after $i$ iterations.

\[
\begin{align*}
b_1 &= b_0 - (\text{projection of } [1, 1] \text{ along } [1, 0]) \\
     &= b_0 - [1, 0] \\
     &= [0, 1] \\
b_2 &= b_1 - (\text{projection of } [0, 1] \text{ along } \left[ \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2} \right]) \\
     &= b_1 - \left[ \frac{1}{2}, \frac{1}{2} \right] \\
     &= \left[ -\frac{1}{2}, \frac{1}{2} \right] \text{ which is not orthogonal to } [1, 0]
\end{align*}
\]
How to repair `project_orthogonal(b, vlist)`?

```python
def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b
```

```python
b = [1,1]
vlist = [[1,0], [\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}]]
```

Final vector is not orthogonal to `[1, 0]`

Maybe the problem will go away if the algorithm

- first finds the projection of `b` along each of the vectors in `vlist`, and
- only afterwards subtracts all these projections from `b`.

```python
def classical_project_orthogonal(b, vlist):
    w = all-zeroes-vector
    for v in vlist:
        w = w + project_along(b, v)
    return b - w
```

Alas, this procedure also does not work. For the inputs

```python
b = [1, 1], vlist = [ [1, 0], [\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}] ]
```

the output vector is `[-1, 0]`

which is orthogonal to neither of the two vectors in `vlist`. 
What to do with \texttt{project\_orthogonal}(b, vlist)?

Try it with two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ that are orthogonal...

\[
\begin{align*}
\mathbf{v}_1 &= [1, 2, 1] \\
\mathbf{v}_2 &= [-1, 2, -1] \\
\mathbf{b} &= [1, 1, 2] \\
\mathbf{b}_1 &= \mathbf{b}_0 - \frac{\mathbf{b}_0 \cdot \mathbf{v}_1}{\mathbf{v}_1 \cdot \mathbf{v}_1} \mathbf{v}_1 \\
&= [1, 1, 2] - \frac{5}{6} [1, 2, 1] \\
&= \begin{bmatrix} 1 \\ 6 \\ \frac{1}{6} \end{bmatrix} - \begin{bmatrix} -4 \\ 6 \\ 6 \end{bmatrix} \\
\mathbf{b}_2 &= \mathbf{b}_1 - \frac{\mathbf{b}_1 \cdot \mathbf{v}_2}{\mathbf{v}_2 \cdot \mathbf{v}_2} \mathbf{v}_2 \\
&= \begin{bmatrix} 1 \\ 6 \\ \frac{1}{6} \end{bmatrix} - \frac{1}{2} [-1, 0, 1] \\
&= \begin{bmatrix} 2 \\ 3 \\ -2 \end{bmatrix}
\end{align*}
\]

and note $\mathbf{b}_2$ is orthogonal to $\mathbf{v}_1$ and $\mathbf{v}_2$. 
What to do with `project_orthogonal(b, vlist)`?

Try it with two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ that are orthogonal...

$$\mathbf{v}_1 = [1, 2, 1]$$
$$\mathbf{v}_2 = [-1, 2, -1]$$
$$\mathbf{b} = [1, 1, 2]$$

$$\mathbf{b}_1 = \mathbf{b}_0 - \frac{\mathbf{b}_0 \cdot \mathbf{v}_1}{\mathbf{v}_1 \cdot \mathbf{v}_1} \mathbf{v}_1$$

$$= [1, 1, 2] - \frac{5}{6} [1, 2, 1]$$
$$= \left[ \frac{1}{6}, -\frac{4}{6}, \frac{7}{6} \right]$$

$$\mathbf{b}_2 = \mathbf{b}_1 - \frac{\mathbf{b}_1 \cdot \mathbf{v}_2}{\mathbf{v}_2 \cdot \mathbf{v}_2} \mathbf{v}_2$$

$$= \left[ \frac{1}{6}, -\frac{4}{6}, \frac{7}{6} \right] - \frac{1}{2} [-1, 0, 1]$$
$$= \left[ \frac{2}{3}, -\frac{2}{3}, \frac{2}{3} \right]$$

and note $\mathbf{b}_2$ is orthogonal to $\mathbf{v}_1$ and $\mathbf{v}_2$. 
Maybe project_orthogonal(b, vlist) works with $v_1, v_2$ orthogonal?

Assume $\langle v_1, v_2 \rangle = 0$.
Remember: $b_i$ is value of $b$ after $i$ iterations

First iteration:
$$b_1 = b_0 - \sigma_1 v_1$$
gives $b_1$ such that $\langle v_1, b_1 \rangle = 0$.

Second iteration:
$$b_2 = b_1 - \sigma_1 v_2$$
gives $b_2$ such that $\langle v_2, b_2 \rangle = 0$

But what about $\langle v_1, b_2 \rangle$?

$$\langle v_1, b_2 \rangle = \langle v_1, b_1 - \sigma v_2 \rangle$$
$$= \langle v_1, b_1 \rangle - \langle v_1, \sigma v_2 \rangle$$
$$= \langle v_1, b_1 \rangle - \sigma \langle v_1, v_2 \rangle$$
$$= 0 + 0$$

Thus $b_2$ is orthogonal to $v_1$ and $v_2$
Don't fix `project_orthogonal(b, vlist)`. Fix the spec.

```python
def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b
```

Instead of trying to fix the flaw by changing the procedure, we will change the spec we expect the procedure to fulfill.

Require that `vlist` consists of **mutually orthogonal** vectors:
the $i^{th}$ vector in the list is orthogonal to the $j^{th}$ vector in the list for every $i \neq j$.

**New spec:**

- **input**: a vector $b$, and a list `vlist` of **mutually orthogonal** vectors
- **output**: the projection $b \perp$ of $b$ orthogonal to the vectors in `vlist`
Loop invariant of `project_orthogonal(b, vlist)`

```python
def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b
```

**Loop invariant:** Let `vlist = [v_1, ..., v_n]`
For `i = 0, ..., n`, let `b_i` be the value of the variable `b` after `i` iterations. Then `b_i` is the projection of `b` orthogonal to `Span \{v_1, ..., v_i\}`. That is,
- `b_i` is orthogonal to the first `i` vectors of `vlist`, and
- `b - b_i` is in the span of the first `i` vectors of `vlist`

We use induction to prove the invariant holds.

For `i = 0`, the invariant is trivially true:
- `b_0` is orthogonal to each of the first 0 vectors (every vector is), and
- `b - b_0` is in the span of the first 0 vectors (because `b - b_0` is the zero vector).
Proof of loop invariant of \texttt{project\_orthogonal}(b, [v_1, \ldots, v_n])

\( b_i = \) projection of \( b \) orthogonal to 
\( \text{Span} \{ v_1, \ldots, v_i \} \):

- \( b_i \) is orthogonal to \( v_1, \ldots, v_i \), and
- \( b - b_i \) is in \( \text{Span} \{ v_1, \ldots, v_i \} \)

Assume invariant holds for \( i = k - 1 \) iterations, and prove it for \( i = k \) iterations.

In \( k^{th} \) iteration, algorithm computes \( b_k = b_{k-1} - \sigma_k v_k \)

By induction hypothesis, \( b_{k-1} \) is the projection of \( b \) orthogonal to \( \text{Span} \{ v_1, \ldots, v_{k-1} \} \)

Must prove

\begin{itemize}
  \item \( b_k \) is orthogonal to \( v_1, \ldots, v_k \), \( \checkmark \)
  \item and \( b - b_k \) is in \( \text{Span} \{ v_1, \ldots, v_k \} \), \( \checkmark \)
\end{itemize}

Choice of \( \sigma_k \) ensures that \( b_k \) is orthogonal to \( v_k \).

Must show \( b_k \) also orthogonal to \( v_j \) for \( j = 1, \ldots, k - 1 \)

\[
\langle b_k, v_j \rangle = \langle b_{k-1} - \sigma_k v_k, v_j \rangle \\
= \langle b_{k-1}, v_j \rangle - \sigma_k \langle v_k, v_j \rangle \\
= 0 - \sigma_k \langle v_k, v_j \rangle \\
= 0 - \sigma_k 0
\]

by the inductive hypothesis

by mutual orthogonality

Shows \( b_k \) orthogonal to \( v_1, \ldots, v_k \)
Correctness of `project_orthogonal(b, vlist)`

```python
def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b
```

We have proved:
If \( \mathbf{v}_1, \ldots, \mathbf{v}_n \) are mutually orthogonal then
output of `project_orthogonal(b, [\mathbf{v}_1, \ldots, \mathbf{v}_n])` is the vector \( \mathbf{b}^\perp \) such that
\[
\begin{align*}
\mathbf{b} &= \mathbf{b}^\parallel + \mathbf{b}^\perp \\
\mathbf{b}^\parallel &\text{ is in Span } \{\mathbf{v}_1, \ldots, \mathbf{v}_n\} \\
\mathbf{b}^\perp &\text{ is orthogonal to } \mathbf{v}_1, \ldots, \mathbf{v}_n.
\end{align*}
\]

Change to zero-based indexing::
If \( \mathbf{v}_0, \ldots, \mathbf{v}_n \) are mutually orthogonal then
output of `project_orthogonal(b, [\mathbf{v}_0, \ldots, \mathbf{v}_n])` is the vector \( \mathbf{b}^\perp \) such that
\[
\begin{align*}
\mathbf{b} &= \mathbf{b}^\parallel + \mathbf{b}^\perp \\
\mathbf{b}^\parallel &\text{ is in Span } \{\mathbf{v}_0, \ldots, \mathbf{v}_n\} \\
\mathbf{b}^\perp &\text{ is orthogonal to } \mathbf{v}_0, \ldots, \mathbf{v}_n.
\end{align*}
\]
Augmenting `project_orthogonal`

Since $\mathbf{b}^\parallel = \mathbf{b} - \mathbf{b}^\perp$ is in \text{Span} \{\mathbf{v}_0, \ldots, \mathbf{v}_n\}, there are coefficients $\alpha_0, \ldots, \alpha_n$ such that

$$\mathbf{b} - \mathbf{b}^\perp = \alpha_0 \mathbf{v}_0 + \cdots + \alpha_n \mathbf{v}_n$$

$b = \alpha_0 \mathbf{v}_0 + \cdots + \alpha_n \mathbf{v}_n + 1 \mathbf{b}^\perp$

Write as

$$\begin{bmatrix} \mathbf{b} \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 & \cdots & \mathbf{v}_n & \mathbf{b}^\perp \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_n \\ 1 \end{bmatrix}$$

The procedure `project_orthogonal(b, vlist)` can be augmented to output the vector of coefficients.

