Physics 2660

Lecture 13

Today
- Convolution
- Fourier Analysis
- Cryptography

Part 1: Convolution

Today we'll skim lightly over a few topics we haven't been able to cover yet in the course. Maybe they'll give you ideas for future projects, or at least give you a better understanding of some of the things computers do.

Convolution Integrals:

Given two functions, *f(t)* and *g(t)*, we can define the convolution of those functions as:

$$h(t) = (f * g)(t) \equiv \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

Convolution integrals turn up in many places:

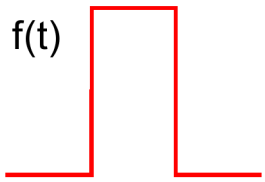
They help us predict the response of an optical system or an electrical circuit to an input signal, they're useful for creating filters for digital image or audio processing, and they help us describe the time-evolution of a system of particles through quantum mechanics.

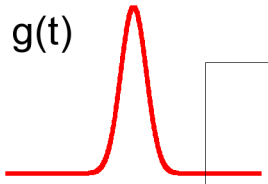
3

Convolution integrals are very common in science and engineering. You'll find them all over the place.

Let's look at a couple of examples that illustrate what it means to convolve two functions together.

An Example:






```
int main () {
  const int NPOINTS = 1000;

  double tmin = -15.0;
  double tmax = 15.0;
  double step = (tmax-tmin)/NPOINTS;

  for (int i=0;i<NPOINTS;i++) {
    double fg = 0.0;
    double t = tmin + step*i;
    for (int j=0;j<NPOINTS;j++) {
      double tau = tmin + step*j;
      fg += square(tau)*gaussian(t-tau);
    }
    printf("%lf %lf\n",t,fg);
  }
}
```

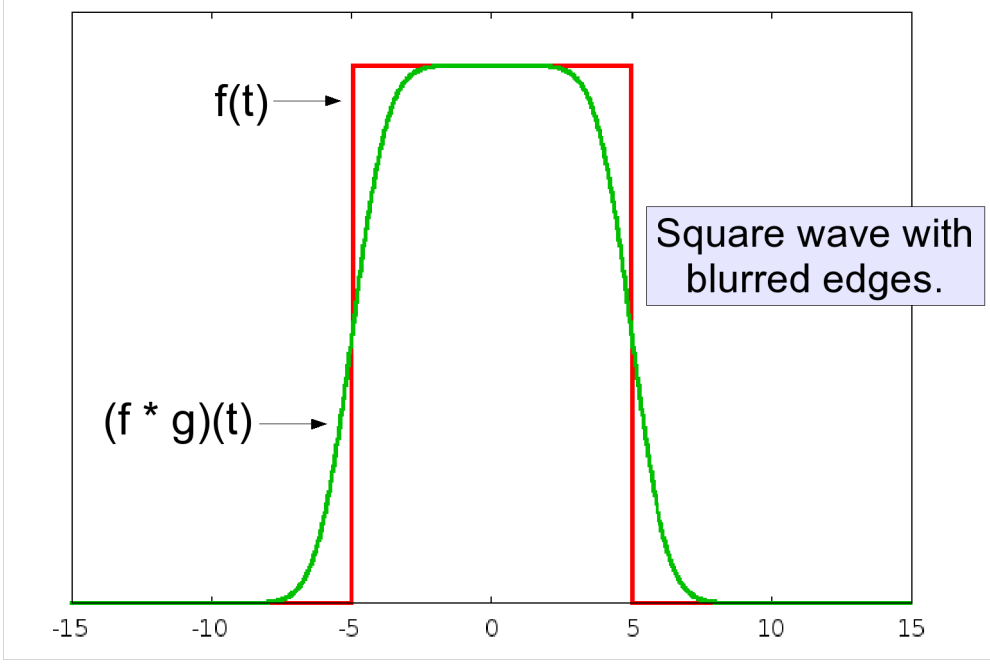
$$\int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

4

Here's a simple program that computes (f*g)(t) for a range of values of t. The function f(t) is a square pulse (its value is zero everywhere except in a small range, where it has a value of one). The function g(t) is a gaussian with a standard deviation of 1.

The inner loop (shown by the bracket) computes the value of the convolution integral for each t value.

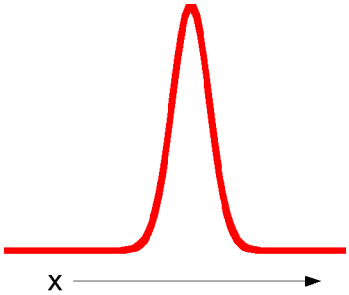
Square Pulse Convolved with Gaussian:



As you can see, the resulting function, (f*g)(t), is like the square pulse, but with rounded edges. It has some of the characteristics of each of the two functions that went into it.

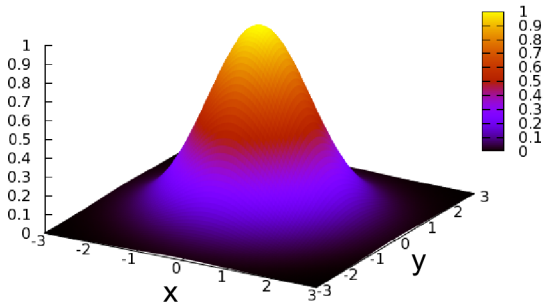
Convolutions in Two Dimensions:

1-Dimensional



$$g(x) = e^{-\frac{x^2}{2\sigma^2}}$$

2-Dimensional



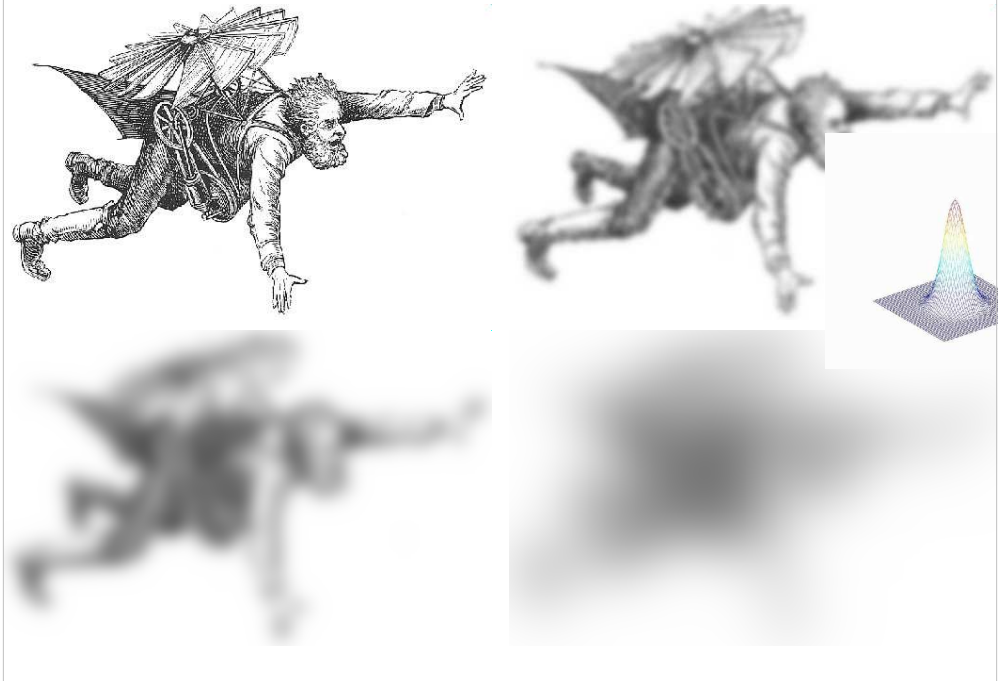
$$g(x,y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

6

Let's extend this example from 1 dimension into 2 dimensions. The figure on the right shows a 2-dimensionl gaussian function.

Let's look at what happens when we convolve some sharp 2-dimensional data with this function.

