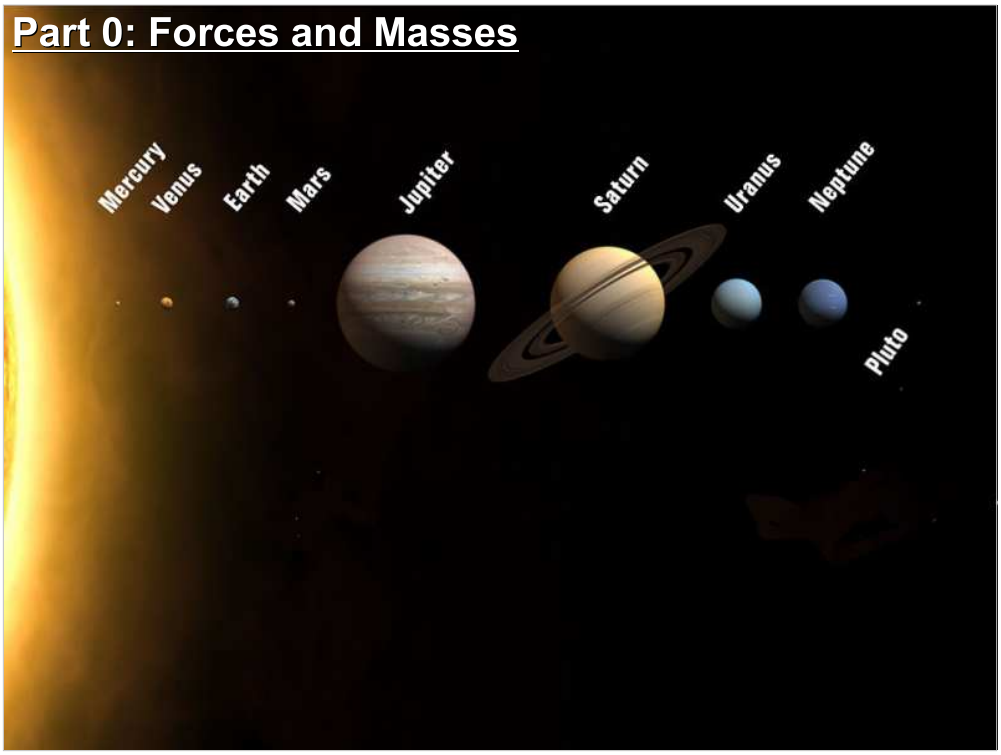
Physics 2660

Lecture 8

Today

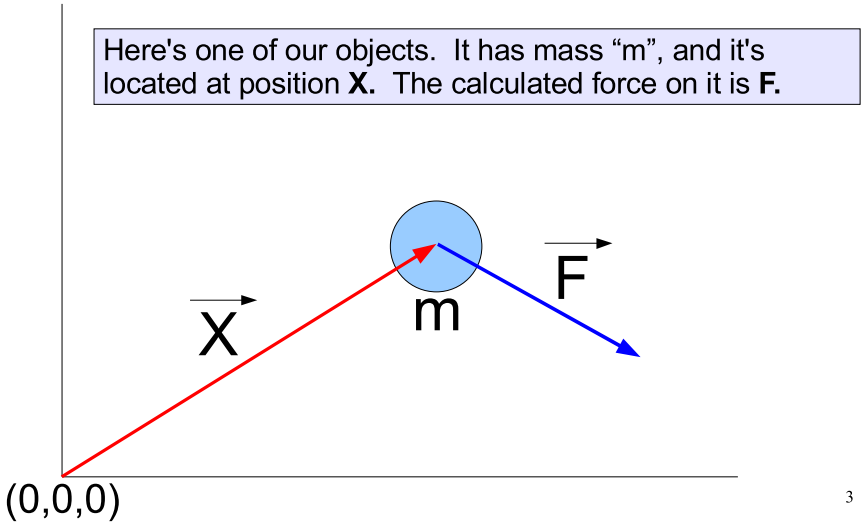
- A taste of C++, functions in structures -> classes
- Searching & Sorting
- Interpreting experimental uncertainties
- Combining / propagating uncertainties in experiments



Let's start out today by talking about the gravity problems you've been working on in the last couple of homework sets.