For technical reasons, we will represent the vector of coefficients as a dictionary, not a Vec.
Augmenting project_orthogonal

\[
\begin{bmatrix}
  b \\
  v_0 \ | \ \cdots \ | \ v_n \ | \ b^\perp
\end{bmatrix}
= 
\begin{bmatrix}
  \alpha_0 \\
  \vdots \\
  \alpha_n \\
  1
\end{bmatrix}
\]

We reuse code from two prior procedures.

```python
def project_along(b, v):
    sigma = ((b*v)/(v*v)) \n        if v*v != 0 else 0
    return sigma * v

def project_orthogonal(b, vlist):
    for v in vlist:
        b = b - project_along(b, v)
    return b

def aug_project_orthogonal(b, vlist):
    alphadict = {len(vlist):1}
    for i in range(len(vlist)):
        v = vlist[i]
        sigma = (b*v)/(v*v) \n            if v*v > 0 else 0
        alphadict[i] = sigma
        b = b - sigma*v
    return (b, alphadict)
```

Must create and populate a dictionary.
- One entry for each vector in vlist
- One additional entry, 1, for $b^\perp$

Initialize dictionary with the additional entry.
Building an orthogonal set of generators

**Original stated goal:**
Find the projection of \( \mathbf{b} \) orthogonal to the space \( \mathcal{V} \) spanned by arbitrary vectors \( \mathbf{v}_1, \ldots, \mathbf{v}_n \).

So far we know how to find the projection of \( \mathbf{b} \) orthogonal to the space spanned by mutually orthogonal vectors.

This would suffice if we had a procedure that, given arbitrary vectors \( \mathbf{v}_1, \ldots, \mathbf{v}_n \), computed mutually orthogonal vectors \( \mathbf{v}^*_1, \ldots, \mathbf{v}^*_n \) that span the same space.

We consider a new problem: **orthogonalization**:

- **input**: A list \([\mathbf{v}_1, \ldots, \mathbf{v}_n]\) of vectors over the reals
- **output**: A list of mutually orthogonal vectors \( \mathbf{v}^*_1, \ldots, \mathbf{v}^*_n \) such that

\[
\text{Span} \left\{ \mathbf{v}^*_1, \ldots, \mathbf{v}^*_n \right\} = \text{Span} \left\{ \mathbf{v}_1, \ldots, \mathbf{v}_n \right\}
\]

How can we solve this problem?
The orthogonalize procedure

**Idea:** Use `project_orthogonal` iteratively to make a longer and longer list of mutually orthogonal vectors.

- First consider $\mathbf{v}_1$. Define $\mathbf{v}^*_1 := \mathbf{v}_1$ since the set $\{\mathbf{v}^*_1\}$ is trivially a set of mutually orthogonal vectors.

- Next, define $\mathbf{v}^*_2$ to be the projection of $\mathbf{v}_2$ orthogonal to $\mathbf{v}^*_1$.

- Now $\{\mathbf{v}^*_1, \mathbf{v}^*_2\}$ is a set of mutually orthogonal vectors.

- Next, define $\mathbf{v}^*_3$ to be the projection of $\mathbf{v}_3$ orthogonal to $\mathbf{v}^*_1$ and $\mathbf{v}^*_2$, so $\{\mathbf{v}^*_1, \mathbf{v}^*_2, \mathbf{v}^*_3\}$ is a set of mutually orthogonal vectors.

In each step, we use `project_orthogonal` to find the next orthogonal vector.

In the $i^{th}$ iteration, we project $\mathbf{v}_i$ orthogonal to $\mathbf{v}^*_1, \ldots, \mathbf{v}^*_{i-1}$ to find $\mathbf{v}^*_i$.

```python
def orthogonalize(vlist):
    vstarlist = []
    for v in vlist:
        vstarlist.append(project_orthogonal(v, vstarlist))
    return vstarlist
```
def orthogonalize(vlist):
    vstarlist = []
    for v in vlist:
        vstarlist.append(project_orthogonal(v, vstarlist))
    return vstarlist

**Lemma:** Throughout the execution of `orthogonalize`, the vectors in `vstarlist` are mutually orthogonal.

In particular, the list `vstarlist` at the end of the execution, which is the list returned, consists of mutually orthogonal vectors.

**Proof:** by induction, using the fact that each vector added to `vstarlist` is orthogonal to all the vectors already in the list. QED
Example of orthogonalize

**Example:** When orthogonalize is called on a vlist consisting of vectors
\[ \mathbf{v}_1 = [2, 0, 0], \mathbf{v}_2 = [1, 2, 2], \mathbf{v}_3 = [1, 0, 2] \]
it returns the list vstarlist consisting of
\[ \mathbf{v}_1^* = [2, 0, 0], \mathbf{v}_2^* = [0, 2, 2], \mathbf{v}_3^* = [0, -1, 1] \]

(1) In the first iteration, when \( \mathbf{v} \) is \( \mathbf{v}_1 \), vstarlist is empty, so the first vector \( \mathbf{v}_1^* \) added to vstarlist is \( \mathbf{v}_1 \) itself.

(2) In the second iteration, when \( \mathbf{v} \) is \( \mathbf{v}_2 \), vstarlist consists only of \( \mathbf{v}_1^* \). The projection of \( \mathbf{v}_2 \) orthogonal to \( \mathbf{v}_1^* \) is
\[
\mathbf{v}_2 - \frac{\langle \mathbf{v}_2, \mathbf{v}_1^* \rangle}{\langle \mathbf{v}_1^*, \mathbf{v}_1^* \rangle} \mathbf{v}_1^* = [1, 2, 2] - \frac{2}{4} [2, 0, 0] \\
= [0, 2, 2]
\]
so \( \mathbf{v}_2^* = [0, 2, 2] \) is added to vstarlist.

(3) In the third iteration, when \( \mathbf{v} \) is \( \mathbf{v}_3 \), vstarlist consists of \( \mathbf{v}_1^* \) and \( \mathbf{v}_2^* \). The projection of \( \mathbf{v}_3 \) orthogonal to \( \mathbf{v}_1^* \) is \( [0, 0, 2] \), and the projection of \( [0, 0, 2] \) orthogonal to \( \mathbf{v}_2^* \) is
\[
[0, 0, 2] - \frac{1}{2} [0, 2, 2] = [0, -1, 1]
\]
so \( \mathbf{v}_3^* = [0, -1, 1] \) is added to vstarlist.
Correctness of the orthogonalize procedure, Part II

**Lemma:** Consider orthogonalize applied to an $n$-element list $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$. After $i$ iterations of the algorithm, $\text{Span } \mathbf{v}_{\text{star list}} = \text{Span } \{\mathbf{v}_1, \ldots, \mathbf{v}_i\}$.

**Proof:** by induction on $i$.

The case $i = 0$ is trivial.

After $i - 1$ iterations, $\mathbf{v}_{\text{star list}}$ consists of vectors $\mathbf{v}_1^*, \ldots, \mathbf{v}_{i-1}^*$.

Assume the lemma holds at this point. This means that

$$\text{Span } \{\mathbf{v}_1^*, \ldots, \mathbf{v}_{i-1}^*\} = \text{Span } \{\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}\}$$

By adding the vector $\mathbf{v}_i$ to sets on both sides, we obtain

$$\text{Span } \{\mathbf{v}_1^*, \ldots, \mathbf{v}_{i-1}^*, \mathbf{v}_i\} = \text{Span } \{\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}, \mathbf{v}_i\}$$

... It therefore remains only to show that

$$\text{Span } \{\mathbf{v}_1^*, \ldots, \mathbf{v}_{i-1}^*, \mathbf{v}_i^*\} = \text{Span } \{\mathbf{v}_1^*, \ldots, \mathbf{v}_{i-1}^*, \mathbf{v}_i\}.$$}

The $i^{th}$ iteration computes $\mathbf{v}_i^*$ using $\text{project_orthogonal}(\mathbf{v}_i, [\mathbf{v}_1^*, \ldots, \mathbf{v}_{i-1}^*])$.

There are scalars $\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{i,i-1}$ such that

$$\mathbf{v}_i = \alpha_{i1}\mathbf{v}_1^* + \cdots + \alpha_{i-1,i}\mathbf{v}_{i-1}^* + \mathbf{v}_i^*$$

This equation shows that any linear combination of $\mathbf{v}_1^*, \ldots, \mathbf{v}_{i-1}^*$ can be transformed into a linear combination of $\mathbf{v}_1^*, \ldots, \mathbf{v}_{i-1}^*, \mathbf{v}_i^*$ and vice versa.

QED
Order in orthogonalize

Order matters!

Suppose you run the procedure `orthogonalize` twice, once with a list of vectors and once with the reverse of that list.

The output lists will **not** be the reverses of each other.

Contrast with `project_orthogonal(b, vlist)`.

The projection of a vector \( b \) orthogonal to a vector space is unique, so in principle the order of vectors in \( vlist \) doesn’t affect the output of `project_orthogonal(b, vlist)`. 
Matrix form for orthogonalize

For project orthogonal, we had
\[
\begin{bmatrix}
    b \\
\end{bmatrix} =
\begin{bmatrix}
    v_0 & \cdots & v_n & b^\perp
\end{bmatrix}
\begin{bmatrix}
    \alpha_0 \\
    \vdots \\
    \alpha_n \\
    1
\end{bmatrix}
\]

For orthogonalize, we have
\[
\begin{bmatrix}
    v_0 \\
\end{bmatrix} =
\begin{bmatrix}
    v_0^* & 1
\end{bmatrix}
\begin{bmatrix}
    v_0 & v_1 & v_2 & v_3
\end{bmatrix} =
\begin{bmatrix}
    v_0^* & v_1^* & v_2^* & v_3^*
\end{bmatrix}
\begin{bmatrix}
    1 & \alpha_{01} & \alpha_{02} & \alpha_{03} \\
    \alpha_{01} & 1 & \alpha_{12} & \alpha_{13} \\
    \alpha_{02} & \alpha_{12} & 1 & \alpha_{23} \\
    \alpha_{03} & \alpha_{13} & \alpha_{23} & 1
\end{bmatrix}
\]

The two matrices on the right are special:
- Columns of first one are mutually orthogonal.
- Second is upper triangular.

We will use these properties in algorithms.
Example of matrix form for orthogonalize

**Example:** for vlist consisting of vectors

\[
\begin{bmatrix}
2 \\
0 \\
0
\end{bmatrix},
\begin{bmatrix}
1 \\
2 \\
2
\end{bmatrix},
\begin{bmatrix}
1 \\
0 \\
2
\end{bmatrix}
\]

we saw that the output list vstarlist of orthogonal vectors consists of

\[
\begin{bmatrix}
2 \\
0 \\
0
\end{bmatrix},
\begin{bmatrix}
0 \\
2 \\
2
\end{bmatrix},
\begin{bmatrix}
0 \\
-1 \\
1
\end{bmatrix}
\]

The corresponding matrix equation is

\[
\begin{bmatrix}
2 & 0 & 0 \\
0 & 2 & -1 \\
0 & 2 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0.5 & 0.5 \\
1 & 0.5 & 
\end{bmatrix}
\]
Solving closest point in the span of many vectors

Let $\mathcal{V} = \text{Span}\{v_0, \ldots, v_n\}$.