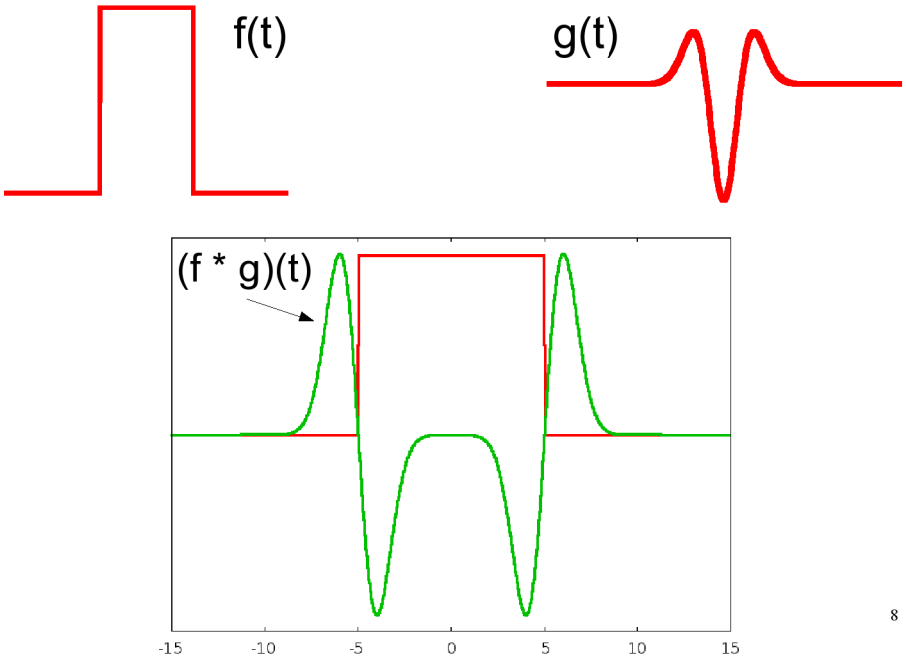
Gaussian Blur:



Starting with the original image at the upper left, we convolve the image with gaussian functions of various widths. This is what image-editing programs call a “gaussian blur”.

As the standard deviation of the gaussian gets bigger, we blur the data across a wider area.

Convolution with the Laplacian of a Gaussian:



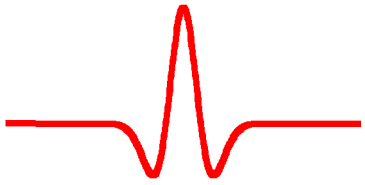
Here's another interesting choice for g(t). This is the 2nd derivative of a gaussian. If we convolve our square pulse with this function, we get the result shown in the bottom figure.

Notice that the (f*g)(t) is zero at locations far away from the square pulse, and it's zero in the middle, where the pulse is flat. The (f*g)(t) function only has big non-zero values in the regions where f(t) is changing rapidly.

2-Dimensional Case:

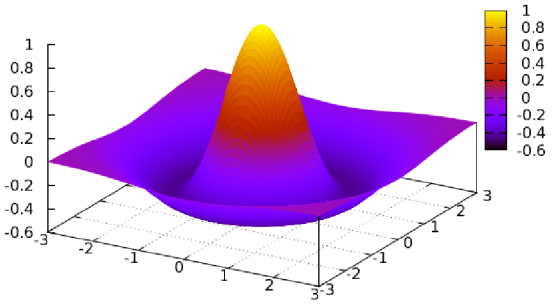
This function in 2 dimensions is often called the “mexican hat” function, because it resembles a sombrero. I've inverted it in the pictures below, to show this.

1-Dimensional



$$\frac{x^2 - \sigma^2}{\sigma^4} e^{-\frac{x^2}{2\sigma^2}}$$

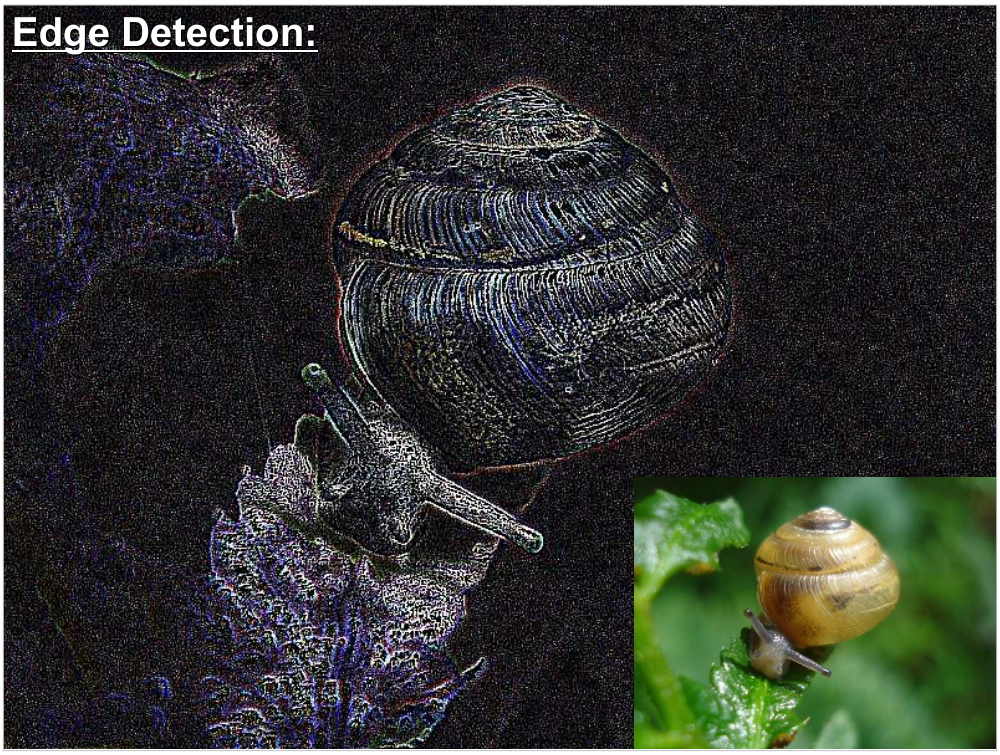
2-Dimensional



$$\frac{x^2 + y^2 - \sigma^2}{\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

9

Again, here's the 2-dimensional version of the same function. Let's see what happens if we convolve some image data with this function.



As we saw with the square pulse, the resulting convolution has a value near zero wherever the image isn't changing very rapidly, but has large values wherever the image makes an abrupt change.

This allows us to detect the edges of things in the image. This sort of thing is interesting artistically, but it's also useful in science and engineering. Imagine, for example, that you were looking at a microscope slide showing some cells. An edge-detection process like this could be used to find the outlines of the cells in the image. This would be useful information for a program that counted the cells or calculated their area.

Part 2: Fourier Analysis

This is an enormous topic, so we're only going to skip very lightly across its waves.

Fourier analysis is something else that turns up all over science and engineering. One application of it is in decomposing signals into their component frequencies. That's what we'll look at here.

Series Expansion of a Function:

We know that many functions can be approximated by summing a series of terms. For example, we often use the Taylor series:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

$$= f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots$$

As we add terms, the sum gets closer and closer to the true value of *f(x)*.

The Fourier Series:

As it turns out, if *f(t)* is any piecewise-continuous function defined in the interval -T to T, we can write *f(t)* as a sum of cosines and sines. The series of terms comprising this sum is called the Fourier series:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n cos\frac{n\pi t}{T} + b_n sin\frac{n\pi t}{T} \right)$$

where:

$$a_n = \frac{1}{T}\int_{-T}^{T} f(t) cos\frac{n\pi t}{T} dt, \; n = 0, 1, 2, ...$$

$$b_n = \frac{1}{T}\int_{-T}^{T} f(t) sin\frac{n\pi t}{T} dt, \; n = 1, 2, ...$$

13

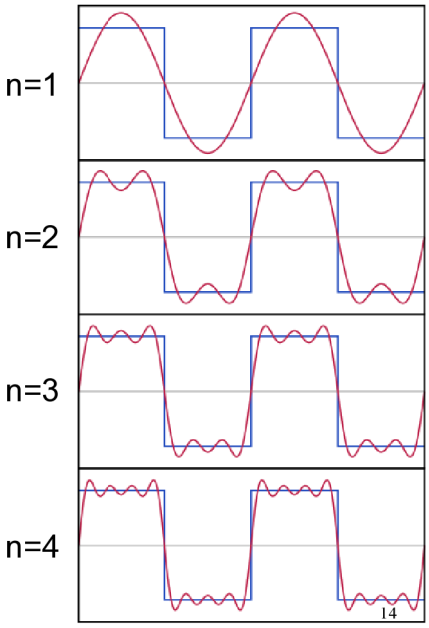
This series breaks the function up into components of various frequencies. The coefficients a_n and b_n are the relative intensities of the different frequencies.

