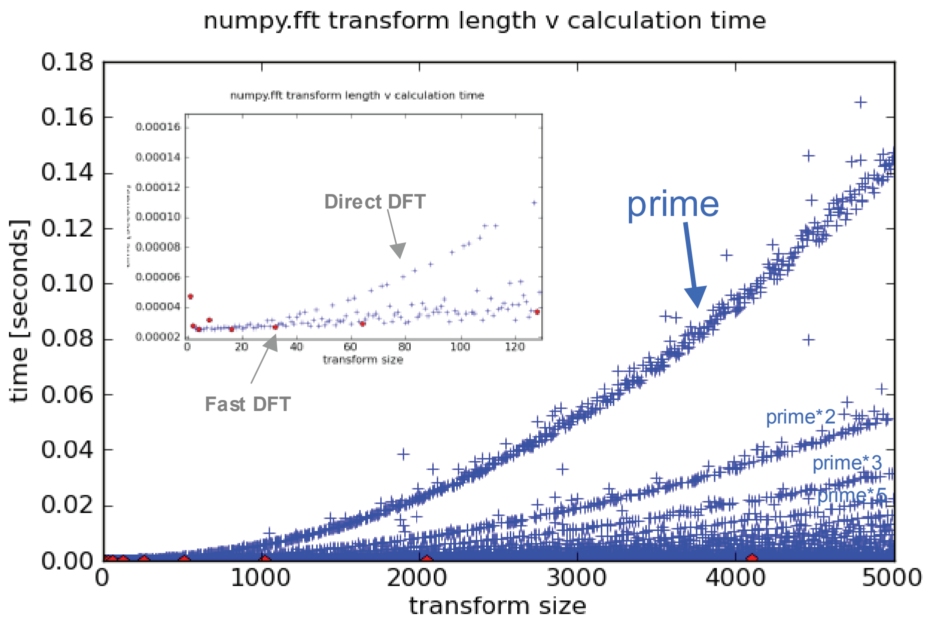
Music 422, 2010, Homework 3, Question 1

See attached source code for MDCT/IMDCT/MDCTslow/IMDCTslow implementation.

Implementation speed

FFT-based transforms are much faster at larger transform lengths when the transform length is a power of two. Tranform lengths consisting of small prime factors also have a speed advantage over direct implementation of the DFT. Large prime-length transforms cannot be implemented with the FFT since there are no symmetries in the calculations to take advantage of.



```
import pylab as pl
import numpy as np
import numpy.fft as npf
import time
N = 5000
domain = np.arange(1,N+1)
range  = np.zeros(N)
dummy  = np.zeros(N)
pow2i  = [2**n-1 for n in xrange(1 \
     +int(np.log2(N)))]

for i in xrange(N):
   time1 = time.clock()
   dummy = npf.fft(np.arange(domain[i]))
   time2 = time.clock()
   range[i] = time2 - time1

pl.plot(domain, range, 'b+')
pl.plot(domain[pow2i], range[pow2i], 'rp')
pl.xlabel('transform size')
pl.ylabel('time [seconds]')
pl.suptitle('numpy.fft length v calculation time')
pl.show()
```

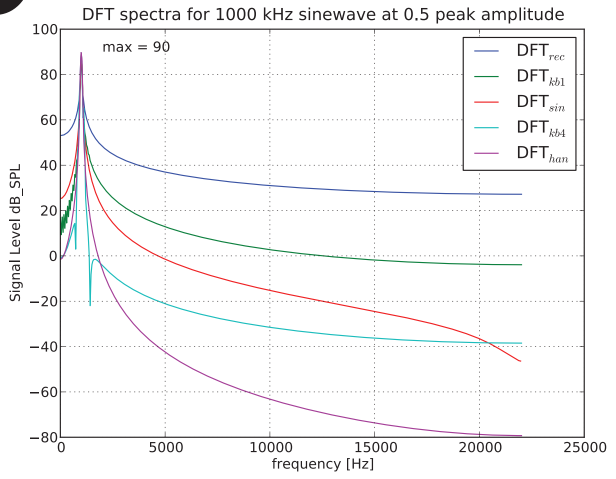
The TDAC process can be done with a window which is a constant 1/sqrt(2) since the resulting window overlap is constant (and one):

$$\underbrace{1/\text{sqrt}(2) * 1/\text{sqrt}(2)}_{\textit{from window } n} + \underbrace{1/\text{sqrt}(2) * 1/\text{sqrt}(2)}_{\textit{from window } n+1} = 1/2 + 1/2 = 1$$

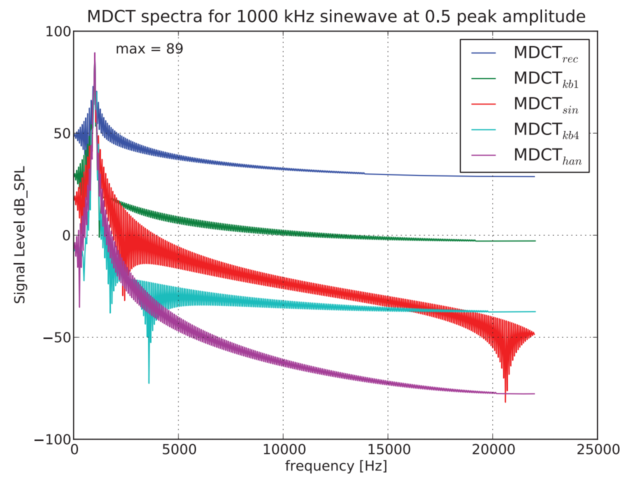
Question 2: See source code for NBC functions for Sine, Hanning, and KBD windows.

Music 422, 2010, Homework 3, Question 3

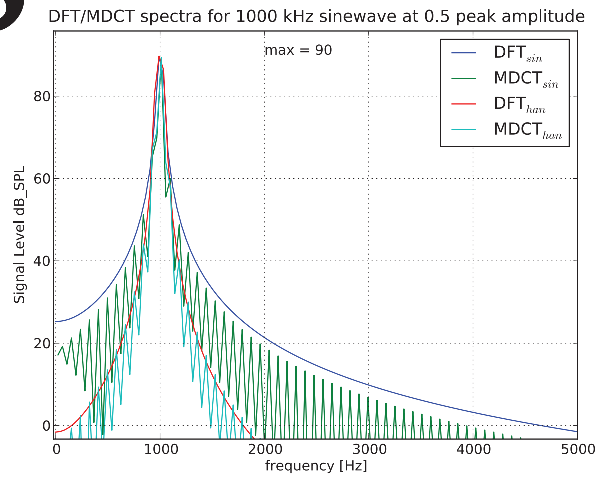
❶



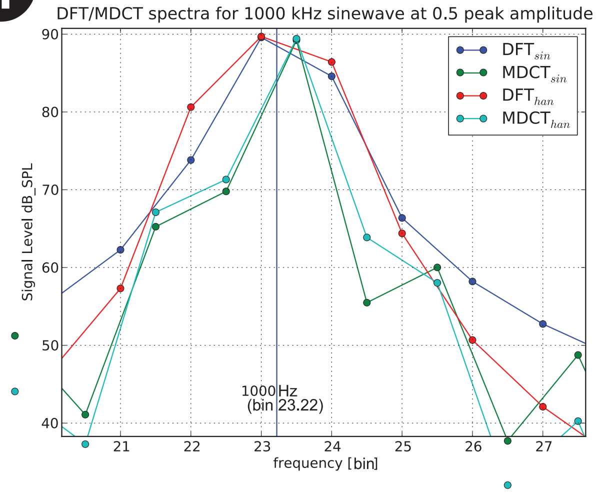
❷



❸



❹



Both the DFT and the MDCT have approximately the same frequency resolution when using a sine window. The MDCT's main lobe is very slightly narrower (see plot ❹), so it does have a slightly better frequency resolution.

