# Digital Signal Processing

## Lecture 8 – Discrete Cosine Transform

상명대학교
컴퓨터과학과
강상욱 교수

# DCT vs. DFT

- DCT is similar to DFT, which has been used for spectral analysis.
- DFT
  - DFT is designed for processing complex-valued waveforms, and is always produce a complex-valued spectrum.
  - In DFT, cosine (real part) or sine (imaginary part) functions alone do not constitute a complete set of basis functions.
  - A real-valued signal has a symmetric Fourier spectrum, so only one half of the spectral coefficients need to be computed without losing any signal information.
- DCT (Discrete Cosine Transform)
  - Uses only cosine functions of various frequencies as basis functions
  - To compress data in MP3, JPEG and MPEG.
  - Operates on real-valued signals and spectral coefficients.
  - There is also a discrete sine transform (DST).

# DCT formula

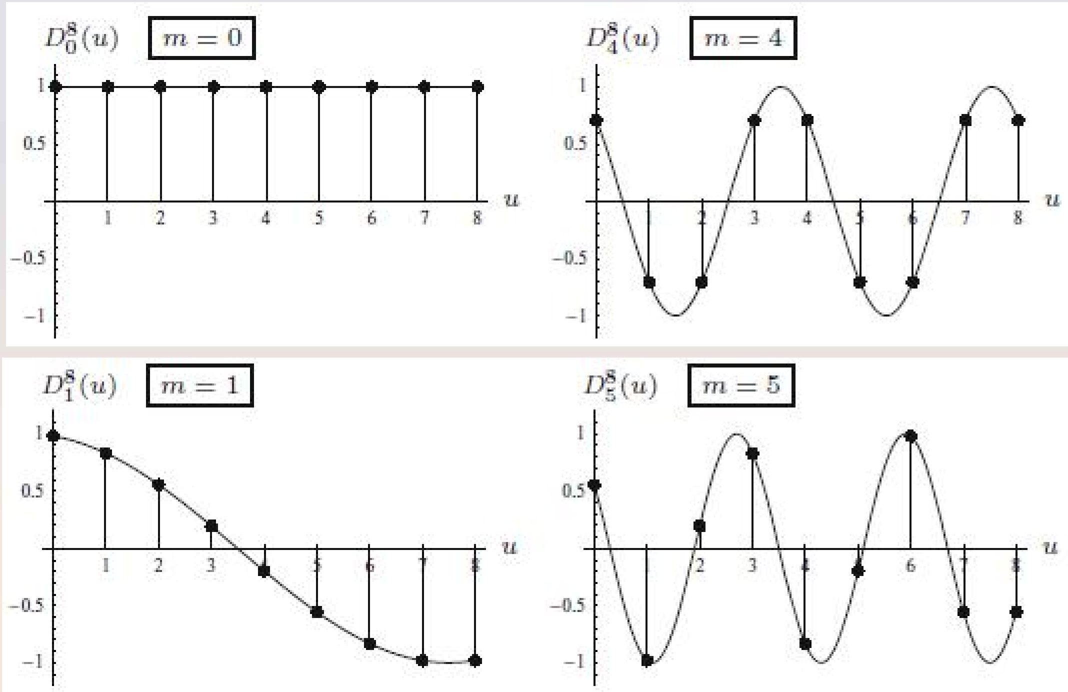
- In 1D case of a signal f(u) of length N

$$F(m) = \sqrt{\frac{2}{N}} \cdot \sum_{u=0}^{N-1} f(u) \cdot c_m \cdot \cos\left(2\pi \frac{m(u+0.5)}{2N}\right), c_m = \begin{cases} \frac{1}{\sqrt{2}} & for \ \ m = 0 \\ 1 & otherwise \end{cases}$$

- DCT basis functions
  - DFT cosine part : $C_m^N(u) = \cos\left(2\pi \frac{mu}{N}\right)$
  - DCT : $D_m^N(u) = \cos\left(2\pi \frac{m(u+0.5)}{2N}\right)$
  - The period of DCT basis functions is double ($\tau_m = 2\frac{N}{m}$).
  - DCT basis functions are phase-shifted by 0.5 units.