The vector in $\mathcal{V}$ closest to $b$ is $b_{\parallel\mathcal{V}}$, which is $b - b_{\perp\mathcal{V}}$.

There are two equivalent ways to find $b_{\perp\mathcal{V}}$,

- **One method:**
  
  Step 1: Apply $\text{orthogonalize}$ to $v_0, \ldots, v_n$, and obtain $v_0^*, \ldots, v_n^*$. (Now $\mathcal{V} = \text{Span}\{v_0^*, \ldots, v_n^*\}$)
  
  Step 2: Call $\text{project_orthogonal}(b, [v_0^*, \ldots, v_n^*])$ and obtain $b_{\perp\mathcal{V}}$ as the result.

- **Another method:** Exactly the same computations take place when $\text{orthogonalize}$ is applied to $[v_0, \ldots, v_n, b]$ to obtain $[v_0^*, \ldots, v_n^*, b^*]$. In the last iteration of $\text{orthogonalize}$, the vector $b^*$ is obtained by projecting $b$ orthogonal to $v_0^*, \ldots, v_n^*$. Thus $b^* = b_{\perp\mathcal{V}}$. 
We’ve shown how orthogonalize can be used to find the vector in \( \text{Span} \{ \mathbf{v}_0, \ldots, \mathbf{v}_n \} \) closest to \( \mathbf{b} \), namely \( \mathbf{b}^\parallel \).

Later we give an algorithm to find the coordinate representation of \( \mathbf{b}^\parallel \) in terms of \( \{ \mathbf{v}_0, \ldots, \mathbf{v}_n \} \).

First we will see how we can use orthogonalization to solve other computational problems.

We need to prove something about mutually orthogonal vectors....
Proposition: Mutually orthogonal nonzero vectors are linearly independent.

Proof: Let \( \mathbf{v}_0^*, \mathbf{v}_2^*, \ldots, \mathbf{v}_n^* \) be mutually orthogonal nonzero vectors.
Suppose \( \alpha_0, \alpha_1, \ldots, \alpha_n \) are coefficients such that

\[
0 = \alpha_0 \mathbf{v}_0^* + \alpha_1 \mathbf{v}_1^* + \cdots + \alpha_n \mathbf{v}_n^*
\]

We must show that therefore the coefficients are all zero.
To show that \( \alpha_0 \) is zero, take inner product with \( \mathbf{v}_0^* \) on both sides:

\[
\langle \mathbf{v}_0^*, 0 \rangle = \langle \mathbf{v}_0^*, \alpha_0 \mathbf{v}_0^* + \alpha_1 \mathbf{v}_1^* + \cdots + \alpha_n \mathbf{v}_n^* \rangle = \alpha_0 \langle \mathbf{v}_0^*, \mathbf{v}_0^* \rangle + \alpha_1 \langle \mathbf{v}_0^*, \mathbf{v}_1^* \rangle + \cdots + \alpha_n \langle \mathbf{v}_0^*, \mathbf{v}_n^* \rangle = \alpha_0 \| \mathbf{v}_0^* \|^2 + \alpha_1 0 + \cdots + \alpha_n 0 = \alpha_0 \| \mathbf{v}_0^* \|^2
\]

The inner product \( \langle \mathbf{v}_0^*, 0 \rangle \) is zero, so \( \alpha_0 \| \mathbf{v}_0^* \|^2 = 0 \). Since \( \mathbf{v}_0^* \) is nonzero, its norm is nonzero, so the only solution is \( \alpha_0 = 0 \).
Can similarly show that \( \alpha_1 = \cdots = \alpha_n = 0 \). QED
Computing a basis

**Proposition:** Mutually orthogonal nonzero vectors are linearly independent.

What happens if we call the `orthogonalize` procedure on a list \( \text{vlist} = [\mathbf{v}_0, \ldots, \mathbf{v}_n] \) of vectors that are linearly dependent?

\[
\dim \text{Span} \{\mathbf{v}_0, \ldots, \mathbf{v}_n\} < n + 1.
\]

`orthogonalize([\mathbf{v}_0, \ldots, \mathbf{v}_n])` returns \([\mathbf{v}_0^*, \ldots, \mathbf{v}_n^*]\)

The vectors \(\mathbf{v}_0^*, \ldots, \mathbf{v}_n^*\) are mutually orthogonal.

They can't be linearly independent since they span a space of dimension less than \(n + 1\).

Therefore some of them must be zero vectors.

Leaving out the zero vectors does not change the space spanned...

Let \(S\) be the subset of \(\{\mathbf{v}_0^*, \ldots, \mathbf{v}_n^*\}\) consisting of nonzero vectors.

\(\text{Span } S = \text{Span } \{\mathbf{v}_0^*, \ldots, \mathbf{v}_n^*\} = \text{Span } \{\mathbf{v}_0, \ldots, \mathbf{v}_n\}\)

Proposition implies that \(S\) is linearly independent.

Thus \(S\) is a basis for \(\text{Span } \{\mathbf{v}_0, \ldots, \mathbf{v}_n\}\).
Computing a basis

Therefore in principle the following algorithm computes a basis for Span \{v_0, \ldots, v_n\}:

```python
def find_basis([v_0, \ldots, v_n]):
    "Return the list of nonzero starred vectors."
    [v^*_0, \ldots, v^*_n] = orthogonalize([v_0, \ldots, v_n])
    return [v^*_v for v^*_v in [v^*_0, \ldots, v^*_n] if v^*_v is not the zero vector]
```

Example:
Suppose \text{orthogonalize}([v_0, v_1, v_2, v_3, v_4, v_5, v_6]) returns [v^*_0, v^*_1, v^*_2, v^*_3, v^*_4, v^*_5, v^*_6] and the vectors v^*_2, v^*_4, and v^*_5 are zero.
Then the remaining output vectors v^*_0, v^*_1, v^*_3, v^*_6 form a basis for Span \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}.

Recall

\textbf{Lemma:} Every finite set \( T \) of vectors contains a subset \( S \) that is a basis for Span \( T \).

What about finding a subset of \( v_0, \ldots, v_n \) that is a basis?

\textbf{Proposed algorithm:}

```python
def find_subset_basis([v_0, \ldots, v_n]):
    "Return the list of original vectors that correspond to nonzero starred vectors."
    [v^*_0, \ldots, v^*_n] = orthogonalize([v_0, \ldots, v_n])
    return [v_v for v_v in [v_0, \ldots, v_n] if v_v is not the zero vector]
```
Computing a basis

Therefore in principle the following algorithm computes a basis for \( \text{Span} \{\mathbf{v}_0, \ldots, \mathbf{v}_n\} \):

```python
def find_basis([\mathbf{v}_0, \ldots, \mathbf{v}_n]):
    "Return the list of nonzero starred vectors."
    [\mathbf{v}^*_0, \ldots, \mathbf{v}^*_n] = orthogonalize([\mathbf{v}_0, \ldots, \mathbf{v}_n])
    return [\mathbf{v}^* for \mathbf{v}^* in [\mathbf{v}^*_0, \ldots, \mathbf{v}^*_n] if \mathbf{v}^* is not the zero vector]
```

Recall

**Lemma:** Every finite set \( T \) of vectors contains a subset \( S \) that is a basis for \( \text{Span} \ T \).

What about finding a subset of \( \mathbf{v}_0, \ldots, \mathbf{v}_n \) that is a basis?

**Proposed algorithm:**

```python
def find_subset_basis([\mathbf{v}_0, \ldots, \mathbf{v}_n]):
    "Return the list of original vectors that correspond to nonzero starred vectors."
    [\mathbf{v}^*_0, \ldots, \mathbf{v}^*_n] = orthogonalize([\mathbf{v}_0, \ldots, \mathbf{v}_n])
    Return [\mathbf{v}_i for i in \{0, \ldots, n\} if \mathbf{v}_i^* is not the zero vector]
```

Is this correct?
Correctness of find_subset_basis

```python
def find_subset_basis([v_0, ..., v_n]):
    [v_0^*, ..., v_n^*] = orthogonalize([v_0, ..., v_n])
    Return [v_i for i in {0, ..., n} if v_i^* is not the zero vector]

def orthogonalize(vlist):
    vstarlist = []
    for v in vlist:
        vstarlist.append(project_orthogonal(v, vstarlist))
    return vstarlist

Example: orthogonalize([v_0, v_1, v_2, v_3, v_4, v_5, v_6]) returns [v_0^*, v_1^*, v_2^*, v_3^*, v_4^*, v_5^*, v_6^*]
```

Suppose v_2^*, v_4^*, and v_5^* are zero vectors.

In iteration 3 iteration of orthogonalize, project_orthogonal(v_3, [v_0^*, v_1^*, v_2^*]) computes v_3^*:

- subtract projection of v_3 along v_0^*,
- subtract projection along v_1^*,
- subtract projection along v_2^*—but since v_2^* = 0, the projection is the zero vector

Result is the same as project_orthogonal(v_3, [v_0^*, v_1^*]). Zero starred vectors are ignored.

Thus orthogonalize([v_0, v_1, v_3, v_6]) would return [v_0^*, v_1^*, v_3^*, v_6^*].

Since [v_0^*, v_1^*, v_3^*, v_6^*] is a basis for \( V = \text{Span} \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\} \)
and [v_0^*, v_1^*, v_3^*, v_6^*] are the same space would be the same cardinality.
Correctness of find_subset_basis

def find_subset_basis([v_0, ..., v_n]):
    [v_0^*, ..., v_n^*] = orthogonalize([v_0, ..., v_n])
    Return [v_i for i in \{0, ..., n\} if v_i^* is not the zero vector]

def orthogonalize(vlist):
    vstarlist = []
    for v in vlist:
        vstarlist.append(
            project_orthogonal(v, vstarlist))
    return vstarlist

Suppose v_2^*, v_4^*, and v_5^* are zero vectors.

In iteration 3 iteration of orthogonalize, project_orthogonal(v_3, [v_0^*, v_1^*, v_2^*])
computes v_3^*:
  ▶ subtract projection of v_3 along v_0^*,
  ▶ subtract projection along v_1^*,
  ▶ subtract projection along v_2^*—but since v_2^* = \textbf{0}, the projection is the zero vector

Result is the same as project_orthogonal(v_3, [v_0^*, v_1^*]). Zero starred vectors are ignored.