The MDCT contains a lot of local junk peaks every other frequency bin, related to time-domain aliasing. This makes it harder to identify true sinewave peaks without further processing of the spectrum (which may also disturb the dB_{SPL} measurements).

Notice that the roll-off for the Hanning window in plots ❶ and ❷ is much better than for the sine window (down 150 dB by 10 kHz compared to down 100 dB for the sine window). By degree of roll-off, the spectra using various windows in plots ❶ and ❷ are:

(1) Rectangular window
(2) Kaiser-Bessel Derived window with α = 1
(3) Sine window
(4) Kaiser-Bessel Derived window with α = 4
(5) Hanning window

For all plots, the windowed signal length is 1024 samples, and the sampling rate is 44100 Hz. Time domain signal: cos(2π 1000 n/44100) for n=[0..1023]

Question 3 (continued)

Here are two functions which can be used to numerically calculate the reference amplitude needed in order to set the spectral amplitude to match the peak amplitude of a sinewave in the time domain. The reference amplitude is equivalent to the integration of the window (summation of window divided by N) times the normalization value for the transform which is 1 for the already normalized MDCT, and N/2 for the DFT (factor of N to normalize DFT, and factor of 1/2 due to only 1/2 of energy being in positive frequencies).

```
def calculateDFTreferenceAmp(window, peakamp=1.0):
  window  = np.array(window)
  peakamp = float(peakamp)
  N       = len(window)
  freq    = int(N/4)
  signal  = peakamp * np.cos(2.0 * np.pi * freq * np.arange(N) / N)
  spectrum= npf.fft(signal * window)
  return max(np.abs(spectrum))
```

```
def calculateMDCTreferenceAmp(window, peakamp=1.0):
  window  = np.array(window)
  peakamp = float(peakamp)
  N       = len(window)
  freq    = int(N/4)-0.5
  signal  = peakamp * np.cos(2.0 * np.pi * freq * np.arange(N) / N)
  spectrum= MDCT(signal * window)
  return max(np.abs(spectrum))
```

DFT reference amps for 1.0 peak amp. sinewave

```
rectangular     : 512.00    (1       * N/2)
sine            : 325.95    (2/π     * N/2)
Hanning         : 256.00    (1/2     * N/2)
KB Derived (1)  : 330.67    (0.64583 * N/2)
KB Derived (4)  : 303.39    (0.59256 * N/2)
```

normalized MDCT reference amps for 1.0 peak amp. sinewave

```
rectangular     : 1.00000
sine            : 0.63662  (2/π)
Hanning         : 0.50000
KB Derived (1)  : 0.64583
KB Derived (4)  : 0.59256
```