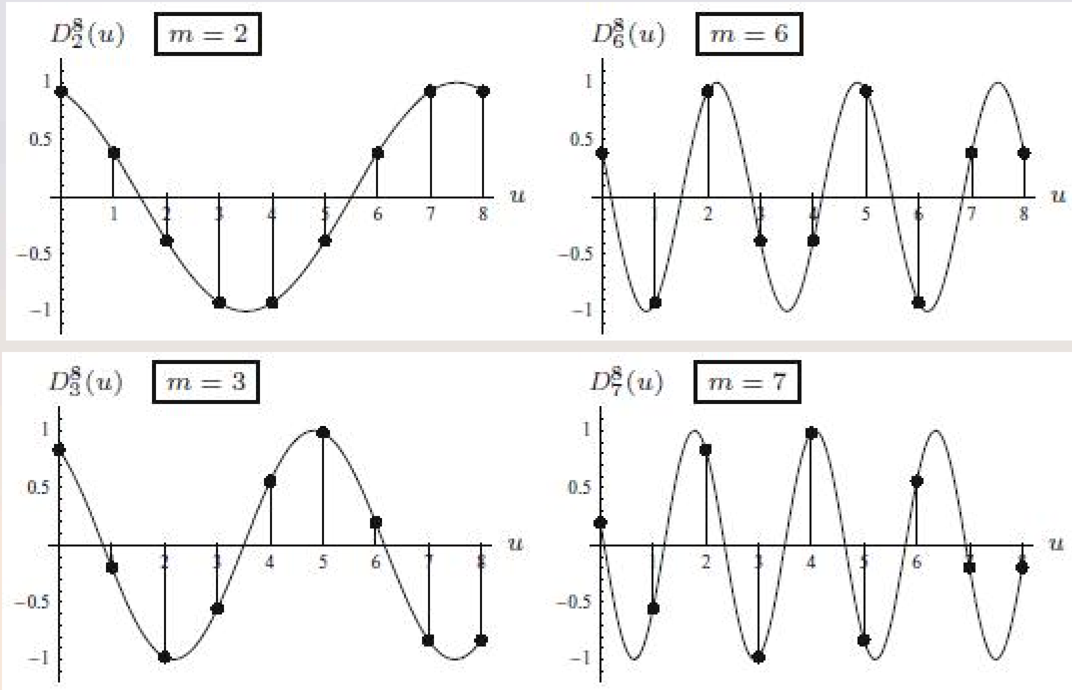
# DCT basis functions I

- For N=8 and wave numbers m=0,…,7
  - $D_0^8(u) = 1$
  - $D_1^8(u) = \cos\left(2\pi \frac{1(u+0.5)}{16}\right)$, so $\tau_1 = 16$ units
  - $D_7^8(u) = \cos\left(2\pi \frac{7(u+0.5)}{16}\right)$, so $\tau_1 = \frac{16}{7}$ units

