ELE 201, Spring 2014
Laboratory No. 3
Filtering, Sampling, and Quantization

1 Introduction

The first part of this lab focuses on filtering. "Filtering" is a general term for generating an output signal by applying some operator (often a convolution) to an input signal. A good example is the averaging filter, where the output pixel intensity is simply the mean intensity of some of the nearest neighbors of the input pixel. Many filters are *time-invariant*, which means that the same operator is applied at every sample of the input, instead of the operator changing depending on which sample is being processed. Linear time-invariant filters are generally the easiest to analyze and sometimes have implementation advantages in the real world. We will concentrate on those types of filters in this lab. For these filters, we can use the powerful tools of frequency domain analysis.

Though filters can be quite simple, they can accomplish a surprising range of tasks: reducing the noise in a corrupted signal, enhancing the signal to achieve a perceptual effect, or even separating two signals which occupy the same segment of time. As we shall see, depending on the task at hand, it may be easier to design the filter in either the time or frequency domain.

The second part of this lab involves digitizing a signal. We will investigate the effects of sampling and quantization. Filtering and frequency domain analysis are important components of sampling.

2 Lab Procedure - Part 1

This part of the lab is for experimenting and developing intuition for convolution and filtering. We will filter unwanted noise, separate signals, and enhance sound and images. The “extras” section has additional operations for images, beyond convolution.

2.1 Convolution of 1-Dimensional Signals

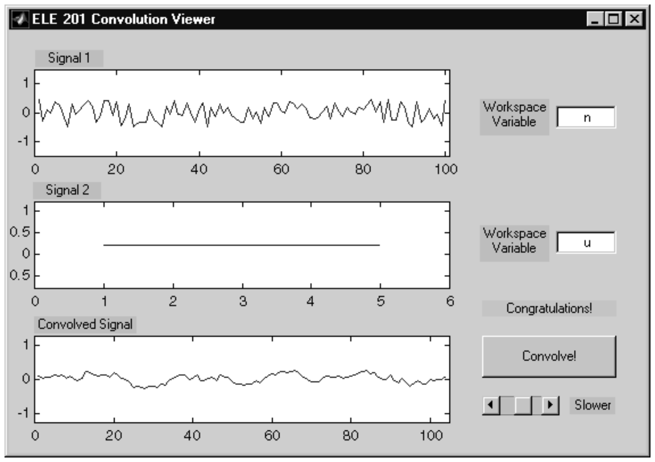


Figure 1: ELE 201 Convolution Viewer.

Bring up the ELE 201 Convolution Viewer by typing `convview`. This is a graphical interface to the Matlab `conv` function that illustrates the way two signals are convolved. You can change the speed at which the animation plays by moving the slider (for longer signals, you should make the animation go faster.) Create some short test signals, including:

- u, a 2-element vector of 1s
- v, a 5-element vector of 1s
- w, a 10 element vector of five -1s followed by five 1s
- n, a 101-element vector of *zero-mean* uniform noise
- r, the ramp `1:5`
- s, a slowly varying sine wave corrupted by noise, e.g. `sin(2*pi*4*[0:.01:1])+n`

Now, look at the following convolutions, and sketch the results in your write-up:

```
u * u    u * v    r * v
v * w    u * s    v * s
```

Comment especially on the results of convolving the noisy sine wave with u and v. Also, try convolving one or two of your own test signals. Comment on anything interesting you see. D1

2.2 Convolutions of Images

We can also apply convolution to images using the `conv2` command. Load the Chrysler building image from **artdeco.mat** and try convolving the image with the small matrix blow. The result will be a local averaging.

$$lp = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Use the command `conv2(im,lp,'same')` to perform the convolution. The third input, 'same,' just tells `conv2` to trim the edges to keep the image the same size as `im`. What does the resulting image look like and why (use `imshow`)? What happens to the DFT of the convolved image and why? You need to use the 2-D command `fft2` as you did in Lab 1. Look at the log of the magnitude, and use `fftshift`. $\boxed{\text{D2}}$

Compare the DFT of the image before and after convolution. Also, calculate the DFT of the matrix *lp* itself. Pad zeros around the matrix to make it the same size as the image. That is, define a zero matrix of the same size as the original image, and change the 5 by 5 submatrix in the center to *lp*. If you do not pad zeros, the DFT will be a sampling of the Fourier transform that is too course to be interesting. $\boxed{\text{M1}}$