Fourier Series Terms:

By adding up the terms in the Fourier series, we can get successively better approximations to the original function.

In fact, we could say that all the terms in the series, taken together, are just another way of representing the original function.

Instead of, say, a function of time like f(t), we could look at the a_n and b_n terms of the corresponding Fourier series as the values of some other function F(frequency) that gives us the same information, but in terms of frequency instead of time.



The Fourier Transform:

To make this time/frequency correspondence even clearer, we can define the "Fourier Transform", *H(f)*, of a function *h(t)*:

$$H(f) = \int_{-\infty}^{\infty} h(t)[\cos(2\pi ft) + i\sin(2\pi ft)]\, dt$$

where *H(f)* is, in general, a complex number. (The original function *h(t)* may be complex, too.)

We can express the original function in terms of its Fourier Transform like this:

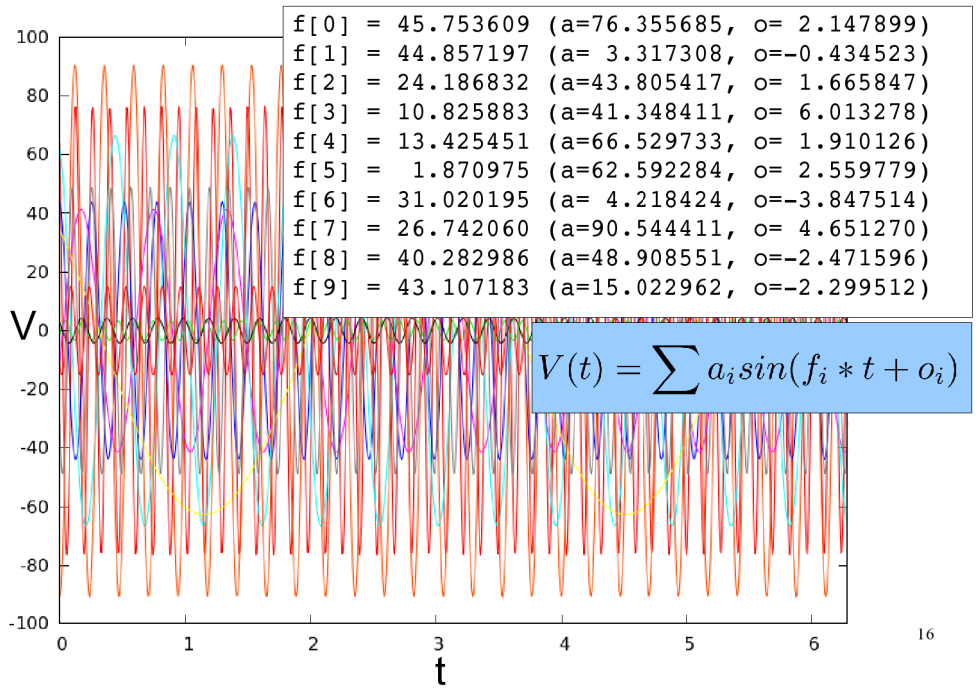
$$h(t) = \int_{-\infty}^{\infty} H(f)[\cos(2\pi ft) - i\sin(2\pi ft)]\, df$$

(Note that this is just one specific type of Fourier transform. We can do similar things with other functions besides sines and cosines.)

h(t) and H(f) are just two different ways of looking at the same thing. We say that h(t) represents the function in the "time domain" and H(f) represents the same function in the "frequency domain".

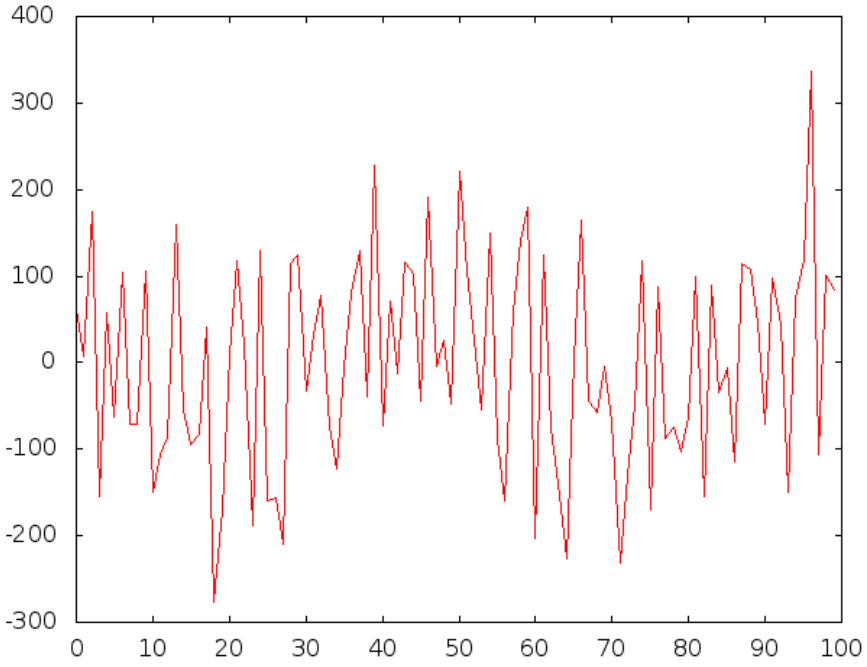
That's all very interesting, but what is it good for? Let's look at a specific example.

A Random Set of Sine Waves:



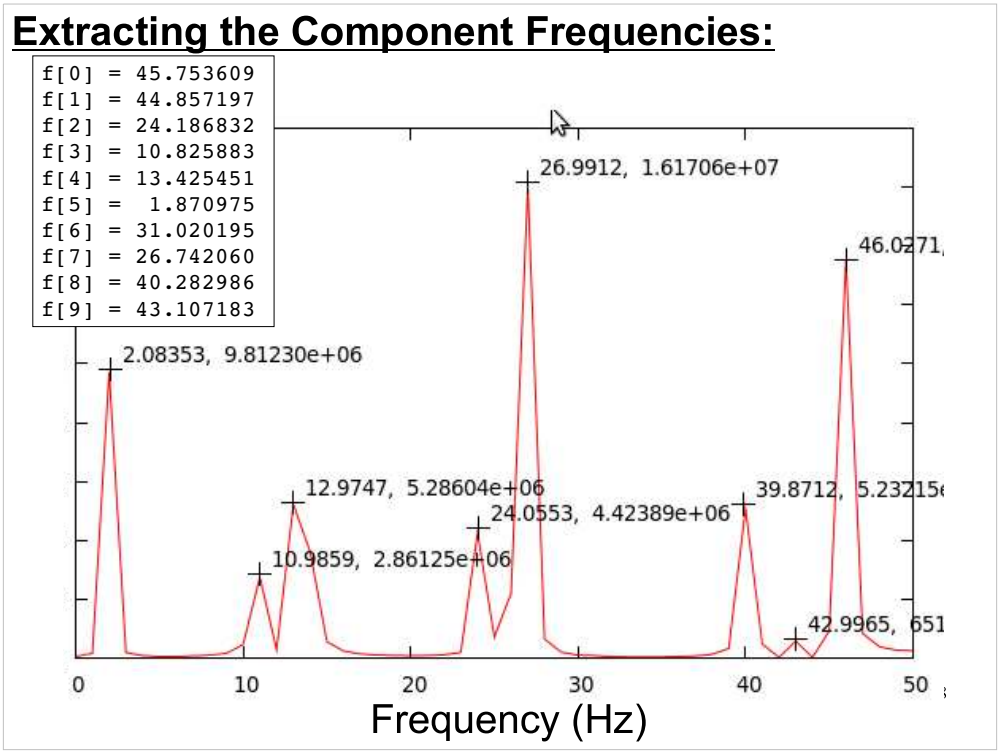
Here, I've generated ten different random sine waves. Each wave has a different frequency, amplitude and phase, given by f, a and o, respectively.