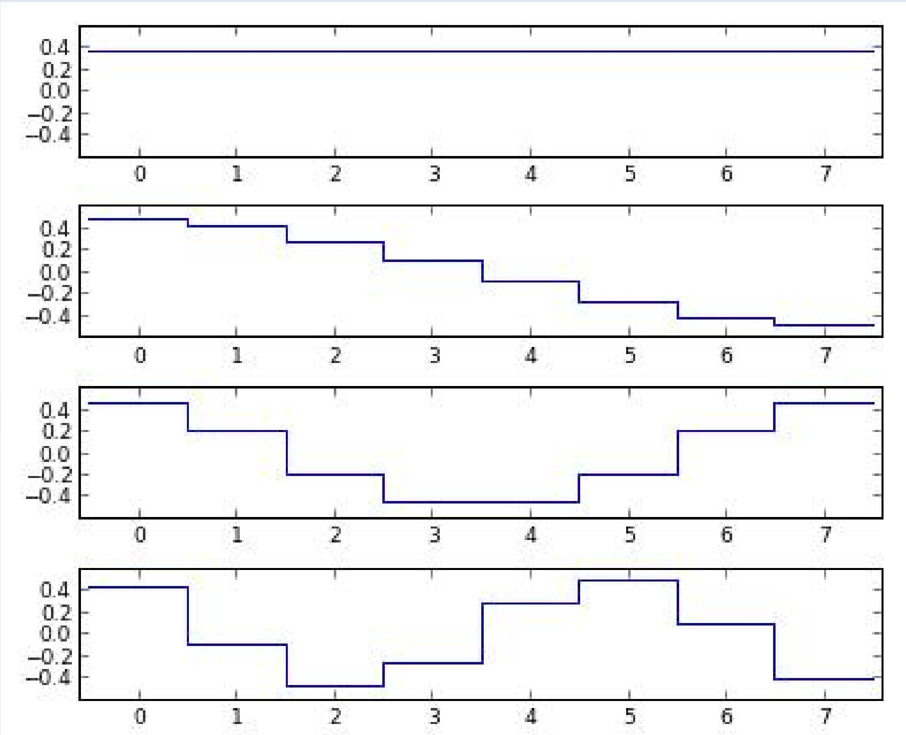


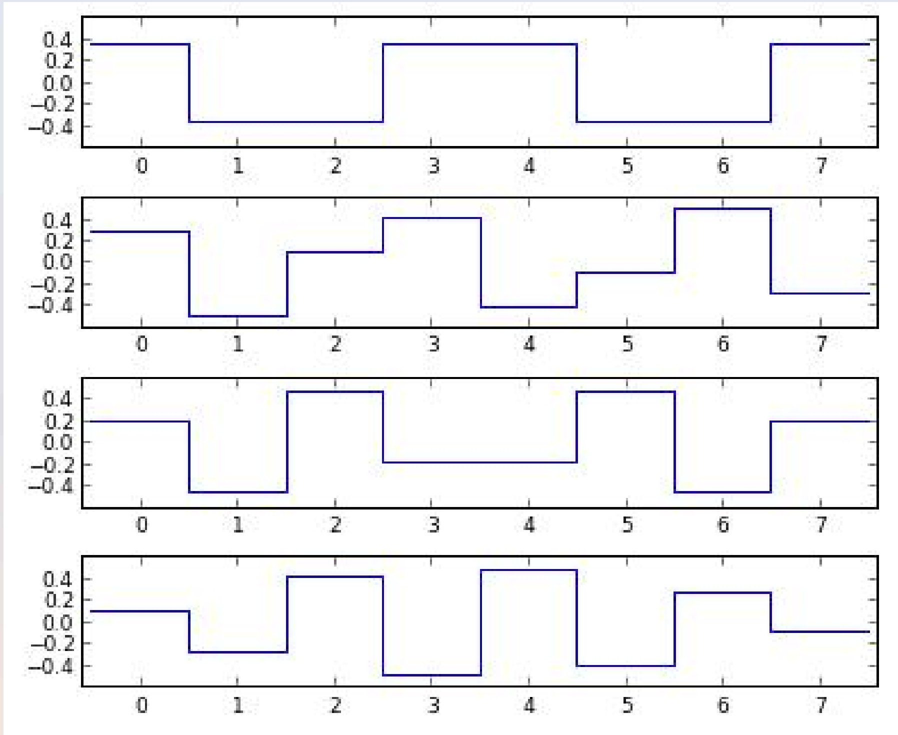
# DCT basis functions 2

- For N=8 and wave numbers m=0,…,7
  - $D_0^8(u) = 1$
  - $D_1^8(u) = \cos\left(2\pi \frac{1(u+0.5)}{16}\right)$, so $\tau_1 = 16$ units
  - $D_7^8(u) = \cos\left(2\pi \frac{7(u+0.5)}{16}\right)$, so $\tau_1 = \frac{16}{7}$ units



# DCT basis functions 3



# Approach to DCT

- Steps to learn DCT.
  - Synthesis : given a set of freq. components and their amplitudes, how can we construct a wave.
  - Rewrite the synthesis problem using Numpy arrays.
  - Analysis : given a signal and a set of frequencies, how can we find the amplitude of each freq. component?
  - Find more efficient analysis algorithm using some linear algebra principles.