2.3 Time Domain vs. Frequency Domain Filtering

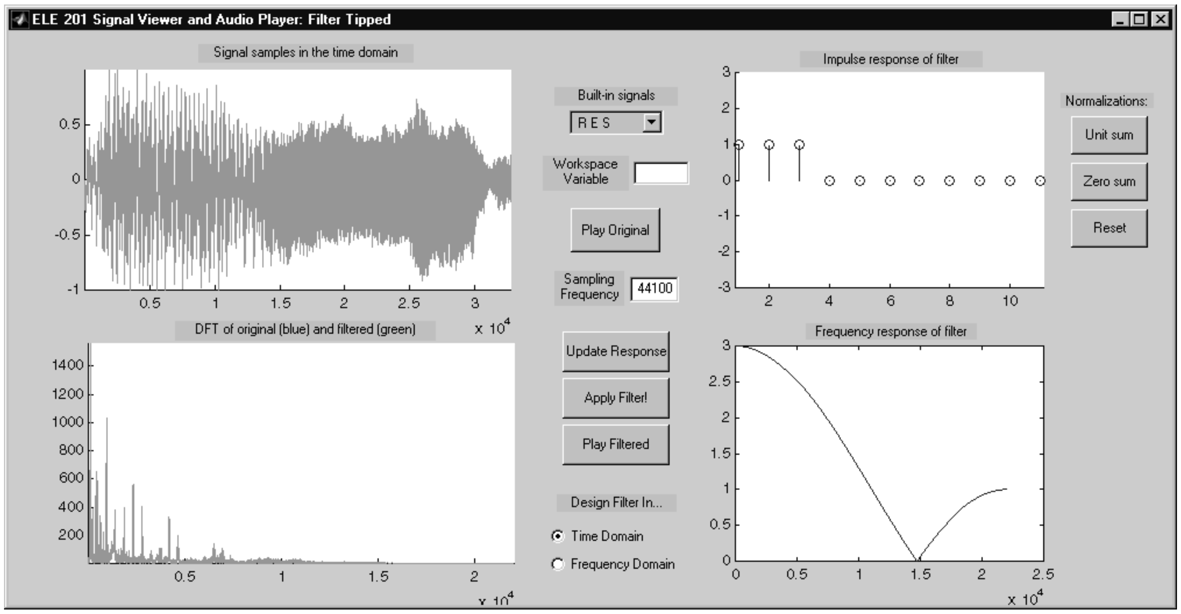


Figure 2: Yet another version of `splay`.

Run `splay4.m`, which will allow you to apply filters to signals. You can design filters in either the time or frequency domains. The radio buttons at the middle bottom of the window control which domain you design in. In either domain, clicking "Update Response" updates either the impulse or frequency response of the filter, but doesn't apply the filter to the signal (this way you can experiment with getting the filter characteristics you want before you go to the trouble of applying it.) "Apply Filter" both updates the responses and applies the filter to the signal.

When designing in the time domain, you can drag the filter taps up and down (the value will "snap" to a hidden grid.) In some cases, it's useful to normalize the filter coefficients so that they either sum to 1 or sum to 0, which are what the right-hand buttons are for, and you can reset the filter back to a delta function with the Reset button.

In the frequency domain design mode, you can drag the transparent bars up and down, much as you would adjust a graphic equalizer on your stereo. You see the DFT of the original signal in red behind the bars so that you can design your filter to dampen or enhance certain regions of the spectrum. The filter that `splay4` produces will not actually be a sharp as the interface suggests. The actual frequency response will be smoothed out a bit. For finer control of the filter design, you can enter low and high thresholds in the boxes to the right of the window, which narrows the range over which the equalizers operate. (**Note:** when designing over a narrower range, the left-most equalizer sets the frequency response for the left-most bin *and* all bins to the left; the right-most equalizer sets the frequency response for the right-most bin *and* all bins to the right. This allows you to design band-pass filters easily.) Again, the reset button returns all the equalizer bars to unit gain.

When you switch to a new signal, the filter automatically resets, regardless of the mode you're in.

Now, a few exercises to get familiar with designing filters and looking at their responses.

In time domain, design the following four filters:

1. A filter with impulse response $[\frac{1}{3}\ \frac{1}{3}\ \frac{1}{3}]$
2. A filter with impulse response $\frac{1}{10}\cdot$ `ones(1,10)`
3. A filter with impulse response $[1\ -1]$
4. A filter with impulse response $[\ 1\quad -2\quad 2.5\quad -2\quad 1\ ]$

In your write-up, sketch the frequency response of these four filters, and note whether the filter is low-pass, high-pass, or something else. D3

Now, in the frequency domain, design the following two filters (keep the band edges at 1 and 5000 Hz):

1. A low pass filter (e.g. the left 5 bars at 1, the rest at 0)
2. A high pass filter (e.g. the right 2 bars at 1, the rest at 0)

In your write-up, describe (very roughly!) the impulse responses of these two filters, and note roughly how long they are. What does this tell you about the trade-offs between designing filters in the time domain or in the frequency domain? D4

Demonstrate these results. M2

2.4 Noise Reduction

In Lab 1, you experimented with adding noise to a signal. Now we'll investigate ways to use filters to remove noise from a signal. Use the built-in signal "Noisy onions" for this section. This is a version of "Onions" with some uniform random noise added. Compare the two versions. What does the noise look like in the frequency domain? Using a suitable frequency-domain filter, try to remove as much of the noise as you can while keeping the original signal. What are the trade-offs involved? How does your cleaned-up signal sound? Describe in your write-up what kind of filter you used. Q1 D5

Load the mat-file **lab3.mat**, which will place a set of images in your workspace. The names of these images are the numbers `one` through `five`.