Sum of Random Sine Waves:



In this graph I've just added up the ten sine waves. The result looks like a mess. If we were presented with this data, could we ever hope to find out what the original ten sine waves were?

Extracting the Component Frequencies:



Yes! By taking the Fourier Transform of the data.

The graph above shows the square of the Fourier Transform of the data in the preceding graph. As you can see, the component frequencies are clearly visible as peaks here.

What else can we do with this? Remember that we can also do the inverse transform, to get h(t) from H(f). So, if we decide we want to eliminate a certain range of frequencies from our data we can just set that range of the graph to zero, and then do the inverse transform.

Imagine, for example, that we have some audio data with an annoying 60 Hz hum in it. We could take the Fourier Transform of the data, cut out the peak at 60 Hz, and transform back to a new, clean version of our data.

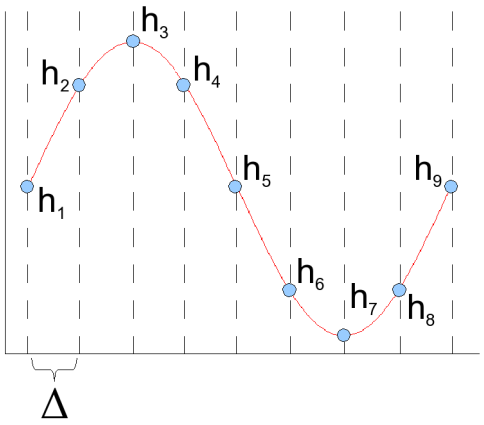
The Discrete Fourier Transform:

When doing experiments, we often don't have a continuous function to transform. Instead, we have a bunch of discrete points.

Say, for example, that we have a group of N evenly-spaced samples, h_0 through h_N, separated by time interval Δ.

We can define the "Discrete Fourier Transform" (DFT) H_n as shown below.

H_n is defined only for a set of discrete frequencies, f_0 through f_N.

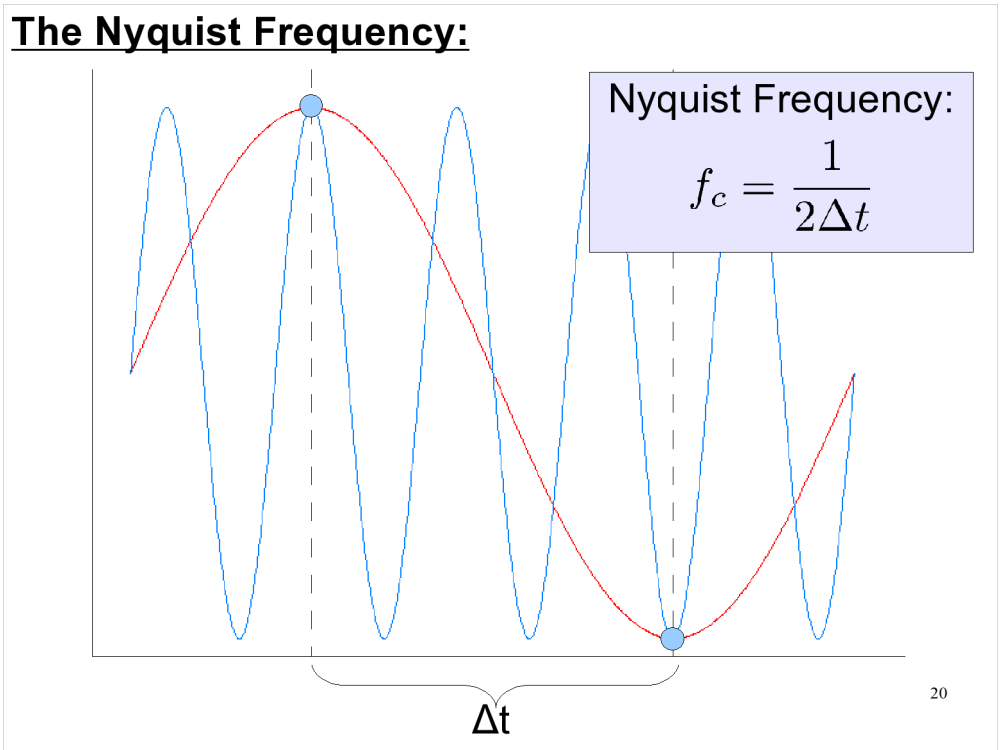


$$H_n = \sum_{k=0}^{N-1} h_k[\cos(2\pi kn/N) + i\sin(2\pi kn/N)]$$

$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n[\cos(2\pi kn/N) - i\sin(2\pi kn/N)]$$

19

This is like the audio data we experimented with in lab. In that case, we had an audio signal that was sampled at 44,100 times per second. (That is, Δ = 1/44,100th of a second.)



The Nyquist critical frequency (f_c) is the highest frequency that our sampling system can measure. It corresponds to sampling the red wave at its peak and it trough. Higher frequencies (say, those that squiggle a million times during Δt) are mysteries to us.

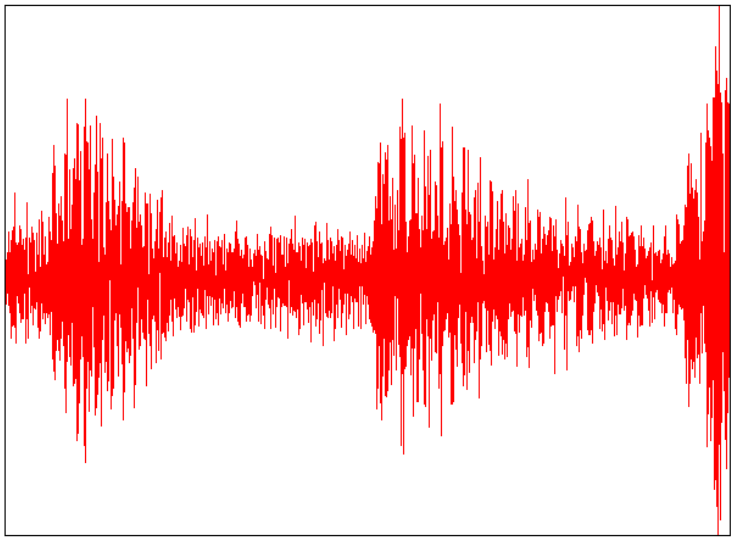
If the data we're sampling happens to be limited to frequencies below the Nyquist frequency, then the DFT is an exact representation of the original signal. (That is, the signal could be exactly reconstructed, given the DFT.)

If any frequencies are above the Nyquist frequency, sampling causes those frequencies get “aliased” to lower frequencies below the Nyquist frequency.

Consider, for example, the blue curve, above. It has peaks and troughs in the same samples as the red curve. There's no way we can distinguish between these two curves using a Δt of this size.

A Real Signal:

What if we try to apply this to some real data? The figure below shows the sound intensity values during a section of Tiny Tim's “Tiptoe Through the Tulips”:



Let's look at a real-world example (assuming that Tiny Tim belonged to the real world).

The FFTW Library:

Instead of writing our own code for doing DFT's we can use the excellent and widely-available FFTW (for the “Fastest Fourier Transform in the West”) library.

To use FFTW in your programs, you'll need to

- include the `<fftw3.h>` header file, and
- link your programs with “`-lfftw3`”.

http://www.fftw.org/

We could write a program to calculate the Fourier Transform of this audio data, but fortunately, somebody else has already done the hard work for us.

Writing an FFTW Program:

```
# include <fftw3.h>

...
 double *in;
 fftw_complex *out;
 fftw_plan plan_forward;

...
// Allocate space for some real (no imaginary parts) input data:
 in = (double *)fftw_malloc (sizeof ( double ) * nwords);
...
// Read in data...

// With N real input data points, the number of Fourier coefficients
// will be N/2 + 1.
// Allocate space for output data:
 nc = ( nwords / 2 ) + 1;
 out = (fftw_complex *)fftw_malloc (sizeof ( fftw_complex ) * nc);
...
 plan_forward = fftw_plan_dft_r2c_1d (nwords, in, out, FFTW_ESTIMATE);
 fftw_execute ( plan_forward );   // Execute the plan.
```

The fftw3.h header file defines several new data types, such as fftw_complex and fftw_plan.

