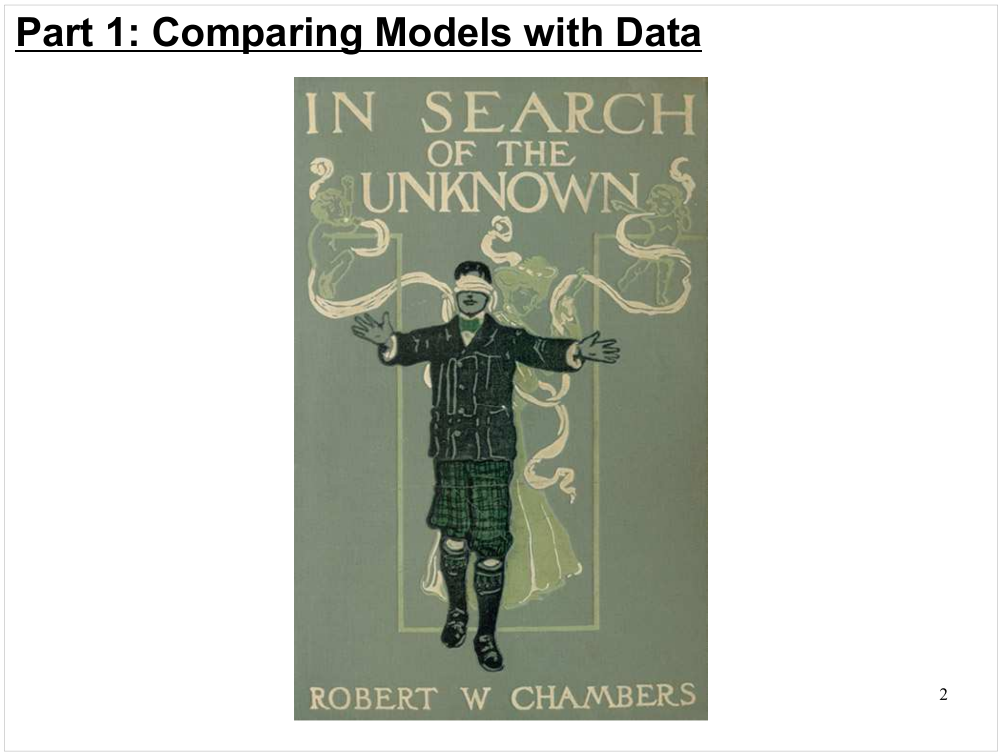
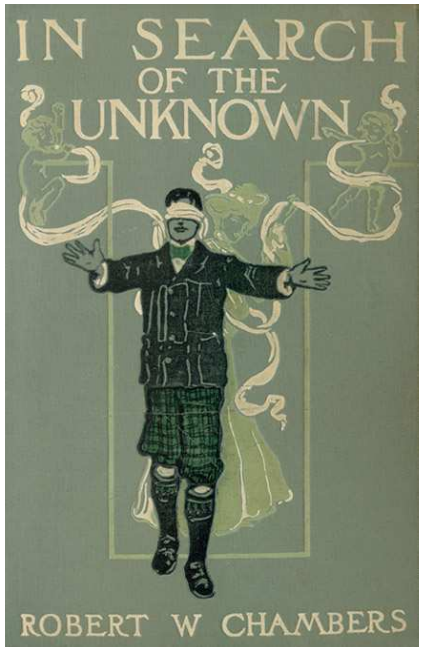
# Physics 2660

## Lecture 10

**Today**
- **Comments on chi-squared fits**
- **Bitwise operators and binary files**
- **Generating arbitrary distributions of pseudo-random numbers for functions that can be inverted and binned data**



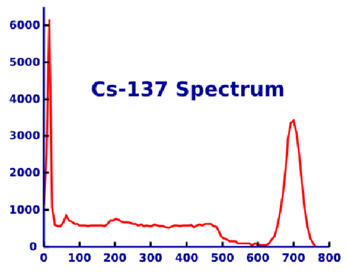
## Part 1: Comparing Models with Data

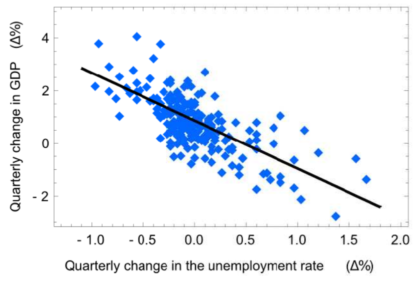


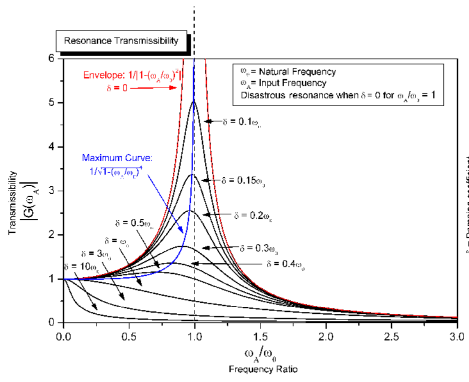
As scientists, we're also “in search of the unknown”. We're looking for the physical laws of the universe. We can't see what's far out in front of us, though, so we have to slowly feel our way along, one experiment at a time.

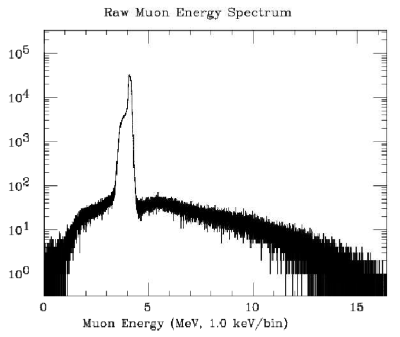
The tools that tell us if there's something solid in front of us, or just air, are statistical measures like chi-squared. Today we'll start out by looking at other tools, and some of the limitations of chi-squared.

## The Importance of Fitting:









3

First, a note about the importance of fitting models to data.

Fitting is widely used in every field of science and engineering, including the physical sciences, biology and the social sciences. It's one of the most important computational techniques for you to learn. As soon as you start collecting data of any kind, you'll want to fit some theoretical model to it, and you'll want to know how well that model fits.

That's why we're spending a substantial amount of time talking about it.

## The Art of Curve Fitting:

There are many things that may make it difficult or impossible to get a model to fit your data well.  Some of them are:

• Using an incorrect model to represent data.

• Making a poor choice of starting parameters.
  Perhaps they are too far from correct values? Also, some programs have trouble with starting parameters at 0.0.

• Sometimes parameters land on unphysical values during the $\chi^2$ minimization process:  1/0,  log(-1), 10^300, sqrt(negative #), ...

• Sometimes the fitting program has difficulty settling into stable values for the parameters (convergence):

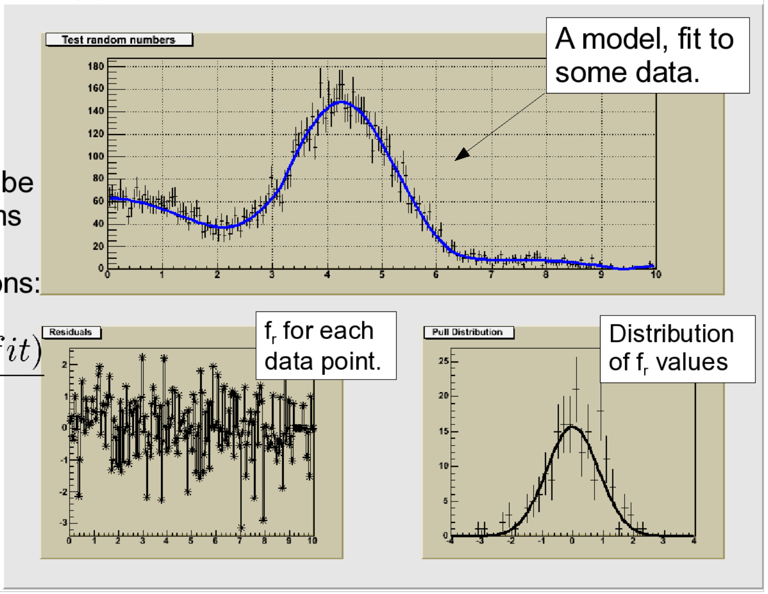
- Maybe you're fitting too many parameters at once, while far from the minimum $\chi^2$.
- Maybe you've chosen a poor set of model parameters: high correlations, large differences in scale among parameters (leading to rounding errors), ...

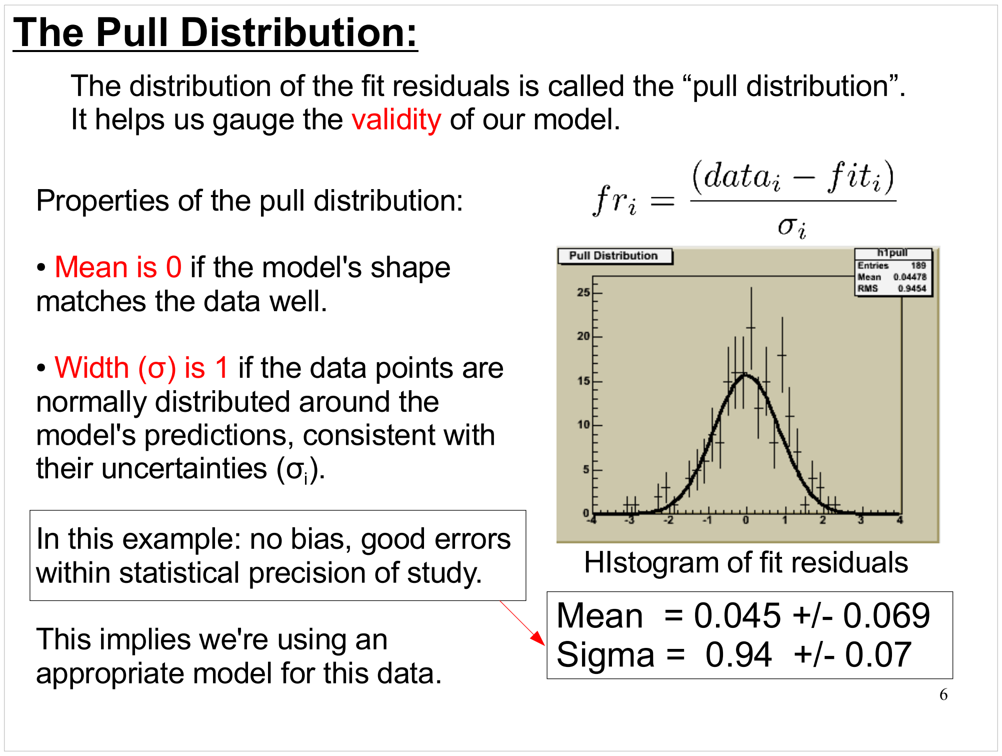
## Deviations from the Model:

One way of gauging the quality of your fit is by looking at the "fit residuals". These are the deviations of your data points from the values predicted by your model.

Fit residuals can be measured in terms of number of standard deviations:

$$f_r = \frac{(data - fit)}{\sigma}$$





## The Pull Distribution:

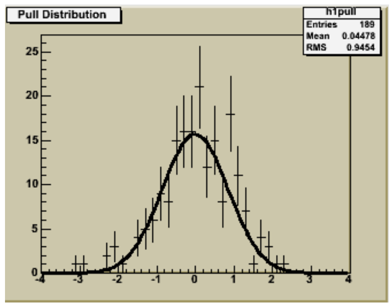
The distribution of the fit residuals is called the “pull distribution”. It helps us gauge the validity of our model.

Properties of the pull distribution:

$$fr_i = \frac{(data_i - fit_i)}{\sigma_i}$$

• Mean is 0 if the model's shape matches the data well.

• Width ($\sigma$) is 1 if the data points are normally distributed around the model's predictions, consistent with their uncertainties ($\sigma_i$).



HIstogram of fit residuals

In this example: no bias, good errors within statistical precision of study.

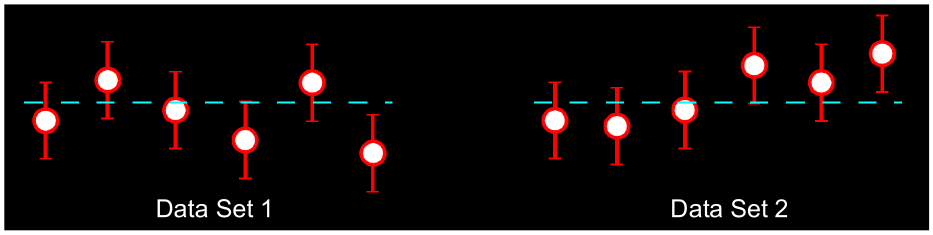
This implies we're using an appropriate model for this data.

Mean = 0.045 +/- 0.069
Sigma = 0.94 +/- 0.07

The term “pull distribution” is used in Physics, but I'm not sure how widely used this term is in other fields. Some people just call it the “histogram of the fit residuals”.

## Looking for Bias:

Consider the following two data distributions:
They've both resulted in the same fit, with the same $\chi^2$ :

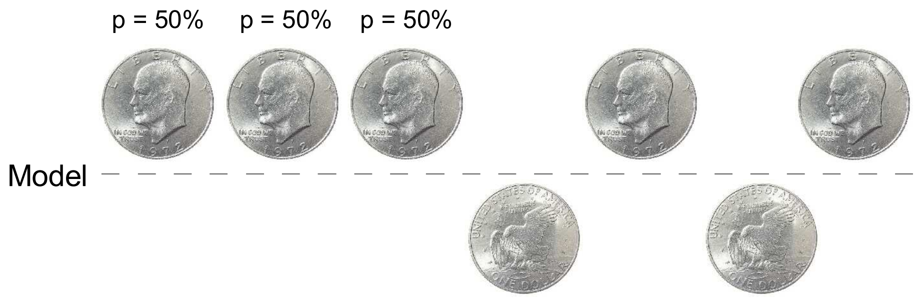


If large groups of points cluster above or below the best fit, this may indicate a problem with your choice of model.

The $\chi^2$ statistic just adds up the squares of the deviations. It won't notice clusters of points like this.

What is the probability of n adjacent points fluctuating above or below the nominal value at random?

## Probability of Clusters Above/Below the Mean:



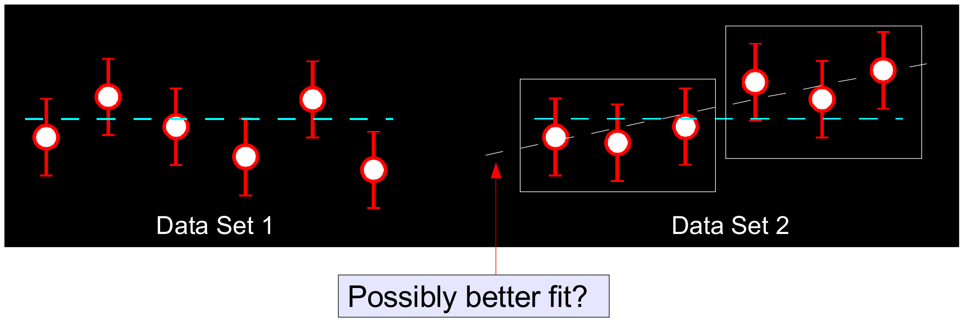
If the model is well-matched to the data, the probability of getting three heads in a row (or, equivalenty, of three consecutive data points above the predicted values) is:

$$P(3) = 0.5 * 0.5 * 0.5 = 0.125$$

The probability of n points in a row above or below the line is:

$$P(n) = 2^{-n}$$

## Clusters of Points:

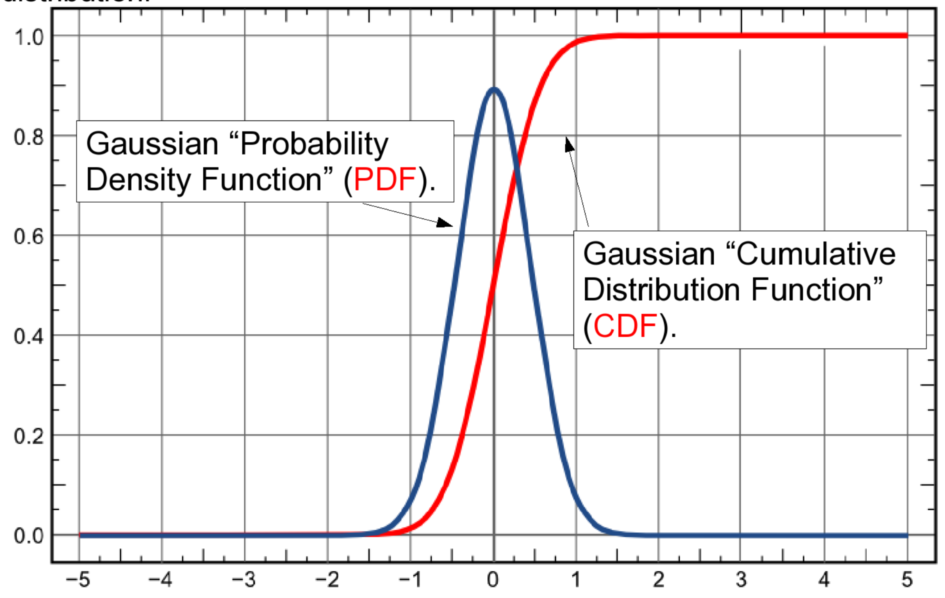


In data set 2, a bias of low results on one end and high results on the other end may indicate that we should use a line with a slope.

A good indication for this is if we add a slope and see a significant reduction in $\chi^2$ .

## Cumulative Distribution Functions (CDFs):

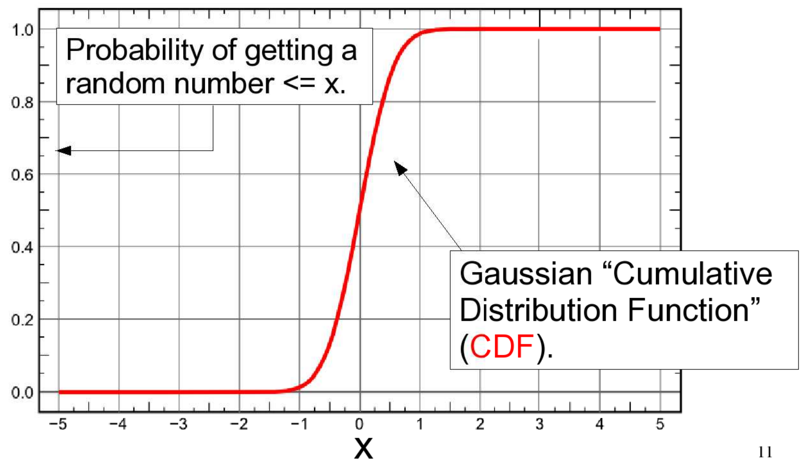
For every probability distribution, there's an associated “Cumulative Distribution Function”. This is the integral from -∞ to x of the probability distribution.