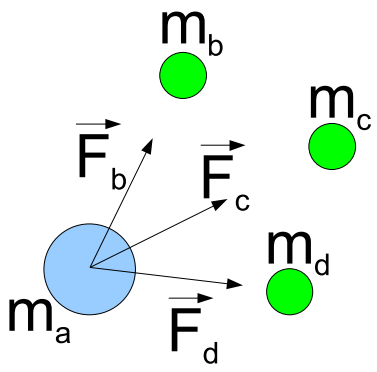
The Problem:

We want to read in the position vectors, initial velocities, and masses of a bunch of objects. Then, using this data, we want to calculate the gravitational force on each object, due to the others. (For the first part, we'll ignore the initial velocities.)



Adding the Forces:

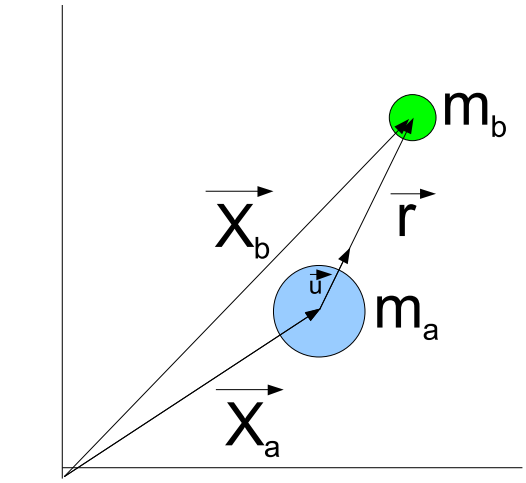
To find the total force on one mass, we just add the force vectors due to each of the other forces.



$$\vec{F} = \vec{F}_b + \vec{F}_c + \vec{F}_d$$

Finding Distance and Direction:

We'll need to know the distance and direction to each other object.



This is the vector from a to b:

$$\vec{r} = \vec{X}_b - \vec{X}_a$$

The magnitude of this vector gives us the distance:

$$r = |\vec{r}|$$

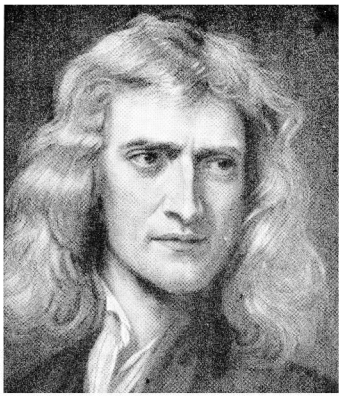
Once we know these, we can make a unit vector pointing from a to b:

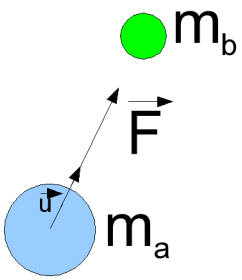
$$\vec{u} = \vec{r} / r$$

Calculating a Single Force:

Newton tells us that the magnitude of the gravitational force between two objects is:

$$F = G\frac{m_a m_b}{r^2}$$





The force will point toward the other object, so the force vector will just be:

$$\vec{F} = F\,\hat{u}$$

Data Structure:

To solve this problem programmatically, we'll first need a data structure to store information about each body:

```
typedef struct{
  double s_vec[3];  // space(position) vector
  double v_vec[3];  // velocity vector
  double f_vec[3];  // force vector
  double mass;
} body;

const int MAX_BODIES = 100;
body bodies[MAX_BODIES];  // array of bodies
```

Reading Data from a File:

```
int read_data(char* file, body *bodies){
  int num=0; // number of entries read from file
  int status;
  FILE *file_p = fopen(file,"r");

  while(num<MAX_BODIES) {
    status=fscanf(file_p,"%lf %lf %lf %lf %lf %lf %lf",
                  &bodies[num].s_vec[0],
                  &bodies[num].s_vec[1],
                  &bodies[num].s_vec[2],
                  &bodies[num].v_vec[0],
                  &bodies[num].v_vec[1],
                  &bodies[num].v_vec[2],
                  &bodies[num].mass);
    if (status==EOF) break;
    num++;
  }
  return num;
}
```

Position

Velocity

Mass

8

Here's an example of a function that can read data from a file and fill the data structure on the previous page. Notice that it takes a file name as an argument, then just opens the file and reads the contents. (If we were better programmers, we'd also take the trouble to close the file when we were done with it.)