# Synthesis I

- Suppose I gave you a list of amplitudes and a list of frequencies, and ask you to construct a signal that is the sum of these freq. components.

```
def synthesize1(amps, fs, ts):
    components = [thinkdsp.CosSignal(freq, amp)
                  for amp, freq in zip(amps, fs)]
    signal = thinkdsp.SumSignal(*components)

    ys = signal.evaluate(ts)
    return ys
```

`amps` is a list of amplitudes, `fs` is the list of frequencies, and `ts` is the sequence of times where the signal should be evaluated.

`components` is a list of `CosSignal` objects, one for each amplitude-frequency pair. `SumSignal` represents the sum of these frequency components.

# Synthesis 2

- Usage of synthesize1 function

```
amps = np.array([0.6, 0.25, 0.1, 0.05])
fs = [100, 200, 300, 400]
framerate = 11025

ts = np.linspace(0, 1, framerate)
ys = synthesize1(amps, fs, ts)
wave = thinkdsp.Wave(ys, framerate)
```

This example makes a signal that contains a fundamental frequency at 100 Hz and three harmonics (100 Hz is a sharp G2). It renders the signal for one second at 11,025 frames per second and puts the results into a Wave object.

# Synthesis with array 1

```
def synthesize2(amps, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    ys = np.dot(M, amps)
    return ys
```

1. `np.outer` computes the **outer product** of `ts` and `fs`. The result is an array with one row for each element of `ts` and one column for each element of `fs`. Each element in the array is the product of a frequency and a time, $ft$.

2. We multiply `args` by $2\pi$ and apply `cos`, so each element of the result is $\cos(2\pi ft)$. Since the `ts` run down the columns, each column contains a cosine signal at a particular frequency, evaluated at a sequence of times.

3. `np.dot` multiplies each row of `M` by `amps`, element-wise, and then adds up the products. In terms of linear algebra, we are multiplying a matrix, `M`, by a vector, `amps`. In terms of signals, we are computing the weighted sum of frequency components.

# Synthesis with array 2

Each row of M corresponds to a time from 0.0 to 1.0 seconds

$a = \cos[2\pi(100)t_n]$

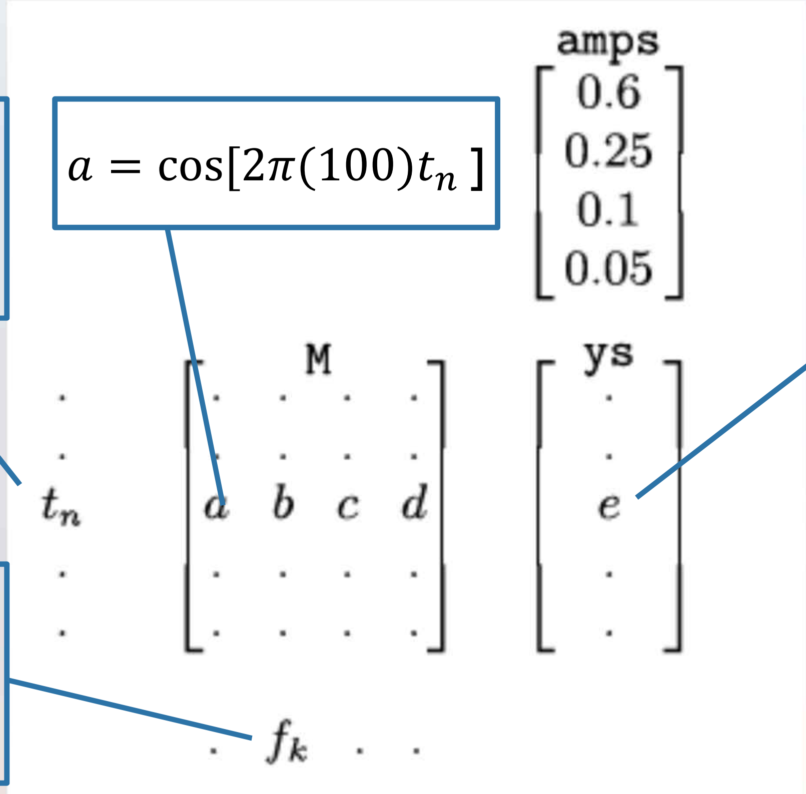



Figure 6.1: Synthesis with arrays.