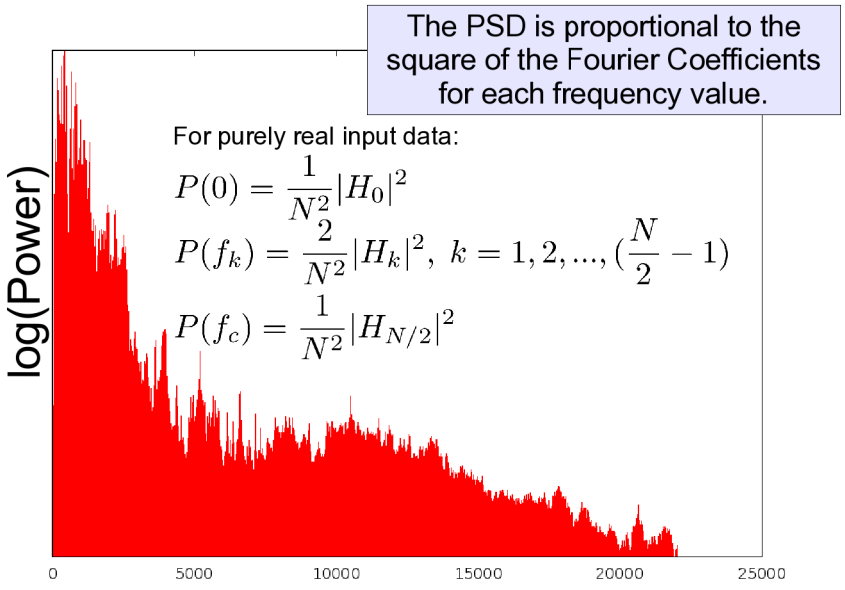
The fftw library includes its own special version of malloc.

Before doing the transform, you first create a "plan". In this step, FFTW looks at the data and decides how best to transform it.

FFTW is moderately complicated to use, but it's well documented at the project's web site. The function "fftw_plan_dft_r2c_1d" can be read as "Discrete Fourier Transform, Real input to Complex output, 1-dimensional data".

The format of the output returned by functions like "fftw_plan_dft_r2c_1d" will differ from one function to another, but it will always be an array of values.

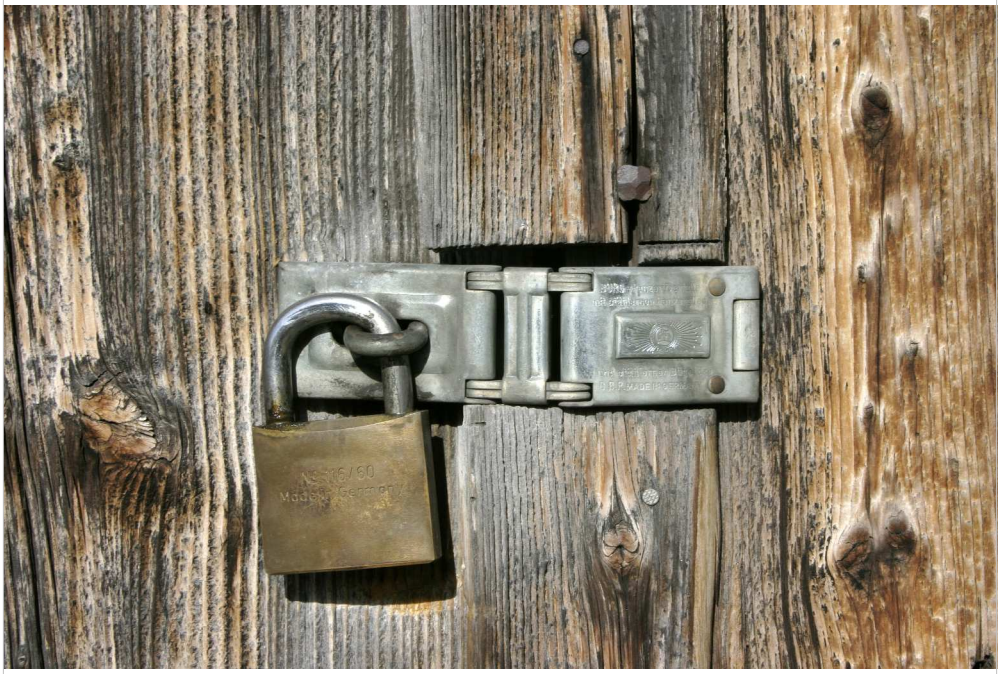
Power Spectral Density (PSD):



Notice that the spectrum trails off at values above about 22,000 Hz. This is because our sample rate (44,100 samples per second) gives us a Nyquist frequency of 22,050 Hz.

This partially explains why the CD audio standard uses 44,100 samples per second. The range of human hearing is about 20 Hz to 20,000 Hz, so the people setting the standard wanted the Nyquist frequency to be above 20,000 Hz.

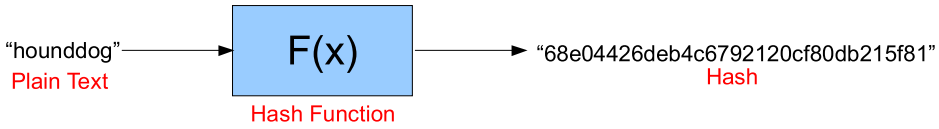
Part 3: Some Encryption Principles



The following is useful for understanding how passwords are stored, and how things like ssh work, but it applies to a lot of other things, too: secure web connections, personal certificates, and e-mail encryption are some examples.

Cryptographic Hashes:

(also called “message digests”)



A cryptographic hash function, F(x), has the following properties:

• It's deterministic. For any particular input, F(x) will always produce the same output (called a “hash” or a “message digest”).

• The output of F(x) is always the same length.

• The output is almost always unique. Different inputs are very unlikely to produce the same output, even if they differ only slightly.

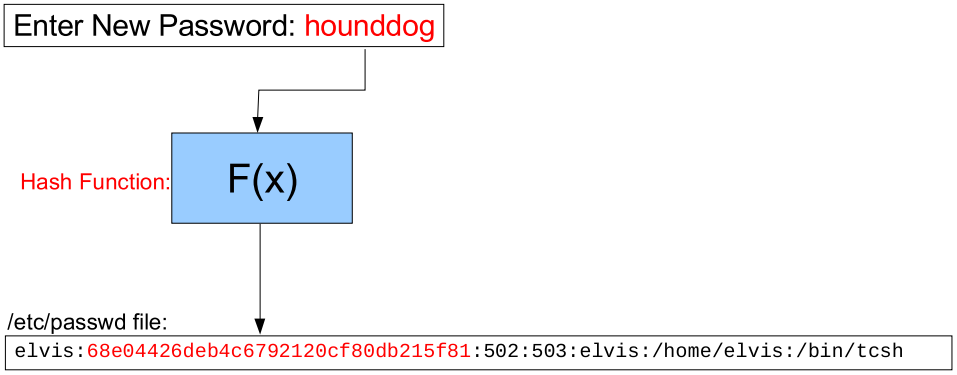
• F(x) is relatively easy to compute.

• The inverse function is extremely difficult to compute. It's very hard (ideally, impossible) to determine what plain text produced a given hash. This is called “trapdoor encryption”.

So why is this relevant to what we're talking about?
We'll see in a minute.

