# The Dot Product



- The "dot product" is a key concept! It is central to geometric interpretations of matrix operations. Two vectors are orthogonal if their dot product is zero

$$\underline{\mathbf{x}}_1 \cdot \underline{\mathbf{x}}_2 = \sum_{i=1}^{n} \underline{\mathbf{x}}_1(i)\underline{\mathbf{x}}_2(i)$$

- We also have:

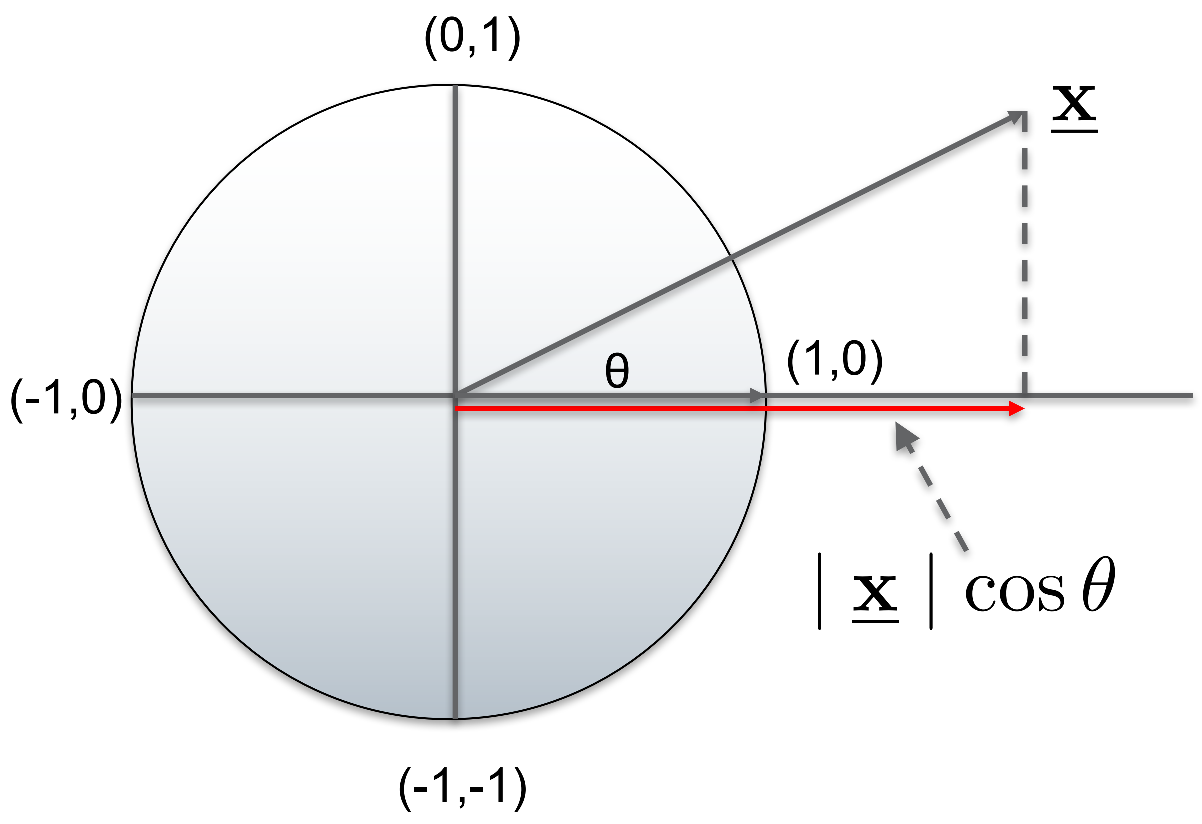
$$|\ \underline{\mathbf{x}}\ | = \sqrt{\underline{\mathbf{x}} \cdot \underline{\mathbf{x}}}$$

$$a\underline{\mathbf{x}} \cdot b\underline{\mathbf{y}} = (ab)(\underline{\mathbf{x}} \cdot \underline{\mathbf{y}}) = ab\underline{\mathbf{x}} \cdot \underline{\mathbf{y}}$$

# Dot Product and Projection



- If $\underline{u}$ is a unit vector, the dot product $\underline{x} \cdot \underline{u}$ tells us the projection of $\underline{x}$ in the direction of $\underline{u}$



# Orthonormal Basis



- **A set of vectors forms an <u>orthonormal basis</u> if**
  - Each vector is a unit vector (magnitude 1)
  - Each vector is orthogonal to every other vector

Mathematically,

$$\underline{\mathbf{u}}_i \cdot \underline{\mathbf{u}}_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{else} \end{cases}$$

# Transforms are Simple! Really!



- A transform like the DFT (and there are many, many different transforms) is defined by a set of basis vectors
- The vectors are orthonormal
- Any vector can be projected onto each of these basis vectors to get the transformed values, i.e., transform coefficients
- NOTE: the dot product of $\underline{x}$ with $\underline{y}$ when $\underline{x}$ and $\underline{y}$ are complex valued is $\underline{x}$ "dot" $\underline{y}^*$ where $\underline{y}^*$ is the complex conjugate vector (conjugate each component)

# Matrix description of a Transform



- **Transforms ("change of basis") can be written as a matrix equation**
  - Make the basis vectors the rows of a "transform matrix"

$$\underline{\mathbf{c}} = A\underline{\mathbf{x}}$$

$$= \begin{bmatrix} \underline{\mathbf{u}}_1^T \\ \underline{\mathbf{u}}_2^T \\ \vdots \\ \underline{\mathbf{u}}_N^T \end{bmatrix} \underline{\mathbf{x}}$$

The transform coefficients

Vector being "transformed" (i.e. for which we are computing projections onto a new basis set)