Sometimes we assume that our data has been generated according to a particular probability distribution (a Gaussian distribution, for example). We can check whether this is likely to be true by looking at the distribution's Cumulative Distribution Function.

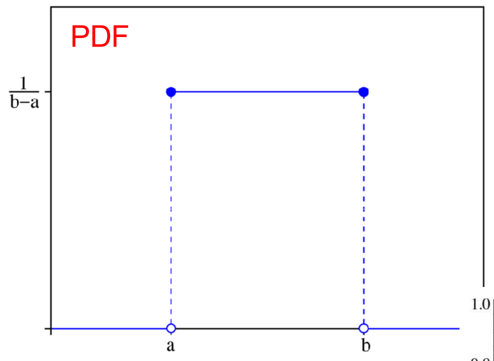
## Properties of the CDF:

At any point x, the CDF tells us the probability of observing a random number less than or equal to x. As x approaches infinity, the value of the CDF approaches 1. (In other words, it's certain that we'll get a random number less than infinity!)



This particular example shows the CDF of a Gaussian distribution centered at zero. Notice that the value of the CDF is 0.5 when x=0. This just tells us that there's a 50% chance of observing an x value less than zero.

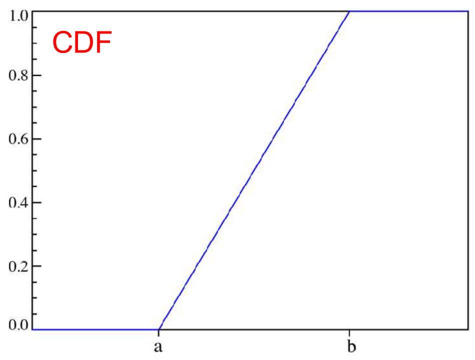
## PDF and CDF for a Uniform Distribution:



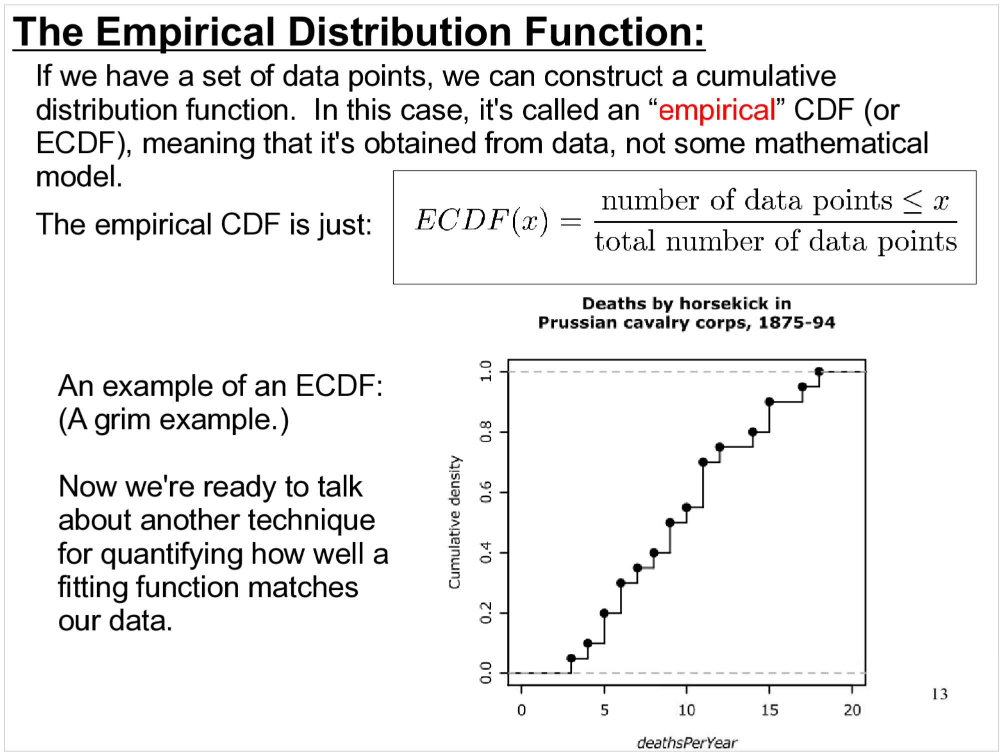
We can compute the CDF for other probability distributions, too.

At the left is the PDF for a uniform distribution, for a < x < b.

And here is the associated CDF. Notice that there's zero probability of x < a, and a probability of 1 for x < b.



The pages on Wikipedia for the various probability distrubutions are great. Each one shows graphs of the distribution's PDF and CDF, along with their functional forms.



## The Empirical Distribution Function:

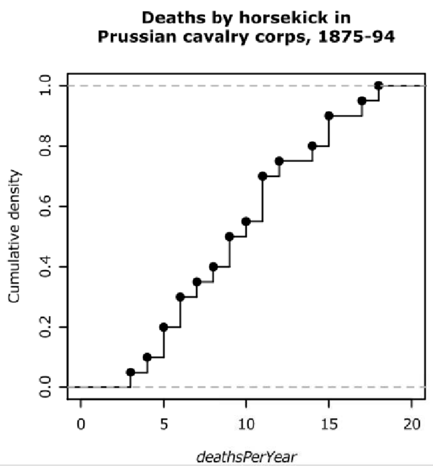
If we have a set of data points, we can construct a cumulative distribution function. In this case, it's called an "empirical" CDF (or ECDF), meaning that it's obtained from data, not some mathematical model.

The empirical CDF is just:

$$ECDF(x) = \frac{\text{number of data points} \leq x}{\text{total number of data points}}$$

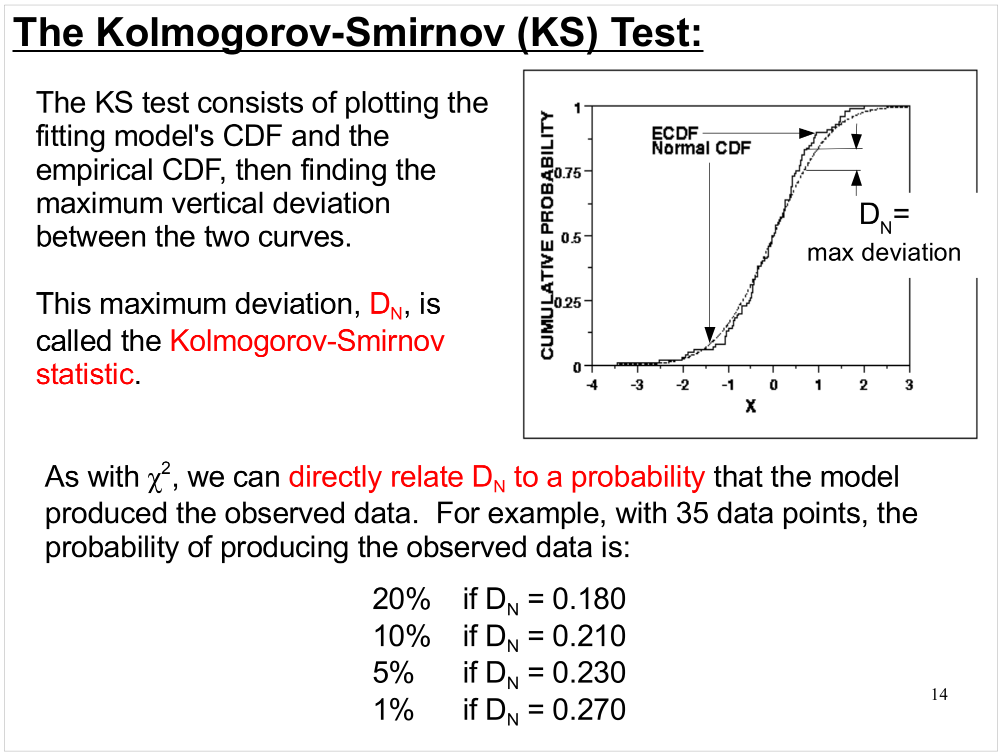
An example of an ECDF: (A grim example.)

Now we're ready to talk about another technique for quantifying how well a fitting function matches our data.



The ECDF is computed from "unbinned" data (i.e., you don't histogram the data and look at the bins, you look at the raw, unhistogrammed numbers).

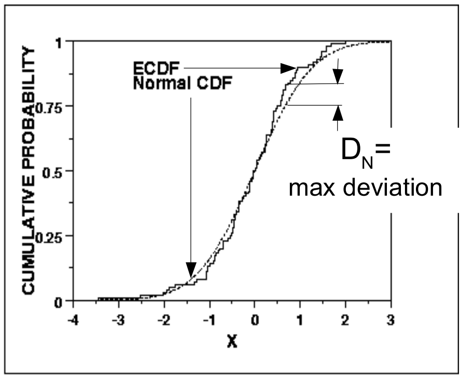
To construct an ECDF just sort the data numbers, $x_i$, in order of increasing value. Then, if the number of values is N, plot $x_i$ on the x-axis and i/N on the y-axis.



## The Kolmogorov-Smirnov (KS) Test:

The KS test consists of plotting the fitting model's CDF and the empirical CDF, then finding the maximum vertical deviation between the two curves.

This maximum deviation, $D_N$, is called the Kolmogorov-Smirnov statistic.

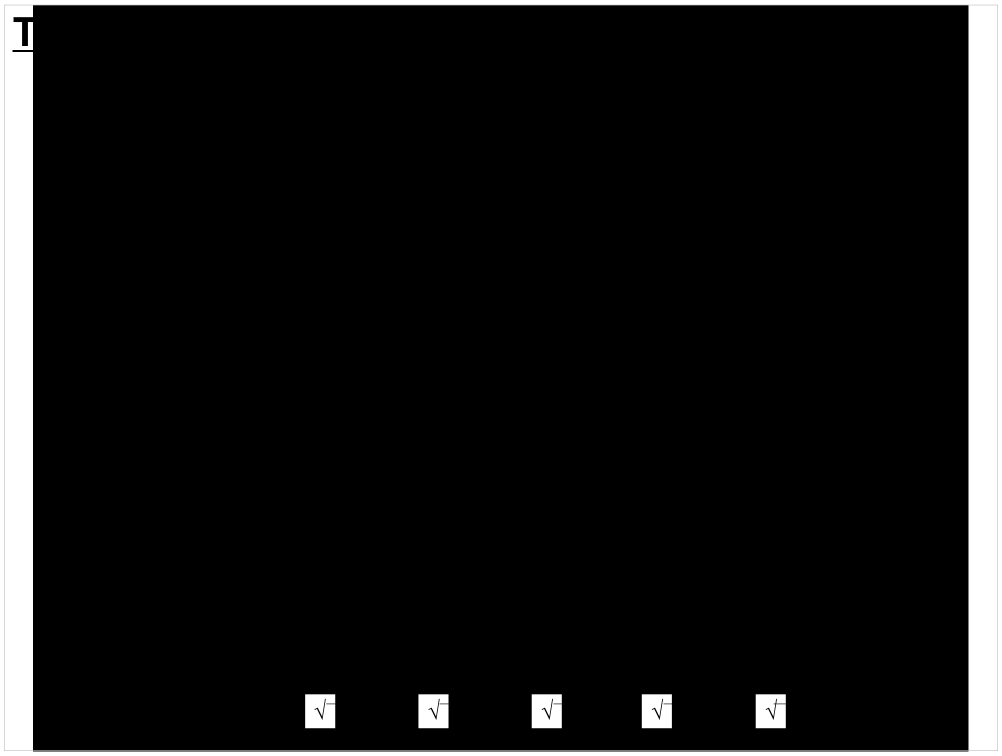


As with $\chi^2$, we can directly relate $D_N$ to a probability that the model produced the observed data. For example, with 35 data points, the probability of producing the observed data is:

| | |
|---|---|
| 20% | if $D_N = 0.180$ |
| 10% | if $D_N = 0.210$ |
| 5% | if $D_N = 0.230$ |
| 1% | if $D_N = 0.270$ |

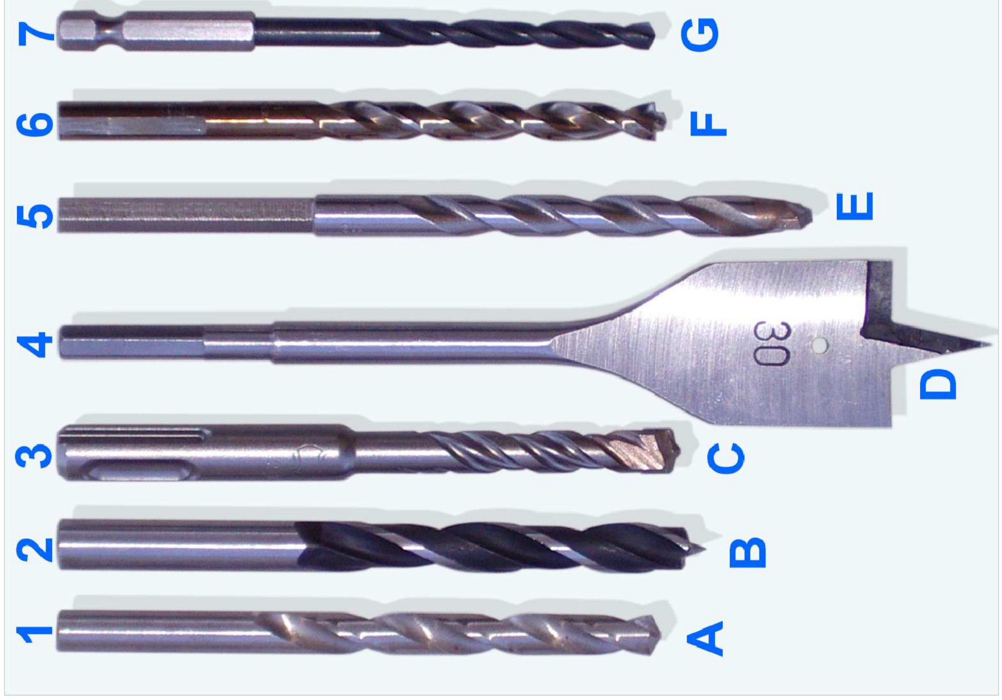
14

The Kolmogorov-Smirnov (KS) test provides us with a mathematical way of testing to see if a set of data is likely to have been generated by a given probability distribution.



Here's a table of such probabilities for various values of $D_N$ and N.

## Part 2: Binary Data



The next section deals with manipulating individual bits within data. This is a topic we haven't talked about yet, and we'll spend some time on it in the next couple of lectures, a lab, and a homework assignment.

## Bits and Bytes:

Until now, we've only dealt with data in 8-bit chunks called “bytes”. All of the variable types we've used have sizes that are integer multiples of 8 bits.

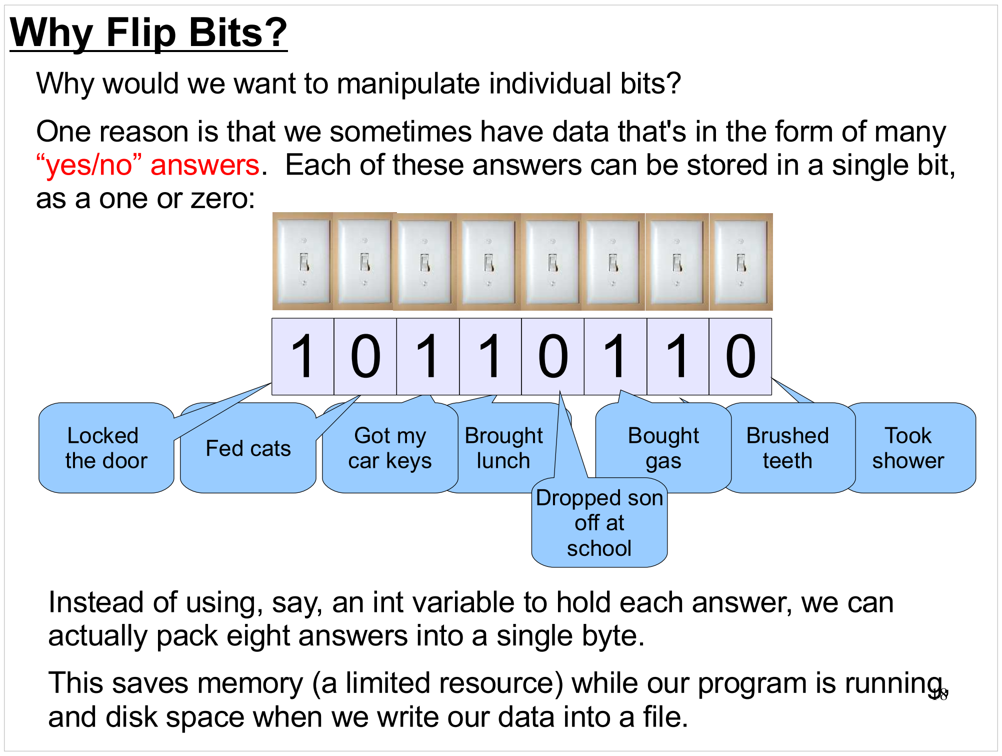
**8 bits:**

$$\underbrace{\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}}_{\text{1 Byte}}$$

**For g++ on Galileo:**

| | | |
|---|---|---|
| `sizeof(int)` | returns 4 | 4 bytes used to store an integer |
| `sizeof(double)` | returns 8 | 8 bytes used to store a double |
| `sizeof(char)` | returns 1 | 1 byte used to store a char |

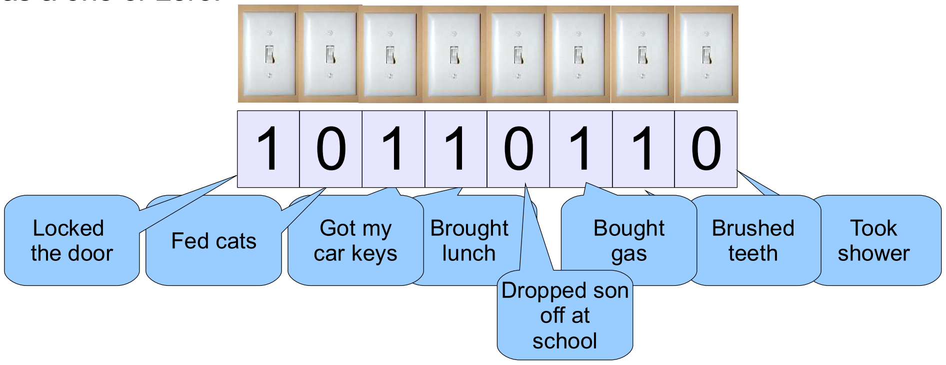
But sometimes we want to flip individual bits. Let's look at how that can be done.