Thus orthogonalize([v_0, v_1, v_3, v_6]) would return [v_0^*, v_1^*, v_3^*, v_6^*].

Since [v_0^*, v_1^*, v_3^*, v_6^*] is a basis for V = \text{Span} \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}
and [v_0, v_1, v_3, v_6] spans the same space, and has the same cardinality
[v_0, v_1, v_3, v_6] is also a basis for V.
Correctness of find_subset_basis

Another way to justify find_subset_basis...

Here's the matrix equation expressing original vectors in terms of starred vectors:

\[
\begin{bmatrix}
v_0 & v_1 & v_2 & \cdots & v_n
\end{bmatrix}
= 
\begin{bmatrix}
v_0^* & v_1^* & v_2^* & \cdots & v_n^*
\end{bmatrix}
\begin{bmatrix}
1 & \alpha_{01} & \alpha_{02} & \cdots & \alpha_{0n}
1 & \alpha_{12} & 1 & \cdots & \alpha_{1n}
1 & \alpha_{2n} & 1 & \cdots & 1
\end{bmatrix}
\]
Correctness of find_subset_basis

Let $\mathcal{V} = \text{Span} \{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6\}$. Suppose $\mathbf{v}_2^*, \mathbf{v}_4^*$, and $\mathbf{v}_5^*$ are zero vectors.

$$\begin{align*}
\begin{bmatrix}
\mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 & \mathbf{v}_4 & \mathbf{v}_5 & \mathbf{v}_6
\end{bmatrix}
= \\
\begin{bmatrix}
\mathbf{v}_0^* & \mathbf{v}_1^* & \mathbf{v}_2^* & \mathbf{v}_3^* & \mathbf{v}_4^* & \mathbf{v}_5^* & \mathbf{v}_6^*
\end{bmatrix}
\end{align*}
$$

Delete zero columns and the corresponding rows of the triangular matrix. Shows $\text{Span} \{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6\} \subseteq \text{Span} \{\mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_3^*, \mathbf{v}_6^*\}$ so $\{\mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_3^*, \mathbf{v}_6^*\}$ is a basis for $\mathcal{V}$.

Delete corresponding original columns $\mathbf{v}_2, \mathbf{v}_4, \mathbf{v}_5$.

Resulting triangular matrix is invertible. Move it to other side. Shows $\text{Span} \{\mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_2^*, \mathbf{v}_6^*\} \subseteq \text{Span} \{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_6\}$ so $\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_6\}$ is basis for $\mathcal{V}$.
Correctness of find_subset_basis

Let $\mathcal{V} = \text{Span}\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6\}$.

$$\begin{bmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_3 & \mathbf{v}_6 \end{bmatrix}$$

Delete zero columns and the corresponding rows of the triangular matrix. Shows $\text{Span}\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6\} \subseteq \text{Span}\{\mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_3^*, \mathbf{v}_6^*\}$ so $\{\mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_3^*, \mathbf{v}_6^*\}$ is a basis for $\mathcal{V}$.

Delete corresponding original columns $\mathbf{v}_2, \mathbf{v}_4, \mathbf{v}_5$.

Resulting triangular matrix is invertible. Move it to other side.

Shows $\text{Span}\{\mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_2^*, \mathbf{v}_6^*\} \subseteq \text{Span}\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_6\}$ so $\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_6\}$ is basis for $\mathcal{V}$. 

$$
\begin{bmatrix}
1 & \alpha_{01} & \alpha_{03} & \alpha_{06} \\
1 & \alpha_{13} & \alpha_{16} \\
1 & \alpha_{36} \\
1
\end{bmatrix}
$$
Correctness of find_subset_basis

Let $\mathcal{V} = \text{Span} \{ \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6 \}$.

Delete zero columns and the corresponding rows of the triangular matrix. Shows $\text{Span} \{ \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6 \} \subseteq \text{Span} \{ \mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_3^*, \mathbf{v}_6^* \}$ so $\{ \mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_3^*, \mathbf{v}_6^* \}$ is a basis for $\mathcal{V}$.

Delete corresponding original columns $\mathbf{v}_2, \mathbf{v}_4, \mathbf{v}_5$.

Resulting triangular matrix is invertible. Move it to other side.

Shows $\text{Span} \{ \mathbf{v}_0^*, \mathbf{v}_1^*, \mathbf{v}_2^*, \mathbf{v}_6^* \} \subseteq \text{Span} \{ \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_6 \}$ so $\{ \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_6 \}$ is basis for $\mathcal{V}$.
Roundoff error in computing a basis

In principle the following algorithm computes a basis for $\text{Span} \{ \mathbf{v}_1, \ldots, \mathbf{v}_n \}$:

```python
def find_basis([\mathbf{v}_1, \ldots, \mathbf{v}_n])
    Use orthogonalize to compute $[\mathbf{v}_1^*, \ldots, \mathbf{v}_n^*]$
    Return the list consisting of the nonzero vectors in this list.
```

However: the computer uses floating-point calculations. Due to round-off error, the vectors that are supposed to be zero won’t be exactly zero. Instead, consider a vector $\mathbf{v}$ to be zero if $\mathbf{v} \cdot \mathbf{v}$ is very small (e.g. smaller than $10^{-20}$):

```python
def find_basis([\mathbf{v}_1, \ldots, \mathbf{v}_n])
    Use orthogonalize to compute $[\mathbf{v}_1^*, \ldots, \mathbf{v}_n^*]$
    Return the list consisting of vectors in this list whose squared norms are greater than $10^{-20}$
```

Can use this procedure in turn to define $\text{rank}(\mathbf{vlist})$ and $\text{is\_independent} (\mathbf{vlist})$. Use same idea in other procedures such as $\text{find\_subset\_basis}$.
Algorithm for finding basis for null space

Now let's find null space of matrix with columns $v_1, \ldots, v_n$.

Here's the matrix equation expressing original vectors in terms of starred vectors:

$$
\begin{bmatrix}
v_0 & v_1 & v_2 & \cdots & v_n
\end{bmatrix}
= 
\begin{bmatrix}
v_0^* & v_1^* & v_2^* & \cdots & v_n^*
\end{bmatrix}
\begin{bmatrix}
1 & \alpha_{01} & \alpha_{02} & \cdots & \alpha_{0n} \\
1 & \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\
1 & \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & & & & 1
\end{bmatrix}
$$

Can transform this to express starred vectors in terms of original vectors.

$$
\begin{bmatrix}
v_0 & v_1 & v_2 & \cdots & v_n
\end{bmatrix}
\begin{bmatrix}
1 & \alpha_{01} & \alpha_{02} & \cdots & \alpha_{0n} \\
1 & \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\
1 & \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & & & & 1
\end{bmatrix}^{-1}
= 
\begin{bmatrix}
v_0^* & v_1^* & v_2^* & \cdots & v_n^*
\end{bmatrix}
$$
Basis for null space

\[
\begin{bmatrix}
\mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 & \mathbf{v}_4 & \mathbf{v}_5 & \mathbf{v}_6 \\
\end{bmatrix}
\begin{bmatrix}
1 & \alpha_{01} & \alpha_{02} & \alpha_{03} & \alpha_{04} & \alpha_{05} & \alpha_{06} \\
\alpha_{12} & 1 & \alpha_{13} & \alpha_{14} & \alpha_{15} & \alpha_{16} \\
\alpha_{23} & \alpha_{24} & 1 & \alpha_{25} & \alpha_{26} \\
\alpha_{34} & \alpha_{35} & \alpha_{36} & 1 \\
\alpha_{45} & \alpha_{46} & \alpha_{56} & 1 \\
\end{bmatrix}^{-1}
\]

= \begin{bmatrix}
\mathbf{v}_0^* & \mathbf{v}_1^* & \mathbf{v}_2^* & \mathbf{v}_3^* & \mathbf{v}_4^* & \mathbf{v}_5^* & \mathbf{v}_6^* \\
\end{bmatrix}

Suppose \( \mathbf{v}_2^* \), \( \mathbf{v}_4^* \), and \( \mathbf{v}_5^* \) are (approximately) zero vectors.

- Corresponding columns of inverse triangular matrix are nonzero vectors of the null space of the leftmost matrix.
- These columns are clearly linearly independent so they span a basis of dimension 3.
- Rank-Nullity Theorem shows that the null space has dimension 3 so these columns are a basis for null space.
Orthogonal complement

Let $\mathcal{U}$ be a subspace of $\mathcal{W}$. For each vector $b$ in $\mathcal{W}$, we can write $b = b^\| + b^\perp$ where

- $b^\|$ is in $\mathcal{U}$, and
- $b^\perp$ is orthogonal to every vector in $\mathcal{U}$.

Let $\mathcal{V}$ be the set $\{b^\perp : b \in \mathcal{W}\}$.

**Definition:** We call $\mathcal{V}$ the *orthogonal complement* of $\mathcal{U}$ in $\mathcal{W}$.

**Easy observations:**

- Every vector in $\mathcal{V}$ is orthogonal to every vector in $\mathcal{U}$.
- Every vector $b$ in $\mathcal{W}$ can be written as the sum of a vector in $\mathcal{U}$ and a vector in $\mathcal{V}$.

Maybe $\mathcal{U} \oplus \mathcal{V} = \mathcal{W}$? To show direct sum of $\mathcal{U}$ and $\mathcal{V}$ is defined, we need to show that the only vector that is in both $\mathcal{U}$ and $\mathcal{V}$ is the zero vector.

Any vector $w$ in both $\mathcal{U}$ and $\mathcal{V}$ is orthogonal to itself. Thus $0 = \langle w, w \rangle = \|w\|^2$.

By Property N2 of norms, that means $w = 0$.

Therefore $\mathcal{U} \oplus \mathcal{V} = \mathcal{W}$. **Recall:** $\dim \mathcal{U} + \dim \mathcal{V} = \dim \mathcal{U} \oplus \mathcal{V}$.
**Example:** Let $\mathcal{U} = \text{Span} \{ [1, 1, 0, 0], [0, 0, 1, 1] \}$. Let $\mathcal{V}$ denote the orthogonal complement of $\mathcal{U}$ in $\mathbb{R}^4$. What vectors form a basis for $\mathcal{V}$?

Every vector in $\mathcal{U}$ has the form $[a, a, b, b]$.

Therefore any vector of the form $[c, -c, d, -d]$ is orthogonal to every vector in $\mathcal{U}$.

Every vector in $\text{Span} \{ [1, -1, 0, 0], [0, 0, 1, -1] \}$ is orthogonal to every vector in $\mathcal{U}$. ... so $\text{Span} \{ [1, -1, 0, 0], [0, 0, 1, -1] \}$ is a subspace of $\mathcal{V}$, the orthogonal complement of $\mathcal{U}$ in $\mathbb{R}^4$.

Is it the whole thing?