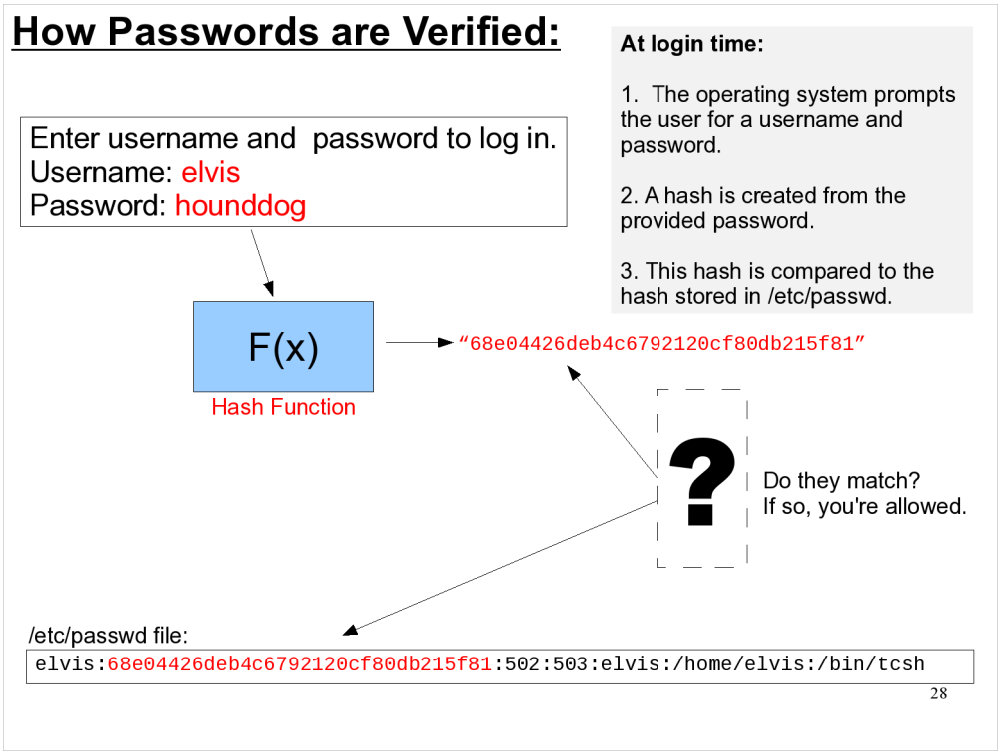
How Passwords are Stored:

When a new account is created, or when a password is changed, here's what happens:



The plain text password is never stored. Only the cryptographic hash is stored. Because hash functions are almost impossible to invert, nobody but “elvis” can know what his password is. So how can the system verify that a user knows his or her own password?

I'm simplifying things a little here. We'll explore some of the complications soon.



This means that the operating system doesn't need to know the password in order the verify that the user knows the password. Clever!

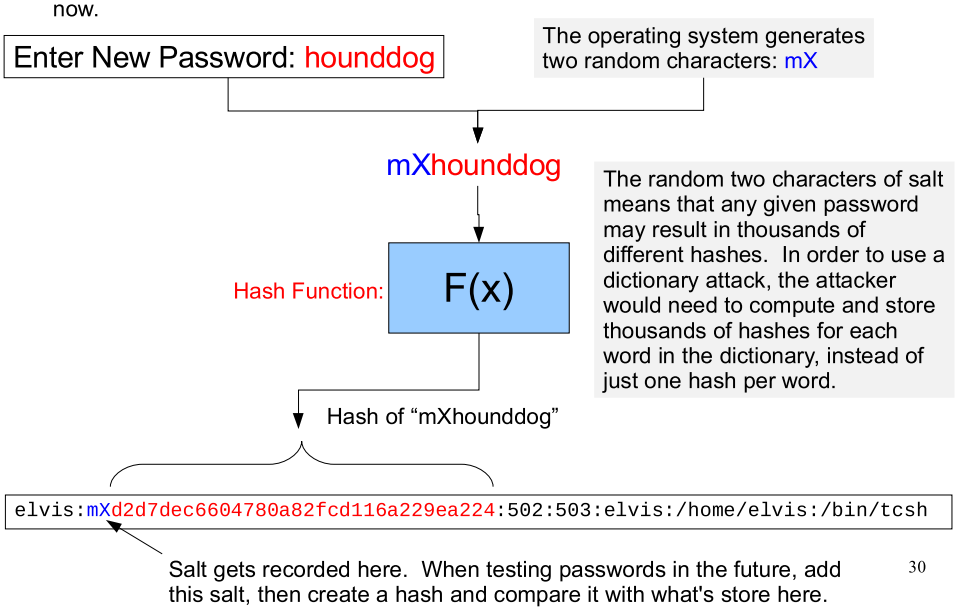
Dictionary Attacks:

Word	Hash
aardvark	88571e5d5e13a4a60f82cea7802f6255
abnegation	f8ec9b74a9b9aa1131abbfc0b8dca989
acrimony	a246c59fdccea42bc48202156a5f72de
adumbration	e6b65039a16dc9ffd4cca73e7f1b973f
advil	a3d4b48aebd5c6b9aaf57583601f1857
aerie	70f1f8799ad6af309af5434cb065bd59
affinity	1474047fb00b2d8d95646f7436837ed0
agnatha	c6bf04438cd39591695454ea4c755acb
aherne	e177eedf9e5b91c39d0ec9940c9870b9
aida	2991a6ba1f1420168809c49ed39dba8b
ajax	2705a83a5a0659cce34583972637eda5
AKKA	004ab7976e8b4799a9c56589838d97a6
algae	4360ef4885ef72b644fb783634a7f958
anthocyanin	7c425ea7f2e8113f6ca1e6e5c3a554a9
antidisestablishmentarianism	2a3ec66488847e798c29e6b500a1bcc6
anxiety	d3af37c0435a233662c1e99dbff0664d
apple	1f3870be274f6c49b3e31a0c6728957f
aurora	99c8ef576f385bc322564d5694df6fc2

Even though it's computationally expensive to compute the inverse of F(x), it's easy to generate hashes for thousands of words and store the results in a file. Once we've created this dictionary of hashes, we can just look up a given hash on the right, and match it with the plain text on the left. Since users are inclined to use common words as passwords, this is a security problem.

Adding Salt:

One way to reduce the effectiveness of dictionary attacks is to add some random "salt" to the plain text before creating the hash. This is always done now.



Password Hash Algorithms:

DES:
Based on National Bureau of Standards' Data Encryption Standard (DES).

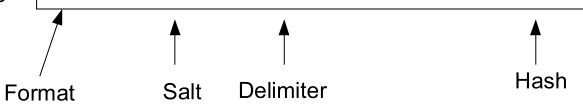
"hounddog" = mX1EcouR8fX7w

MD5:
RSA's "Message Digest 5" algorithm.

"hounddog" = 1SG.2TL0i$vQuZcYnvA7kgMwg3gEB2RA

SHA-256:
NSA "Secure Hash Algorithm".

"hounddog" = 5SG.2TL0i$NScvuF0OzU/lkdIModlq8Sn59mU

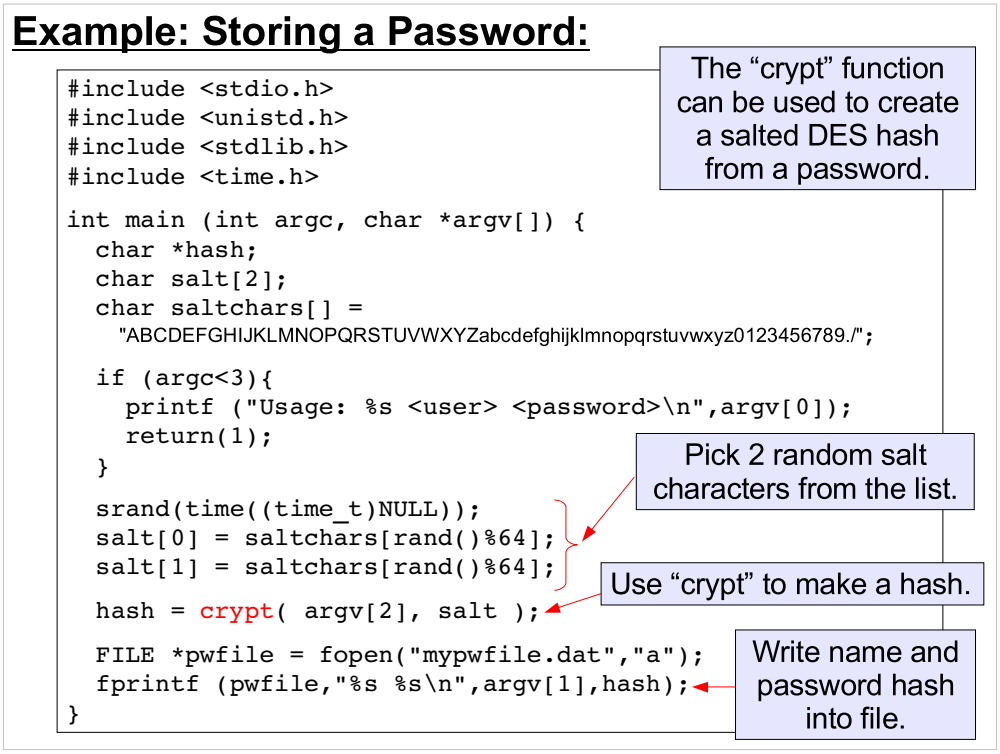


31

These are the formats you'll really see in /etc/password or /etc/shadow.