Calculate dBspl for DFT/MDCT using reference for a specific N and window type, and transform:

```
sploffset    = 96
DFTspectrum  = numpy.fft.fft(signal * window)
DFTspl       = sploffset + 20 * numpy.log10(numpy.abs(DFTspectrum / dftwinref))
MDCTspectrum = MDCT(signal, N/2, N/2)
MDCTspl      = sploffset + 20 * numpy.log10(numpy.abs(MDCTspectrum / mdctwinref))
```

```
#!usr/bin/env python
#
# Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
# Creation Date: Wed Feb  3 10:22:19 PST 2010
# Last Modified: Thu Mar  4 18:16:56 PST 2010
# Filename:      craigspl.py
# Syntax:        Python 2.6
#
"""
Full-scale reference amplitude calculations for coordinating the
amplitudes of the MDCT and FFT spectra.  Source code for answering
Question 3 on HW 3 is found in the 'if __name__ == "__main__":' section.

"""

import numpy     as np
import numpy.fft as npf
from craigwindow import *
from craigmdct   import *



##############################
##
## calculateDFTreferenceAmp --  Calculate the peak bin amplitude
##   when a sinewave is aligned exactly on a frequency bin.  Using
##   a frequency at 1/2 of the Nyquist frequency to avoid interactions
##   with the negative frequency component.
##

def calculateDFTreferenceAmp(window, peakamp=1.0):
   """calculateDFTreferenceAmp -- calculate the corresponding peak
   amplitude in a DFT spectrum for a given peak sinewave amplitude
   in the time domain.  This amplitude is also dependent on the
   window type which is the second input.
   Arguments:
      window  -- array of numbers containing the windowing function.
      peakamp -- peak amplitude of time-domain sinewave to uas as reference.
   Returns: the reference amplitude of the peak in the spectrum
      for the given peak amplitude.
   """
   window  = np.array(window)
   peakamp = float(peakamp)
   N       = len(window)
   freq    = int(N/4)
   signal  = peakamp * np.cos(2.0 * np.pi * freq * np.arange(N) / N)
   spectrum= npf.fft(signal * window)
   return max(np.abs(spectrum))



##############################
##
## calculateMDCTreferenceAmp -- Calculate the peak bin amplitude
##   when a sinewave is aligned exactly on a frequency bin.  Using
##   a frequency as about 1/2 of the Nyquist frequency to avoid
##   interactions with the negative frequency component.  The MDCT frequency
##   bins are shifted 1/2 bin higher than for the DFT, so the frequency
##   is on a 1/2 bin in the DFT due to the 1/2 bin shift in the
##   freq variable below.
##

def calculateMDCTreferenceAmp(window, peakamp=1.0):
   """calculateMDCTreferenceAmp -- calculate the corresponding peak
   amplitude in an MDCT spectrum for a given peak sinewave amplitude
   in the time domain.  This amplitude is also dependent on the
   window type which is the second input.
   Arguments:
      window  -- array of numbers containing the windowing function.
      peakamp -- peak amplitude of time-domain sinewave to use as reference.
   Returns: the reference amplitude of the peak in the spectrum
      for the given peak amplitude.
   """
   window  = np.array(window)
   peakamp = float(peakamp)
   N       = len(window)
   freq    = int(N/4)-0.5
   signal  = peakamp * np.cos(2.0 * np.pi * freq * np.arange(N) / N)
   spectrum= MDCT(signal * window)
   return max(np.abs(spectrum))




###########################################################################
#
# Testing code:
#

if __name__ == "__main__":
   """
   Homework 3, problem 3
   """

   import pylab as pl
   import numpy as np
   import numpy.fft as npf
   from craigwindow import *
   from craigmdct   import *

   N          = 1024
   freq       = 1000.0
   srate      = 44100.0
   peakamp    = 1.0
   amp        = 0.5
   sploffset  = 96
   signal     = amp * np.cos(2.0*np.pi*freq*np.arange(N)/srate)
   recwin     = np.ones(N)
   sinwin     = SineWindow(N)
   hanwin     = HanningWindow(N)
   kb1win     = KBDWindow(N, 1.0)
   kb4win     = KBDWindow(N, 4.0)

   recDFTref  = calculateDFTreferenceAmp(recwin, peakamp)
   sinDFTref  = calculateDFTreferenceAmp(sinwin, peakamp)
   hanDFTref  = calculateDFTreferenceAmp(hanwin, peakamp)
   kb1DFTref  = calculateDFTreferenceAmp(kb1win, peakamp)
   kb4DFTref  = calculateDFTreferenceAmp(kb4win, peakamp)

   recMDCTref = calculateMDCTreferenceAmp(recwin, peakamp)
   sinMDCTref = calculateMDCTreferenceAmp(sinwin, peakamp)
   hanMDCTref = calculateMDCTreferenceAmp(hanwin, peakamp)
   kb1MDCTref = calculateMDCTreferenceAmp(kb1win, peakamp)
   kb4MDCTref = calculateMDCTreferenceAmp(kb4win, peakamp)

   ## DFT amplitudes are unormalized (1/2 factor included
   ## since not doubling the positive frequency amplitudes).
   # recDFTref  # 512.0                 # 1       * N/2
   # sinDFTref  # 325.94945128396904    # 2/pi    * N/2
   # hanDFTref  # 256.0                 # 1/2     * N/2
   # kb1DFTref  # 330.66742639151016    # 0.64583 * N/2
   # kb4DFTref  # 303.39089615553968    # 0.59256 * N/2

   ## MDCT amplitudes are normalized (factor of 2 included in
   ## transform to double amplitude of positive frequecies)
   # recMDCTref # 0.99999882345170776   # 1
   # sinMDCTref # 0.63661927301775245   # 2/pi
   # hanMDCTref # 0.49999941172585388   # 1/2
   # kb1MDCTref # 0.64583405728406651
   # kb4MDCTref # 0.59255964687733975

   # ignore negative frequencies in DFT spectra (negative frequencies
   # have Hermetian symmetry (real:symmetric, imaginary:antisymmetric)
   absDFTrec  = np.split(np.abs(npf.fft(signal*recwin)), 2)[0]
   absDFTsin  = np.split(np.abs(npf.fft(signal*sinwin)), 2)[0]
   absDFThan  = np.split(np.abs(npf.fft(signal*hanwin)), 2)[0]
   absDFTkb1  = np.split(np.abs(npf.fft(signal*kb1win)), 2)[0]
   absDFTkb4  = np.split(np.abs(npf.fft(signal*kb4win)), 2)[0]

   # MDCT output is only positive frequencies (second half of
   # full transform is anti-symmetric version of first half)
   absMDCTrec = np.abs(MDCT(signal*recwin))
   absMDCTsin = np.abs(MDCT(signal*sinwin))
   absMDCThan = np.abs(MDCT(signal*hanwin))
   absMDCTkb1 = np.abs(MDCT(signal*kb1win))
   absMDCTkb4 = np.abs(MDCT(signal*kb4win))

   # convert DFT spectra to DB_SPL
   splDFTrec  = sploffset + 20 * np.log10(absDFTrec/recDFTref)
   splDFTsin  = sploffset + 20 * np.log10(absDFTsin/sinDFTref)
   splDFThan  = sploffset + 20 * np.log10(absDFThan/hanDFTref)
   splDFTkb1  = sploffset + 20 * np.log10(absDFTkb1/kb1DFTref)
   splDFTkb4  = sploffset + 20 * np.log10(absDFTkb4/kb4DFTref)

   # convert MDCT spectra to DB_SPL
   splMDCTrec = sploffset + 20 * np.log10(absMDCTrec/recMDCTref)
   splMDCTsin = sploffset + 20 * np.log10(absMDCTsin/sinMDCTref)
   splMDCThan = sploffset + 20 * np.log10(absMDCThan/hanMDCTref)
   splMDCTkb1 = sploffset + 20 * np.log10(absMDCTkb1/kb1MDCTref)
   splMDCTkb4 = sploffset + 20 * np.log10(absMDCTkb4/kb4MDCTref)