$\mathcal{U} \oplus \mathcal{V} = \mathbb{R}^4$ so $\dim \mathcal{U} + \dim \mathcal{V} = 4$.

$\{ [1, 1, 0, 0], [0, 0, 1, 1] \}$ is linearly independent so $\dim \mathcal{U} = 2$... so $\dim \mathcal{V} = 2$

$\{ [1, -1, 0, 0], [0, 0, 1, -1] \}$ is linearly independent

so $\dim \text{Span} \{ [1, -1, 0, 0], [0, 0, 1, -1] \}$ is also 2....

so $\text{Span} \{ [1, -1, 0, 0], [0, 0, 1, -1] \} = \mathcal{V}$. 

Example: Find a basis for the null space of $A = \begin{bmatrix} 1 & 0 & 2 & 4 \\ 0 & 5 & 1 & 2 \\ 0 & 2 & 5 & 6 \end{bmatrix}$

By the dot-product definition of matrix-vector multiplication, a vector $v$ is in the null space of $A$ if the dot-product of each row of $A$ with $v$ is zero.

Thus the null space of $A$ equals the orthogonal complement of Row $A$ in $\mathbb{R}^4$.

Since the three rows of $A$ are linearly independent, we know $\dim \text{Row } A = 3$...

so the dimension of the orthogonal complement of Row $A$ in $\mathbb{R}^4$ is $4 - 3 = 1$...

The vector $[1, \frac{1}{10}, \frac{13}{20}, \frac{-23}{40}]$ has a dot-product of zero with every row of $A$...

so this vector forms a basis for the orthogonal complement.

and thus a basis for the null space of $A$. 
Using orthogonalization to find intersection of geometric objects

**Example:** Find the intersection of

- the plane spanned by $[1, 2, -2]$ and $[0, 1, 1]$  
- the plane spanned by $[1, 0, 0]$ and $[0, 1, -1]$

The orthogonal complement in $\mathbb{R}^3$ of the first plane is $\text{Span} \{[4, -1, 1]\}$. Therefore first plane is $\{[x, y, z] \in \mathbb{R}^3 : [4, -1, 1] \cdot [x, y, z] = 0\}$

The orthogonal complement in $\mathbb{R}^3$ of the second plane is $\text{Span} \{[0, 1, 1]\}$. Therefore second plane is $\{[x, y, z] \in \mathbb{R}^3 : [0, 1, 1] \cdot [x, y, z] = 0\}$

The intersection of these two sets is the set $\{[x, y, z] \in \mathbb{R}^3 : [4, -1, 1] \cdot [x, y, z] = 0 \text{ and } [0, 1, 1] \cdot [x, y, z] = 0\}$

Since the annihilator of the annihilator is the original space, a basis for this vector space is a basis for the null space of $A = \begin{bmatrix} 4 & -1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$

The null space of $A$ is the orthogonal complement of $\text{Span} \{[4, -1, 1], [0, 1, 1]\}$ in $\mathbb{R}^3$... which is $\text{Span} \{[1, 2, -2]\}$
Computing the orthogonal complement

Suppose we have a basis \( u_1, \ldots, u_k \) for \( \mathcal{U} \) and a basis \( w_1, \ldots, w_n \) for \( \mathcal{W} \). How can we compute a basis for the orthogonal complement of \( \mathcal{U} \) in \( \mathcal{W} \)?

One way: use \texttt{orthogonalize(vlist)} with

\[
vlist = [u_1, \ldots, u_k, w_1, \ldots, w_n]
\]

Write list returned as \( [u_1^*, \ldots, u_k^*, w_1^*, \ldots, w_n^*] \)

These span the same space as input vectors \( u_1, \ldots, u_k, w_1, \ldots, w_n \), namely \( \mathcal{W} \), which has dimension \( n \).

Therefore exactly \( n \) of the output vectors \( u_1^*, \ldots, u_k^*, w_1^*, \ldots, w_n^* \) are nonzero.

The vectors \( u_1^*, \ldots, u_k^* \) have same span as \( u_1, \ldots, u_k \) and are all nonzero since \( u_1, \ldots, u_k \) are linearly independent.

Therefore exactly \( n - k \) of the remaining vectors \( w_1^*, \ldots, w_n^* \) are nonzero.

Every one of them is orthogonal to \( u_1, \ldots, u_n \)

so they are orthogonal to every vector in \( \mathcal{U} \)

so they lie in the orthogonal complement of \( \mathcal{U} \).

By Direct-Sum Dimension Lemma, orthogonal complement has dimension \( n - k \), so the remaining nonzero vectors are a basis for the orthogonal complement.
Finding basis for null space using orthogonal complement

To find basis for null space of an $m \times n$ matrix $A = \begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix}$, find orthogonal complement of $\text{Span} \{a_1, \ldots, a_m\}$ in $\mathbb{R}^n$:

- Let $e_1, \ldots, e_n$ be the standard basis vectors $\mathbb{R}^n$.
- Let $[a_1^*, \ldots, a_m^*, e_1^*, \ldots, e_n^*] = \text{orthogonalize}([a_1, \ldots, a_m, e_1, \ldots, e_n])$
- Find the nonzero vectors among $e_1^*, \ldots, e_n^*$
Algorithm for finding basis for null space

Another approach to find basis of null space of a matrix: Write matrix in terms of its columns $v_0, \ldots, v_n$.

Here's the matrix equation expressing original vectors in terms of starred vectors:

\[
\begin{bmatrix}
  v_0 & v_1 & v_2 & \cdots & v_n \\
\end{bmatrix}
= \begin{bmatrix}
  v_0^* & v_1^* & v_2^* & \cdots & v_n^* \\
\end{bmatrix}
\begin{bmatrix}
  1 & \alpha_{01} & \alpha_{02} & \cdots & \alpha_{0n} \\
  1 & \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\
  1 & \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\
  \vdots & & & \ddots & \vdots \\
  1 & & & & 1 \\
\end{bmatrix}
\]

Can transform this to express starred vectors in terms of original vectors.

\[
\begin{bmatrix}
  v_0 & v_1 & v_2 & \cdots & v_n \\
\end{bmatrix}
\begin{bmatrix}
  1 & \alpha_{01} & \alpha_{02} & \cdots & \alpha_{0n} \\
  1 & \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\
  1 & \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\
  \vdots & & & \ddots & \vdots \\
  1 & & & & 1 \\
\end{bmatrix}^{-1}
= \begin{bmatrix}
  v_0^* & v_1^* & v_2^* & \cdots & v_n^* \\
\end{bmatrix}
\]
Basis for null space

\[
\begin{bmatrix}
v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6
\end{bmatrix}
= \begin{bmatrix}
v_0^* & v_1^* & v_2^* & v_3^* & v_4^* & v_5^* & v_6^*
\end{bmatrix}
\begin{bmatrix}
1 & \alpha_{01} & \alpha_{02} & \alpha_{03} & \alpha_{04} & \alpha_{05} & \alpha_{06} \\
1 & \alpha_{12} & \alpha_{13} & \alpha_{14} & \alpha_{15} & \alpha_{16} \\
1 & \alpha_{23} & \alpha_{24} & \alpha_{25} & \alpha_{26} \\
1 & \alpha_{34} & \alpha_{35} & \alpha_{36} & \\
1 & \alpha_{45} & \alpha_{46} & \\
1 & \alpha_{56} & 
\end{bmatrix}
\]

Suppose \(v_2^*, v_4^*,\) and \(v_5^*\) are (approximately) zero vectors.

- Corresponding columns of inverse triangular matrix are nonzero vectors of the null space of the leftmost matrix.
- These columns are clearly linearly independent so they span a basis of dimension 3.
- Rank-Nullity Theorem shows that the null space has dimension 3 so these columns are a basis for null space.
Basis for null space

\[
\begin{bmatrix}
  v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6
\end{bmatrix}
\]

\[
\begin{bmatrix}
  1 & \alpha_{01} & \alpha_{02} & \alpha_{03} & \alpha_{04} & \alpha_{05} & \alpha_{06} \\
  1 & \alpha_{12} & \alpha_{13} & \alpha_{14} & \alpha_{15} & \alpha_{16} \\
  1 & \alpha_{23} & \alpha_{24} & \alpha_{25} & \alpha_{26} \\
  1 & \alpha_{34} & \alpha_{35} & \alpha_{36} \\
  1 & \alpha_{45} & \alpha_{46} \\
  1 & \alpha_{56} \\
\end{bmatrix}^{-1}
\]

\[
= \begin{bmatrix}
  v_0^* & v_1^* & v_2^* & v_3^* & v_4^* & v_5^* & v_6^*
\end{bmatrix}
\]

def find_null_space(A):
    vstarlist = orthogonalize(columns of A)
    find upper triangular matrix \( T \) such that
    \[
    A = (\text{matrix with columns } v\text{starlist}) \times T
    \]
    return list of columns of \( T^{-1} \) corresponding to zero vectors in vstarlist

How to find matrix \( T \)? How to find its inverse?
Augmenting orthogonalize(vlist)

We will write a procedure `aug_orthogonalize(vlist)` with the following spec:

- **input**: a list \([\mathbf{v}_1, \ldots, \mathbf{v}_n]\) of vectors
- **output**: the pair \(\left( [\mathbf{v}^*_1, \ldots, \mathbf{v}^*_n], [r_1, \ldots, r_n] \right)\) of lists of vectors such that \(\mathbf{v}^*_1, \ldots, \mathbf{v}^*_n\) are mutually orthogonal vectors whose span equals \(\text{Span} \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}\), and

\[
\begin{bmatrix}
\mathbf{v}_1 & \cdots & \mathbf{v}_n \\
\end{bmatrix}
= 
\begin{bmatrix}
\mathbf{v}^*_1 & \cdots & \mathbf{v}^*_n \\
\end{bmatrix}
\begin{bmatrix}
r_1 & \cdots & r_n \\
\end{bmatrix}
\]

```python
def orthogonalize(vlist):
    vstarlist = []
    for v in vlist:
        vstarlist.append(
            project_orthogonal(v, vstarlist))
    return vstarlist

def aug_orthogonalize(vlist):
    vstarlist = []
    r_vecs = []
    D = set(range(len(vlist)))
    for v in vlist:
        (vstar, alphadict) =
            aug_project_orthogonal(v, vstarlist)
        vstarlist.append(vstar)
        r_vecs.append(Vec(D, alphadict))
    return vstarlist, r_vecs
```
Using `aug_orthogonalize` to find null space

```python
def find_null_space(A):
    vstarlist = orthogonalize(columns of A)
    find upper triangular matrix $T$ such that
    $A$ equals (matrix with columns $vstarlist$) $\times T$
    return list of columns of $T^{-1}$ corresponding to zero vectors in $vstarlist$
```

```python
def find_null_space(A):
    vstarlist, r_vecs = aug_orthogonalize(columns of A)
    let $T$ be matrix with columns given by the vectors of $r_vecs$
    return list of columns of $T^{-1}$ corresponding to zero vectors in $vstarlist$
```