Images can be noisy too; consider the image *three*, which is an original image corrupted by additive Gaussian noise. Can you de-noise the image using the same ideas you used to de-noise the sound? For this, it's easiest to design a filter h in the spatial domain and use `conv2(three,h,'same')`. Describe in your write-up what filter you used and why, and what the resulting image looks like. D6

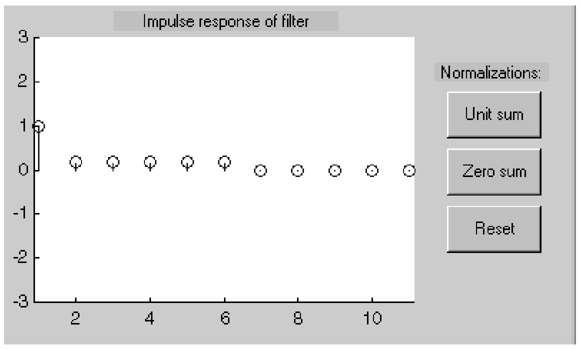
The image *four* is an image corrupted by impulsive black and white noise (sometimes called "salt and pepper" noise). How does the filter that you used for the *three* image work here? Probably not so well. Instead, try applying what's called a *median filter*. This is a nonlinear operation which looks at a neighborhood or each input pixel and assigns the output pixel value to be the median, or middle, intensity value in that neighborhood. This operation is encapsulated in the Matlab command `medfilt2`; use the syntax `fourm = medfilt2(four)`.[1] How does the resulting image look? Why does median filtering work so well for this kind of noise? $\boxed{\text{D7}}$

Conversely, try applying median filtering to the *three* image. How does the filtered image look, and why? $\boxed{\text{D8}}$

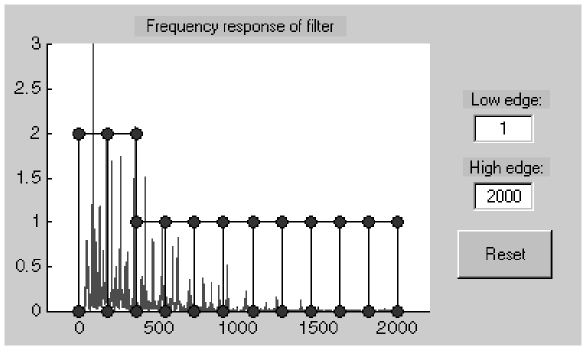
Demonstrate these results. $\boxed{\text{M3}}$

2.5 Enhancement

Sometimes filters are used not to remove noise from a signal, but to enhance certain aspects of it. For example, on lots of CD players and car stereos, there's a "bass boost" button which has the effect of making the music "bass-heavy". Try bass-boosting the built-in signal "Onions" in two ways:



(a)



(b)

Figure 3: Bass boosting filters

1. Use the filter in the time domain you get by starting with [1 1 1 1 1 1], clicking "Unit sum", and dragging the first filter tap up to 1 (see Figure 3a).
2. Use the filter in the frequency domain you get by setting the low edge to 1, the high edge to 2000, the left 2 bars at 2, and the rest at 1 (see Figure 3b).

How does each filtered signal sound, and why? Which do you think sounds better? $\boxed{\text{D9}}$

Images can be enhanced in various ways, too. One example is called "high emphasis" filtering, in which a high-pass-filtered version of the image is added to the original to produce a certain effect. One spatial-domain approximation of this effect convolution with the following matrix:

$$hefilt = \begin{bmatrix} 0 & -1.5 & 0 \\ -1.5 & 7 & -1.5 \\ 0 & -1.5 & 0 \end{bmatrix}$$

Take the image *five* and convolve it with `hefilt`. Describe what the filter seems to do to the image? $\boxed{\text{M4}}$ $\boxed{\text{D10}}$

[1]By default, the `medfilt2` function uses a 3×3 neighborhood of each pixel.

2.6 Signal Separation

If two signals are well-localized in the frequency domain, it may be possible to separate them. For example, consider the signal called "Phone Darth", which consists of various familiar sounds. Using the filter design tools in `splay4`, and what you remember about the frequency content of these sounds, can you isolate Darth from the other signals? What filter did you use, and why? $\boxed{\text{D11}}$

3 Lab Extras

If you have time, here are some extra activities to try. This section is not to be turned in for credit but is highly recommended.