$e = 0.6a + 0.25b + 0.1c + 0.05d$

Each column of M corresponds to a freq. from 100 to 400 Hz.

So, each element of ys is the sum of four frequency components, evaluated at a point in time, and multiplied by the corresponding amplitudes.

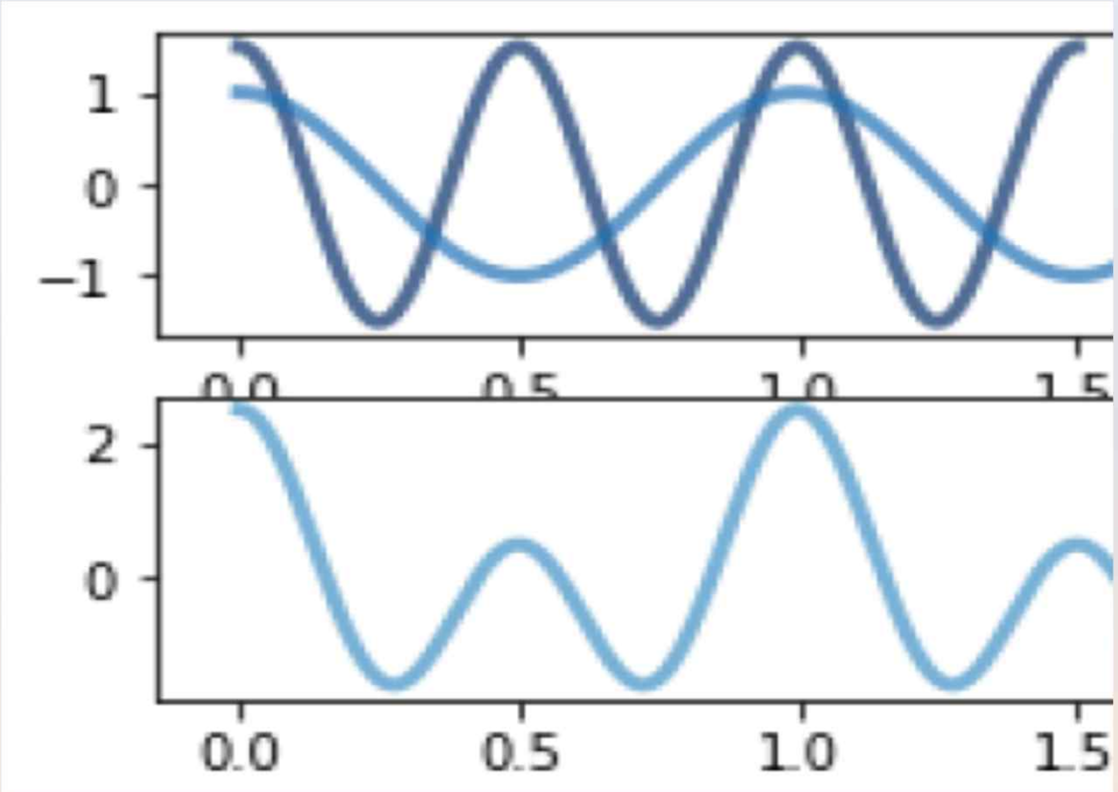
$M = \cos(2\pi T \otimes F)$

$Y = M\, A$

Size of matrices

T : 1 x N, F : 1 x K, M: N x K, A : K x 1, Y : N x 1

# Think in digital manner



# Analysis I

- Suppose I gave you a wave and tell you that it is the sum of cosines with a given set of frequencies.

- How would you find the amplitude for each freq. component? (Given ys, ts and fs, recover amps)
  - The first step : compute $M = \cos(2\pi T \otimes F)$.
  - Find $A$ so that $Y = M\,A$.