How to find a column of $T^{-1}$?
How to find a column of $T^{-1}$?

$$
\begin{bmatrix}
 v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6
\end{bmatrix}
$$

$$
= \begin{bmatrix}
 v_0^* & v_1^* & v_2^* & v_3^* & v_4^* & v_5^* & v_6^*
\end{bmatrix}
\begin{bmatrix}
 1 & \alpha_{01} & \alpha_{02} & \alpha_{03} & \alpha_{04} & \alpha_{05} & \alpha_{06} \\
 1 & \alpha_{12} & \alpha_{13} & \alpha_{14} & \alpha_{15} & \alpha_{16} \\
 1 & \alpha_{23} & \alpha_{24} & \alpha_{25} & \alpha_{26} \\
 1 & \alpha_{34} & \alpha_{35} & \alpha_{36} \\
 1 & \alpha_{45} & \alpha_{46} \\
 1 & \alpha_{56} \\
 1
\end{bmatrix}
$$

The matrix $T$ is square and upper triangular, with nonzero diagonal elements

$$
\begin{bmatrix}
 T \\
 T^{-1}
\end{bmatrix}
= \begin{bmatrix}
 1 & \ldots \\
 \ldots & 1
\end{bmatrix}
$$

To find column $j$ of $T^{-1}$, solve

$$
\begin{bmatrix}
 T
\end{bmatrix}
\begin{bmatrix}
 x
\end{bmatrix} = e_j
$$

Use `triangular_solve`
Towards QR factorization

We will now develop the *QR factorization*. We will show that certain matrices can be written as the product of matrices in special form. Matrix factorizations are useful mathematically and computationally:

- **Mathematical**: They provide insight into the nature of matrices—each factorization gives us a new way to think about a matrix.
- **Computational**: They give us ways to compute solutions to fundamental computational problems involving matrices.
Matrices with mutually orthogonal columns

\[
\begin{bmatrix}
  \mathbf{v}_1^T \\
  \vdots \\
  \mathbf{v}_n^T
\end{bmatrix}
\begin{bmatrix}
  \mathbf{v}_1^* \\
  \cdots \\
  \mathbf{v}_n^*
\end{bmatrix}
= \begin{bmatrix}
  \|\mathbf{v}_1\|^2 \\
  \vdots \\
  \|\mathbf{v}_n\|^2
\end{bmatrix}
\]

Cross-terms are zero because of mutual orthogonality.
To make the product into the identity matrix, can normalize the columns.

Normalizing a vector means scaling it to make its norm 1.
Just divide it by its norm.

```python
>>> def normalize(v): return v/sqrt(v*v)
>>> q = normalize(list2vec[1,1,1])
>>> q * q
1.0000000000000002
>>> print(q)
0 1 2
------------------
0.577 0.577 0.577
```
Matrices with mutually orthogonal columns

\[
\begin{bmatrix}
\mathbf{v}_1^* T \\
\vdots \\
\mathbf{v}_n^* T \\
\end{bmatrix}
\begin{bmatrix}
\mathbf{v}_1^* & \cdots & \mathbf{v}_n^* \\
\end{bmatrix} = 
\begin{bmatrix}
\|\mathbf{v}_1\|^2 \\
\|\mathbf{v}_n\|^2 \\
\end{bmatrix}
\]

Cross-terms are zero because of mutual orthogonality.
To make the product into the identity matrix, can \textit{normalize} the columns.

Normalize columns \[
\begin{bmatrix}
\mathbf{v}_1^* & \cdots & \mathbf{v}_n^* \\
\end{bmatrix} \Rightarrow \begin{bmatrix}
\mathbf{q}_1 & \cdots & \mathbf{q}_n \\
\end{bmatrix}
\]
Matrices with mutually orthogonal columns

\[
\begin{bmatrix}
q_1^T \\
\vdots \\
q_n^T
\end{bmatrix}
\begin{bmatrix}
q_1 & \cdots & q_n
\end{bmatrix}
= \begin{bmatrix}
1 & \cdots & 1
\end{bmatrix}
\]

Normalize columns

\[
\begin{bmatrix}
v_1^* & \cdots & v_n^*
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
q_1 & \cdots & q_n
\end{bmatrix}
\]
Matrices with mutually orthogonal columns

\[
\begin{bmatrix}
q_1^T \\
\vdots \\
q_n^T
\end{bmatrix}
\begin{bmatrix}
q_1 & \cdots & q_n
\end{bmatrix} =
\begin{bmatrix}
1 \\
\vdots \\
1
\end{bmatrix}
\]

**Proposition:** If columns of \( Q \) are mutually orthogonal with norm 1 then \( Q^T Q \) is identity matrix.

**Definition:** Vectors that are mutually orthogonal and have norm 1 are *orthonormal*.

**Definition:** If columns of \( Q \) are orthonormal then we call \( Q \) a *column-orthogonal* matrix.

**Definition:** If \( Q \) is square and column-orthogonal, we call \( Q \) an *orthogonal* matrix.

**Proposition:** If \( Q \) is an orthogonal matrix then its inverse is \( Q^T \).
Projection onto columns of a column-orthogonal matrix

Suppose \( q_1, \ldots, q_n \) are orthonormal vectors.

Projection of \( b \) onto \( q_j \) is \( b\|q_j = \sigma_j q_j \) where \( \sigma_j = \frac{\langle q_j, b \rangle}{\langle q_j, q_j \rangle} = \langle q_j, b \rangle \)

Vector \( [\sigma_1, \ldots, \sigma_n] \) can be written using dot-product definition of matrix-vector multiplication:

\[
\begin{bmatrix}
\sigma_1 \\
\vdots \\
\sigma_n
\end{bmatrix} =
\begin{bmatrix}
q_1 \cdot b \\
\vdots \\
q_n \cdot b
\end{bmatrix} =
\begin{bmatrix}
q_1^T \\
\vdots \\
q_n^T
\end{bmatrix}
\begin{bmatrix}
\sigma_1 \\
\vdots \\
\sigma_n
\end{bmatrix}
\]

and linear combination \( \sigma_1 q_1 + \cdots + \sigma_n q_n =
\begin{bmatrix}
q_1 \\
\vdots \\
q_n
\end{bmatrix}
\begin{bmatrix}
\sigma_1 \\
\vdots \\
\sigma_n
\end{bmatrix} \)
Towards QR factorization

Orthogonalization of columns of matrix $A$ gives us a representation of $A$ as product of

- matrix with mutually orthogonal columns
- invertible triangular matrix

$$
\begin{bmatrix}
v_1 & v_2 & v_3 & \cdots & v_n
\end{bmatrix}
= 
\begin{bmatrix}
v_1^* & v_2^* & v_3^* & \cdots & v_n^*
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & \cdots & \alpha_{12} & \alpha_{13} & \cdots & \alpha_{1n}
0 & 1 & 0 & \cdots & \alpha_{23} & 1 & \cdots & \alpha_{2n}
0 & 0 & 1 & \cdots & \alpha_{3n} & \cdots & \alpha_{3n} & 1
\end{bmatrix}
$$

Suppose columns $v_1, \ldots, v_n$ are linearly independent. Then $v_1^*, \ldots, v_n^*$ are nonzero.

- Normalize $v_1^*, \ldots, v_n^*$ (Matrix is called $Q$)
- To compensate, scale the rows of the triangular matrix. (Matrix is $R$)

The result is the QR factorization.

$Q$ is a column-orthogonal matrix and $R$ is an upper-triangular matrix.
Towards QR factorization

Orthogonalization of columns of matrix $A$ gives us a representation of $A$ as product of
- matrix with mutually orthogonal columns
- invertible triangular matrix

$$
\begin{bmatrix}
  v_1 & v_2 & v_3 & \cdots & v_n
\end{bmatrix}
= \begin{bmatrix}
  q_1 & q_2 & q_3 & \cdots & q_n
\end{bmatrix}
\begin{bmatrix}
  \|v_1^*\| & \beta_{12} & \beta_{13} & \cdots & \beta_{1n} \\
  \|v_2^*\| & \beta_{23} & \cdots & \cdots & \beta_{2n} \\
  \|v_3^*\| & \cdots & \cdots & \cdots & \beta_{3n} \\
  \|v_n^*\| & \cdots & \cdots & \cdots & \cdots
\end{bmatrix}
$$

Suppose columns $v_1, \ldots, v_n$ are linearly independent. Then $v_1^*, \ldots, v_n^*$ are nonzero.
- Normalize $v_1^*, \ldots, v_n^*$ (Matrix is called $Q$)
- To compensate, scale the rows of the triangular matrix. (Matrix is $R$)

The result is the QR factorization.
$Q$ is a column-orthogonal matrix and $R$ is an upper-triangular matrix.
Using the QR factorization to solve a matrix equation $Ax = b$

First suppose $A$ is square and its columns are linearly independent. Then $A$ is invertible.
It follows that there is a solution (because we can write $x = A^{-1}b$)

**QR Solver Algorithm** to find the solution in this case:

- Find $Q, R$ such that $A = QR$ and $Q$ is column-orthogonal and $R$ is triangular
- Compute vector $c = Q^Tb$
- Solve $R = c$ using backward substitution, and return the solution.

Why is this correct?

- Let $\hat{x}$ be the solution returned by the algorithm.
- We have $R\hat{x} = Q^Tb$
- Multiply both sides by $Q$: $Q(R\hat{x}) = Q(Q^Tb)$
- Use associativity: $(QR)\hat{x} = (QQ^T)b$
- Substitute $A$ for $QR$: $A\hat{x} = (QQ^T)b$
- Since $Q$ and $Q^T$ are inverses, we know $QQ^T$ is identity matrix: $A\hat{x} = 1b$

Thus $A\hat{x} = b$. 
Solving $Ax = b$

What if columns of $A$ are not independent?

Let $v_1, v_2, v_3, v_4$ be columns of $A$.

Suppose $v_1, v_2, v_3, v_4$ are linearly dependent.

Then there is a basis consisting of a subset, say $v_1, v_2, v_4$

$$\left\{ \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} : x_1, x_2, x_3, x_4 \in \mathbb{R} \right\} =$$

$$\left\{ \begin{bmatrix} v_1 & v_2 \\ v_1 & v_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_4 \end{bmatrix} : x_1, x_2, x_4 \in \mathbb{R} \right\}$$

Therefore: if there is a solution to $Ax = b$ then there is a solution to $A'x' = b$ where columns of $A'$ are a subset basis of columns of $A$ (and $x'$ consists of corresponding variables).
The least squares problem

Suppose $A$ is an $m \times n$ matrix and its columns are linearly independent.