   # Examine dB_SPL maxima in both spectra with various windows.
   # If the input peak amplitude is 1.0, then the dB_SPL max for
   # the spectra should be 96 dB (the dbsploffset variable).
   # However, since the 1000 Hz peak is at bin 23.22, the true
   # maxima is not directly visible in the sampled spectra. So the
   # maximum observed amplitudes will be slightly lower (up to 2 dB
   # lower for sine windowing and 1.4 dB for Hanning windowing in
   # the worst case).

   ## For amp = 1.0:
   ## To get exactly 96, you will have to examine the DTFT (continuous
   ## form of the DFT) more closely for the 1000 Hz tone at bin 23.22...
   # max(splDFTrec)  # 95.310978661932765
   # max(splDFTsin)  # 95.604489405903408
   # max(splDFThan)  # 95.728134825841494
   # max(splDFTkb1)  # 95.581067925020477
   # max(splDFTkb4)  # 95.701368021646104
   #
   # max(splMDCTrec) # 94.73811335908681
   # max(splMDCTsin) # 95.237247650043884
   # max(splMDCThan) # 95.438770142575223
   # max(splMDCTkb1) # 95.199062795941018
   # max(splMDCTkb4) # 95.394811095218699

   ## For amp = 0.5: (6 dB lower than for amp=1.0, so expect 90 dB)
   ## Notice that an amplitude decrease of 1/2 is equivalent to a decrease
   ## of 6 dB: 20 * log_10(0.5) = -6.0205999132 dB.
   ## To get exactly 89.9794, you will have to examine the DTFT (continuous
   ## form of the DFT) more closely for the 1000 Hz tone at bin 23.22...
   # max(splDFTrec)  # 89.29037874865314
   # max(splDFTsin)  # 89.583889492623783
   # max(splDFThan)  # 89.707534912561869
   # max(splDFTkb1)  # 89.560468011740852
   # max(splDFTkb4)  # 89.680768108366479
   #
   # max(splMDCTrec) # 88.717513445807185
   # max(splMDCTsin) # 89.216647736764259
   # max(splMDCThan) # 89.418170229295598
   # max(splMDCTkb1) # 89.178462882661393
   # max(splMDCTkb4) # 89.374211181939074

   # Frequency axis of DFT and MDCT are rotated from each other by 1/2 bin:
   DFTfreq  = srate *  np.arange(N/2)       / N
   MDCTfreq = srate * (np.arange(N/2)+0.5) / N

   #####                                           #####
   ##### Generating plots for HW3, Question 3      #####
   #####                                           #####

   ##### Examine Sine/Hanning Windows in DFT/MDCT #####
   ##### Plot 3 in Solutions (HW3, Q3)             #####

   pl.clf()
   pl.plot(                    \
      DFTfreq,   splDFTsin,    \
      MDCTfreq,  splMDCTsin,   \
      DFTfreq,   splDFThan,    \
      MDCTfreq,  splMDCThan,   \
   )
   pl.xlabel("frequency [Hz]")
   pl.ylabel("Signal Level dB_SPL")
   pl.title("DFT/MDCT spectra for 1000 kHz sinewave at 0.5 peak amplitude")
   maxval = int(max(splDFThan+0.5))
   pl.text(2000, maxval, 'max = ' + `maxval`)
   pl.legend((                 \
         "DFT$_{sin}$",    \
         "MDCT$_{sin}$",   \
         "DFT$_{han}$",    \
         "MDCT$_{han}$",   \
      ), loc='upper right')
   pl.grid(True)
   pl.show()

   ##### Looking in terms of DFT bins on x-axis:  #####
   ##### Plot 4 in Solutions (HW3, Q3)             #####

   DFTbins  = np.arange(N/2)
   MDCTbins = np.arange(N/2) + 0.5
   targetbin = freq * N / srate   # 23.219954648526077

   pl.clf()
   pl.plot(                                  \
      DFTbins,   splDFTsin,  'b-',   \
      DFTbins,   splDFTsin,  'bo',   \
      MDCTbins,  splMDCTsin, 'g-',   \
      MDCTbins,  splMDCTsin, 'go',   \
      DFTbins,   splDFThan,  'r-',   \
      DFTbins,   splDFThan,  'ro',   \
      MDCTbins,  splMDCThan, 'c-',   \
      MDCTbins,  splMDCThan, 'co',   \
   )
   pl.xlabel("frequency [bin]")
   pl.ylabel("Signal Level dB_SPL")
   pl.title("DFT/MDCT spectra for 1000 kHz sinewave at 0.5 peak amplitude")
   pl.text(targetbin, 40, `freq`)
   pl.axvline(targetbin, 0, 100)
   pl.legend((                \
         "DFT$_{sin}$",  "", \
         "MDCT$_{sin}$", "", \
         "DFT$_{han}$",  "", \
         "MDCT$_{han}$", "", \
      ), loc='upper right')
   pl.grid(True)
   pl.show()


   ##### Examine various windows in DFT           #####
   ##### Plot 1 in Solutions (HW3, Q3)            #####

   pl.clf()
   pl.plot(                    \
      DFTfreq,  splDFTrec,     \
      DFTfreq,  splDFTkb1,     \
      DFTfreq,  splDFTsin,     \
      DFTfreq,  splDFTkb4,     \
      DFTfreq,  splDFThan,     \
   )
   pl.xlabel("frequency [Hz]")
   pl.ylabel("Signal Level dB_SPL")
   pl.title("DFT spectra for 1000 kHz sinewave at 0.5 peak amplitude")
   maxval = int(max(splDFThan+0.5))
   pl.text(2000, maxval, 'max = ' + `maxval`)
   pl.legend((                 \
         "DFT$_{rec}$",   \
         "DFT$_{kb1}$",   \
         "DFT$_{sin}$",   \
         "DFT$_{kb4}$",   \
         "DFT$_{han}$",   \
      ), loc='upper right')
   pl.grid(True)
   pl.show()


   ##### Examine various windows in MDCT          #####
   ##### Plot 2 in Solutions (HW3, Q3)            #####

   pl.clf()
   pl.plot(                      \
      MDCTfreq,  splMDCTrec,     \
      MDCTfreq,  splMDCTkb1,     \
      MDCTfreq,  splMDCTsin,     \
      MDCTfreq,  splMDCTkb4,     \
      MDCTfreq,  splMDCThan,     \
   )
   pl.xlabel("frequency [Hz]")
   pl.ylabel("Signal Level dB_SPL")
   pl.title("MDCT spectra for 1000 kHz sinewave at 0.5 peak amplitude")
   maxval = int(max(splMDCThan+0.5))
   pl.text(2000, maxval, 'max = ' + `maxval`)
   pl.legend((                 \
         "MDCT$_{rec}$",   \
         "MDCT$_{kb1}$",   \
         "MDCT$_{sin}$",   \
         "MDCT$_{kb4}$",   \
         "MDCT$_{han}$",   \
      ), loc='upper right')
   pl.grid(True)
   pl.show()
```

```
#!usr/bin/env python
#
# Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
# Creation Date: Mon Feb  1 21:31:24 PST 2010
# Last Modified: Mon Mar  1 16:08:56 PST 2010
# Filename:      craigmdct.py
# Syntax:        Python 2.6
#
"""
MDCT Interface Functions for PAC system:

MDCT(windowedsignal, leftoverlap, rightoverlap, isInverse=False)
IMDCT(halfspectrum, leftoverlap, rightoverlap)
   Forward and inverse MDCT transform, with normalization of 1/N
   factor applied to forward MDCT rather than IMDCT.  Implemented
   using the numpy.fft.fft/numpy.fft.ifft functions which utilize the
   "Fast Fourier Transform" algorithm of the DFT.

MDCTslow(windowedsignal, leftoverlap, rightoverlap, isInverse=False)
IMDCTslow(halfspectrum, leftoverlap, rightoverlap)
   Direct implementation of the MDCT with O(N**2) computational efficiency.
   Performs much slower as length of the transform is increased when
   compared the the above pair of functions.

"""

import numpy     as np
import numpy.fft as npf


def MDCT(windowedsignal, a=-1, b=-1, isInverse=False):
   if (isInverse is False):
      return MDCTpositive (windowedsignal, a, b)
   else:
      return IMDCTpositive(windowedsignal, a, b)


def IMDCT(halfspectrum, a=-1, b=-1):
      return IMDCTpositive(halfspectrum, a, b)


def MDCTslow(windowedsignal, a=-1, b=-1, isInverse=False):
   if (isInverse is False):
      return np.split(DirectMDCT(windowedsignal, a, b), 2)[0]
   else:
      return IMDCTSlow(windowedsignal, a, b)


def IMDCTslow(halfspect, a=-1, b=-1):
      fullspectrum = np.concatenate([halfspect, -np.flipud(halfspect)])
      return DirectIMDCT(fullspectrum, a, b)



###########################################################################


##############################
##
## MDCTpositive -- MDCT implemented using numpy.fft.fft().
##    numpy.fft.fft() can actually receive data of any length,
##    so no need to check for powers of two for the classical implementation
##    of the Fast Fourier Transform.  numpy.fft.fft() is "fast" when
##    the prime factors of the length are about less than 11.  For
##    larger prime factors, the "direct" implementation of the Discrete
##    Fourier Transform is used by numpy.fft.fft()
##
## Example input/output:
```

```
##    MDCTpositive([0, 1, 2, 3, 4, 5, 6, 7])
##    array([-5.44915121, -0.87556153,  0.81515624,  0.61291387])
##
## Asymmetric 50% overlapping case:
##    MDCTpositive([0, 1, 2, 3, 4, 5], 2, 4)
##    array([-3.99001577,  0.23570226,  0.72402944])
##

def MDCTpositive(input, a=-1, b=-1, normalize=True):
   """MDCTpositive: MDCT implemented with FFT, returning first 1/2 of spectrum
      array (positive frequencies).  The MDCT includes the normalization
      factor rather than the IMDCT.
      Arguments:
         input:  pre-windowed signal (using sine or kbd window).