## Why Flip Bits?

Why would we want to manipulate individual bits?

One reason is that we sometimes have data that's in the form of many "yes/no" answers. Each of these answers can be stored in a single bit, as a one or zero:



Instead of using, say, an int variable to hold each answer, we can actually pack eight answers into a single byte.

This saves memory (a limited resource) while our program is running, and disk space when we write our data into a file.

We often think of a set of bits like this as a checklist, where we can "set" a bit, by making it a 1, or "unset" it by making it a 0.

## Bitwise Operators:

C provides several operators for manipulating individual bits:

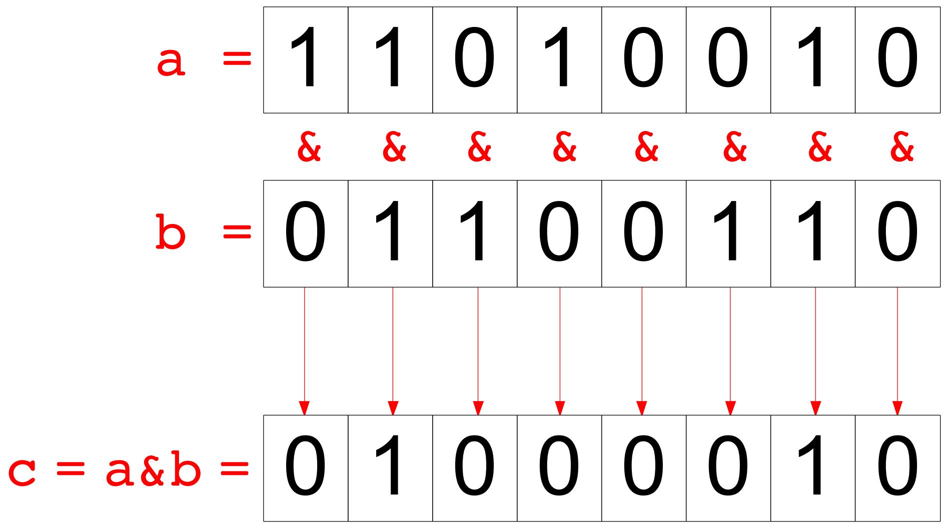
| | | |
|---|---|---|
| `&` | `a&b` | Bitwise *and* |
| `\|` | `a\|b` | Bitwise *or* |
| `<<` | `a<<b` | Left shift |
| `>>` | `a>>b` | Right shift |
| `~` | `~a` | Bitwise inverse |

Don't confuse the `&` and `|` operators with the `&&` and `||` operators we've used before.

Let's see what these new operators do.

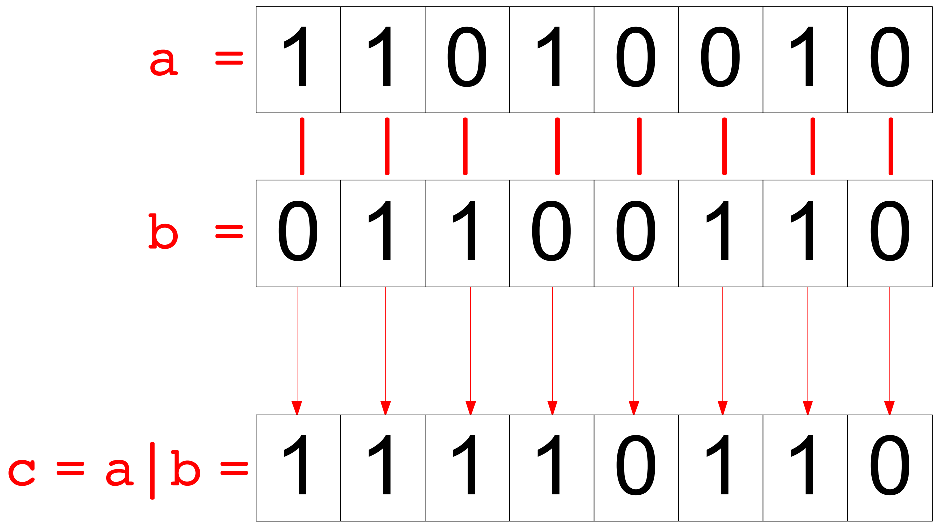
## Bitwise And:

The “&” operator performs a “bitwise and” on its two arguments. The bits of the returned value are computed by “and”ing together the corresponding bits of the two arguments. If both bits are “1”, then the resulting bit is “1”, otherwise, it's “0”.



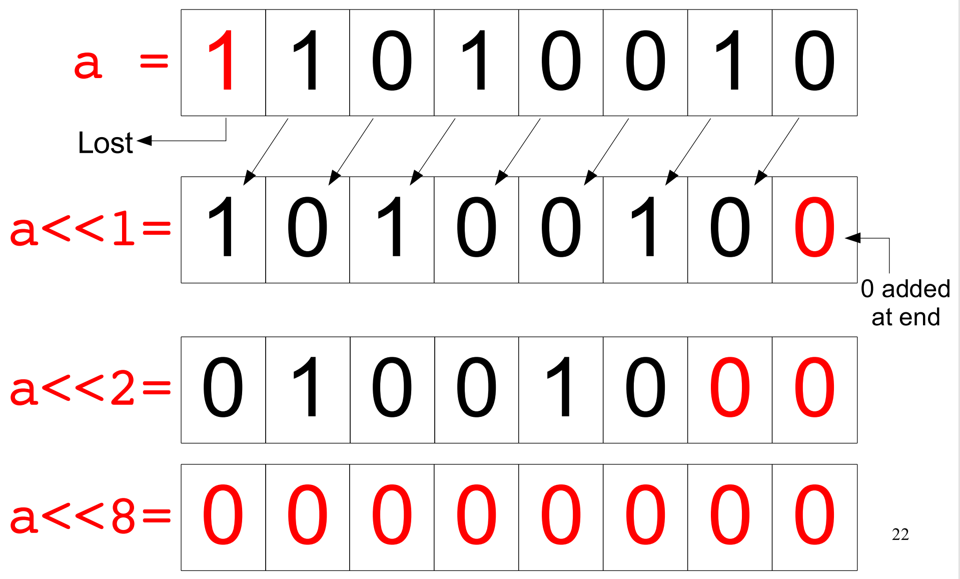
## Bitwise Or:

The "|" operator performs a "bitwise or" on its two arguments. That is, the bits of the returned value are computed by "or"ing together the corresponding bits of the two arguments. If either bit is "1", then the resulting bit is "1", otherwise it's "0".



## Left Shift:

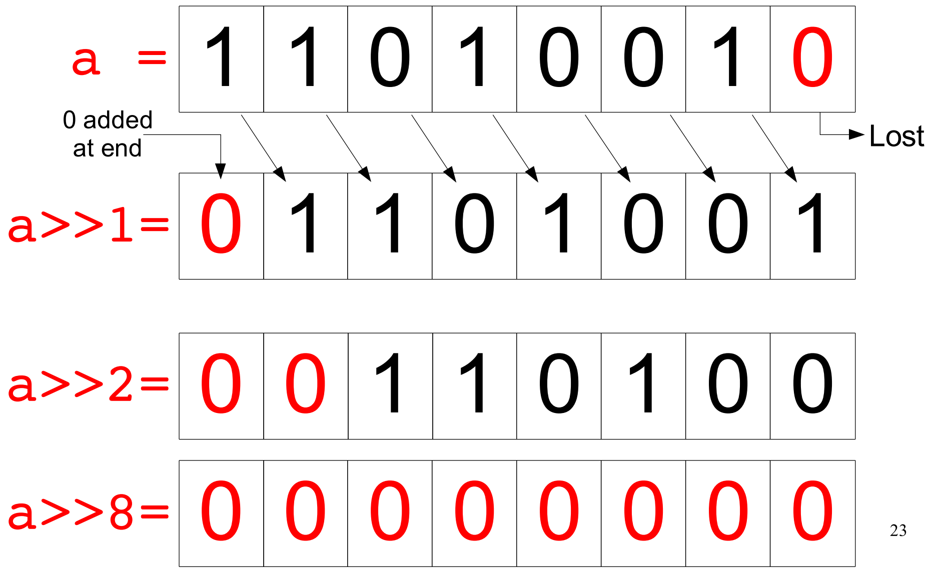
the “<<” operator shifts all of the bits to the left by a specified number of slots and returns the result. Bits shifted past the end of the byte are lost, and empty slots on the right-hand side are padded with zeros.

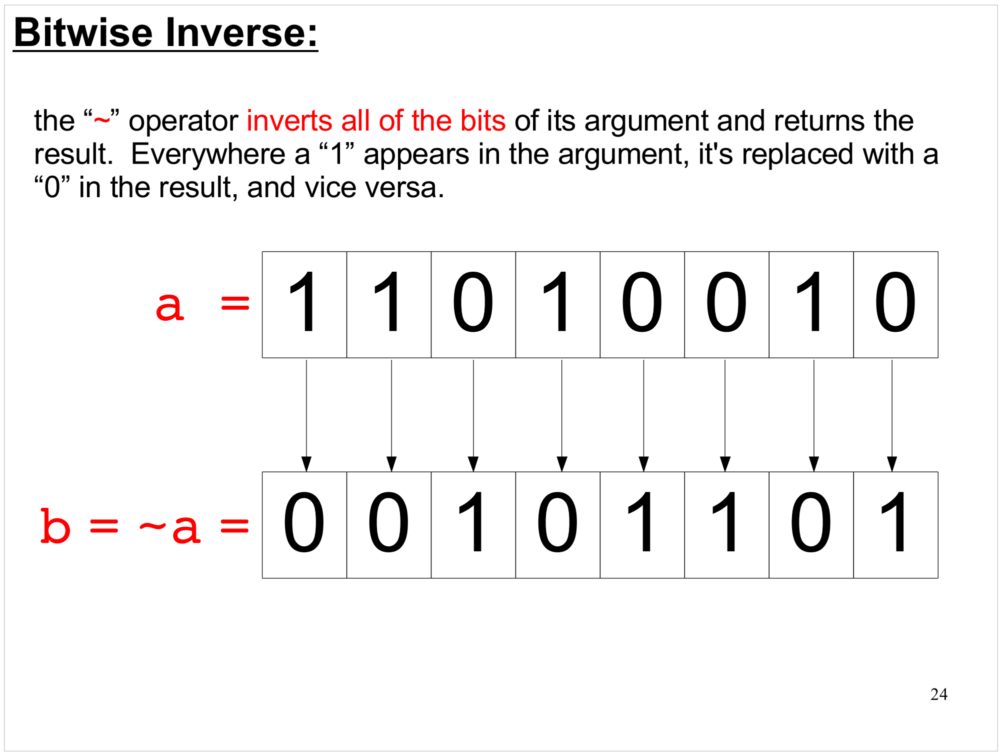


This seems like an odd thing to want to do, but we'll see what it's good for soon.

## Right Shift:

the “>>” operator shifts all of the bits to the right by a specified number of slots and returns the result.  Bits shifted past the end of the byte are lost, and empty slots on the left-hand side are padded with zeros.



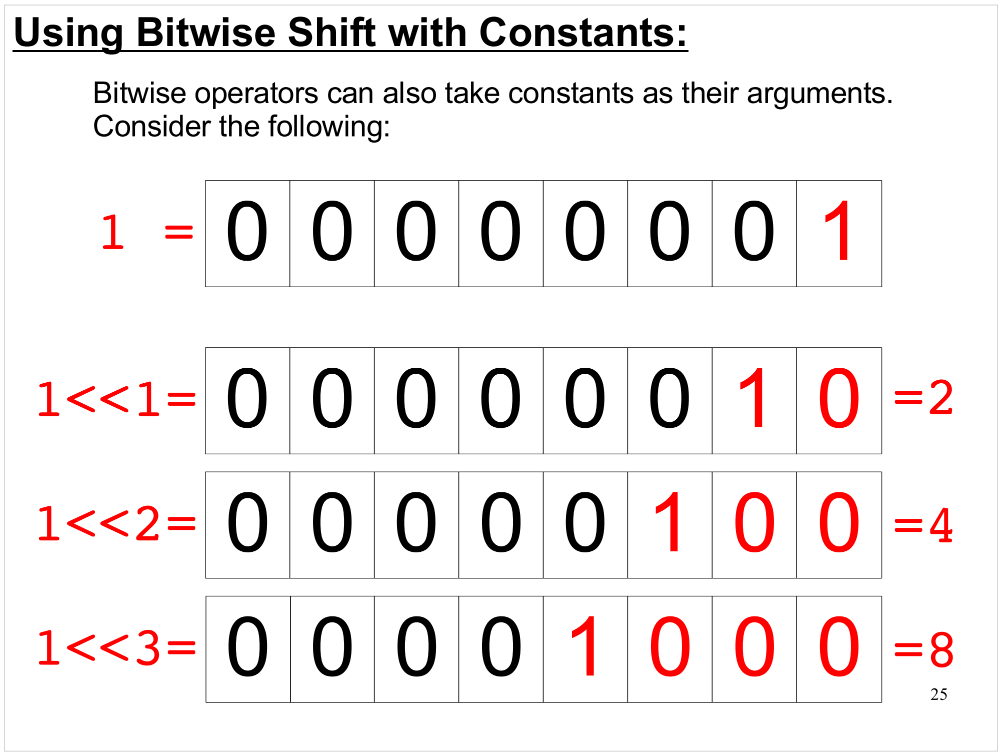


## Bitwise Inverse:

the “~” operator inverts all of the bits of its argument and returns the result. Everywhere a “1” appears in the argument, it's replaced with a “0” in the result, and vice versa.

a = 1 1 0 1 0 0 1 0
b = ~a = 0 0 1 0 1 1 0 1

Don't confuse this with the logical “!” operator.



## Using Bitwise Shift with Constants:

Bitwise operators can also take constants as their arguments. Consider the following:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | |
| 1<<1= | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | =2 |
| 1<<2= | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | =4 |
| 1<<3= | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | =8 |

25

Also note that 1<<0 is just equal to 1.

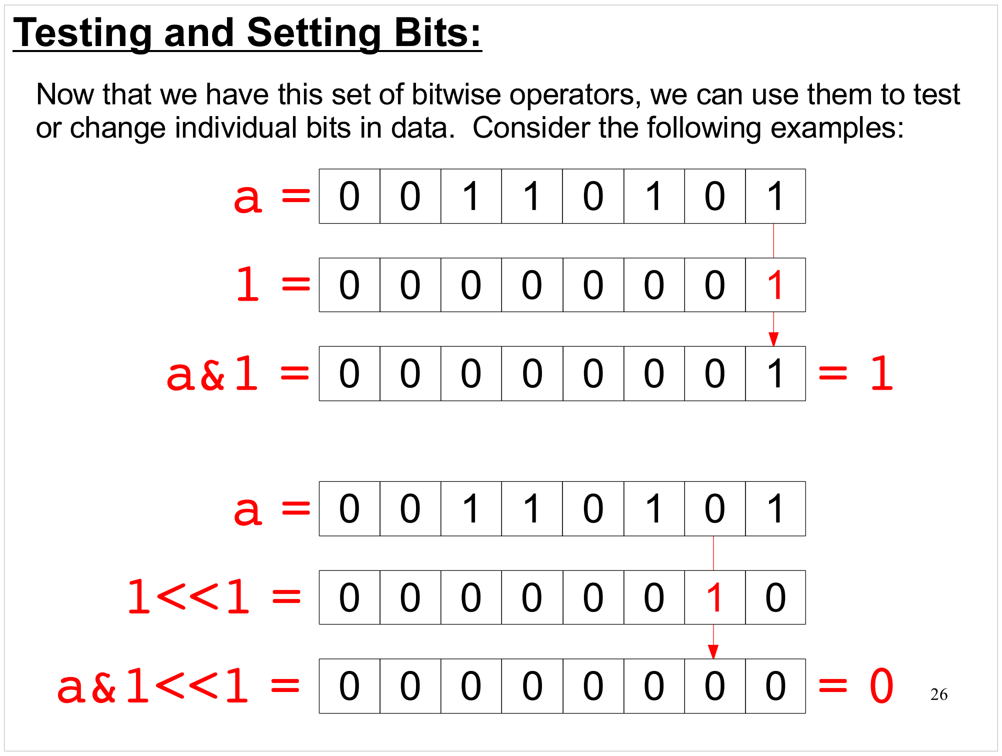
Notice that this gives us an easy way to make powers of two. Instead of, for example, writing:

```
pow(2,4)
```

for 2^4, we could just write:

```
1<<4
```

And, as you might imagine, the latter is much faster.



## Testing and Setting Bits:

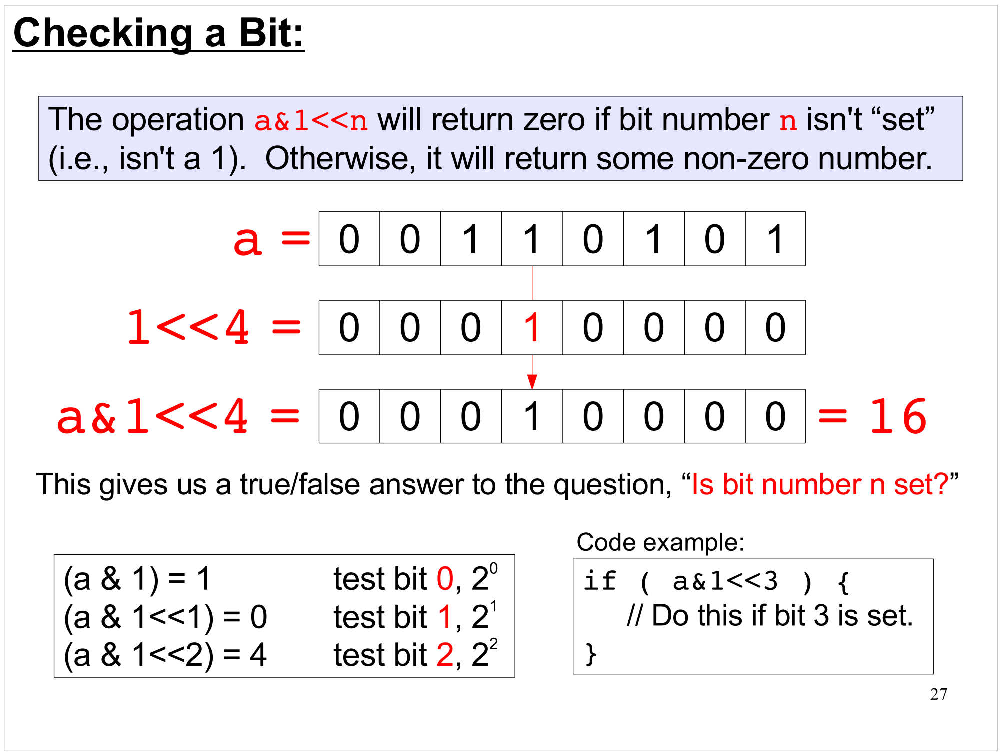
Now that we have this set of bitwise operators, we can use them to test or change individual bits in data. Consider the following examples:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a = | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | |
| 1 = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | |
| a&1 = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a = | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | |
| 1<<1 = | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| a&1<<1 = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 0 |

See what's happening here? Most of the bits in “1<<n” are zero (there's only one non-zero bit). When a bit from “a” gets anded with one of these zeros, the result will always be zero, since “anything & 0” is zero.