3.1 Memoryless Point Operations

Even though many filters operate on a neighborhood of samples around the input sample, many interesting effects can be achieved by simply applying a function $f(\cdot)$ to each input sample to produce the corresponding output sample, independent of the time/location of the sample and independent of other samples. These types of operations are called *memoryless point* operations. To begin with, we'll consider a few of these simple operations, using images as our signals.

3.2 Increasing the Dynamic Range

Take a look at the image *one* using the function `imshow`. How are its intensities distributed? Hint: use the image histogram command `imhst` to answer this question more precisely.

Unless there's a compelling reason not to, we would generally like to increase the *dynamic range* of the image so that the darkest pixel is black (intensity 0) and the lightest pixel is white (intensity 1). One way to do this is by stretching the histogram of the image so it takes up all of the $[0, 1]$ range. You should be able to do this with a few simple Matlab commands. (Hint: You'll need to use the minimum and maximum intensities of the image, given by `min(x(:))` and `max(x(:))` respectively. Use a linear (affine) function to spread the range from 0 to 1.)

Apply your range mapping commands to the image *one* and describe how it looks. Sketch (very roughly) in your lab what the "before" and "after" histograms look like.

3.3 Histogram Equalization

Take a look at what happens when you apply your dynamic range routine to the image *two*. Again, sketch the before and after histograms in your lab. Why is the result still somewhat unsatisfactory?

A way to avoid the problem seen above is to not only stretch the histogram to make it take up all of the $[0, 1]$ range, but to also make it as flat as possible, so that the intensities are roughly uniformly distributed. Lucky for you, the function `hsteq` does this already. The syntax is `out = hsteq(in)`. Use this function to histogram-equalize *two*, and take a look at the output image and its histogram.

3.4 Digital Negative

You know what a film negative looks like; how would you make the negative of a digital image? That is, white should become black and vice versa. You should be able to do this with a simple line of Matlab code. What does this do to the histogram of the image?

4 Lab Procedure - Part 2

This part of the lab procedure is about sampling and quantization.

In order for computer software like Matlab to process signals, analog signals must be made digital. Conversely, in many systems, digital signals at some stage are generally made analog.

When someone sings into a microphone, the analog signal is transmitted along a wire into a mixing console which may be analog or digital. The mixed version can be transferred to LP or audio tape (analog) or CD or MP3 (digital). Finally, the signal must be recreated as an analog signal to be played on your headphones or stereo speakers. The same applies to images and video: Film can for most purposes be considered analog, but digital processing is often used to add special effects in post-production and is increasingly used for storage. The digital signal needs to be converted to analog to be displayed on movie screens or older TV sets, or remains in the digital domain to be viewed on computer monitors or HDTV sets. However, between the viewing device and your brain is the analog human audio-visual system.

The processes of sampling and quantization are at the heart of the conversion between the analog and digital domains. The goals of this lab are for you to:

- Hear the effects of sampling (and undersampling) on audio.
- Observe the effects of sampling an image and investigate simple subsampling algorithms.
- Hear and see the effects of quantization in sounds and images.
- Learn about how dithering can improve the perceptual quality of a quantized signal.
- Learn how grayscale images are halftoned for viewing in newspapers and laser printouts.
- See some recent computer graphics research related to these problems.

Since this lab is mostly demonstrations, your grade depends on your responses to the many questions sprinkled throughout this handout. Be sure to answer them all in your write-up!

NOTE: Don't spend time in lab sketching or printing out each signal and Fourier transform. The idea is for you to experiment freely, and comment on interesting things you discovery.

4.1 Sampling

4.1.1 Audio

How should an analog signal be converted to the digital domain? A reasonable idea is to take equally spaced samples of the signal, as illustrated in Figure 4a. Digital audio is commonly sampled at 44100 samples per second.

As noted in the introduction, to play a digital signal as a sound through a speaker, an analog version must be reconstructed. Figure 4b illustrates reconstruction by *zero-order hold*, which makes an analog stair-step function out of the data. Another idea is to connect the samples with straight lines (sometimes called *first-order hold*) as in Figure 4c. Higher level interpolation using more complicated interpolating functions results in an even smoother reconstructed signal as in Figure 4d. However, if the original signal is not sampled often enough, as in Figure 4e, important detail is lost and the reconstructed signal is a poor approximation to the original, as in Figure 4f.

Yes, the `splay` function has been changed again... this time, you'll use `splay3`, an interface which allows you to change the subsampling factor and reconstruction method for audio signals.