Since each column is an $m$-vector, dimension of column space is at most $m$, so $n \leq m$.

What if $n < m$? How can we solve the matrix equation $Ax = b$?

Remark: There might not be a solution:
- Define $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ by $f(x) = Ax$
- Dimension of $\text{Im } f$ is $n$
- Dimension of co-domain is $m$
- Thus $f$ is not onto.

Goal: An algorithm that, given equation $Ax = b$, where columns are linearly independent, finds the vector $\hat{x}$ minimizing $\|b - A\hat{x}\|$.

Solution: Same algorithm as we used for square $A$
Recall...

**High-Dimensional Fire Engine Lemma:** The point in a vector space $\mathcal{V}$ closest to $b$ is $b^{\parallel \mathcal{V}}$ and the distance is $\|b^{\perp \mathcal{V}}\|$. 

Given equation $A\mathbf{x} = \mathbf{b}$, let $\mathcal{V}$ be the column space of $A$.

We need to show that the QR Solver Algorithm returns the vector $\hat{\mathbf{x}}$ such that $A\hat{\mathbf{x}} = b^{\parallel \mathcal{V}}$. 
The least squares problem

Suppose $A$ is an $m \times n$ matrix and its columns are linearly independent.

Since each column is an $m$-vector, dimension of column space is at most $m$, so $n \leq m$.

What if $n < m$? How can we solve the matrix equation $Ax = b$?

Remark: There might not be a solution:
- Define $f : \mathbb{R}^n \to \mathbb{R}^m$ by $f(x) = Ax$
- Dimension of $\text{Im} f$ is $n$
- Dimension of co-domain is $m$.
- Thus $f$ is not onto.

Goal: An algorithm that, given a matrix $A$ whose columns are linearly independent and given $b$, finds the vector $\hat{x}$ minimizing $\|b - A\hat{x}\|$.

Solution: Same algorithm as we used for square $A$
The least squares problem

Recall...

**High-Dimensional Fire Engine Lemma:** The point in a vector space $V$ closest to $b$ is $b\|V$ and the distance is $\|b\perp V\|$.

Given equation $Ax = b$, let $V$ be the column space of $A$.

We need to show that the QR Solver Algorithm returns $b\|V$. 
Representation of $b^\parallel$ in terms of columns of $Q$

Let $Q$ be a column-orthogonal matrix. Let $b$ be a vector, and write $b = b^\parallel + b^\perp$ where $b^\parallel$ is projection of $b$ onto Col $Q$ and $b^\perp$ is projection orthogonal to Col $Q$.

Let $u$ be the coordinate representation of $b^\parallel$ in terms of columns of $Q$.

By linear-combinations definition of matrix-vector multiplication,

$$
\begin{bmatrix}
  \parallel b \\
\end{bmatrix} = \begin{bmatrix}
  Q \\
\end{bmatrix} \begin{bmatrix}
  u \\
\end{bmatrix}
$$

Multiply both sides on the left by $Q^T$:

$$
\begin{bmatrix}
  Q^T \\
\end{bmatrix} \begin{bmatrix}
  \parallel b \\
\end{bmatrix} = \begin{bmatrix}
  Q^T \\
\end{bmatrix} \begin{bmatrix}
  Q \\
\end{bmatrix} \begin{bmatrix}
  u \\
\end{bmatrix}
$$
QR Solver Algorithm for $Ax \approx b$

**Summary:**
- $QQ^Tb = b$\|$

**Proposed algorithm:**

Find $Q, R$ such that $A = QR$ and $Q$ is column-orthogonal and $R$ is triangular
Compute vector $c = Q^Tb$
Solve $Rx = c$ using backward substitution, and return the solution $\hat{x}$.

**Goal:** To show that the solution $\hat{x}$ returned is the vector that minimizes $\|b - A\hat{x}\|$.

Every vector of the form $Ax$ is in Col $A$ ($= \text{Col } Q$)

By the High-Dimensional Fire Engine Lemma, the vector in Col $A$ closest to $b$ is $b$\|, the projection of $b$ onto Col $A$.

Solution $\hat{x}$ satisfies $R\hat{x} = Q^Tb$

Multiply by $Q$: $QR\hat{x} = QQ^Tb$

Therefore $A\hat{x} = b$\|
The Normal Equations

Let $A$ be a matrix with linearly independent columns. Let $QR$ be its QR factorization. We have given one algorithm for solving the least-squares problem $Ax \approx b$:

Find $Q, R$ such that $A = QR$ and $Q$ is column-orthogonal and $R$ is triangular
Compute vector $c = Q^T b$
Solve $Rx = c$ using backward substitution, and return the solution $\hat{x}$.

However, there are other ways to find solution.

Not hard to show that

- $A^T A$ is an invertible matrix
- The solution to the matrix-vector equation $(A^T A)x = A^T b$ is the solution to the least-squares problem $Ax \approx b$
- Can use another method (e.g. Gaussian elimination) to solve $(A^T)x = A^T b$

The linear equations making up $A^T A x = A^T b$ are called the normal equations
Application of least squares: linear regression

Finding the line that best fits some two-dimensional data.

Data on age versus brain mass from the Bureau of Made-up Numbers:

<table>
<thead>
<tr>
<th>age</th>
<th>brain mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>4 lbs.</td>
</tr>
<tr>
<td>55</td>
<td>3.8</td>
</tr>
<tr>
<td>65</td>
<td>3.75</td>
</tr>
<tr>
<td>75</td>
<td>3.5</td>
</tr>
<tr>
<td>85</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Let $f(x)$ be the function that predicts brain mass for someone of age $x$.

Hypothesis: after age 45, brain mass decreases linearly with age, i.e. that $f(x) = mx + b$ for some numbers $m, b$.

**Goal:** find $m, b$ to as to minimize the sum of squares of prediction errors

The observations are $(x_1, y_1) = (45, 4), (x_2, y_2) = (55, 3.8), (x_3, y_3) = (64, 3.75), (x_4, y_4) = (75, 3.5), (x_5, y_5) = (85, 3.3)$.

The prediction error on the the $i^{th}$ observation is $|f(x_i) - y_i|$.

The sum of squares of prediction errors is $\sum_i (f(x_i) - y_i)^2$.

For each observation, measure the difference between the predicted and observed $y$-value.

In this application, this difference is measured in pounds.

Measuring the distance from the point to the line wouldn’t make sense.
Application of least squares: linear regression

Finding the line that best fits some two-dimensional data.

Data on age versus brain mass from the Bureau of Made-up Numbers:

<table>
<thead>
<tr>
<th>age</th>
<th>brain mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>4 lbs.</td>
</tr>
<tr>
<td>55</td>
<td>3.8</td>
</tr>
<tr>
<td>65</td>
<td>3.75</td>
</tr>
<tr>
<td>75</td>
<td>3.5</td>
</tr>
<tr>
<td>85</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Let \( f(x) \) be the function that predicts brain mass for someone of age \( x \).

Hypothesis: after age 45, brain mass decreases linearly with age, i.e. that \( f(x) = mx + b \) for some numbers \( m, b \).

**Goal:** find \( m, b \) to as to minimize the sum of squares of prediction errors

The observations are \((x_1, y_1) = (45, 4), (x_2, y_2) = (55, 3.8), (x_3, y_3) = (64, 3.75), (x_4, y_4) = (75, 3.5), (x_5, y_5) = (85, 3.3)\).

The prediction error on the the \( i^{th} \) observation is \( |f(x_i) - y_i| \).

The sum of squares of prediction errors is \( \sum_i (f(x_i) - y_i)^2 \).

For each observation, measure the difference between the predicted and observed \( y \)-value. In this application, this difference is measured in pounds. Measuring the distance from the point to the line wouldn’t make sense.
Application of least squares: linear regression

Finding the line that best fits some two-dimensional data.

Data on age versus brain mass from the Bureau of Made-up Numbers:

<table>
<thead>
<tr>
<th>age</th>
<th>brain mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>4 lbs.</td>
</tr>
<tr>
<td>55</td>
<td>3.8</td>
</tr>
<tr>
<td>65</td>
<td>3.75</td>
</tr>
<tr>
<td>75</td>
<td>3.5</td>
</tr>
<tr>
<td>85</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Let \( f(x) \) be the function that predicts brain mass for someone of age \( x \).

Hypothesis: after age 45, brain mass decreases linearly with age, i.e. that \( f(x) = mx + b \) for some numbers \( m, b \).

**Goal:** find \( m, b \) to as to minimize the sum of squares of prediction errors.

The observations are \((x_1, y_1) = (45, 4), (x_2, y_2) = (55, 3.8), (x_3, y_3) = (64, 3.75), (x_4, y_4) = (75, 3.5), (x_5, y_5) = (85, 3.3)\).

The prediction error on the the \( i^{th} \) observation is \(|f(x_i) - y_i|\).

The sum of squares of prediction errors is \( \sum_i(f(x_i) - y_i)^2 \).

For each observation, measure the difference between the predicted and observed \( y \)-value.

In this application, this difference is measured in pounds.

Measuring the distance from the point to the line wouldn’t make sense.
Linear regression

To find the best line for given data \((x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4), (x_5, y_5)\), solve this least-squares problem:

\[
\begin{bmatrix}
  x_1 & 1 \\
  x_2 & 1 \\
  x_3 & 1 \\
  x_4 & 1 \\
  x_5 & 1 \\
\end{bmatrix}
\begin{bmatrix}
  m \\
  b \\
\end{bmatrix}
\approx
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
  y_5 \\
\end{bmatrix}
\]

The dot-product of row \(i\) with the vector \([m, b]\) is \(mx_i + b\), i.e. the value predicted by \(f(x) = mx + b\) for the \(i^{th}\) observation.

Therefore, the vector of predictions is \(A\begin{bmatrix} m \\ b \end{bmatrix}\).

The vector of differences between predictions and observed values is \(A\begin{bmatrix} m \\ b \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}\), and the sum of squares of differences is the squared norm of this vector. Therefore the method of least squares can be used to find the pair \((m, b)\) that minimizes the sum of squares, i.e. the line that best fits the data.
Application of least squares: coping with approximate data

Recall the *industrial espionage* problem: finding the number of each product being produced from the amount of each resource being consumed.