This means that all but one of the bits in the result will always be zero. The only bit in question is the bit that gets anded with “1”. If this bit of “a” is zero, then the result has zero. Otherwise, the result has a one.



## Checking a Bit:

The operation `a&1<<n` will return zero if bit number `n` isn't “set” (i.e., isn't a 1). Otherwise, it will return some non-zero number.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| `a =` | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | |
| `1<<4 =` | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| `a&1<<4 =` | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | `= 16` |

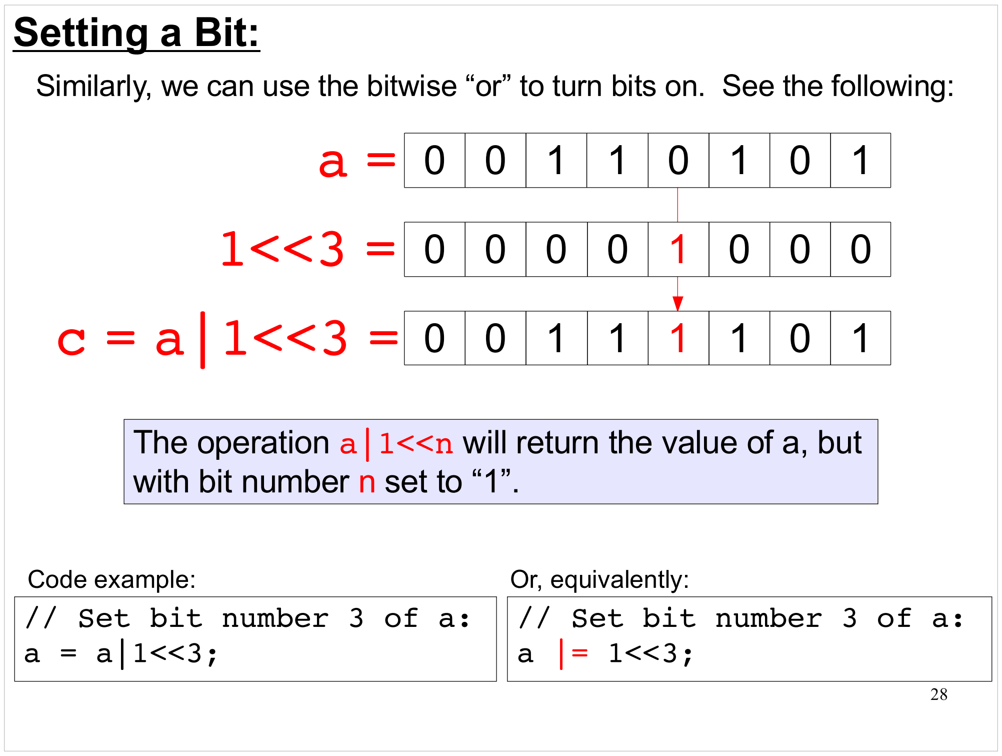
This gives us a true/false answer to the question, “Is bit number n set?”

| | |
|---|---|
| (a & 1) = 1 | test bit 0, $2^0$ |
| (a & 1<<1) = 0 | test bit 1, $2^1$ |
| (a & 1<<2) = 4 | test bit 2, $2^2$ |

Code example:

```
if ( a&1<<3 ) {
    // Do this if bit 3 is set.
}
```

You can think of “1<<4” as a “mask” that only lets one bit through. (Bit number 4, in this case.) The “1” in the bit pattern of “1<<4” is like a hole in the mask, allowing us to see through to the value of the corresponding bit in “a”, underneath.



## Setting a Bit:

Similarly, we can use the bitwise "or" to turn bits on.  See the following:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a = | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1<<3 = | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| c = a\|1<<3 = | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

The operation a|1<<n will return the value of a, but with bit number n set to "1".

Code example:

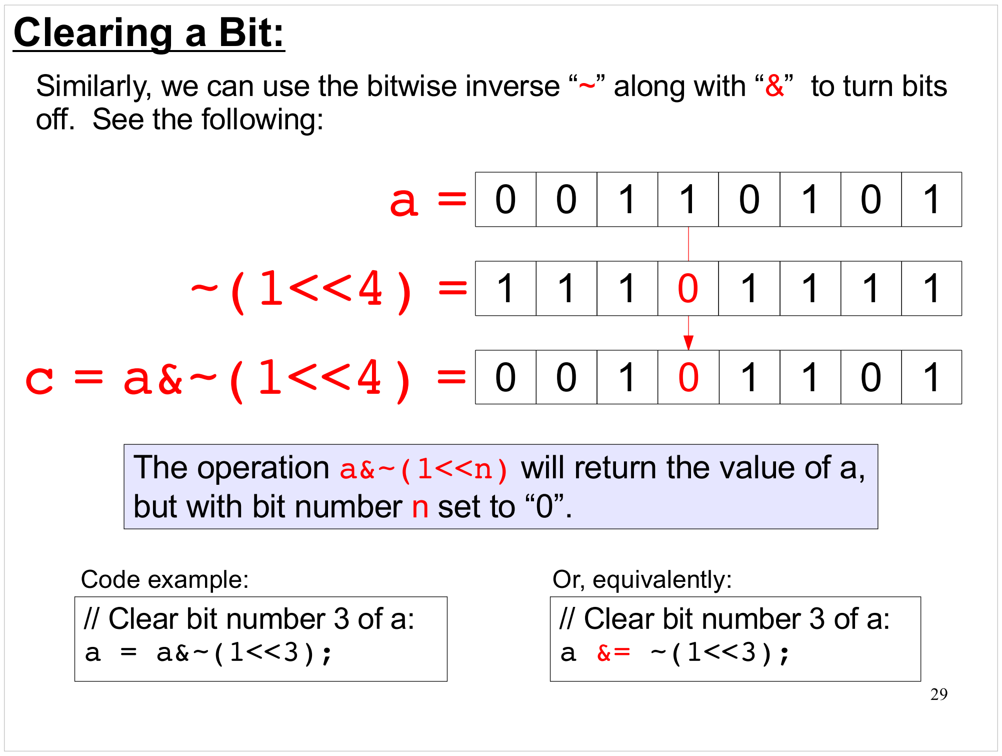
```
// Set bit number 3 of a:
a = a|1<<3;
```

Or, equivalently:

```
// Set bit number 3 of a:
a |= 1<<3;
```

In this example, all but one of the bits in "1<<3" is zero. Since "anything | 0" is just "anything", most of the bits in the result will have the same values they have in "a".

The one exception is the bit of "a" that's above the "1" in "1<<3". Since "anything | 1" is always 1, this bit of the result will always be 1.



## Clearing a Bit:

Similarly, we can use the bitwise inverse “~” along with “&” to turn bits off. See the following:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a = | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| ~(1<<4) = | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| c = a&~(1<<4) = | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |

The operation `a&~(1<<n)` will return the value of a, but with bit number n set to “0”.

Code example:

```
// Clear bit number 3 of a:
a = a&~(1<<3);
```

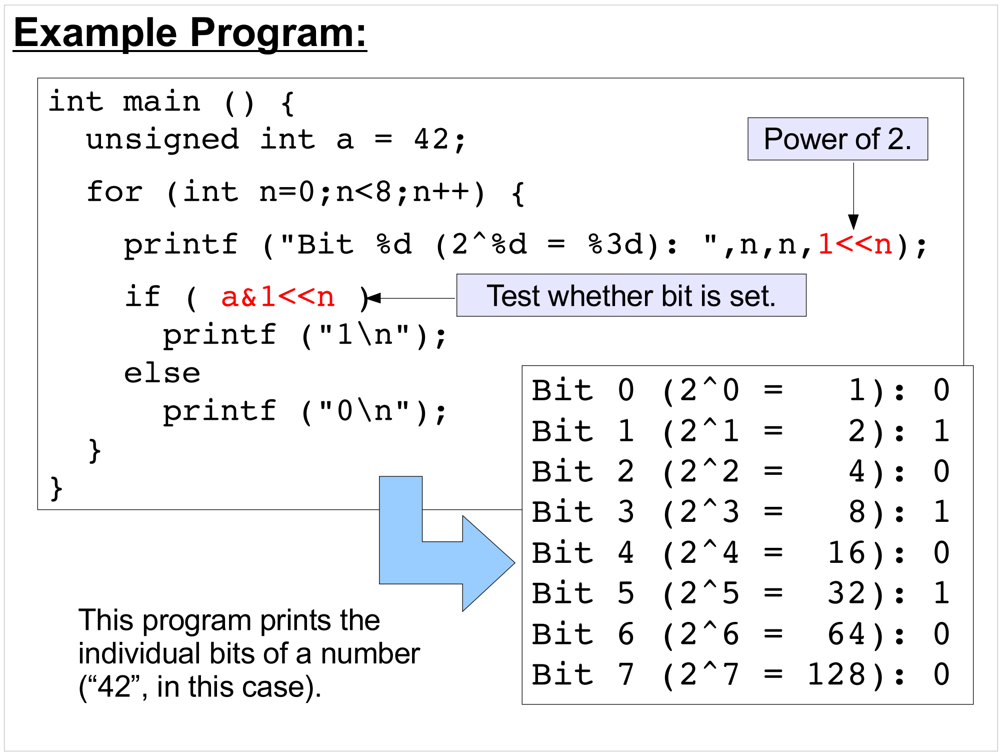
Or, equivalently:

```
// Clear bit number 3 of a:
a &= ~(1<<3);
```

29

Most of the bits in ~(1<<4) are ones. There's only one zero bit. Since “anything & 1” is just “anything”, most of the bits in the result will have the same values they have in “a”.

The one exception is the bit of “a” above the “0”. Since “anything & 0” is always zero, this bit will always be zero.



## Example Program:

```
int main () {
  unsigned int a = 42;
  for (int n=0;n<8;n++) {
    printf ("Bit %d (2^%d = %3d): ",n,n,1<<n);
    if ( a&1<<n )
      printf ("1\n");
    else
      printf ("0\n");
  }
}
```

Power of 2.

Test whether bit is set.

```
Bit 0 (2^0 =   1): 0
Bit 1 (2^1 =   2): 1
Bit 2 (2^2 =   4): 0
Bit 3 (2^3 =   8): 1
Bit 4 (2^4 =  16): 0
Bit 5 (2^5 =  32): 1
Bit 6 (2^6 =  64): 0
Bit 7 (2^7 = 128): 0
```

This program prints the individual bits of a number (“42”, in this case).

Note how we use 1<<n to write out the powers of two.

## Hexadecimal Representation:

Binary numbers are long, and it's easy to mis-type a 1 or 0. Because of this, we often represent binary numbers in "hexadecimal" (base 16) form. Hex notation works better than our normal "decimal" (base 10) system for this because each hex digit is equivalent to exactly 4 bits (half a byte).

In hex notation, there are 16 digits instead of the 10 we usually use, or the 2 (0 and 1) that are used in binary. Here's how you count in hex:

$$\boxed{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}$$

Decimal equivalents $\longrightarrow$ 10,11,12,13,14,15

| Decimal | Binary | Hex |
|---|---|---|
| 1 | 00000001 | 01 |
| 10 | 00001010 | 0A |
| 42 | 00101010 | 2A |
| 100 | 01100100 | 64 |
| 128 | 10000000 | 80 |
| 255 | 11111111 | FF |

Here are some examples showing decimal, binary and hex equivalents for a few numbers. $\longleftarrow$

## Decimal, Hex and Binary Table:

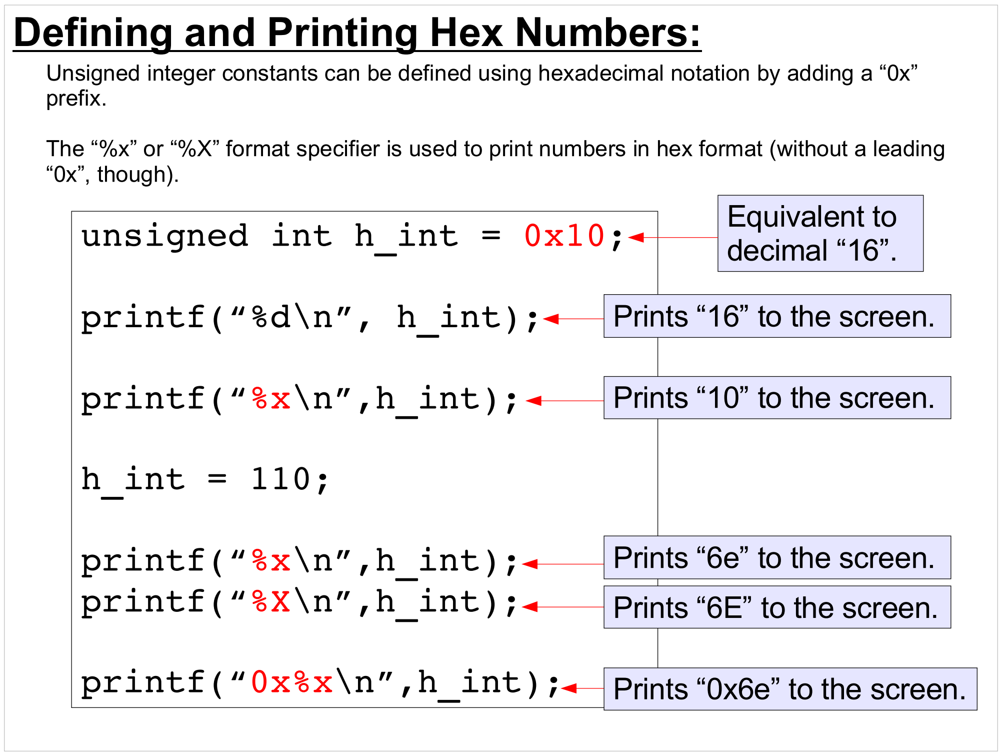
Some more examples. Note again that one hex digit is equivalent to four binary digits (half a byte).

| DECIMAL | HEX | BINARY |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |
| 16 | 10 | 10000 |
| | ... | |

All bits zero.

0 through F covers all the possible combinations of four bits.

All bits one.



## Defining and Printing Hex Numbers:

Unsigned integer constants can be defined using hexadecimal notation by adding a “0x” prefix.

The “%x” or “%X” format specifier is used to print numbers in hex format (without a leading “0x”, though).

```
unsigned int h_int = 0x10;

printf(“%d\n”, h_int);

printf(“%x\n”,h_int);

h_int = 110;

printf(“%x\n”,h_int);
printf(“%X\n”,h_int);

printf(“0x%x\n”,h_int);
```

Equivalent to decimal “16”.

Prints “16” to the screen.

Prints “10” to the screen.

Prints “6e” to the screen.

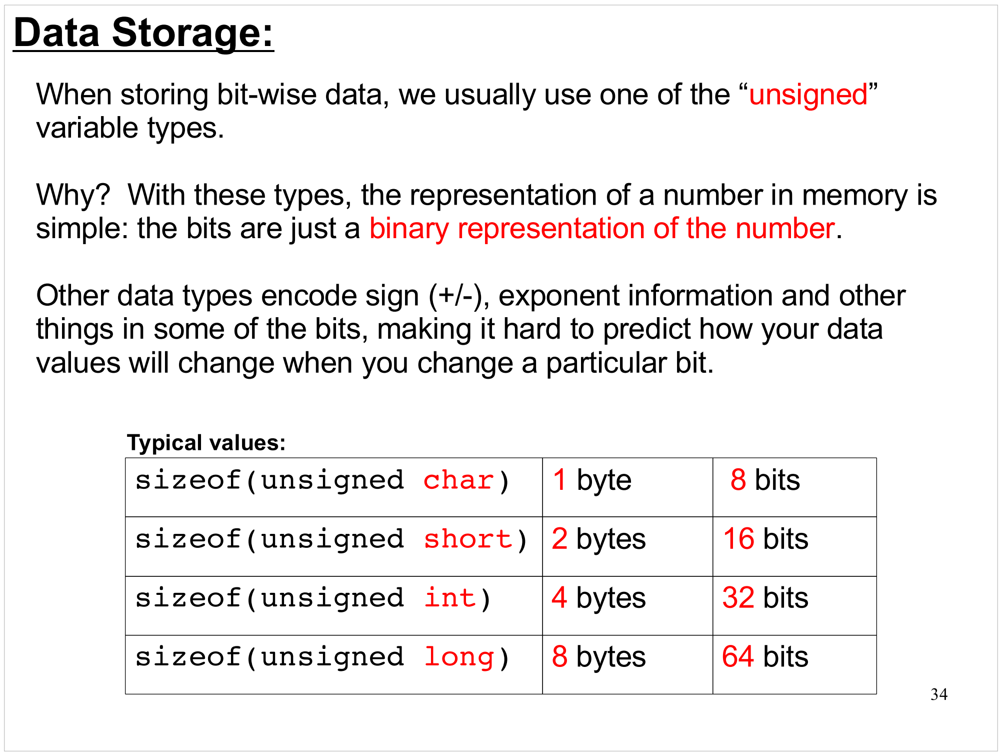
Prints “6E” to the screen.

Prints “0x6e” to the screen.

The history of the “0x” notation goes like this: a long time ago, people often used “octal” (base 8) notation with computers. In C, you can write octal numbers by just prefixing a zero in front of the number, like “012”. The leading zero indicates that this number is to be interpreted in something other than base 10.

To accomodate hexadecimal numbers, C added an “x” after the zero. (For “heXadecimal”.) The C compiler interprets “0x12” something like this:

“Here comes a non-decimal number. Oh, and by the way, it's hexadecimal. And the digits are 1,2.”



## Data Storage:

When storing bit-wise data, we usually use one of the "unsigned" variable types.

Why? With these types, the representation of a number in memory is simple: the bits are just a binary representation of the number.

Other data types encode sign (+/-), exponent information and other things in some of the bits, making it hard to predict how your data values will change when you change a particular bit.