First, choose the "Tiny sine" built-in signal. It displays in the upper window in blue. If you play it, you'll hear a very short beep. The green circles indicate the locations where the original signal is subsampled, and the solid green line indicates the reconstructed signal. If you change the reconstruction method, you'll see the green signal change. Note that the reconstructed signal always passes through the locations where the

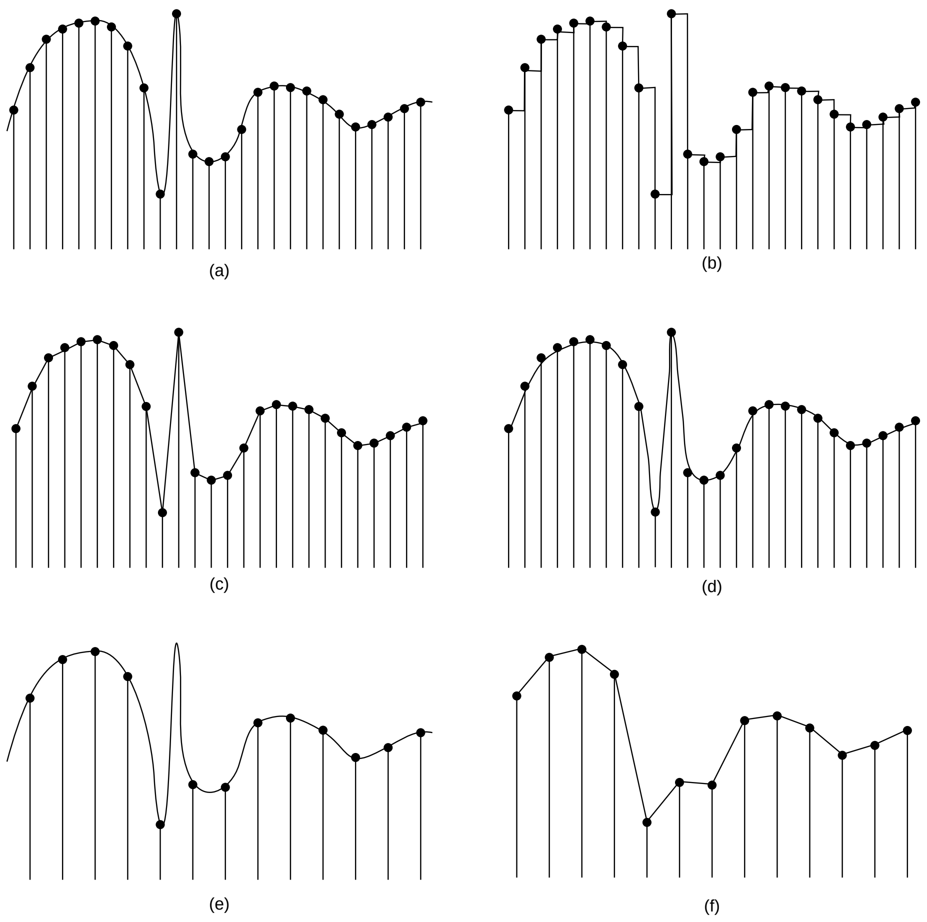
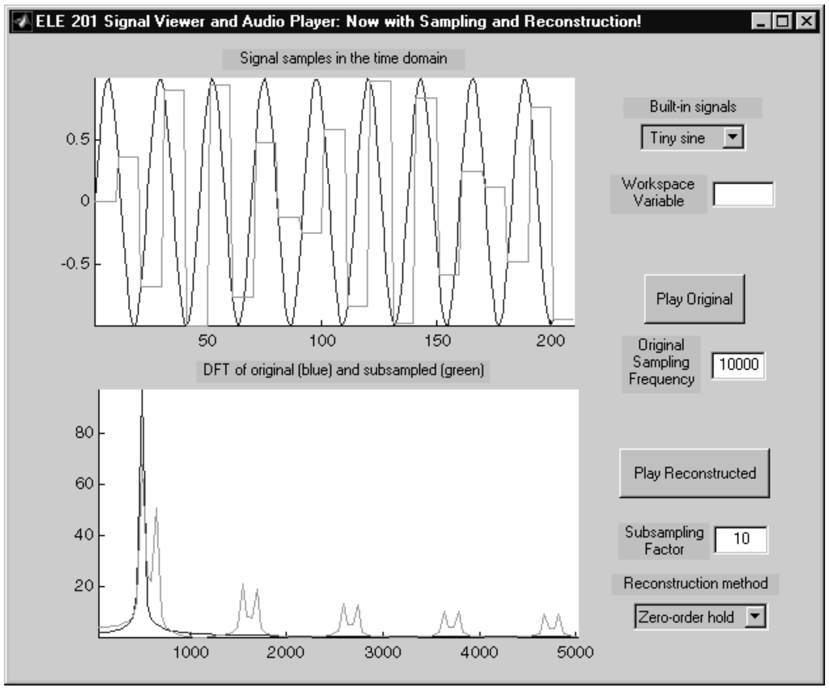


Figure 4: Sampling a signal. (a) Original signal and samples. (b) Zero-order hold. (c) Linear interpolation. (d) Higher-level interpolation. (e) Under-sampling. (f) A poor reconstruction.