Slow

```
def analyze1(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.linalg.solve(M, ys)
    return amps
```

Solving a linear system

In general, we can only solve a system of linear equations if the matrix is square (the number of equations (rows) = the number of unknowns (columns)

# Analysis 2

In this example, we have only 4 frequencies, but we have evaluated the signal at 11,025 times. So we have many more equations than unknowns.

```
n = len(fs)
amps2 = analyze1(ys[:n], fs, ts[:n])
```

Because we know that ys were generated by adding only 4 freq. components

Also, we need to know frequency components used to make the signal in advance.

Impossible!

# Orthogonal matrices I

- To solve the equation $\mathrm{Y} = MA$,
  - $M^{-1}Y = M^{-1}MA = A$
- If we can compute $M^{-1}$ efficiently, we can find $A$ with a simple matrix multiplication.
  - This algorithms has $n^2$ complexity, witch is better than $n^3$.
- Inverting a matrix is slow, but some special cases are faster.
  - If $\mathrm{M}$ is orthogonal, the $M^{-1}$ is just the $M^T$, which implies $M^T M = I$.

# Orthogonal matrices 2

- Synthesize2( )
  - Since M has 11,025 rows, let's work with a smaller example.

time_unit drops out of the computation.

Since there are 4 freq. components, we chose 4 samples in time That way, M is square.

time_unit = 0.001 is a arbitrary choice.

Since the sampling time is time_unit/N, the period of the signal should be at least $\frac{2 \cdot time\_unit}{N}$, or maximum freq $\frac{N}{2 \cdot time\_unit}$

```
def test1():
    amps = np.array([0.6, 0.25, 0.1, 0.05])
    N = 4.0
    time_unit = 0.001
    ts = np.arange(N) / N * time_unit
    max_freq = N / time_unit / 2
    fs = np.arange(N) / N * max_freq
    ys = synthesize2(amps, fs, ts)
```

Since N samples per time_unit, the framerate is N/time_unit and the Nyquist freq. is framerate/2, which is 2000Hz. So fs is a vector of equally spaced frequencies between 0 and 2000Hz. [0, 500, 1000, 1500]

# Orthogonal matrices 3

Amps = [ 0.6, 0.25, 0.1, 0.05]
Ts = [0, 1/4000, 2/4000, 3/4000]
Fs = [0, 500, 1000, 1500]

$$M = \cos\left(\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \pi/4 & \pi/2 & 3\pi/4 \\ 0 & \pi/2 & \pi & 3\pi/2 \\ 0 & 3\pi/4 & 3\pi/2 & 9\pi/4 \end{bmatrix}\right)$$

```
def test1():
    amps = np.array([0.6, 0.25, 0.1, 0.05])
    N = 4.0
    time_unit = 0.001
    ts = np.arange(N) / N * time_unit
    max_freq = N / time_unit / 2
    fs = np.arange(N) / N * max_freq
    ys = synthesize2(amps, fs, ts)
```

$$M = \begin{bmatrix} 1. & 1. & 1. & 1. \\ 1. & 0.707 & 0. & -0.707 \\ 1. & 0. & -1. & 0. \\ 1. & -0.707 & 0. & 0.707 \end{bmatrix}$$

$$0.707 = \frac{\sqrt{2}}{2} = \cos\frac{\pi}{4}$$

$$M^T = M$$

$$M^T M = \begin{bmatrix} 4. & 1. & 0. & 1. \\ 1. & 2 & 1. & 0. \\ 0. & 1. & 2. & 1. \\ 1. & 0. & 1. & 2. \end{bmatrix}$$

*M is not orthogonal.*

# DCT-IV I

- If we choose ts and fs carefully, we can make M orthogonal.
  - There are several ways to do it, so there are several versions of the Discrete Cosine Transform (DCT).
  - One simple option is to shift ts and fs by a half unit. (DCT-IV)

Two changes :
1. time_unit is canceled out.
2. 0.5 is added both to ts and fs.

```
def test2():
    amps = np.array([0.6, 0.25, 0.1, 0.05])
    N = 4.0
    ts = (0.5 + np.arange(N)) / N
    fs = (0.5 + np.arange(N)) / 2
    ys = synthesize2(amps, fs, ts)
```

$$M = \begin{bmatrix} 0.981 & 0.831 & 0.556 & 0.195 \\ 0.831 & -0.195 & -0.981 & -0.556 \\ 0.556 & -0.981 & 0.195 & 0.831 \\ 0.195 & -0.556 & 0.831 & -0.981 \end{bmatrix}$$

$$M^T M = \begin{bmatrix} 2. & 0. & 0. & 0. \\ 0. & 2 & 0. & 0. \\ 0. & 0. & 2. & 0. \\ 0. & 0. & 0. & 2. \end{bmatrix} = 2\mathrm{I}$$