         a:      number of samples in overlap with previous 50% overlap frame.
         b:      number of samples in overlap with next 50% overlap frame.
      Returns:   The first half (positive) portion of antisymmetric spectrum.
   """
   a = int(a)
   b = int(b)
   input = np.array(input)
   if (a<0 or b<0):
      N = len(input)
      a = N/2
      b = N/2
   else:
      N = a + b
   assert N   == len(input)
   assert N%2 == 0
   n0 = (b+1.0)/2.0
   n  = np.arange(N)
   y  = input * np.e**(-1j*np.pi*n/N)
   Y  = npf.fft(y)
   if (normalize is True):
      # Factor of 1/N is applied to place spectrum in range from -1.0 to +1.0;
      # otherwise the maximum spectral value is N (consider the transform
      # of npf.fft[1,1,1,1] which is [4,0,0,0], so dividing by 4 sets the
      # maximum to 1: npf.fft([1,1,1,1])/4 = [1,0,0,0]
      Y *= 1.0/N

   k = np.arange(N/2) # frequency bins of the fft up to the Nyquist freq.
   output = np.real(Y[k] * np.e**(-2j*np.pi*n0*(k+0.5)/N))
   # Since the MDCT spectrum contains equal energy in the positive
   # and negative frequencies, add an extra factor of 2.0 to increase
   # the positive frequency values to full scale for peak amplitude
   # sinewaves.
   output *= 2.0
   return output



##############################
##
## IMDCTpositive -- Inverse MDCT implemented using numpy.fft.ifft().
##
## Example input/output:
##    IMDCTpositive([0, 1, 2, 3])
##    array([ 3.8076084 , -3.6699737 ,  3.6699737 , -3.8076084 ,
##            7.64674316, -5.05576209, -5.05576209,  7.64674316])
##
## Asymmetric 50% overlapping case:
##    IMDCTpositive([0, 1, 2], 2, 4)
##    array([ 2.44948974, -2.44948974,  4.24264069,
##            -2.44948974, -2.44948974,  4.24264069])
##

def IMDCTpositive(input, a=-1, b=-1, normalize=False):
   """IMDCTpositive: IMDCT implemented with IFFT.
```

```
      Arguments:
         input:  first 1/2 of anti-symmetric MDCT spectrum.
         a:      number of samples in overlap with previous 50% overlap frame.
         b:      number of samples in overlap with next 50% overlap frame.
      Returns:   Full-length time-domain signal (which then needs to be
                 windowed with a sine or KBD window) before 50% overlap-add.
   """
   a = int(a)
   b = int(b)
   input = np.array(input)
   # reconstruct negative frequency portion of MDCT spectrum:
   input = np.concatenate([input, -np.flipud(input)])
   if (a<0 or b<0):
      N = len(input)
      a = N/2
      b = N/2
   else:
      N = a + b
   assert N   == len(input)
   assert N%2 == 0
   n0 = (b+1.0)/2.0
   k  = np.arange(N)
   y  = input * np.e**(1j*2*np.pi*k*n0/N)
   Y  = npf.ifft(y)
   if (normalize is False):
      # The IFFT (for signal processing and as found in numpy) already contains
      # a 1/N normalization factor. However, IMDCTpositive already applied this
      # normalization factor, so remove the normalization from here.
      Y *= N
   n  = np.arange(N)
   output = np.real(Y * np.e**(1j*np.pi*(n+n0)/N))
   return output


##########################################################################


##############################
##
## DirectMDCT -- O(N**2) implementation of the MDCT.  Outputs a full length-N
##   spectrum for a length-N signal (unlike the MDCTpositive above which
##   outputs an N/2 length spectrum from a length-N signal).
##
## Example inputs/outputs:
##
##   DirectMDCT([0, 1, 2, 3, 4, 5, 6, 7])
##   array([-5.44915121, -0.87556153,  0.81515624,  0.61291387,
##          -0.61291387, -0.81515624,  0.87556153,  5.44915121])
##
##   DirectMDCT([0, 1, 2, 3, 4, 5], 2, 4)
##   array([-3.99001577,  0.23570226,  0.72402944,
##          -0.72402944, -0.23570226,  3.99001577])
##

def DirectMDCT(input, a=-1, b=-2, normalize=True):
   """DirectMDCT: Calculate MDCT as two nested N-length summations
      Arguments:
         input:  pre-windowed signal (using sine or kbd window)
         a:      number of samples in overlap with previous 50% overlap frame.
         b:      number of samples in overlap with next 50% overlap frame.
      Returns:   The complete N-length anti-symmetric spectrum.
   """
   input = np.array(input)
   a = int(a)
   b = int(b)
   if (a<0 or b<0):
      N = len(input)
      a = N/2
      b = N/2
   else:
      N = a + b
   assert N%2 == 0
   assert N   == len(input)
   n0     = (b + 1.0)/2.0
   tpN    = 2.0 * np.pi / N
   n      = np.arange(N)
   output = np.zeros(N)
   for k in xrange(N):
      output[k] = np.sum(input * np.cos(tpN*(n+n0)*(k+0.5)))
   if (normalize is True):
      output *= 1.0/N
   # add extra 2.0 normalization factor so pos. freqs. match other MDCT amps.
   return 2.0 * output



##############################
##
## Jason Su’s solution using maxtrix calculations. (positive spectrum output)
##
## Example function calls:
##
## DirectMDCTjs([0, 1, 2, 3, 4, 5, 6, 7], 4, 4)
##    matrix([[-5.44915121],
##            [-0.87556153],
##            [ 0.81515624],
##            [ 0.61291387]])
## DirectMDCTjs([0, 1, 2, 3], 4, 4, True)  # IMDCT
##    matrix([[ 3.8076084 ],
##            [-3.6699737 ],
##            [ 3.6699737 ],
##            [-3.8076084 ],
##            [ 7.64674316],
##            [-5.05576209],
##            [-5.05576209],
##            [ 7.64674316]])
##

def DirectMDCTjs(input, a=-1, b=-2, isInverse=False):
   N      = a+b
   n0     = (b+1)/2.0
   mat1   = (np.matrix(np.arange(N/2))+0.5).T   # N/2x1 column vector
   mat2   = np.matrix(np.arange(N)) + n0        # 1xN row vector
   xfmmat = np.cos(2.0*np.pi/N*mat1*mat2)       # build transform matrix
   # Add extra 2.0 factor to include negative energy
   xfmmat *= 2.0
   if not isInverse:
      xfmmat *= 1.0/N
   else:
      xfmmat = xfmmat.T
   # make sure input a column vector
   input = np.matrix(input)
   if input.shape[0] is 1 and input.shape[1] is not 1:
      input = input.T
   return xfmmat*input



##############################
##
## DirectIMDCT -- O(N**2) implementation of the inverse MDCT.
##
## Example inputs/outputs:
##
##    DirectIMDCT([0, 1, 2, 3, -3, -2, -1, 0])
##    array([ 3.8076084 , -3.6699737 ,  3.6699737 , -3.8076084 ,
##            7.64674316, -5.05576209, -5.05576209,  7.64674316])
##
##    DirectIMDCT([0, 1, 2, -2, -1, 0], 2, 4)
##    array([ 2.44948974, -2.44948974,  4.24264069,
##           -2.44948974, -2.44948974,  4.24264069])
##

def DirectIMDCT(input, a=-1, b=-1, normalize=False):
   """DirectIMDCT: IMDCT implemented with double summation.
      Arguments:
         input:  Full-length anti-symmetric spectrum.
         a:      number of samples in overlap with previous 50% overlap frame.
         b:      number of samples in overlap with next 50% overlap frame.
      Returns:   Full-length time-domain signal (which then needs to be
                 windowed with a sine or KBD window).
   """
   input  = np.array(input)
   # if the input is only the first half of the spectrum, use this line:
   # input  = np.concatenate([input, -np.flipud(input)])
   a = int(a)
   b = int(b)
   if (a<0 or b<0):
      N = len(input)
      a = N/2
      b = N/2
   else:
      N = a + b
   assert N%2 == 0
   assert N   == len(input)
   n0     = (b + 1.0)/2.0
   tpN    = 2.0 * np.pi / N
   k      = np.arange(N)
   output = np.zeros(N)
   for n in xrange(N):
      output[n] += np.sum(input * np.cos(tpN * (n+n0) * (k+0.5)))
   if (normalize is True):
      output *= 2.0 / N;
   return output



###########################################################################
#
# Testing code:
#

###
### Various testing plots
###

if __name__ == "__main__":

   """