  - Every set of orthonormal basis vectors forms a "transform" via a matrix multiply

# Some DFT insights



- **For the DFT, the basis vectors are complex sinusoids**
- **We use N/2+1 DFT coefficients for real-valued signals**
  - Complex conjugacy
- **The spectrogram (STFT)**
  - A series of spectral plots presented one after the other
    - ✓ A pure sinusoid appears as a straight line
  - Each spectral plot is one frame of data
  - The frames are “short” and may overlap

# WAV format for audio



- **Header contains**
  - Sampling frequency
    - ✓ 44.1 kHz, 22.05 kHz, 11.025 kHz, 8 kHz are common
  - Number of channels
    - ✓ 1 and 2 channels are common
  - Number of samples
    - ✓ A “sample” is <u>all</u> the data values for every channel
  - Bits per audio data value
    - ✓ 16 bits and 8 bits are common
      - – 16 bit signed integer format covers -32768 to 32767

# WAV file IO



- **Reading and writing a wav file**

Code

```
bombay% cat zuppy.py
#!/usr/local/bin/python3
import scipy.io.wavfile as spwav

###########
# Here is our main block of code.  Execution starts here
###########

Fs, b = spwav.read("test.wav")
print(f"the sampling rate is {Fs}")
print(f"the number of audio samples is {len(b)}")
print(f"b[50000] = {b[50000]}, b[100000] = {b[100000]}")
print(f"type( b ) = {type(b)}")
print(f"type( b[50000] ) = {type( b[50000] )}")
spwav.write("newtest.wav", Fs, b)
```

Output

```
bombay%
bombay% zuppy.py
the sampling rate is 11025
the number of audio samples is 2317404
b[50000] = -2462, b[100000] = 1767
type( b ) = <class 'numpy.ndarray'>
type( b[50000] ) = <class 'numpy.int16'>
bombay%
```

# Lab 1 Hints



- **Getting a list of files in a directory**
  - If you include “import glob” at the top of your python program then you can get a list of files matching some criteria with a call like this:
    - ✓ needed_files = glob.glob(“*.wav”)
  - Extracting 24 seconds of audio
    - ✓ Compute the sample at which to start extracting via:
      - – num_samps // 2
    - ✓ We want the number of samples extracted to be a multiple of 512 to simplify later analysis. If you extract 516 frames of size 512, this is nearly 24 seconds of data (512*516/11025 = 23.96 s)

# More Hints



- **Don't use a for loop to do the extraction!**
  - Learn to use python "slicing". For example, if x is a 1-D numpy array, then x[8:15] is a new array with x[8] to x[14] inclusive
    - ✓ x[:15] gives all values up to x[14]
    - ✓ x[25:] gives all samples from x[25] to the end
  - I recommend reshaping your 1-D array of audio samples into an array of size 516 x 512
    - ✓ Look at the command numpy.reshape()
    - ✓ If x is a re-shaped 2D array, then x[n] gives you the 512 samples of frame n
      - Note: this means python lets you drop the second index entirely when what you are after is x[n,:]

# More Hints



- **Minimize for() loops!**
  - You can do all of your frame calculations for time domain and spectral features using a single for() loop that indexes frames
    - ✓ You <u>don't</u> need a nested for() loop that looks at individual samples
    - ✓ If you use nested for() loops you <u>aren't</u> using the power of numpy/python (your grade will reflect that)
- **Slicing with -n**
  - x[:-1] gives an array with all samples in x <u>except</u> for the last sample. This will help with ZCR calculations
    - ✓ This syntax extends to x[:-n] when you want to drop n samples off the end

# More Hints



- **Spectrogram calculation (IMPORTANT)**
  - Import this module
    - ✓ “import scipy.signal as spsig”
  - Computing a window
    - ✓ blackman_window = spsig.get_window("blackman", N, False)
  - Use “spsig.spectrogram()”
    - ✓ See the documentation on the web
    - ✓ You can specify the window, for example, ‘blackman’, when invoking the function
    - ✓ use: mode=‘magnitude’
  - Do your plots using dB not linear!
    - ✓ See the sample code on the next slide

# More Hints



- **Spectrogram calculation and graphing examples**
  - Make sure you know what np.where( ) is doing below. It is a super useful function for modifying arrays (also, since I used a 2D array, I ”flattened” it before doing the call)

```
f, t, Sxx = spsig.spectrogram( tmpdat.flatten(), Fs, 'blackman', N, \
                               mode='magnitude')
```

```
Sxx = np.where(Sxx < 10**-3, 10**-3, Sxx)
Sxx = 20 * np.log10( Sxx / np.amax(Sxx) )

plt.figure(3)
# cmaps of interest: inferno, BuPu, coolwarm, bwr, Reds, gray, plasma
plt.pcolormesh(t, f, Sxx, cmap='coolwarm')
plt.ylabel("Frequency (Hz)" )
plt.xlabel("time (s)")
print(f"shape of Sxx is {Sxx.shape}")

plt.show()
```

# More Hints



- **Generating multiple plots**
  - You can generate multiple plots in separate windows by inserting ”plt.figure(n)” in front of the commands for graph n, and putting a single plt.show() command after all figures you want have been constructed
  - You can generate multiple plots in a single window using ”plt.subplot()”

# More Hints



- **Don't forget to apply the blackman filter to each frame of data when computing the spectral features**
- **ERROR IN WRITEUP: the size of your array for the mel filter banks should be Nb by K+1 (the writeup says Nb x K )**
- **The frequency corresponding to the k'th coefficient when doing a DFT is k*Fs/N**
- **The graph in the lab writeup of the frequency responses is WRONG. It gives the right basic shape, however, the heights are wrong**