**Typical values:**

| | | |
|---|---|---|
| `sizeof(unsigned char)` | 1 byte | 8 bits |
| `sizeof(unsigned short)` | 2 bytes | 16 bits |
| `sizeof(unsigned int)` | 4 bytes | 32 bits |
| `sizeof(unsigned long)` | 8 bytes | 64 bits |

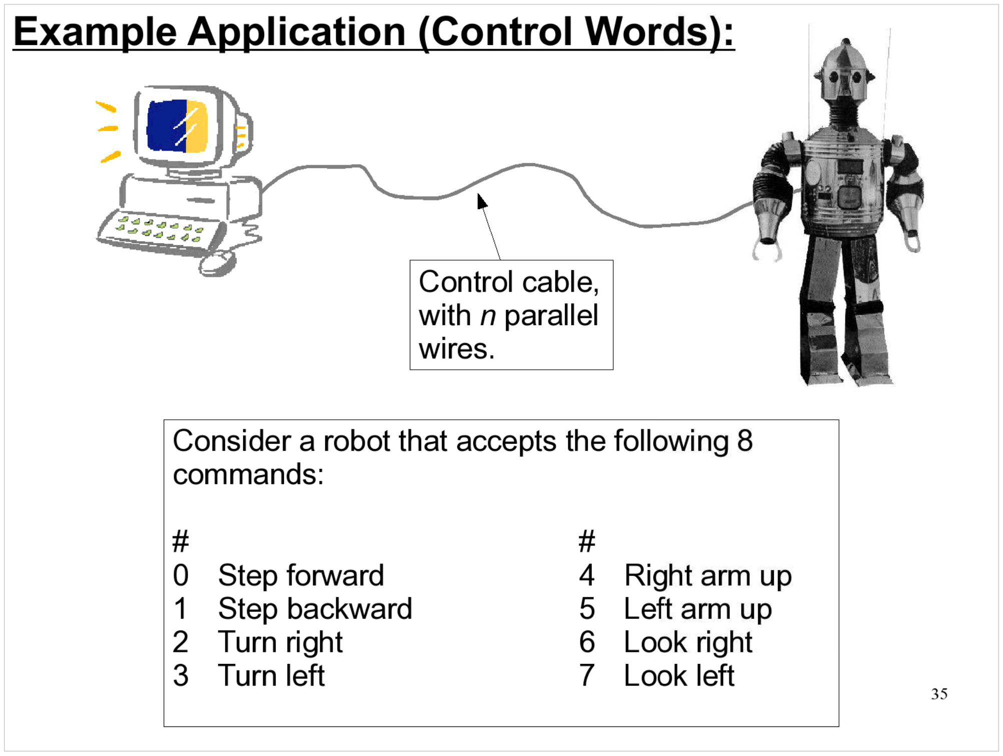
34

Since the "unsigned" variable types have such a simple, direct representation in memory, we can quickly do multiplication or division by powers of 2 on them by using the bitwise shift operators:

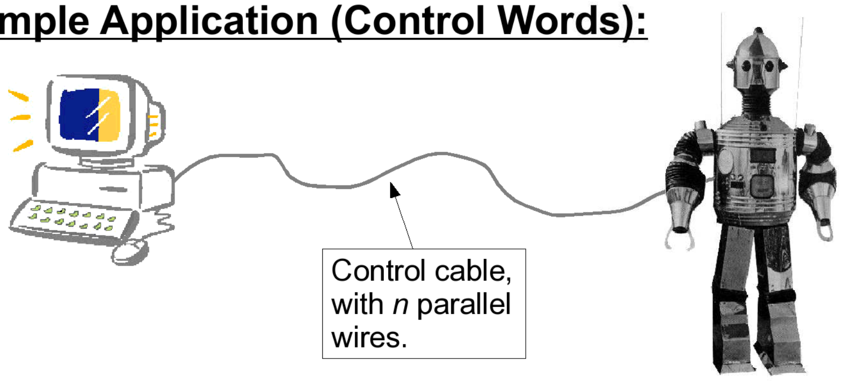
```
unsigned int i = 42;
unsigned int j;

j = i<<1;  // Multiply i by 2.
j = i<<2;  // Multiply i by 4.

j = i>>1;  // Divide i by 2.
j = i>>2;  // Divide i by 4.
```



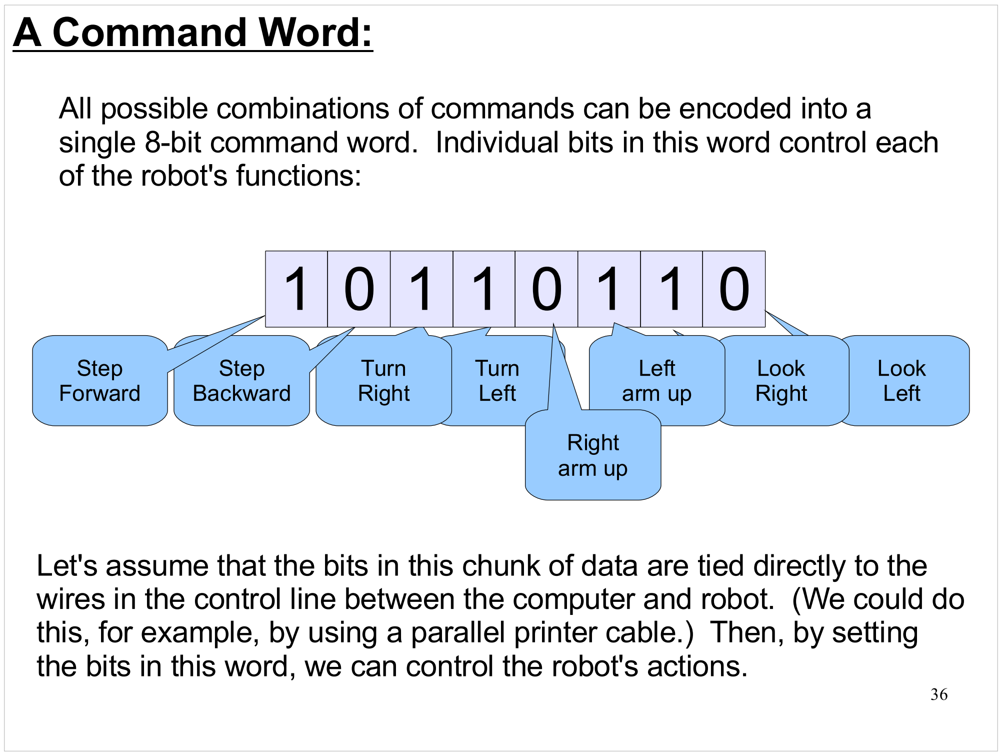
## Example Application (Control Words):



Consider a robot that accepts the following 8 commands:

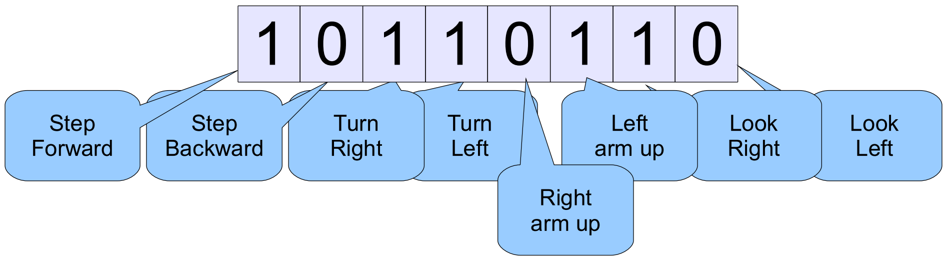
| # |  | # |  |
|---|---|---|---|
| 0 | Step forward | 4 | Right arm up |
| 1 | Step backward | 5 | Left arm up |
| 2 | Turn right | 6 | Look right |
| 3 | Turn left | 7 | Look left |

So, what kinds of things might we use these operators for? Here's an example of one kind of application where bitwise operations might be useful.



## A Command Word:

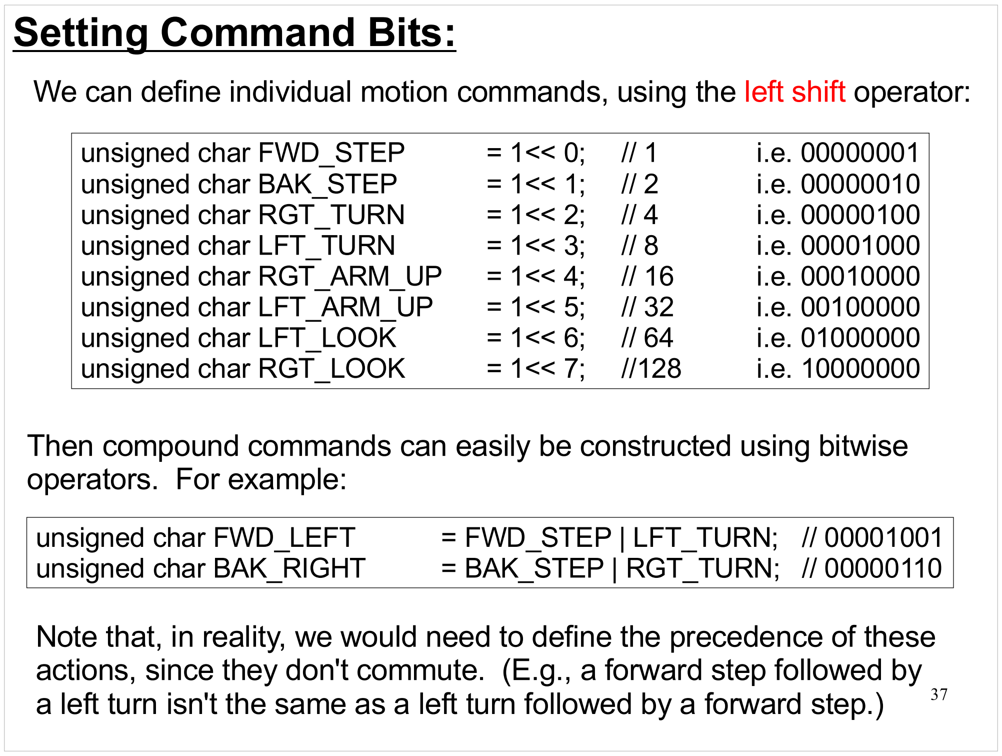
All possible combinations of commands can be encoded into a single 8-bit command word. Individual bits in this word control each of the robot's functions:



Let's assume that the bits in this chunk of data are tied directly to the wires in the control line between the computer and robot. (We could do this, for example, by using a parallel printer cable.) Then, by setting the bits in this word, we can control the robot's actions.

36

If we could flip the bits on and off, like light switches, we could control the robot's behavior.



## Setting Command Bits:

We can define individual motion commands, using the left shift operator:

```
unsigned char FWD_STEP       = 1<< 0;   // 1       i.e. 00000001
unsigned char BAK_STEP       = 1<< 1;   // 2       i.e. 00000010
unsigned char RGT_TURN       = 1<< 2;   // 4       i.e. 00000100
unsigned char LFT_TURN       = 1<< 3;   // 8       i.e. 00001000
unsigned char RGT_ARM_UP     = 1<< 4;   // 16      i.e. 00010000
unsigned char LFT_ARM_UP     = 1<< 5;   // 32      i.e. 00100000
unsigned char LFT_LOOK       = 1<< 6;   // 64      i.e. 01000000
unsigned char RGT_LOOK       = 1<< 7;   //128      i.e. 10000000
```

Then compound commands can easily be constructed using bitwise operators. For example:

```
unsigned char FWD_LEFT        = FWD_STEP | LFT_TURN;   // 00001001
unsigned char BAK_RIGHT       = BAK_STEP | RGT_TURN;   // 00000110
```

Note that, in reality, we would need to define the precedence of these actions, since they don't commute. (E.g., a forward step followed by a left turn isn't the same as a left turn followed by a forward step.)

We'll do more with bitwise operations in this week's lab.

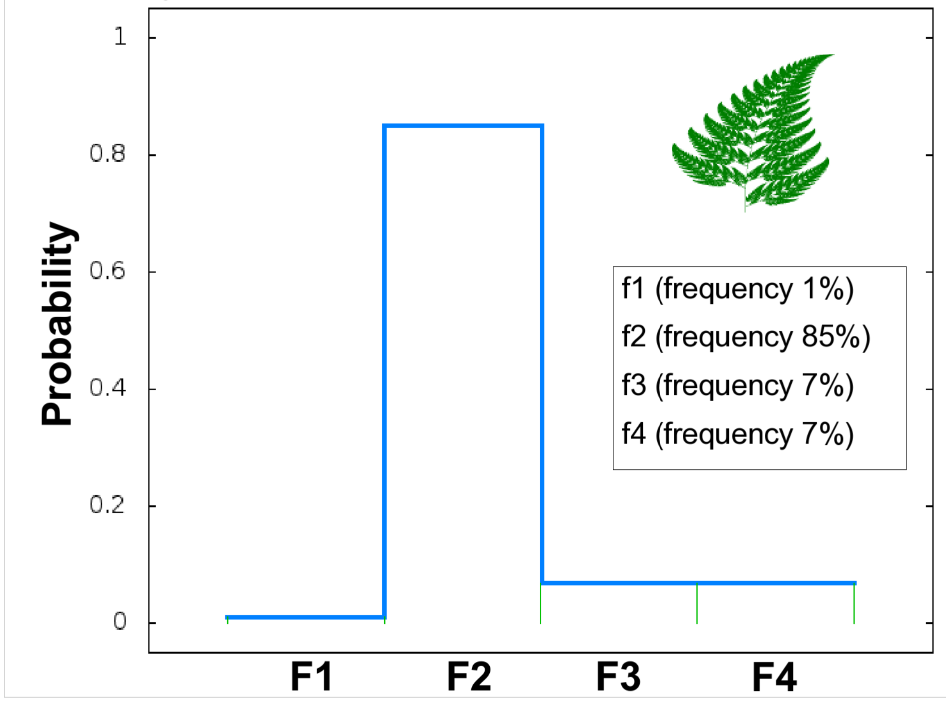
## Part 3: Arbitrary Random Number Distributions

Here's an arbitrary random picture of a bullfrog from the pond down the street.

We've talked a lot about a couple of probability distributions: the Gaussian (Normal) distribution and the uniform distribution.  We've seen how it's possible to generate random numbers according to each of these.

But what if we want to generate random numbers in some other distribution?  Is there some general way to generate random numbers in a given, arbitrary distribution?

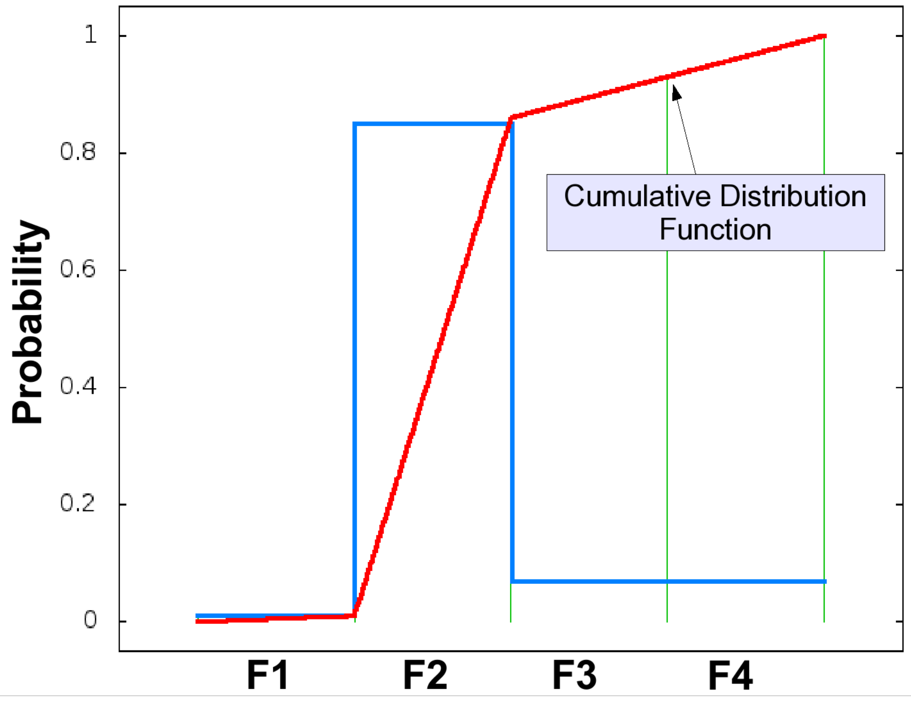
**Barnsley Fern Example:**



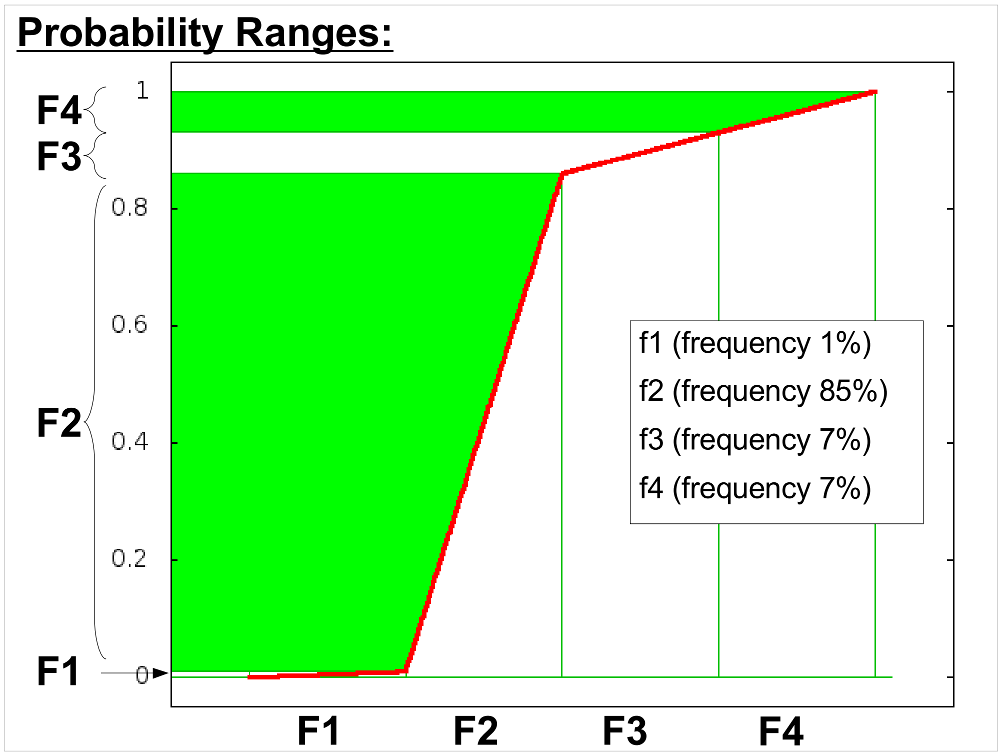
If you remember the Barnsley Fern problem we did a few weeks ago, you may recall that it required us to pick one of four functions (f1, f2, f3, or f4) at random, with a different probability for picking each function.