ELE 201 Signal Viewer and Audio Player: Now with Sampling and Reconstruction!
Signal samples in the time domain
DFT of original (blue) and subsampled (green)
Play Original
Play Reconstructed
0.5
0
-0.5
50
100
150
200
80
60
40
20
1000
2000
3000
4000
5000

Built-in signals: Tiny sine
Workspace Variable:
Original Sampling Frequency: 10000
Subsampling Factor: 10
Reconstruction method: Zero-order hold

Figure 5: Yet another version of `splay`

original was subsampled. The lower window shows the DFT of the original and reconstructed signals. The better the reconstruction is, the closer the DFTs will be.

For the tiny sine, start with a subsampling factor of 5, and try the three reconstruction methods. What happens to the sound of the reconstructed signal? What happens to the DFT? D12

Next try a subsampling factor of 10, then a factor of 15, then a factor of 20. You don't need to describe what you see and hear in detail in your write-up. Just play with the interface to get a feel for what happens. Generally, what happens as you increase the interval between samples? In particular, comment Q2 on the reconstruction for subsampling factor 20 using first-order hold and sinc interpolation. What is this phenomenon called? Q3

Theoretically, under suitable assumptions, signal reconstruction can be achieved from samples with no loss of information. If the signal is sampled faster than a certain rate (called the *Nyquist rate*) the signal can be reconstructed perfectly using sinc interpolation functions. If the signal is sampled below the Nyquist rate, perfect reconstruction is impossible.

Now, we'll do a short experiment with the dial tone, which you may recall consists of two sinusoids at 350 and 440 Hz. Use a subsampling factor of 10 and the three different reconstruction methods. What do you see and hear? Now just use the sinc interpolation method of reconstruction for subsampling factors of 11 and 12. What do you hear, and what do you see in the frequency domain? Can you explain these results in terms of the sampling theorem and the Nyquist rate? (Some calculation is required to get the answer.) D13

Finally, try the signal "R E S" with a subsampling factor of 10 and the three different reconstruction methods. What do you see and hear? For this long signal, you should see that in the frequency domain, the sinc interpolation function is acting like a low-pass filter. D14

4.1.2 Images

We'll now consider the effect of sampling on images. For this part, we'll look at a different issue regarding sampling. Namely, we'll look at how the choice of the sampling process can affect the resulting sampled signal. This issue can be particularly important in cases where upsampling or reconstruction is not done prior to viewing.

Load **artdeco.mat**. Let's investigate a few different ways of making an image four times as small in each direction.

One way is to just take a representative pixel from every 4×4 block, as done in Lab 1.

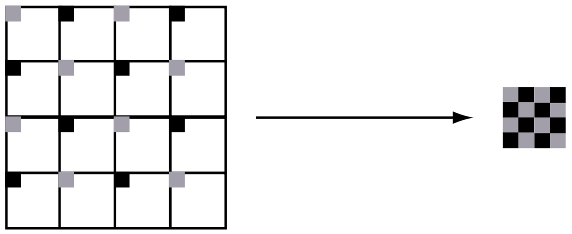


Figure 6: Subsampling by taking the $(1, 1)$ element of each 4×4 block.

Recall from Lab 1 that you can implement this kind of subsampling command in Matlab using

```
≫ im1 = im(1:4:end,1:4:end);
```

Display the result with `imshow`. How does the image look compared to the original? Does it do a good job of capturing the image content? D15

Our choice of the $(1, 1)$ entry in each block was arbitrary; we could have easily chosen a different representative. For example, we can choose the $(4, 2)$ entry of each block using:

```
≫ im1 = im(4:4:end,2:4:end);
```

How does this image look compared to the previous one? What does this tell you about the problems with this type of subsampling? D16

A better idea is to assign a new intensity value to each 4×4 block based on its average intensity. The `blkproc` command can be used to implement this kind of subsampling:

```
≫ f = inline('mean(x(:))');
≫ im2 = blkproc(im,[4 4],f);
```

Compare this image to the image you got from subsampling the $(1, 1)$ entry of each block. Which looks better, and why? D17

You can explore the effects of subsampling at coarser levels by replacing `[4 4]` in the Matlab command with a different block size. Try `[8 8]` and `[16 16]`. For the `[16 16]` image, roll your chair back from the screen a ways and squint (alternatively, resize the figure window to make the image a lot smaller). Remember that the perceptual quality of a subsampled image depends on the size at which it's displayed! M5

4.2 Quantization

The purpose of a quantization routine is to process an input signal so that each sample can take on one of only a finite number of levels. Sometimes the levels are fixed from external constraints (e.g. the quantization of color on a computer monitor), and sometimes they can depend on the signal.

For the first part of the section, we'll use the quantization function `uquant`, which is called this way:

```
≫ qsig = uquant(sig,numlevels);
```

This quantizes the input signal to the number of levels specified in the variable *numlevels.* This routine implements *uniform quantization*, which means that the levels are equally spaced across the range of the data.