Let \[ M = \begin{pmatrix}
    \text{garden gnome} & 0 & 1.3 & .2 & .8 & .4 \\
    \text{hula hoop} & 0 & 0 & 1.5 & .4 & .3 \\
    \text{slinky} & .25 & 0 & 0 & .2 & .7 \\
    \text{silly putty} & 0 & 0 & .3 & .7 & .5 \\
    \text{salad shooter} & .15 & 0 & .5 & .4 & .8 \\
\end{pmatrix} \]

We solved \( u^T M = b \) where \( b \) is vector giving amount of each resource consumed:

\[
b = \begin{pmatrix}
    \text{metal} & \text{concrete} & \text{plastic} & \text{water} & \text{electricity} \\
    226.25 & 1300 & 677 & 1485 & 1409.5
\end{pmatrix}
\]

\( \text{solve}(M^T, b) \) gives us \( u \approx \begin{pmatrix}
    \text{gnome} & \text{hoop} & \text{slinky} & \text{putty} & \text{shooter} \\
    1000 & 175 & 860 & 590 & 75
\end{pmatrix} \)
Application of least squares: industrial espionage problem

More realistic scenario: measurement of resources consumed is approximate

**True amounts:** \( \mathbf{b} = \begin{pmatrix} \text{metal} & \text{concrete} & \text{plastic} & \text{water} & \text{electricity} \\ 226.25 & 1300 & 677 & 1485 & 1409.5 \end{pmatrix} \)

Solving with true amounts gives \( \mathbf{\hat{b}} = \begin{pmatrix} \text{gnome} & \text{hoop} & \text{slinky} & \text{putty} & \text{shooter} \\ 1000 & 175 & 860 & 590 & 75 \end{pmatrix} \)

**Measurements:** \( \tilde{\mathbf{b}} = \begin{pmatrix} \text{metal} & \text{concrete} & \text{plastic} & \text{water} & \text{electricity} \\ 223.23 & 1331.62 & 679.32 & 1488.69 & 1492.64 \end{pmatrix} \)

Solving with measurements gives \( \tilde{\mathbf{\hat{b}}} = \begin{pmatrix} \text{gnome} & \text{hoop} & \text{slinky} & \text{putty} & \text{shooter} \\ 1024.32 & 28.85 & 536.32 & 446.7 & 594.34 \end{pmatrix} \)

Slight changes in input data leads to pretty big changes in output.

**Output data not accurate, perhaps not useful!** (see slinky, shooter)

**Question:** How can we improve accuracy of output without more accurate measurements?

**Answer:** More measurements!
Application of least squares: industrial espionage problem

Have to measure something else, e.g. **amount of waste water produced**

<table>
<thead>
<tr>
<th></th>
<th>metal</th>
<th>concrete</th>
<th>plastic</th>
<th>water</th>
<th>electricity</th>
<th>waste water</th>
</tr>
</thead>
<tbody>
<tr>
<td>garden gnome</td>
<td>0</td>
<td>1.3</td>
<td>.2</td>
<td>.8</td>
<td>.4</td>
<td>.3</td>
</tr>
<tr>
<td>hula hoop</td>
<td>0</td>
<td>0</td>
<td>1.5</td>
<td>.4</td>
<td>.3</td>
<td>.35</td>
</tr>
<tr>
<td>slinky</td>
<td>.25</td>
<td>0</td>
<td>0</td>
<td>.2</td>
<td>.7</td>
<td>0</td>
</tr>
<tr>
<td>silly putty</td>
<td>0</td>
<td>0</td>
<td>.3</td>
<td>.7</td>
<td>.5</td>
<td>.2</td>
</tr>
<tr>
<td>salad shooter</td>
<td>.15</td>
<td>0</td>
<td>.5</td>
<td>.4</td>
<td>.8</td>
<td>.15</td>
</tr>
</tbody>
</table>

Measured: \( \mathbf{\tilde{b}} = \begin{pmatrix} \text{metal} \\ \text{concrete} \\ \text{plastic} \\ \text{water} \\ \text{electricity} \\ \text{waste water} \end{pmatrix} = \begin{pmatrix} 223.23 \\ 1331.62 \\ 679.32 \\ 1488.69 \\ 1492.64 \\ 489.19 \end{pmatrix} \)

Equation \( \mathbf{u} \ast M = \mathbf{\tilde{b}} \) is more constrained \( \Rightarrow \) has **no solution**

but least-squares solution is \( \mathbf{\tilde{u}} = \begin{pmatrix} \text{gnome} \\ \text{hoop} \\ \text{slinky} \\ \text{putty} \\ \text{shooter} \end{pmatrix} = \begin{pmatrix} 1022.26 \\ 191.8 \\ 1005.58 \\ 549.63 \\ 41.1 \end{pmatrix} \)

True amounts: \( \mathbf{\tilde{u}} = \begin{pmatrix} \text{gnome} \\ \text{hoop} \\ \text{slinky} \\ \text{putty} \\ \text{shooter} \end{pmatrix} = \begin{pmatrix} 1000 \\ 175 \\ 860 \\ 590 \\ 75 \end{pmatrix} \)

Better output accuracy with same input accuracy
Application of least squares: Sensor node problem

Recall *sensor node problem*: estimate current draw for each hardware component

Define \( D = \{\text{'radio'}, \text{'sensor'}, \text{'memory'}, \text{'CPU'}\} \).

**Goal:** Compute a \( D \)-vector \( u \) that, for each hardware component, gives the current drawn by that component.

**Four test periods:**

- total mA-seconds in these test periods \( b = [140, 170, 60, 170] \)
- for each test period, vector specifying how long each hardware device was operating:
  
  \[
  \begin{align*}
  \text{duration}_1 &= \text{Vec}(D, \ 'radio':0.1, \ 'CPU':0.3) \\
  \text{duration}_2 &= \text{Vec}(D, \ 'sensor':0.2, \ 'CPU':0.4) \\
  \text{duration}_3 &= \text{Vec}(D, \ 'memory':0.3, \ 'CPU':0.1) \\
  \text{duration}_4 &= \text{Vec}(D, \ 'memory':0.5, \ 'CPU':0.4)
  \end{align*}
  \]

To get \( u \), solve \( Ax = b \) where

\[
A = \begin{bmatrix}
\text{duration}_1 \\
\text{duration}_2 \\
\text{duration}_3 \\
\text{duration}_4
\end{bmatrix}
\]
Application of least squares: Sensor node problem

If measurements are exact, get back true current draw for each hardware component:

$$\mathbf{b} = [140, 170, 60, 170]$$

solve $\mathbf{A}\mathbf{x} = \mathbf{b}$

<table>
<thead>
<tr>
<th>radio</th>
<th>sensor</th>
<th>CPU</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>250</td>
<td>300</td>
<td>100</td>
</tr>
</tbody>
</table>

More realistic: approximate measurement

$$\tilde{\mathbf{b}} = [141.27, 160.59, 62.47, 181.25]$$

solve $\mathbf{A}\mathbf{x} = \tilde{\mathbf{b}}$

<table>
<thead>
<tr>
<th>radio</th>
<th>sensor</th>
<th>CPU</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>421</td>
<td>142</td>
<td>331</td>
<td>98.1</td>
</tr>
</tbody>
</table>

How can we get more accurate results?

**Solution:** Add more test periods and solve least-squares problem
Application of least squares: Sensor node problem

\[ \text{duration}_1 = \text{Vec}(D, \text{'radio': 0.1, 'CPU': 0.3}) \]
\[ \text{duration}_2 = \text{Vec}(D, \text{'sensor': 0.2, 'CPU': 0.4}) \]
\[ \text{duration}_3 = \text{Vec}(D, \text{'memory': 0.3, 'CPU': 0.1}) \]
\[ \text{duration}_4 = \text{Vec}(D, \text{'memory': 0.5, 'CPU': 0.4}) \]
\[ \text{duration}_5 = \text{Vec}(D, \text{'radio': 0.2, 'CPU': 0.5}) \]
\[ \text{duration}_6 = \text{Vec}(D, \text{'sensor': 0.3, 'radio': 0.8, 'CPU': 0.9, 'memory': 0.8}) \]
\[ \text{duration}_7 = \text{Vec}(D, \text{'sensor': 0.5, 'radio': 0.3, 'CPU': 0.9, 'memory': 0.5}) \]
\[ \text{duration}_8 = \text{Vec}(D, \text{'radio': 0.2, 'CPU': 0.6}) \]

Measurement vector is \( \tilde{\mathbf{b}} = \begin{bmatrix} 141.27, 160.59, 62.47, 181.25, 247.74, 804.58, 609.10, 282.09 \end{bmatrix} \)

Let \( \mathbf{A} = \begin{bmatrix} \text{duration}_1 \\ \text{duration}_2 \\ \text{duration}_3 \\ \text{duration}_4 \\ \text{duration}_5 \\ \text{duration}_6 \\ \text{duration}_7 \\ \text{duration}_8 \end{bmatrix} \)

Now \( \mathbf{Ax} = \tilde{\mathbf{b}} \) has no solution

But solution to least-squares problem is

<table>
<thead>
<tr>
<th></th>
<th>radio</th>
<th>sensor</th>
<th>CPU</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>451.40</td>
<td>252.07</td>
<td>314.37</td>
<td>111.66</td>
</tr>
</tbody>
</table>

True solution is

<table>
<thead>
<tr>
<th></th>
<th>radio</th>
<th>sensor</th>
<th>CPU</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>500</td>
<td>250</td>
<td>300</td>
<td>100</td>
</tr>
</tbody>
</table>
Applications of least squares: breast cancer machine-learning problem

**Recall:** breast-cancer machine-learning lab

**Input:** vectors $a_1, \ldots, a_m$ giving features of specimen, values $b_1, \ldots, b_m$ specifying +1 (malignant) or -1 (benign)

**Informal goal:** Find vector $w$ such that sign of $a_i \cdot w$ predicts sign of $b_i$

**Formal goal:** Find vector $w$ to minimize sum of squared errors
\[
(b_1 - a_1 \cdot w)^2 + \cdots + (b_m - a_m \cdot w)^2
\]

**Approach:** Gradient descent

**Results:** Took a few minutes to get a solution with error rate around 7%

Can we do better with least squares?
Applications of least squares: breast cancer machine-learning problem

**Goal:** Find the vector $\mathbf{w}$ that minimizes $(\mathbf{b}[1] - a_1 \cdot \mathbf{w})^2 + \cdots + (\mathbf{b}[m] - a_m \cdot \mathbf{w})^2$

**Equivalent:** Find the vector $\mathbf{w}$ that minimizes $\left\| \begin{bmatrix} \mathbf{b} \\ \vdots \\ \mathbf{a}_m \end{bmatrix} - \begin{bmatrix} a_1 \\ \vdots \\ a_m \end{bmatrix} \begin{bmatrix} \mathbf{x} \end{bmatrix} \right\|^2$

This is the least-squares problem.

Using the algorithm based on QR factorization takes a *fraction of a second* and gets a solution with *smaller error rate*.

Even better solutions using more sophisticated techniques in linear algebra:

- Use an inner product that better reflects the variance of each of the features.
- Use *linear programming*
- Even more general: use *convex programming*