The SHA-256 and SHA-512 hash options are being developed now, but you should see them soon in Linux distributions. The person behind this effort is Ulrich Drepper. Here's his documentation about the SHA password hash standards he's developing:
http://people.redhat.com/drepper/SHA-crypt.txt

Example: Storing a Password:



This example program allows the user to enter a username and a password on the command line, and then it stores the username and a hash of the password in a file.

Example: Checking a Password:

```
int main (int argc, char *argv[]) {
  char *hash;
  char user[80],pw[80];

  if (argc<3){
    printf ("Usage: %s <user> <password>\n",argv[0]);
    return(1);
  }

  FILE *pwfile = fopen("mypwfile.dat","r");
  while (!feof(pwfile)) {
    fscanf(pwfile,"%s %s",user,pw);
    if ( !strcmp(argv[1],user) ) {
      hash = crypt( argv[2], pw );
      if ( !strcmp(hash,pw) ) {
        printf ("Password accepted!\n");
        return(0);
      } else {
        printf ("Wrong password!\n");
        return(1);
      }
    }
  }
}
```

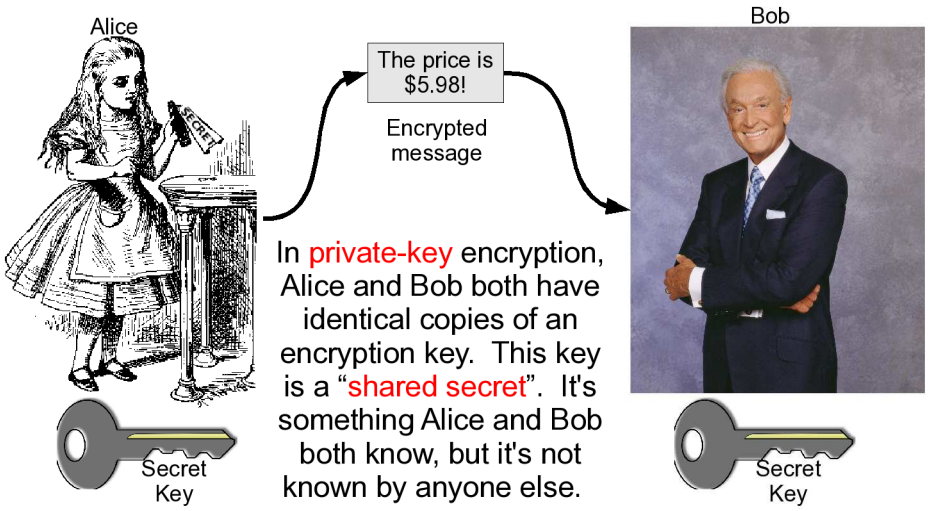
Look for this user in the password file.

Hash the supplied password, using the salt from the 1st 2 chars of the stored hash.

Does this hash match the stored hash?

This program takes a user name and a password on the command line, and then checks the supplied password to see if it is correct.

Private-Key (Symmetric) Encryption:



When Alice wants to send a message to Bob, she encrypts it with this key. After Bob receives the encrypted message, he can decrypt it with the same key.

Since the same key is used for encryption and decryption, this is called "symmetric" encryption.

34

Symmetric encryption has been used since ancient times. It's something every schoolchild comes up with on his or her own, making up secret codes to share messages with friends.

As we'll see, it has some problems.

(Alice and Bob are common in cryptography examples. There's a T-shirt for crypto geeks that says "I know Alice and Bob's shared secret.")

Some Problems with Private-Key Encryption:

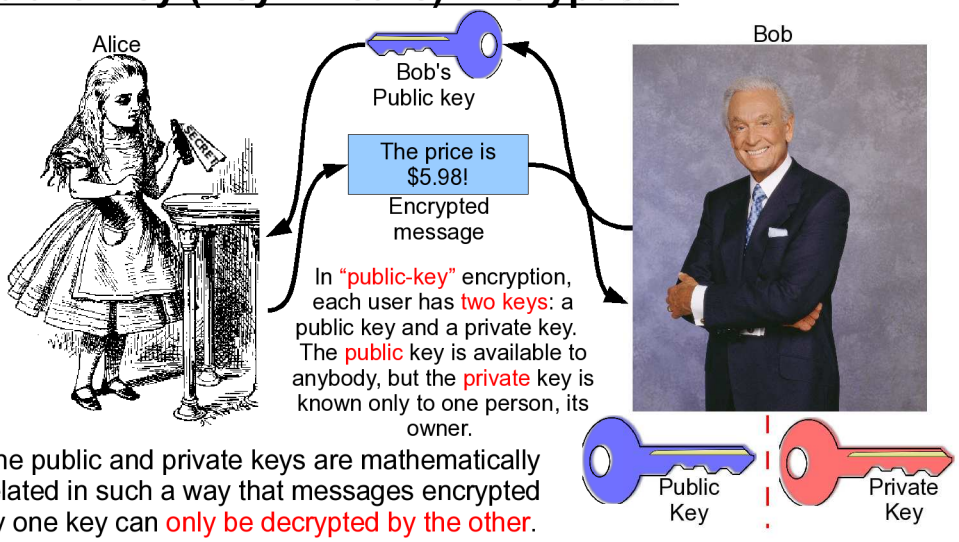
• **Key distribution is hard, and possibly insecure.**
What if Alice is in Portugal and Bob is in Hong Kong? How do they initially get the secret key to each other? What if they need to change it later?

• **Anyone who steals the key can masquerade as Alice or Bob.**
This system has no way of verifying the identity of the sender. If a third party obtains the key, he can send messages that appear to come from Alice or Bob.

• ***"A secret shared by two is compromised. A secret shared by three is no secret."***
Can Alice really trust Bob to keep the key a secret? Can Bob trust Alice? And what if we have a whole organization full of people who all need to know the secret key?

Even if we ignore most of these problems, we still have to face up to the problem of key distribution. This started out with "How do we get keys to our secret agents scattered around the world?". By the 1990s it had changed to "How do we get keys to all of the people on the internet who want to buy things from our e-commerce site?"

Public-Key (Asymmetric) Encryption:



The public and private keys are mathematically related in such a way that messages encrypted by one key can only be decrypted by the other.

When Alice wants to send an encrypted message to Bob, she first obtains his public key. Then she uses this key to encrypt the message. Only Bob, the holder of the matching private key, can read the resulting message.

Because encryption and decryption are done with different keys, this is called "asymmetric" encryption.

In 1974, Malcolm Williamson, working for British Intelligence, came up with a solution to the key distribution problem. It remained secret for a couple of years, though, until it was rediscovered and published by two researchers, Whitfield Diffie and Martin Hellman. In their seminal 1976 paper "New Directions in Cryptography" Diffie and Hellman described public-key cryptography and set the cryptographic world on its ear.

Now, each individual could have a unique pair of keys, generated locally. There was no longer any need to distribute keys.