   #  Verify that the TDAC process returns the original signal, within
   #  machine precision. (HW 3, Question 1)
   #"""

   import numpy     as np
   import numpy.fft as npf
   from craigwindow import *

   sigcase = 2
   wincase = 2

   if (sigcase == 0):
      N = 32
      signal = np.ones(N)
   elif (sigcase == 1):
```

```
   N = 32
   signal = np.sin(2.0 * np.pi * np.arange(N) * 1.0 / N)
else:
   N = 8
   signal = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

if (wincase == 0):
   window = SineWindow(N)
elif (wincase == 1):
   window = KBDWindow(N, 1)
elif (wincase == 2):
   window = np.ones(N)/np.sqrt(2)
elif (wincase == 3):
   window = HanningWindow(N)
else:
   window = np.ones(N)

# Add 1/2 frame of zeros before and after signal.
# Mod of signal length also needs to first be padded to 1/2 frame length.
padding = np.zeros(N/2)
paddedsignal = np.concatenate((padding, signal, padding))
segments = np.split(paddedsignal, len(signal)*2/N+2)
# fcount: number of signal frames
fcount = len(segments)-1;
# frames, 50% overlapping frames of the original signal
frames = [np.concatenate([segments[i], segments[i+1]]) \
            for i in xrange(fcount)]
# blocks = MDCT half-spectra of windowed signal frames
blocks = [MDCT(frames[i] * window) for i in xrange(fcount)]
# Bit optimization and quantization done here...
# iframes = blocks IMDCTed and re-windowed
iframes = np.array([IMDCT(blocks[i]) * window for i in xrange(fcount)])
# isegments = 1/2 length iframes, preparing for 50% overlap
isegments = iframes.flatten()
isegments = np.split(isegments, len(isegments)*2/N)
# drop initial and final 1/2 frames which contain aliasing junk:
isegments = isegments[1:-1]
# isignal: 50% overlap summation to regenerate original signal.
isignal = np.array( [ isegments[2*i] + isegments[2*i+1] \
      for i in xrange(fcount-1) ] ).flatten()
diff = np.abs(signal - isignal)
max(diff)
### 3.5527136788005009e-15
# Using KBDWindow(N, 1):
### 2.886579864025407e-15
# Using 1/sqrt(2) window:
### 7.1054273576010019e-15

# examine reconstruction of [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]:
isignal
### array([  1.22124533e-15,   1.00000000e+00,   2.00000000e+00,
###          3.00000000e+00,   4.00000000e+00,   5.00000000e+00,
###          6.00000000e+00,   7.00000000e+00,   8.00000000e+00,
###          9.00000000e+00,   1.00000000e+01,   1.10000000e+01])




#""
# Verify that Fast and Slow MDCT implementations give the same values.
#""

import random
import numpy as np
N         = 1024
signal    = [random.random() for i in xrange(N)]
fastmdct  = MDCT(signal)
slowmdct  = MDCTslow(signal)
diff      = np.abs(fastmdct - slowmdct)
```

```
max(diff)
### 2.1427304375265521e-14

spectrum  = [random.random() for i in xrange(N/2)]
fastimdct = IMDCT(spectrum)
slowimdct = IMDCTslow(spectrum)
idiff     = np.abs(fastimdct - slowimdct)
max(idiff)
### 2.8705926524708048e-11



#""
# Examine Timing characteristics of numpy.fft.fft ######################
# Homework 3, Question 1
#""

import pylab as pl
import numpy as np
import numpy.fft as npf
import time
N = 5000
domain = np.arange(1,N+1)
range  = np.zeros(N)
dummy  = np.zeros(N)
pow2i  = [2**n-1 for n in xrange(1+int(np.log2(N)))]
for i in xrange(N):
   time1    = time.clock()
   dummy    = npf.fft(np.arange(domain[i]))
   time2    = time.clock()
   range[i] = time2 - time1

pl.clf()
pl.plot(domain, range, 'b+')
pl.plot(domain[pow2i], range[pow2i], 'rp')
pl.xlabel('transform size')
pl.ylabel('time [seconds]')
pl.suptitle('numpy.fft transform length v calculation time')
pl.show()



#""
# Examining Frequency offset between FFT and MDCT ######################
#""

# Maximum possible signal: 0 Hz signal at amplitude of 1:
# (which can be viewed as two 1/2 amplitude sinewaves at 0 Hz)
import pylab     as pl
import numpy     as np
import numpy.fft as npf
N = 1024
dcsig    = np.ones(N)
dcfft    = npf.fft(dcsig)
dcfftabs = np.abs(dcfft)
max(dcfftabs)
### 1024.0
dcifft = npf.ifft(dcfft)
max(dcifft)
### (1+0j)
min(dcifft)
### (1+0j)


# examine in frequency range with minimal positive/negative
# frequency interaction (at bin N/4 in the middle of the positive
# frequency range):
```

```
signal = np.sin(2 * np.pi * (N/4.0) * np.arange(N) / N)  # N/4 freq.
absspectrum = np.abs(npf.fft(signal))
max(absspectrum)) / N
### 0.5
# check to see if the max is where it is expected:
absspectrum[N/4] / N
### 0.5
absspectrum[N/4+1] / N     # look in next higher bin
### 5.103003534765435e-15
absspectrum[N/4-1] / N     # look in next lower bin
### 5.1030035347654397e-15

# Move the test frequency halfway between two bins
signal = np.sin(2 * np.pi * (N/4.0+0.5) * np.arange(N) / N)  # N/4 freq.
absspectrum = np.abs(npf.fft(signal))
max(absspectrum) / N
# 0.31831188357043555
# notice that max is split between two adjacent bins:
absspectrum[N/4] / N
### 0.31830888749775132
absspectrum[N/4+1] / N
### 0.31831188357043544
# and also notice that max is surrounded by lower amplitude bins:
absspectrum[N/4+2] / N
### 0.10610629152320546
absspectrum[N/4-1] / N
### 0.10610329539412655

# Move the test frequency to the next higher bin:
signal = np.sin(2 * np.pi * (N/4.0+1) * np.arange(N) / N)  # N/4 freq.
absspectrum = np.abs(npf.fft(signal))
max(absspectrum) / N
# 0.5
absspectrum[N/4+1] / N
### 0.5
absspectrum[N/4] / N
### 6.5506535623712632e-15
absspectrum[N/4+2] / N
### 6.664158709764428e-15


## MDCT examining frequency offset by 0.5 bins:

# Move the test frequency halfway between two FFT bins, which
# is exactly on a frequency bin in the MDCT.
signal = np.sin(2 * np.pi * (N/4.0+0.5) * np.arange(N) / N)
absspectrum = np.abs(MDCT(signal))
max(absspectrum)
# 0.99999882345170166
# check to see if max is where it is expected:
absspectrum[N/4]
### 0.99999882345170166
# and notice that max is surronded by lower amplitude bins:
absspectrum[N/4+1]
### 1.3437304749009987e-16
absspectrum[N/4-1]
### 8.3154533161029903e-16

# Move the test frequency up by 1/2 bin, so it is centered on FFT bin
# N/4+1 but halfway between MDCT bins N/4 and N/4+1:

signal = np.sin(2.0*np.pi*(N/4+1.0) * np.arange(N) / N)
absspectrum = np.abs(MDCT(signal))
max(absspectrum)
### 0.63662002212358693
absspectrum[N/4-2]
### 0.00058593083592834024
absspectrum[N/4-1]
```

```
### 0.21220334506190164
absspectrum[N/4]
### 0.00097655445718997449
absspectrum[N/4+1]
### 0.63662002212358693
absspectrum[N/4+2]
### 0.0016276571495544786


#""
# Sweep a frequency past an FFT bin to examine continuous Fourier transform
# in the region between bins.
#"

import pylab as pl
import numpy as np
N = 1024;
window = HanningWindow(N)
#window = np.ones(N)
windowfactor = np.sum(window) / N
output  = np.zeros(N)
domain  = N/4.0 - 2.0 + (4.0 * np.arange(N) / N) - 256
for i in xrange(N):
   signal = np.sin(2*np.pi*(N/4.0-2.0 + 4.0*i/N)*np.arange(N)/N)
   absspectrum = np.abs(npf.fft(signal * window)) * 2 / N / windowfactor
   output[i] = absspectrum[N/4]

pl.clf()
pl.plot(domain, output)
pl.show()

# Sweep a frequency past an MDCT bin to examine continuous MDCT transform
# in the region between bins.

import pylab as pl
import numpy as np
N = 1024;
window = SineWindow(N)
#window = np.ones(N)
windowfactor = np.sum(window) / N
output  = np.zeros(N);
domain  = N/4.0 - 2.0 + (4.0 * np.arange(N) / N) - 256
for i in xrange(N):
   signal = np.sin(2*np.pi*(N/4.0+0.5-2.0 + 4.0*i/N)*np.arange(N)/N)
   absspectrum = np.abs(MDCT(signal * window)) / windowfactor
   output[i] = absspectrum[N/4]

pl.plot(domain, output)
pl.show()



#""
# Amplitude/Power scaling due to windowing within a fractional bin range.
#"