4.2.1 Audio

Load the m-file **respect.mat**, which will create a variable r in your workspace. Quantize the signal using $N = 8$ in `uquant`, and play the signal using `soundsc`. Remember to set the sampling frequency to 44100! How does it sound compared to the original signal? D18

Now try quantizing to even fewer levels using $N = 4$ and $N = 2$. How does the sound quality degrade? Sound on a computer will be quantized by the sound card, sometimes to only 8 bits, i.e. $N = 2^8 = 256$ levels, depending on the quality of the sound card. Can you tell the difference between this finely quantized sound and the original? D19 Q4

4.2.2 Images

You may be more familiar with the quantization of images. The settings in your computer's operating system should allow you to change the number of colors which are displayed on your monitor. In order to display an image which contains more colors than the monitor can display, it must be quantized to a lower number of colors.

Again use the image in **artdeco.mat**. Try quantizing this image to 16, 8, and 4 levels, looking at the quantized image after each step. (`uquant` is smart enough to display the image for you.) How does the image quality change? Pay particular attention to what happens to the clouds in the upper right hand corner; this phenomenon is called *contouring.* D20

Now load the file **artdeco2.mat**, which contains a copy of the Art Deco image which has some uniform random noise added to it. (The variable *im2* is the matrix which contains the picture.) What happens to the contouring when you quantize this image to 8 levels? D21

So far we've only dealt with uniform quantization, but for signals whose values are not roughly uniformly distributed, this can be a bad idea. For example, load the file **hhold.mat** and take a look at the image *hh.* What can you say about the distribution of intensities in this image? (Hint: try using the `imhist` function, which will show you a histogram of intensities.) What happens when you uniformly quantize the image to 3 levels? D22 D23

We've also supplied a non-uniform quantization function `quant`, which is called this way:

```
≫ qsig = quant(sig,levels);
```

This quantizes the input signal to the list of levels specified in the vector variable *levels.* (Put the levels you want to use in square brackets. They should be between 0 and 1.)

Based on the histogram of the image intensities, can you select a set of 3 levels that gives a more satisfying quantized image? Use `quant` to do the non-uniform quantization. What levels did you choose? M6 Q5

4.3 Trade-offs between sampling and quantization

Dithering and halftoning are both techniques which try to represent an image using only a small number of intensities. Dithering refers generally to adding noise before quantizing (this is what was going on with the noisy Art Deco image) . Halftoning is more specific in that the goal is to create an output image which contains only two intensities, but more general in that several different algorithms can be used to produce the effect. Here, we'll take a closer look at halftoning. Notice how resolution is used to compensate for a low number of quantization levels.

4.3.1 Halftoning

Laser printers and newspaper presses can only print black dots, yet they can produce images which appear to contain hundreds of gray levels. The secret is a technique called halftoning. If you look closely at a newspaper image like Figure 7 you can see the way different sized black dots contribute to the illusion of many gray levels.

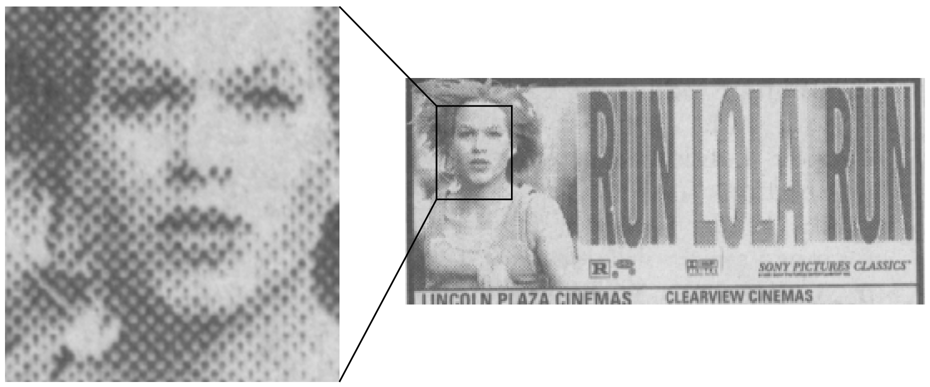


Figure 7: Halftoning in a newspaper.

We'll explore four different halftoning techniques, encapsulated in four **.m**-files. We'll apply each technique to the Art Deco image *im*.

One quick way to halftone an image is simply to quantize it to two levels, which is what `halftone1` does. How does the result look? D24

A better method is to add noise before quantizing (again, this is called dithering), which is what `halftone2` does. How does this new image look? D25

Surprisingly, one trick to making a better halftoned image is to create a small block of uniform random noise (called a *halftone screen*), tile this block over the original image, add the tiled noise to the original, and then quantize. This is encapsulated in the Matlab command `halftone3`:

```
≫ out = halftone3(im,blocksize);
```

The variable *blocksize* is the size of the halftone screen you want to use. Try using `halftone3` with *blocksize* = 4,8, and 16. How do these images look in general? What is the effect of varying the block size? D26

Finally, look at the halftoning result you get with `halftone4`. How does this image look compared to the others?