We could represent that as a Probability Density Function, like the graph above. The height of the blue line just represents the probability of picking each function.

## Barnsley Fern CDF:



We could integrate this PDF and come up with a CDF, like the one shown in red.

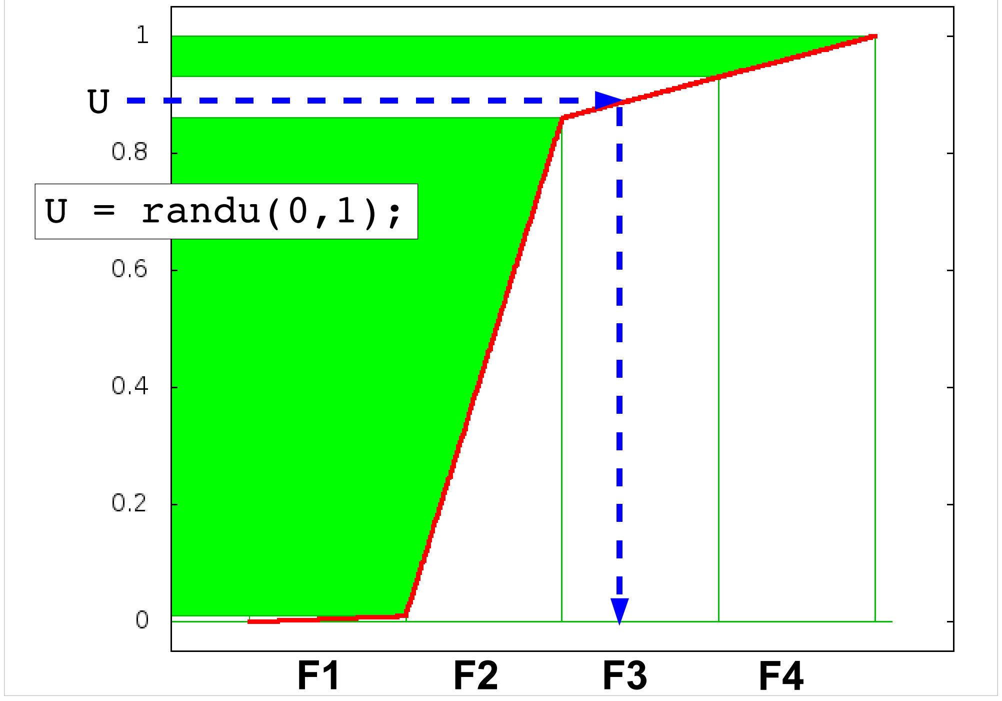


Do you recall how we picked which function to use? We chose a uniformly-distributed random number, U, between 0 and 1, and then said:

```
if U between 0.00 and 0.01
    pick f1
if U between 0.01 and 0.86
    pick f2
if U between 0.86 and 0.93
    pick f3
if U between 0.93 and 1.000
    pick f4
```

When we did this, we were just integrating the PDF in our heads, and creating the CDF.

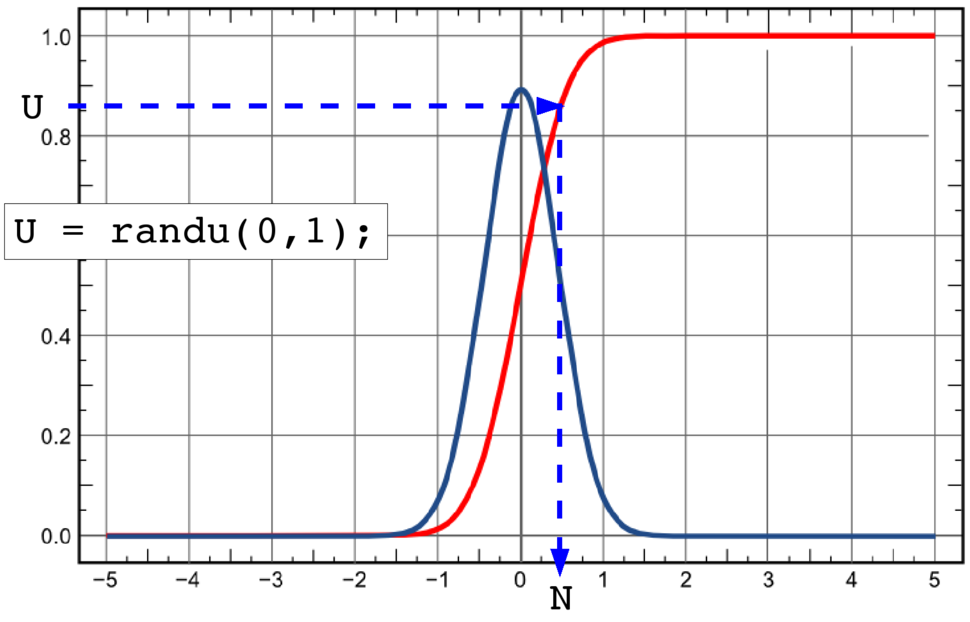
## Picking a Function at Random:



The uniformly-distributed random number we picked just corresponds to the vertical axis on the graph. Following the blue dashed line, it told us which function to pick.

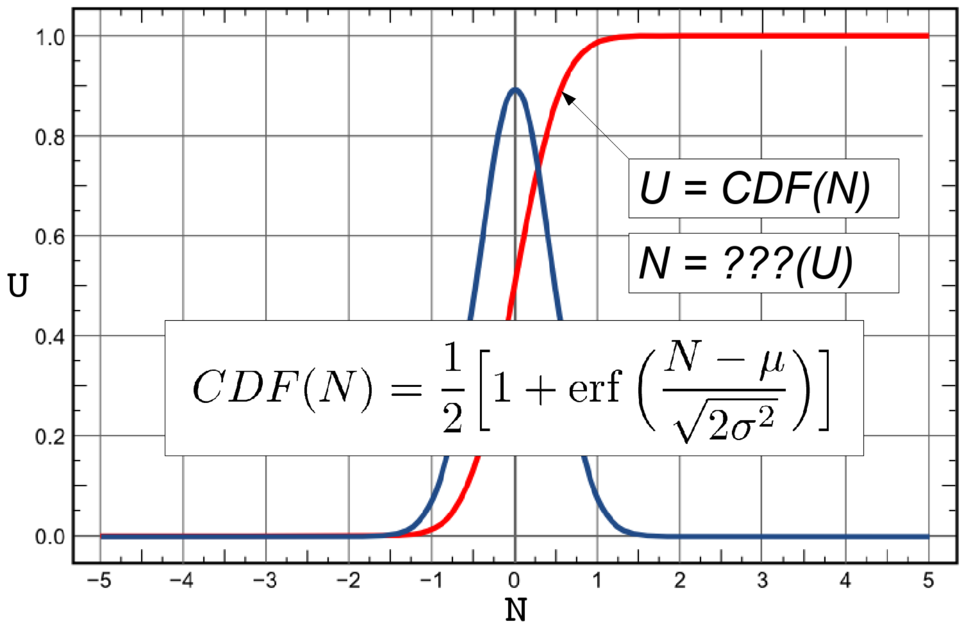
It turns out that you can use the same procedure to generate random numbers in any distribution you want. All you need to know is the distribution's CDF.

**Generating a Normal Distribution from a CDF:**



For example, here's the PDF (dark blue) and CDF (red) for a Gaussian (Normal) distribution. By picking a uniformly-distributed random number between 0 and 1 on the y-axis, we could get a number N from the x-axis. The values of N picked in this way would be distributed in a Gaussian way. If we histogrammed these numbers, we'd see the familiar Gaussian bell curve.

**Functional Form of the Normal CDF:**

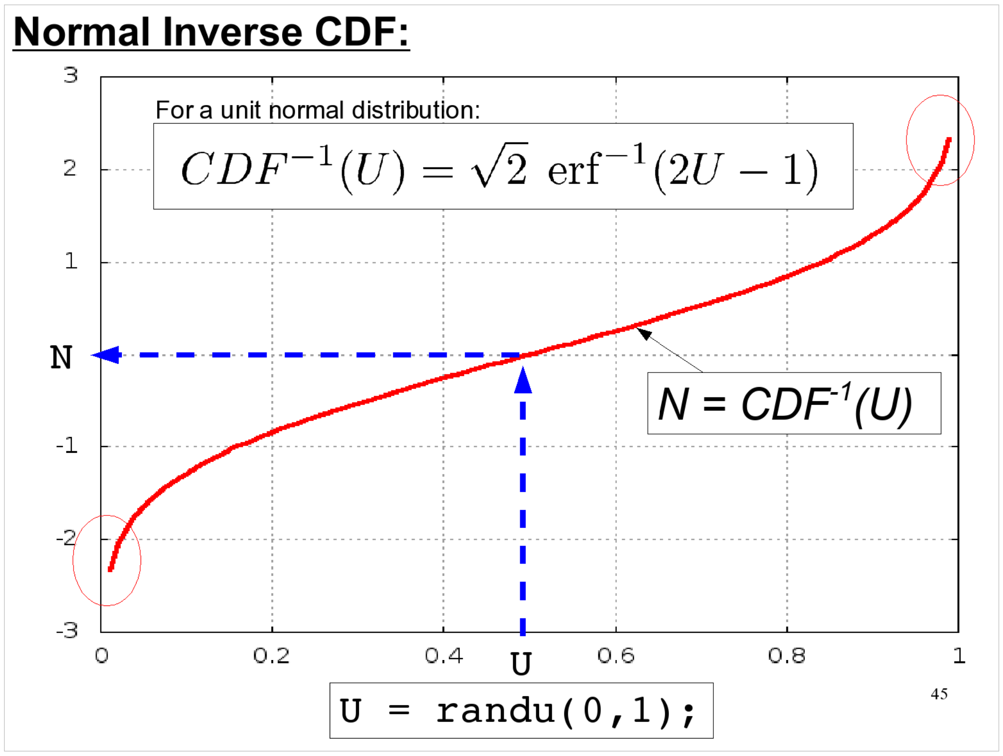


$$CDF(N) = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{N-\mu}{\sqrt{2\sigma^2}}\right)\right]$$

The devil is in the details, though. We'd like to be able to generate U, then stick U into some formula that gives us N. The CDF, though, gives us a formula for getting U from N (the exact opposite of what we want).

Can we invert the CDF to get a function that does what we want?

The actual form of the CDF for a Gaussian is shown in the formula at the bottom. Can we solve this equation for N as a function of U?

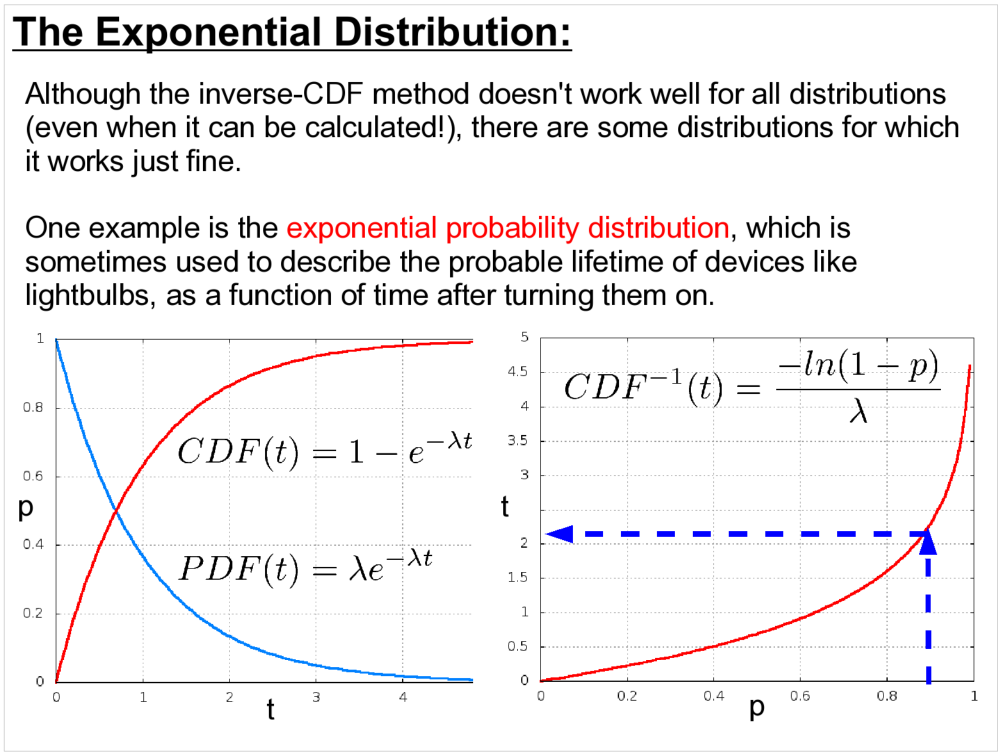


Yes, we can! The formula for the “inverse CDF” of a unit Normal distribution is shown at the top. Using this, we could just generate uniformly-distributed U values on the x axis, and plug those values into the function to get the corresponding normally-distributed N values.

There's a practical problem with this, though. In the regions that are circled at either end of the graph, the function is zooming up (or down) very fast. This means that a small change in U in this region will cause a large change in N. Note that these regions are only two or three standard deviations away from N=0.

Once we're beyond a few sigma away from N=0, we'll find that small computational errors will multiply rapidly, and our results for N will become meaningless.

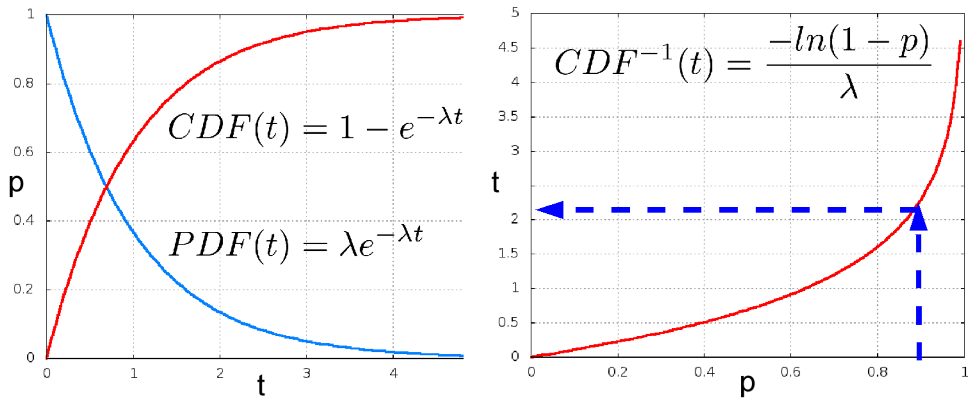
This is why we don't typically use this method to generate normally-distributed numbers. Instead, we use tricks like the Box-Muller method that our “randn” function uses.



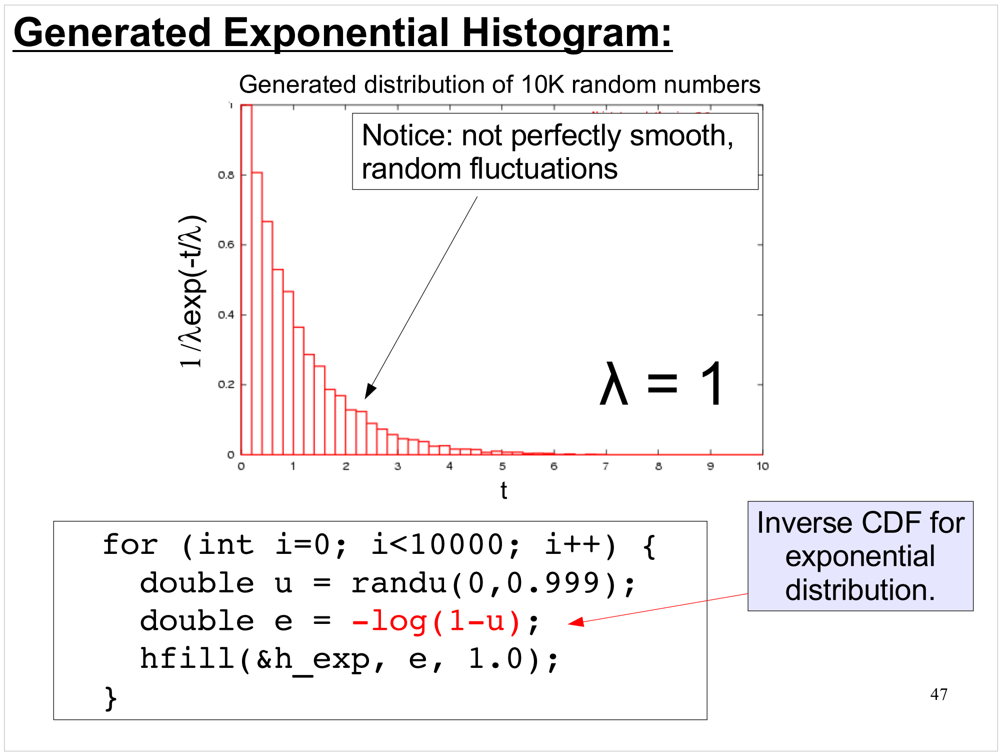
## The Exponential Distribution:

Although the inverse-CDF method doesn't work well for all distributions (even when it can be calculated!), there are some distributions for which it works just fine.

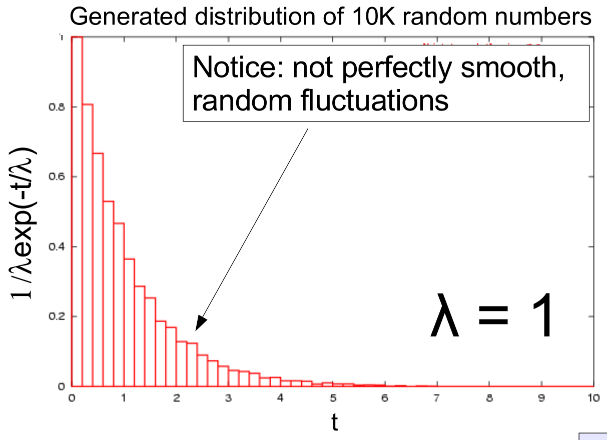
One example is the exponential probability distribution, which is sometimes used to describe the probable lifetime of devices like lightbulbs, as a function of time after turning them on.



Some distributions work better, though. Here's an example of one that does.