Advantages of Public-Key Encryption:

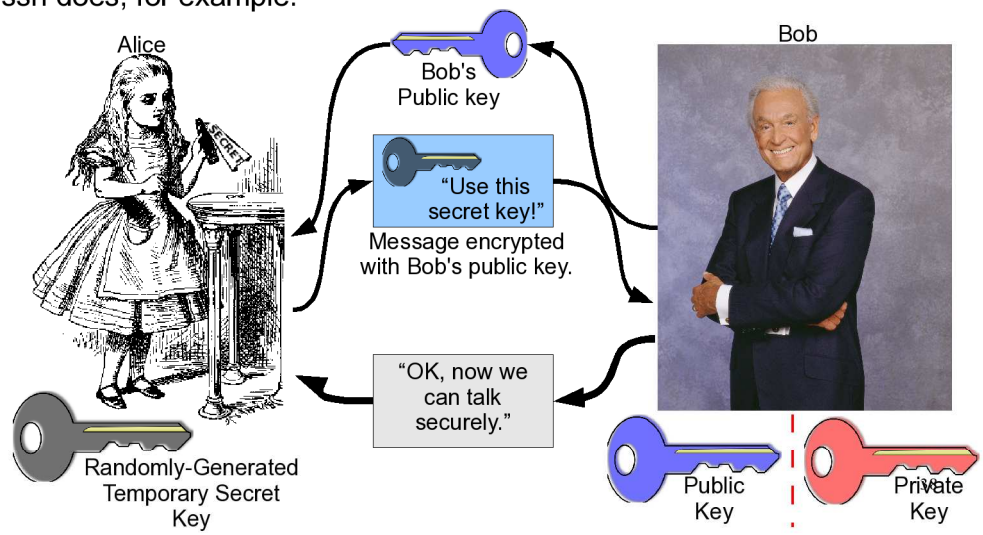
- **No need to distribute secret shared keys.**
  Each person has his or her own public/private key pair, generated locally. Private keys always stay in the hands of their owner, and never need to be transmitted.

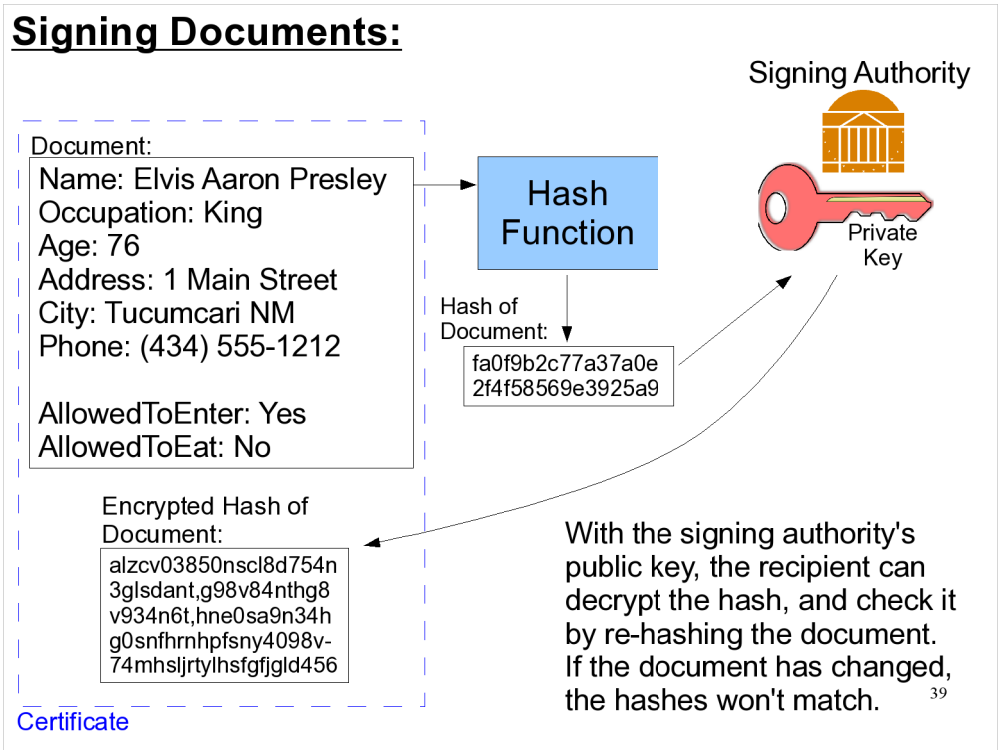
- **The sender's identity can be verified.**
  Since each sender has a unique public/private key pair, we can verify his/her identity.

- **Security risks are limited to a single user.**
  Alice is responsible for the security of her keys, and Bob is responsible for the security of his keys. The theft of one person's key doesn't compromise the security of the other keys.

Key Distribution by Public-key Encryption:

Public-key cryptography is more computationally expensive than symmetric-key cryptography. Usually, public-key encryption is only used to exchange a temporary secret key, which is then used to conduct the remainder of the conversation via symmetric cryptography. This is what ssh does, for example.





This is roughly the way digital certificates are created.

Encryption and Legal Problems:

Ssh adoption was delayed for many years by legal problems in the United States. These problems fell into two categories:

- **Patent Restrictions:**
  The earliest versions of ssh relied on the patented RSA public-key encryption algorithm. The patent-owner provided a free RSA "reference version" (RSAref), but only allowed it to be used for non-commercial applications. OpenSSH worked around this by substituting the freely-available DSA algorithm. The RSA patent expired in 2000, so this is no longer an issue.
- **Export Restrictions:**
  U.S. export restrictions on cryptography were very strict at the time ssh was first written. They've relaxed considerably since then, and primarily apply to a list of seven "terrorist countries". The law is still very confusing, though. Fortunately, OpenSSH is based in Canada, and the U.S. has no import restrictions on cryptography. To avoid possible legal problems, the OpenSSH project does not accept help from software developers in the U.S.

Additionally, In some other countries (e.g., France, Russia, and Pakistan) it may be illegal to use encryption at all.

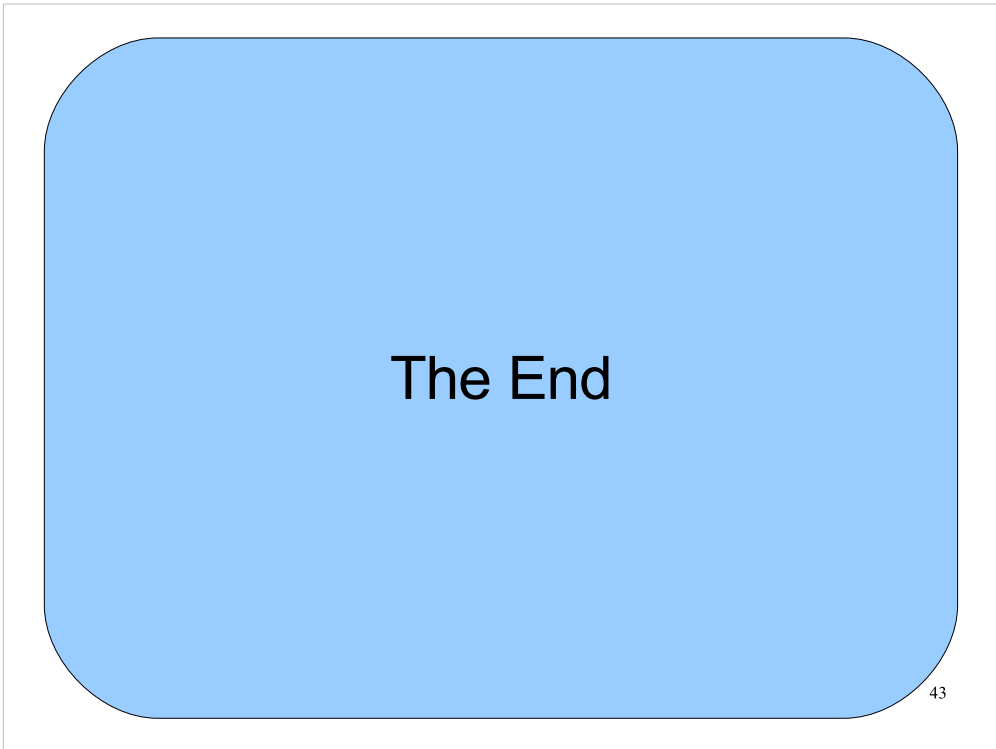
From U.S. export law's point of view, the seven "terrorist countries" are Cuba, Iran, Iraq, Libya, North Korea, Sudan and Syria.

As an example of the dizzying confusion of U.S. law, a 1997 ruling permitted financial-specific cryptographic applications to be exported only if they could, by design, only be used to encrypt financial data. For much more information on U.S. cryptography export law, see:
http://rechten.uvt.nl/koops/cryptolaw/cls2.htm#us

Conclusion:

I hope you feel that you have an enormous repertoire of computing skills now.

Thanks for all of your hard work this semester!



Thanks!