The very last thing: execute the command `ramphalf`, which will show you a grayscale ramp halftoned using various techniques, which may give you some more intuition about what's going on.

Based on your halftoning experiments, why do you think a 1200 dpi (dots-per-inch) printer is better than a 300 dpi printer? Q6

4.4 Food for thought

In the computer graphics community, there's some interesting research on representing an image in black and white in a way that better preserves image features.

One example is an automatic pen-and-ink method which produces "illustrated" images from photographs. Figure 8 shows some of the resulting images.

Another interesting approach is based on techniques used by traditional engravers. You can see this kind of art on dollar bills, stamps, and Barnes and Noble shopping bags, as well as in Wall Street Journal

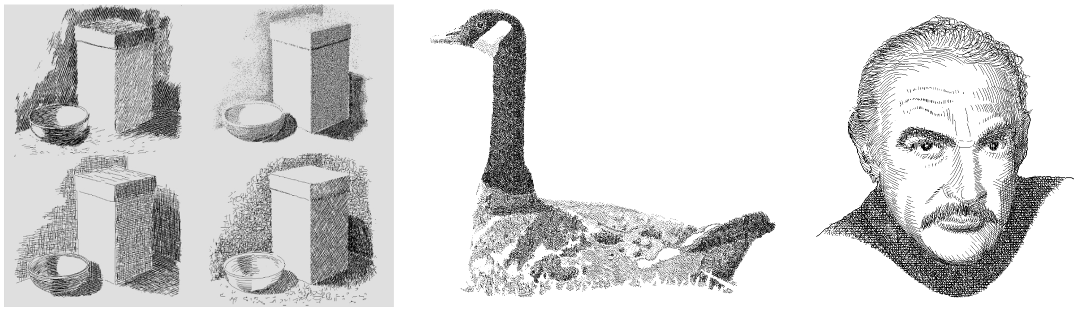
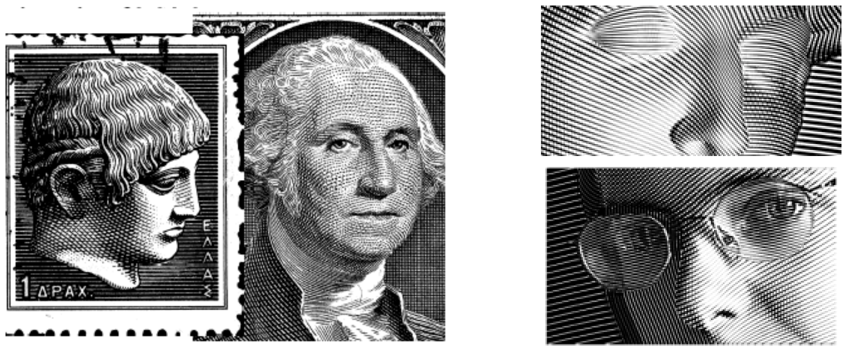

(a) (b) (c)

Figure 8: Pen and ink illustrations created from real images.

illustrations. These "engraved" versions of photographs in Figure 9 were semi-automatically produced using a sophisticated algorithm. Again, the output consists solely of black marks on a white page.



(a) (b)

Figure 9: (a) Traditional engravings. (b) Computer-generated engravings

Finally, you might be familiar with the "photomosaic" technique, in which many smaller images are used as "pixels" to build up a larger image. An example is Figure 10, a picture of a flamingo comprised of many smaller images.

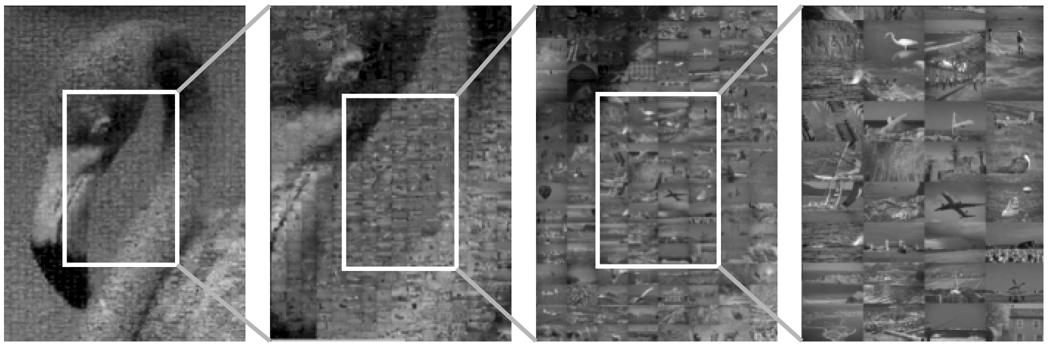


Figure 10: Photomosaic of a flamingo.

Can you relate these techniques to sampling, quantization, dithering, or halftoning? $\boxed{\text{D27}}$