import pylab     as pl
import numpy     as np
import numpy.fft as npf
N = 1024
from craigwindow import *
sinwin = SineWindow(N)
hanwin = HanningWindow(N)

lowrange     = np.zeros(N+1)
lowrangesin  = np.zeros(N+1)
lowrangehan  = np.zeros(N+1)
for i in xrange(N+1):
   sig             = np.sin(2 * np.pi * (2.0+1.0*i/N) * np.arange(N) / N)
   spec            = npf.fft(sig)
   absspec         = np.abs(spec)
   lowrange[i]     = max(absspec)
   spec            = npf.fft(sig*sinwin)
   absspec         = np.abs(spec)
   lowrangesin[i]  = max(absspec)
   spec            = npf.fft(sig*hanwin)
   absspec         = np.abs(spec)
   lowrangehan[i]  = max(absspec)

hirange      = np.zeros(N+1)
hirangesin   = np.zeros(N+1)
hirangehan   = np.zeros(N+1)
for i in xrange(N+1):
   sig             = np.sin(2 * np.pi * (N/4+1.0*i/N) * np.arange(N) / N)
   spec            = npf.fft(sig)
   absspec         = np.abs(spec)
   hirange[i]      = max(absspec)
   spec            = npf.fft(sig*sinwin)
   absspec         = np.abs(spec)
   hirangesin[i]   = max(absspec)
   spec            = npf.fft(sig*hanwin)
   absspec         = np.abs(spec)
   hirangehan[i]   = max(absspec)

pl.clf()
pl.plot(np.arange(N+1.0)/N,lowrange,
        np.arange(N+1.0)/N,hirange,
        np.arange(N+1.0)/N,lowrangesin,
        np.arange(N+1.0)/N,hirangesin,
        np.arange(N+1.0)/N,lowrangehan,
        np.arange(N+1.0)/N,hirangehan)
pl.title("FFT")
pl.xlabel("fractional bin of signal frequency")
pl.ylabel("spectral bin maximum [amplitude]")
pl.show()



#""
# Examine effect of interaction between negative and positive components
# in the MDCT.
#""

import pylab     as pl
import numpy     as np
import numpy.fft as npf
N = 1024
from craigwindow import *
sinwin = SineWindow(N)
hanwin = HanningWindow(N)

lowrangesin  = np.zeros(N+1)
for i in xrange(N+1):
   sig             = np.sin(2 * np.pi * (2.0+1.0*i/N) * np.arange(N) / N)
   spec            = MDCT(sig*sinwin)
   absspec         = np.abs(spec)
   lowrangesin[i]  = max(absspec)

hirangesin   = np.zeros(N+1)
for i in xrange(N+1):
   sig             = np.sin(2 * np.pi * (N/4+1.0*i/N) * np.arange(N) / N)
   spec            = MDCT(sig*sinwin)
   absspec         = np.abs(spec)
   hirangesin[i]   = max(absspec)

pl.clf()
pl.title("MDCT")
pl.plot(np.arange(N+1.0)/N,lowrangesin, 'b-')
pl.plot(np.arange(N+1.0)/N,hirangesin,  'r-')
pl.xlabel("fractional bin of signal frequency")
pl.ylabel("spectral bin maximum [amplitude]")
pl.show()


"""
```

```
#!/usr/bin/env python
#
# Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
# Creation Date: Mon Feb  1 21:31:24 PST 2010
# Last Modified: Wed Feb 17 21:25:28 PST 2010
# Filename:      craigwindow.py
# Syntax:        Python 2.6
#
"""
Signal windowing functions for Music 422.  Implementations of the Sine
Window, the Hanning Window for both even and odd window lengths, as
well as the Kaiser-Bessel Derived Window.  Also contains a test function
called ColaMirrorTest() which can be used to check if a particular window
will generate a constant-amplitude reconstruction at 50% overlap between
adjacent windows.


Compatible importing method:

from craigwindow import ApplySineWindow    as SineWindow
from craigwindow import ApplyHanningWindow as HanningWindow
from craigwindow import ApplyKBDWindow     as KBDWindow


Although it would be more efficient to not regenerate the window
values for each application of the window:

from craigwindow import SineWindow
from craigwindow import HanningWindow
from craigwindow import KBDWindow

And then store the precalculated window in an array to apply to
signals with a multiply, for example:

import random
signal = [random.random() for i in xrange(1024)]
window = SineWindow(1024)
windowedsignal = signal * window

"""

import numpy as np


def ApplySineWindow(signal):       return signal * SineWindow(len(signal))
def ApplyHanningWindow(signal):    return signal * HanningWindow(len(signal))
def ApplyKBDWindow(signal, alpha): return signal * KBDWindow(len(signal), alpha)


###########################################################################


##############################
##
## SineWindow -- page 107, equation 2 (Even-length sine window definition).
##   Also the odd-length window definition is implemented by this function
##   suitable for 50% overlap-add (derived from continuous sine window
##   definition in equation 1 on page 107).
##

def SineWindow(length):
   """Sine Window -- Generates a sine window which, when 50% overlapped,
      generates a constant overlap-add (COLA) amplitude when the window
      is squared (applied twice to a signal).  Shift of 1/2 sample for
      even lengths because window is sampled from the middle of the window,
      not from left end of window.  Similarly, scaling by N-1 in the odd
      case such that the continuous window definition is sampled symmetrically
      around the middle of the window.
      Argument:
         length: Length of the output window array.
   """
   N = int(length)
   assert N > 1
   if (N%2 == 0):
      halfwindow = np.sin((np.arange(N/2)+0.5)*np.pi/N)
      return np.concatenate([halfwindow, np.flipud(halfwindow)])
   else:
      halfwindow = np.sin(np.arange(N/2+1)*np.pi/(N-1))
      # don’t repeat the middle sample of an odd window:
      return np.concatenate([halfwindow, np.flipud(halfwindow[0:-1])])



##############################
##
## HanningWindow -- page 107, last equation.  Even-length window definition.
##    Odd-length version also implemented in this function.
##

def HanningWindow(length, power=2):
    """Hanning Window -- Raised Cosine, Cosine**2 or Sine**2 window.
       Arguments:
          length: Length of the output window array which must be positive.
          power:  power=2 for Hanning window, power=1 for Sine Window.
    """
    return SineWindow(length)**power



##############################
##
## KBDWindowKVH -- from Katarina Van Heusen’s solution, implemented
##                 using numpy.kaiser function, and then deriving KBD.
##

def KBDWindowKVH(length, beta):
   """Kaiser Bessel Derived Window
      Arguments:
         length: Length of output window array.  Must be positive & even.
         beta:   Kaiser window parameter alpha times pi.
   """
   N      = int(length)
   assert N%2 == 0
   assert N > 0
   M      = N/2
   w      = np.kaiser(M+1, beta)
   wMsum  = 1.0 / np.sqrt(w[0:M+1].sum())
   wnsuml = np.sqrt(np.cumsum(w[0:M]))
   wl     = wnsuml * wMsum
   return np.concatenate([wl, np.flipud(wl)])


def KBDWindow(length, alpha):
   return KBDWindowKVH(length, np.pi * alpha)



##############################
##
## KBDWindowCSS -- Craig’s solution, implemented from bessel I_0 function.
##    Based on C program implementation:
##       https://ccrma.stanford.edu/courses/422/projects/kbd/kbdwindow.cpp
##

def KBDWindowCSS(length, alpha):
   """Kaiser Bessel Derived Window
```

```
      Arguments:
         length: Length of output window array.  Must be positive & even.
         alpha:  Kaiser window parameter.
   """
   N         = int(length)
   assert N % 2 == 0
   assert N > 0
   halfN     = N/2
   beta      = np.pi * alpha
   bessels   = np.i0(beta * np.sqrt(1.0-(4.0*np.arange(halfN)/N-1.0)**2.0))
   halfwin   = np.cumsum(bessels)
   normalize = halfwin[-1] + np.i0(beta*np.sqrt(1.0-(4.0*halfN/N-1.0)**2.0))
   halfwin   = np.sqrt(halfwin/normalize)
   return np.concatenate([halfwin, np.flipud(halfwin)])



##############################
##
## KBDWindowCAR -- from Colin Raffel's solution: Implements Kaiser window, then
##   Applys KBD modification.
##

def kaiserWindow(length, alpha):