## Generated Exponential Histogram:

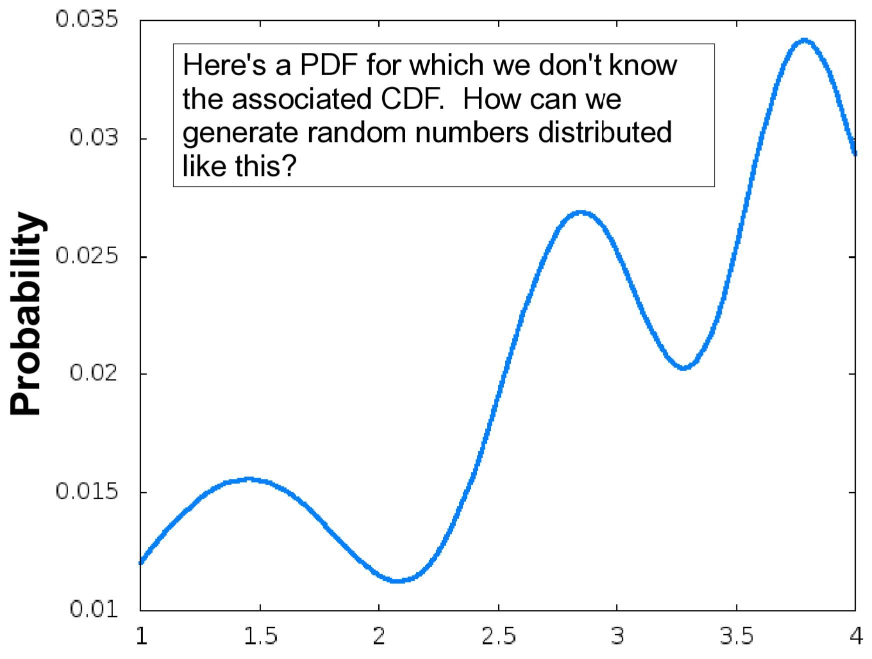


```
for (int i=0; i<10000; i++) {
  double u = randu(0,0.999);
  double e = -log(1-u);
  hfill(&h_exp, e, 1.0);
}
```

Inverse CDF for exponential distribution.

And here's an example, showing how you can use the inverse CDF of the exponential distribution to actually generate numbers distributed this way.

## Arbitrary PDFs:

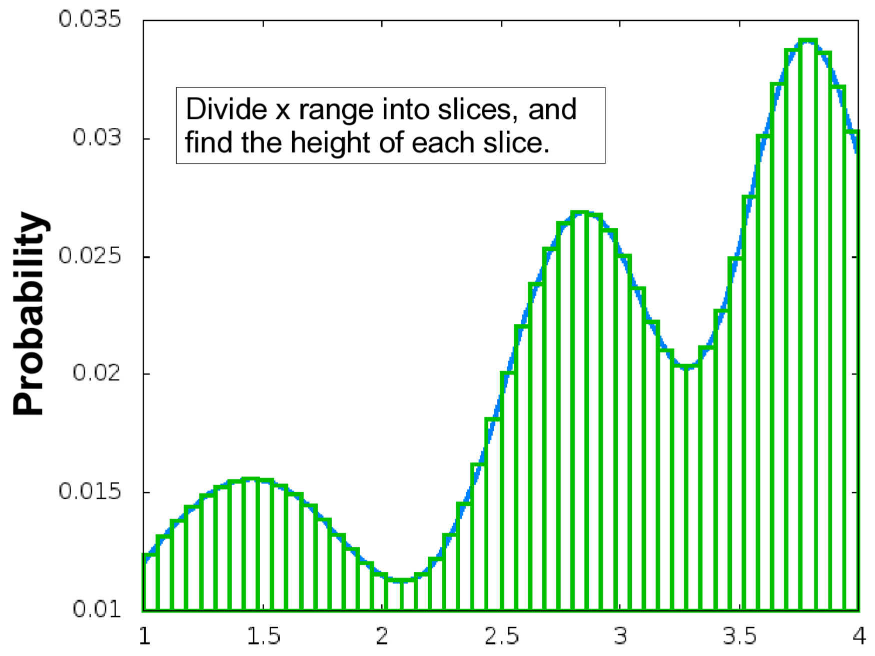


Here's a PDF for which we don't know the associated CDF. How can we generate random numbers distributed like this?

Of course we are often interested in distributions for which the CDF isn't known, or can't be inverted.

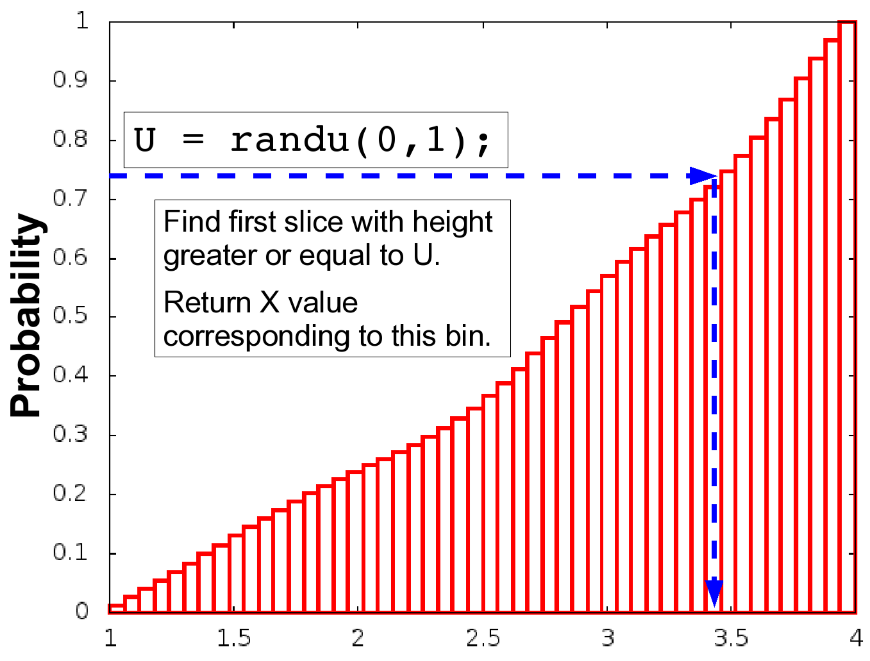
There are a couple of methods we could use to generate numbers distributed in any way at all. First, we could approximate the inverse of the CDF and its inverse by using a fixed number of samples. We've done similar things for the purpose of integration (estimating the area underneath a curve). Let's see how that works.

**Binning Data:**



First, slice up the x axis, and find the height of each slice.

**Calculated CDF:**



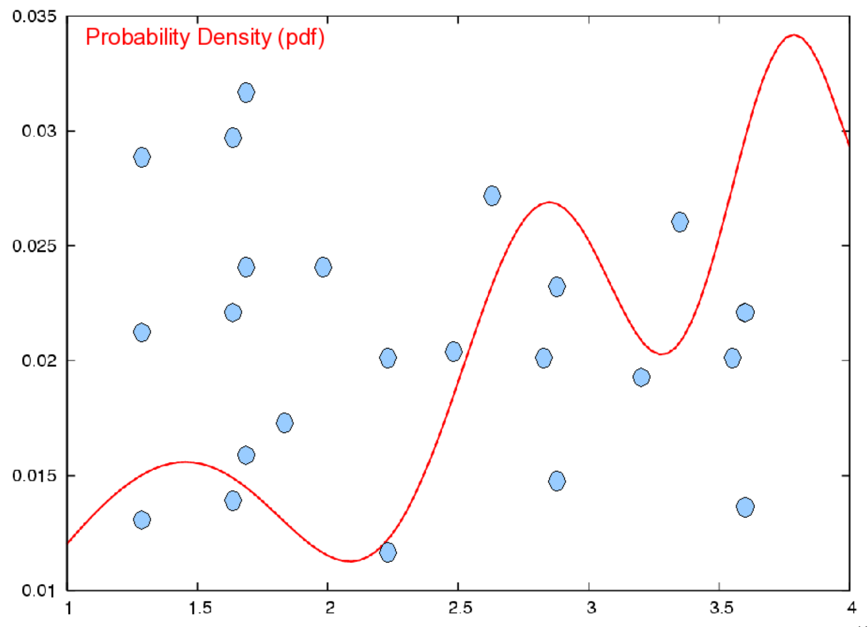
As we're making slices from left to right,  store the sum of all the slices so far into an array element. When we're done, the array will contain our Cumulative Distribution Function (or an approximation of it).

(Of course, we'll need to normalize it to a maximum value of 1. We can do this by dividing all of the aray elements by the last, largest element.)

We can use this CDF to generate random numbers distributed like the original PDF.  Just generate a uniformly-distributed random number, U, then loop through our array elements until we find one with a value greater than U.  The N value corresponding to this array element is the one we want.

By using a smaller step size, we can increase the accuracy.

## A Monte Carlo Method:



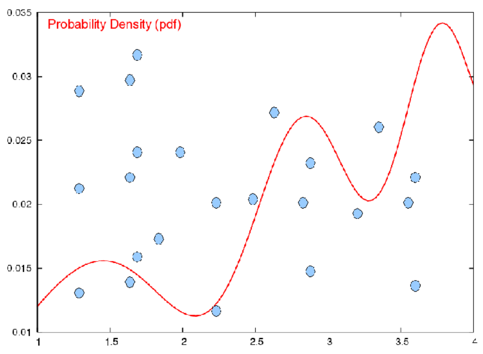
It turns out that there's another, easier but slower way to generate random numbers in any distribution you want. It's essentially another Monte Carlo method, similar to the Monte Carlo integration techniques we looked at earlier.

Say, for example, we want to generate random numbers according to some funky probability distribution like the red line in the graph above. All we need to do is generate some random points. For each point, we check to see if it's underneath the red curve. If it is, then we return this point's x value as our random number. If it's not, we try again.

## Monte Carlo Explanation:

This technique is slow, but it will always work, even for the oddest probability distributions.

How does it work?



As we generate random points, it's more likely for a point to fall under the curve at x values where the curve is high (there's more room underneath), so more of our random numbers will come from these places.

At x values where the curve is low, points will be less likely to fall underneath the curve, so we'll get fewer random numbers from there.

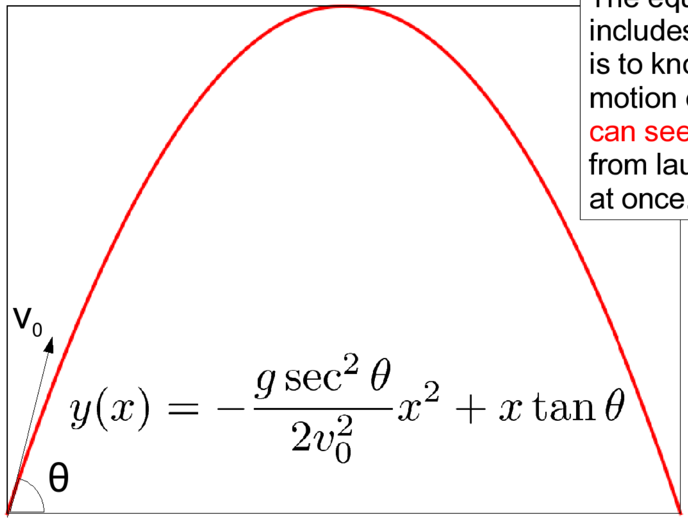
## Part 4: Solving Differential Equations Numerically

Last week, in the homework problem involving a projectile with air resistance, we solved a differential equation numerically, using a technique called Euler's method. Let's look at another way of solving these equations.

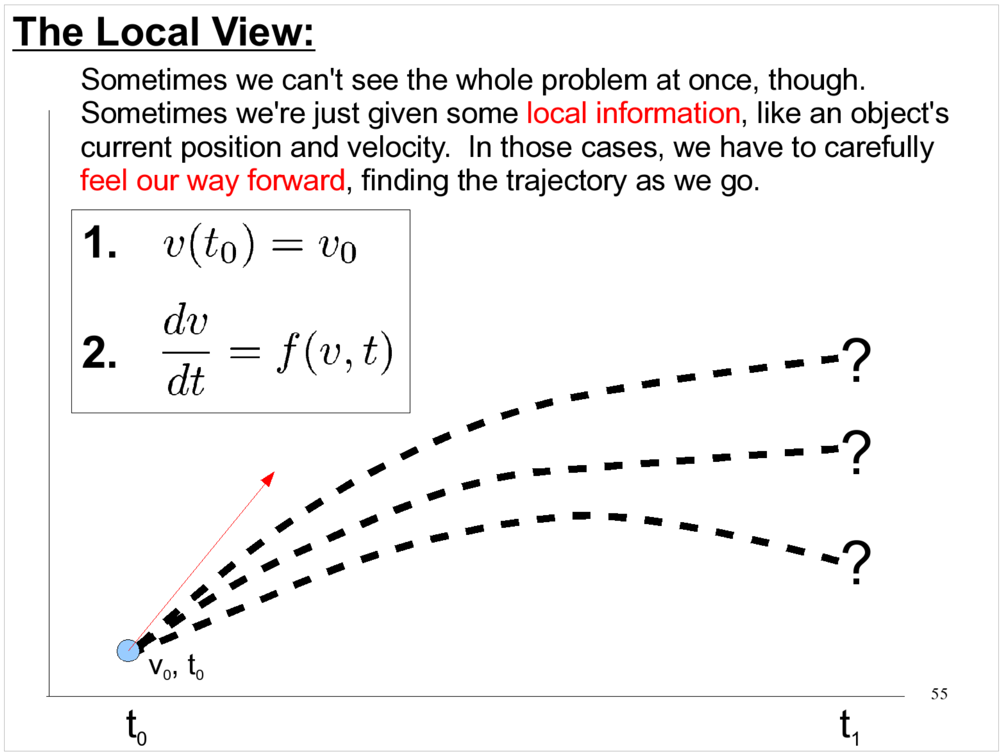
## The Global View:

Sometimes we can look at a whole problem from start to finish. Say, for example, that we're given the equation for the trajectory of a projectile with no air resistance. We can look at it and determine the position of the projectile at any point along its path.

The equation of motion includes everything there is to know about the motion of this object. We can see its whole history, from launch to landing, all at once.



$$y(x) = -\frac{g\sec^2\theta}{2v_0^2}x^2 + x\tan\theta$$

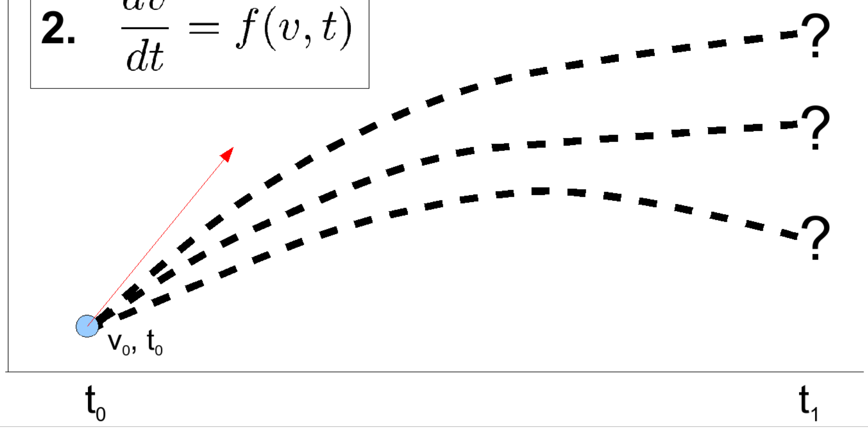


## The Local View:

Sometimes we can't see the whole problem at once, though. Sometimes we're just given some local information, like an object's current position and velocity. In those cases, we have to carefully feel our way forward, finding the trajectory as we go.

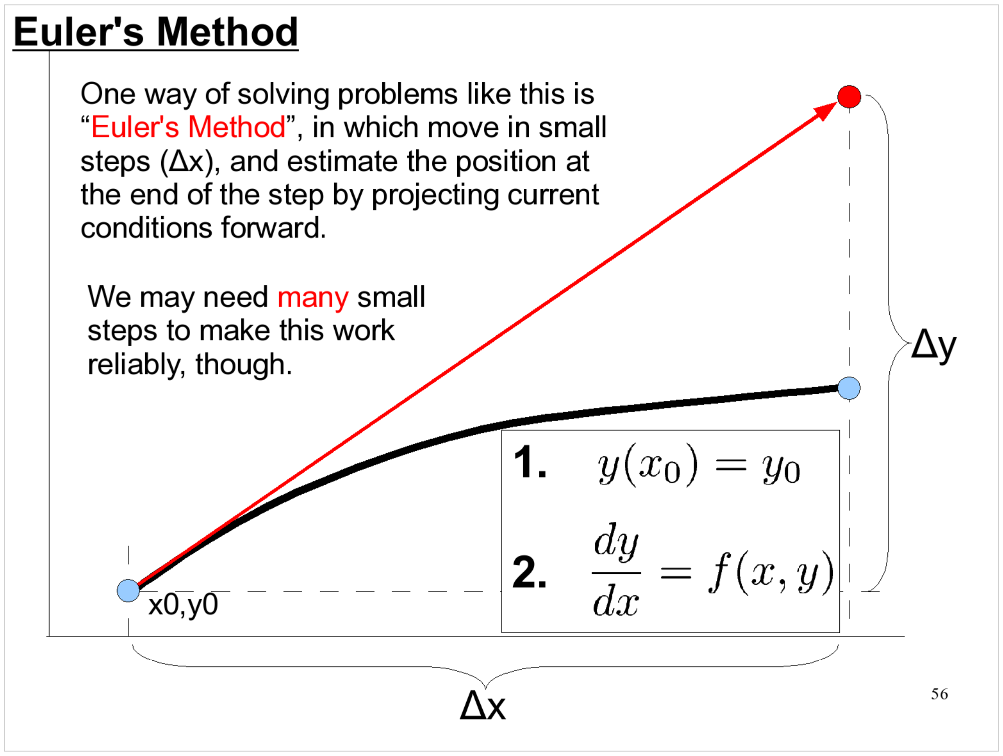
1. $v(t_0) = v_0$

2. $\dfrac{dv}{dt} = f(v, t)$



55

Consider the problem above, involving velocity and time. We aren't explicitly given the relationship between velocity and time, but we're told that dv/dt is some function that we can evaluate, and we're told an initial condition. We want to find v at a later time value.