2I is almost orthogonal . $\frac{M^T M}{2} = I$

# DCT-IV 2

- M is symmetric and almost orthogonal.
  - $M\frac{M^T}{2} = M\frac{M}{2} = I$, so $M^{-1} = \frac{M}{2}$.
  - Now we can write a more efficient version of analyse.

Instead of using np.linalg.solve,
we just multiply by M/2

```
def analyze2(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.dot(M, ys) / 2
    return amps
```

ys is the wave array.
No need of ts and fs

```
def dct_iv(ys):
    N = len(ys)
    ts = (0.5 + np.arange(N)) / N
    fs = (0.5 + np.arange(N)) / 2
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.dot(M, ys) / 2
    return amps
```

# Inverse DCT

- Notice that analyze2( ) and synthesize2( ) are almost identical.

```
def synthesize2(amps, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    ys = np.dot(M, amps)
    return ys
```

/2

*2

```
def analyze2(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.dot(M, ys) / 2
    return amps
```

We can use the above insight to compute the inverse DCT

```
def inverse_dct_iv(amps):
    return dct_iv(amps) * 2
```

The biggest difference is about 1e-16.

```
amps = [0.6, 0.25, 0.1, 0.05]
ys = inverse_dct_iv(amps)
amps2 = dct_iv(ys)
max(abs(amps - amps2))
```

# The DCT class I

- To make a Dct object, you can invoke make_dct on a Wave.

```
signal = thinkdsp.TriangleSignal(freq=400)
wave = signal.make_wave(duration=1.0, framerate=10000)
dct = wave.make_dct()
dct.plot()
```

A negative value in DCT corresponds to a negated cosine.

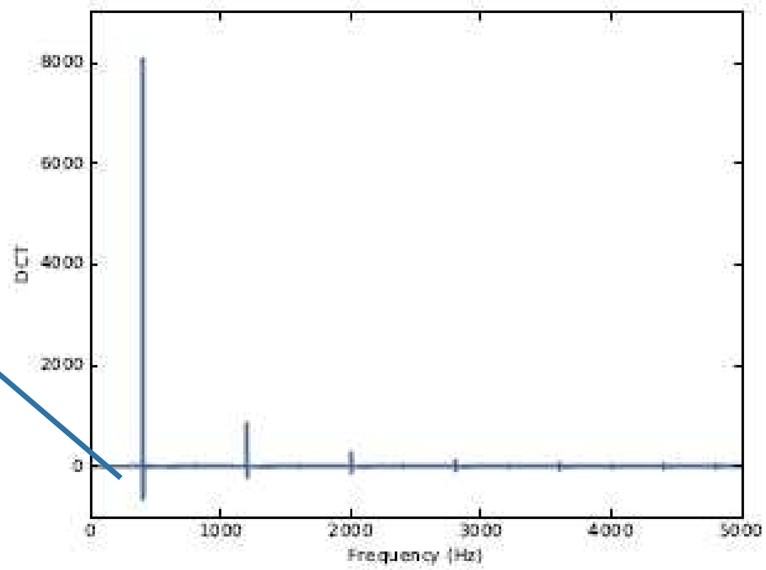



Figure 6.2: DCT of a triangle signal at 400 Hz, sampled at 10 kHz.

# The DCT class 2

- DCT-II in scipy.fftpack

```
import scipy.fftpack

# class Wave:
    def make_dct(self):
        N = len(self.ys)
        hs = scipy.fftpack.dct(self.ys, type=2)
        fs = (0.5 + np.arange(N)) / 2
        return Dct(hs, fs, self.framerate)
```

No difference in result.
Check with this!

```
wave2 = dct.make_wave()
max(abs(wave.ys-wave2.ys))
```