   """Kaiser-Bessel Window
      Arguments:
         length: Length of output window array.  Must be positive & even.
         alpha:  Kaiser window parameter.
   """
   N = int(length)
   n = np.arange(N)
   w = (n - (N-1)/2.0)/((N-1)/2.0)
   w = w**2
   w = np.sqrt(1-w)
   w = np.pi * alpha * w
   w = np.i0(w)
   w = w/np.i0(np.pi*alpha)
   return w


def KBDWindowCAR(length, alpha):
   """Kaiser-Bessel Derived Window
      Arguments:
         length: Length of output window array.  Must be positive & even.
         alpha:  Kaiser window parameter.
   """
   N      = int(length)
   assert N%2 == 0
   assert N > 0
   wKai   = kaiserWindow(N/2+1, alpha)
   wHalf  = np.sqrt(np.cumsum(wKai[0:-1])/np.sum(wKai))
   return np.concatenate([wHalf, np.flipud(wHalf)])




##############################
##
## ColaMirrorTest --
##

def ColaMirrorTest(window, power = 2):
   """ColaMirrorTest -- Test for Constant OverLap-Add requirement in TDAC
      process which is that the square of the window values plus the
      mirror image of each half window squared sums to one (or at least
      a constant).  The central value of an odd-length window will be
      checked agains both ends of the window (resulting in an additional
      middle value in the output array).  Check the max an min values
      of the output array to verify that they are equal (or extremely close).
      Arguments:
         window: array containing a window.
         power:  power value to apply to window values. (1 for hanning,
                 2 for sine or kbd)
   """
   if (type(window) == list): window = np.array(window)
   N       = window.size
   power   = float(power)
   half    = int(N/2.0)
   output  = np.zeros(N, dtype=float)
   powed   = window**power
   if (N % 2 == 0):
      firsthalf  = powed[0:N/2]
      secondhalf = powed[N/2:N]
      return np.concatenate([firsthalf + np.flipud(firsthalf),\
                             secondhalf + np.flipud(secondhalf)])
   else:
      # returns length N+1, since checking middle sample twice
      firsthalf  = powed[0:N/2+1]
      secondhalf = powed[N/2:N]
      return np.concatenate([firsthalf + np.flipud(firsthalf),\
                             secondhalf + np.flipud(secondhalf)])



###########################################################################
#
# Testing code:
#

if __name__ == "__main__":

   """
   # Plot various windows with 1024 length ##############################
   #"""

   import pylab as pl
   N = 1024
   n = np.arange(N)/(N-1.0) # normalized range
   lines = pl.plot(n, SineWindow(N),      \
                   n, KBDWindow(N, 1),    \
                   n, HanningWindow(N),   \
                   n, KBDWindow(N, 4))
   pl.legend(lines, ("Sine Window",       \
                     "KBD Win alpha=1",   \
                     "Hanning Window",    \
                     "KBD Win alpha=4"),  \
                     "lower center")
   pl.axis('tight')
   pl.xlabel('normalized windowlength')
   pl.ylabel('amplitude')
   pl.suptitle('Four examples of windows')
   pl.show()


   #"""
   # Test windows to make sure they are COLA: ###########################
   #"""

   sinewin  = SineWindow(1024)
   sinecola =  ColaMirrorTest(sinewin)
   print "Min value:", np.min(sinecola)
   print "Max value:", np.max(sinecola)
   ### Min value: 1.0
   ### Max value: 1.0

   hanwin  = HanningWindow(1024)
```

```
hancola =  ColaMirrorTest(hanwin, 1.0)
print "Min value:", np.min(hancola)
print "Max value:", np.max(hancola)
### Min value: 1.0
### Max value: 1.0

kai100win  = KBDWindowCSS(1024, 1.00)
kai100cola =  ColaMirrorTest(kai100win)
print "Min value:",   np.min(kai100cola)
print "Max value:",   np.max(kai100cola)
### Min value: 1.0
### Max value: 1.0

kai200win  = KBDWindowKVH(1024, 2.00)
kai200cola =  ColaMirrorTest(kai200win)
print "Min value:",   np.min(kai200cola)
print "Max value:",   np.max(kai200cola)
### Min value: 1.0
### Max value: 1.0

kai399win  = KBDWindowCAR(1024, 3.99)
kai399cola =  ColaMirrorTest(kai399win)
print "Min value:",   np.min(kai399cola)
print "Max value:",   np.max(kai399cola)
### Min value: 1.0
### Max value: 1.0

# Make a transitional window from Sine into KBD:
N = 1024
hywin = np.concatenate([SineWindow(N)[0:N/2], KBDWindow(N,4)[N/2:N]])
hywincola =  ColaMirrorTest(hywin)
print "Min value:",   np.min(hywincola)
print "Max value:",   np.max(hywincola)
### Min value: 1.0
### Max value: 1.0



#""
# Comparison test between KBDWindowKVH and KBDWindowCSS and KBDWindowCAR:
#""

KBDWindowCSS(2048, 1.0)[0:10]
### array([0.01629672,  0.02310258,  0.02836276,  0.03282897,  0.0367916,
###        0.04039919,  0.04373976,  0.04687052,  0.04983111,  0.05265029])

max(np.abs(KBDWindowKVH(2048, np.pi) - KBDWindowCSS(2048, 1.0)))
### 6.6613381477509392e-16

max(np.abs(KBDWindowCAR(2048, np.pi) - KBDWindowCSS(2048, 1.0)))
### 6.6613381477509392e-16

max(np.abs(KBDWindowCAR(2048, 1.0) - KBDWindowKVH(2048, np.pi)))
### 2.2204460492503131e-16

import time
t1 = time.clock()
csswin = KBDWindowCSS(1000000, 1.0)
t2 = time.clock()
print "CSS TIME:", t2 - t1, "seconds"
### CSS TIME: 0.90310835761 seconds

t1 = time.clock()
kvhwin = KBDWindowKVH(1000000, 1.0)
t2 = time.clock()
print "KVH TIME:", t2 - t1, "seconds"
### KVH TIME: 0.900658371609 seconds
```

```
t1 = time.clock()
carwin = KBDWindowCAR(1000000, 1.0)
t2 = time.clock()
print "CAR TIME:", t2 - t1, "seconds"
### CAR TIME: 0.905957039338 seconds



#""
# Test window-application functions which internally generate a window.
#""

import pylab as pl
N = 1024
signal = np.sin(2.0 * np.pi * 3000.0 * np.arange(N) / 44100.0)
n = np.arange(N)/(N-1.0) # normalized range
pl.clf()
lines = pl.plot(n, ApplySineWindow(signal),     \
                n+1, ApplyHanningWindow(signal),  \
                n+2, ApplyKBDWindow(signal, 1),   \
                n+3, ApplyKBDWindow(signal, 4))
pl.axis('tight')
pl.xlabel('normalized windowlength')
pl.ylabel('amplitude')
pl.suptitle('Various windows applied to a sinewave')
pl.text(0.5, 0.0, 'sine\nwindowed', horizontalalignment='center',
                                  verticalalignment='center')
pl.text(1.5, 0.0, 'hanning\nwindowed', horizontalalignment='center',
                                  verticalalignment='center')
pl.text(2.5, 0.0, 'KBD windowed\nalpha=1', horizontalalignment='center',
                                  verticalalignment='center')
pl.text(3.5, 0.0, 'KBD windowed\nalpha=4', horizontalalignment='center',
                                  verticalalignment='center')
pl.show()



#""
# Visualize constant overlapp-add of windows
#""

import pylab as pl
pl.clf()
N = 1024
window = KBDWindow(1024, 4.0)
winl = window[0:N/2]
winr = window[N/2:N]
overlapsum = winl + winr
win2l = winl**2
win2r = winr**2
overlap2sum = win2l + win2r
n = np.arange(N)/(N-1.0) # normalized range
nl = n[0:N/2]
nr = n[N/2:N]
pl.plot(nl,   winl,        'b-')
pl.plot(nl,   winr,        'b-')
pl.plot(nl,   overlapsum,  'r--')
pl.plot(nl+1, win2l,       'b-')
pl.plot(nl+1, win2r,       'b-')
pl.plot(nl+1, overlap2sum, 'r--')
pl.text(0.25, 1.1, 'not COLA$^1$', \
        horizontalalignment='center', verticalalignment='center')
pl.text(1.25, 1.1, 'COLA$^2$', \
        horizontalalignment='center', verticalalignment='center')
pl.text(0.5, 0.5, 'KBD Window, $\\alpha=4$\n50% overlap', \
        horizontalalignment='center', verticalalignment='center')
pl.text(1.0, 0.5, 'KBD Window$^2$, $\\alpha=4$\n50% overlap', \
        horizontalalignment='center', verticalalignment='center')
```

```
pl.show()


"""
```