## Euler's Method

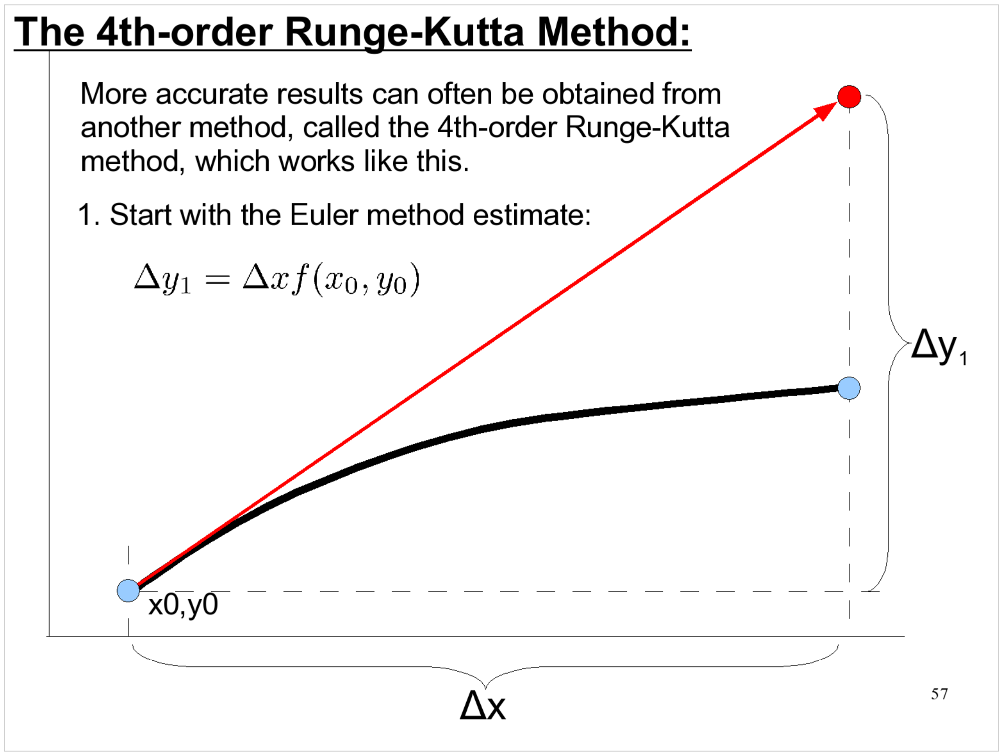
One way of solving problems like this is "Euler's Method", in which move in small steps (Δx), and estimate the position at the end of the step by projecting current conditions forward.

We may need many small steps to make this work reliably, though.

1. $y(x_0) = y_0$

2. $\frac{dy}{dx} = f(x, y)$

As you can see, the estimate in this case isn't very close to the true value at all. Maybe if we divided the x distance up into a hundred steps (instead of one) we'd get a better value. But that would involve a lot of computations, and take a lot of time.



## The 4th-order Runge-Kutta Method:

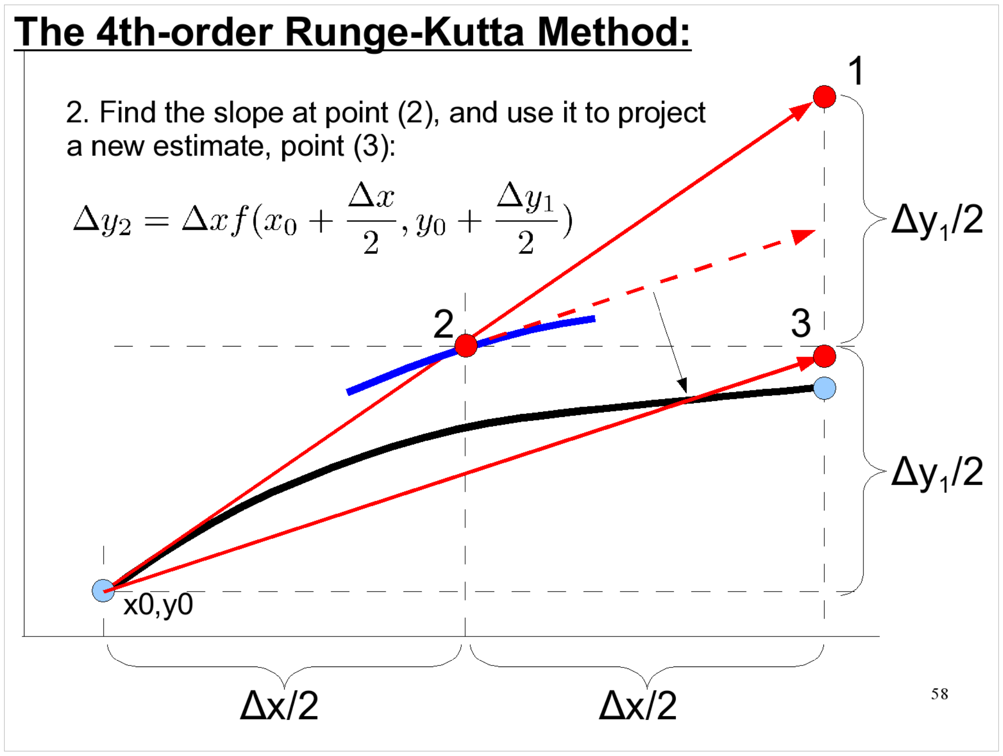
More accurate results can often be obtained from another method, called the 4th-order Runge-Kutta method, which works like this.

1. Start with the Euler method estimate:

$$\Delta y_1 = \Delta x f(x_0, y_0)$$

There's a whole family of Runge-Kutta methods, but generally when people say “the Runge-Kutta method” they mean the 4th order one. It gives good results, and higher-order versions are significantly more difficult to calculate.

As we'll see in the pages that follow, the Runge-Kutta method gives results that are much more accurate than we'd get using a similar number of computations with the Euler method.



## The 4th-order Runge-Kutta Method:

2. Find the slope at point (2), and use it to project a new estimate, point (3):

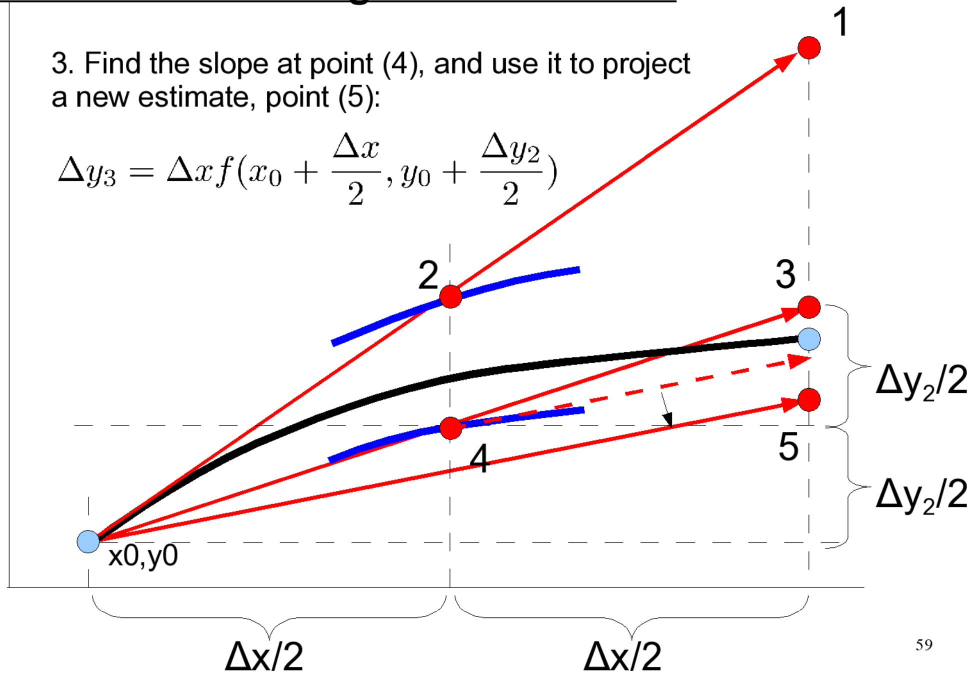
$$\Delta y_2 = \Delta x f(x_0 + \frac{\Delta x}{2}, y_0 + \frac{\Delta y_1}{2})$$

Note that we find the slope by just evaluating our function, f(x,y) at this point.

## The 4th-order Runge-Kutta Method:

3. Find the slope at point (4), and use it to project a new estimate, point (5):

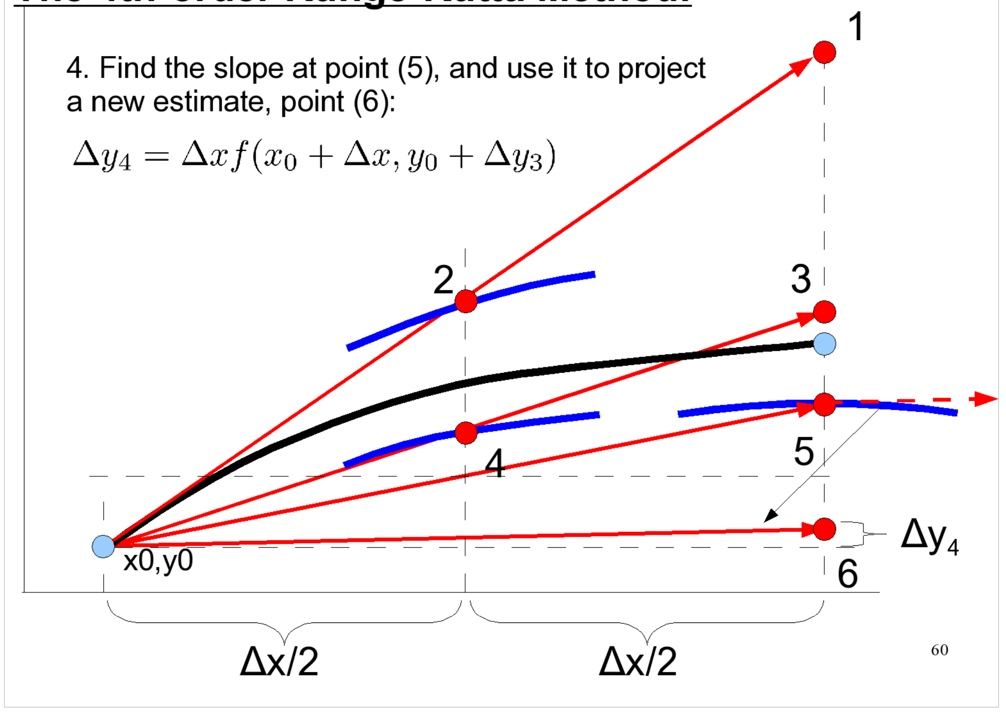
$$\Delta y_3 = \Delta x f(x_0 + \frac{\Delta x}{2}, y_0 + \frac{\Delta y_2}{2})$$



## The 4th-order Runge-Kutta Method:

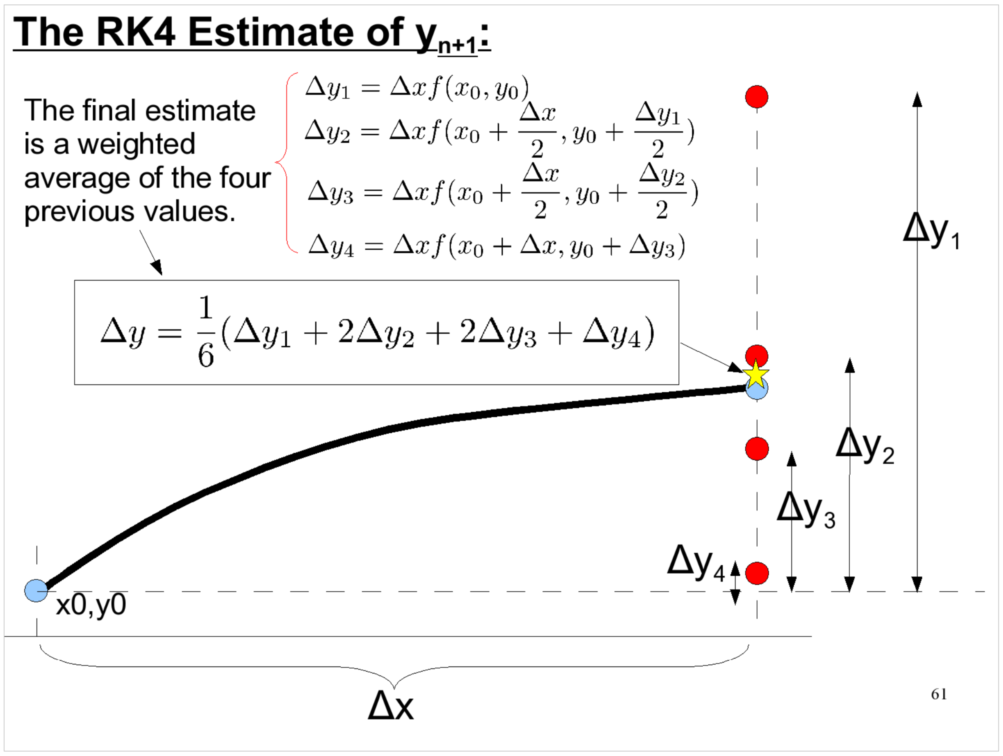
4. Find the slope at point (5), and use it to project a new estimate, point (6):

$$\Delta y_4 = \Delta x f(x_0 + \Delta x, y_0 + \Delta y_3)$$



We're now using slopes obtained from four different points: one point at each end of the trajectory (the original position and point 5), and two points in the middle (points 2 and 4).

Using these slopes, we've found four estimates of the final position: points 1, 3, 5 and 6.



## The RK4 Estimate of $y_{n+1}$:

The final estimate is a weighted average of the four previous values.

$$\Delta y_1 = \Delta x f(x_0, y_0)$$
$$\Delta y_2 = \Delta x f(x_0 + \frac{\Delta x}{2}, y_0 + \frac{\Delta y_1}{2})$$
$$\Delta y_3 = \Delta x f(x_0 + \frac{\Delta x}{2}, y_0 + \frac{\Delta y_2}{2})$$
$$\Delta y_4 = \Delta x f(x_0 + \Delta x, y_0 + \Delta y_3)$$

$$\boxed{\Delta y = \frac{1}{6}(\Delta y_1 + 2\Delta y_2 + 2\Delta y_3 + \Delta y_4)}$$

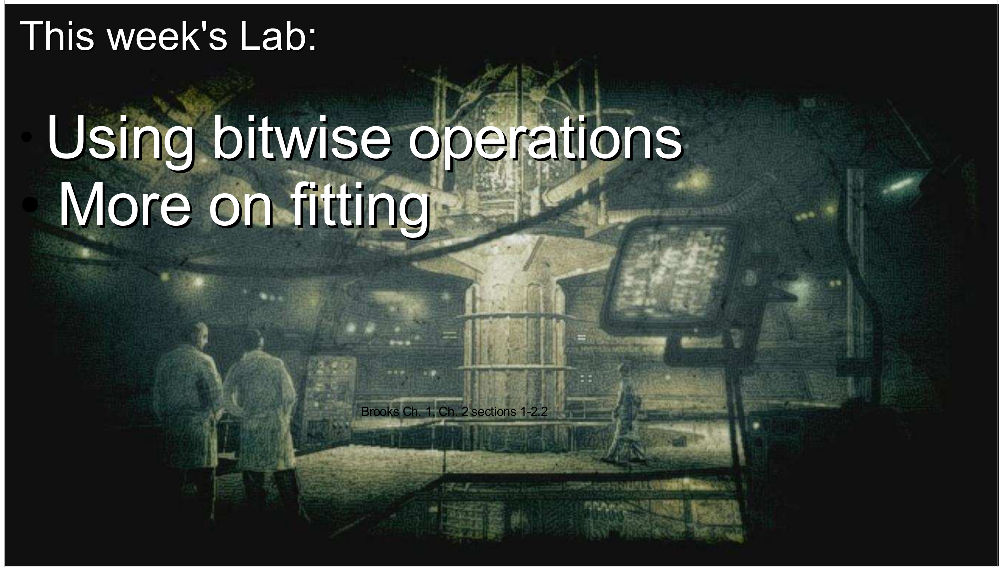
The final weighted average is a pretty good approximation to the true value. Note that it only required a few calculations to get it. Compare this to the possibly hundreds of steps that would be reqired to get a simlar accuracy from the Euler method, and you'll see the appeal of the RK4 method.

**Next Time:**

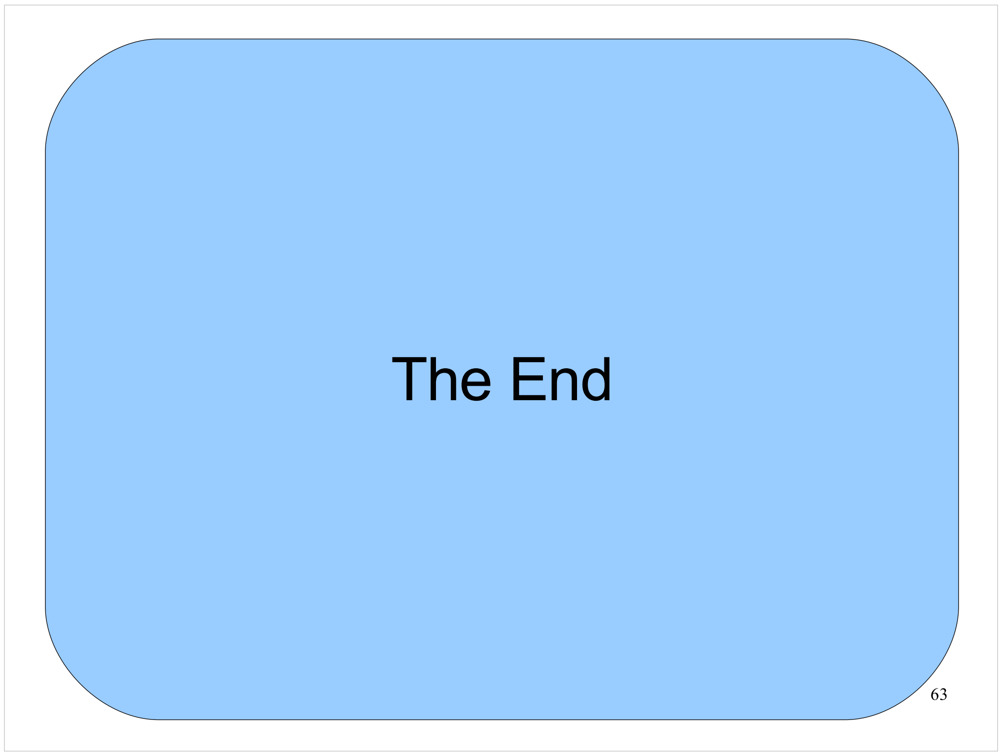
- More on bitwise operations
- Dynamic memory allocation

This week's Lab:

- Using bitwise operations
- More on fitting



Brooks Ch. 1, Ch. 2 sections 1-2.2



The End

63

Thanks!