Notice that the function will only read up to MAX_BODIES bodies, since this is the size of our array. If there are more bodies than this in the file, the remaining ones will be ignored.

If the function gets to the end of the file before it reaches MAX_BODIES, it stops. The function returns “num”, the number of bodies actually read.

Some Useful Functions:

```
// Find distance between two points:
double distance(double *svec1, double *svec2){
  double dist2=0;
  int i;
  for (i=0; i<3; i++)
    dist2 += (svec1[i]-svec2[i])*
             (svec1[i]-svec2[i]);
  return sqrt(dist2);
}
// Find difference of two vectors:
void vsub(double *v1, double *v2, double *v1m2){
  int i;
  for (i=0; i<3; i++)
    v1m2[i] = v1[i]-v2[i];
}
```

$(x_1-x_2)^2 + (y_1-y_2)^2 + (z_1-z_2)^2$

diff = $[(x_1-x_2), (y_1-y_2), (z_1-z_2)]$

The “distance” function just calculates the distance between two points in three-dimensional space. You can think of this as the magnitude of the vector “r” pointing from one body to another.

The “vsub” function subtracts one vector from another, to produce a third vector. You can think of this as the “r” vector itself.

Calculating the Forces:

```
void forces(body *bodies, int nbodies){
  double dist, force;
  double dirvec[3];
  const double G = 6.67e-11;

  for(int i=0; i<nbodies; i++){
    bodies[i].f_vec[0]=0;
    bodies[i].f_vec[1]=0;
    bodies[i].f_vec[2]=0;
    for(int j=0; j<nbodies; j++){
       if ( i!=j ) {
        dist = distance(bodies[i].s_vec,bodies[j].s_vec);
        vsub(bodies[j].s_vec,bodies[i].s_vec,dirvec);
        dirvec[0] /= dist;
        dirvec[1] /= dist;
        dirvec[2] /= dist;
        force = G*bodies[i].mass*bodies[j].mass/(dist*dist);
        for(int k=0; k<3; k++) {
          bodies[i].f_vec[k] += force*dirvec[k];
       }
      }
    }
  }
}
```

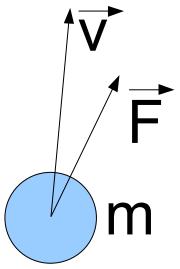

Here's a function that calculates the total force on each object, due to all the other objects. Notice the if statement “(i!=j)” that omits the object itself from the calculation.

The function uses the “distance” and “vsub” functions we saw in the previous slide.

The vector “dirvec” is the unit vector pointing from one mass to the other. We get it by dividing the “r” vector by its length.

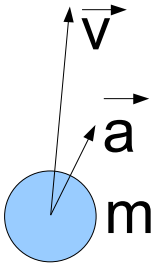
Finally, note that, although this function has a double loop that goes through all the objects, we could really get by with only half as many iterations, since the mutual forces on each pair of objects are equal and opposite. If we were clever, we could take advantage of this and write the function so that we just calculate each pair of forces once.

Trajectories:



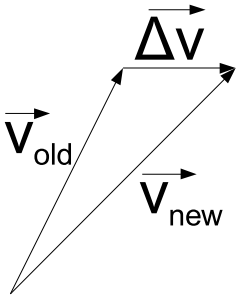
$$\vec{F} = m\,\vec{a}$$

$$\vec{a} = \vec{F} / m$$



$$\Delta\vec{V} = \vec{a}\,\Delta t$$

$$\vec{V}_{new} = \vec{V}_{old} + \Delta\vec{V}$$



11

That takes care of most of the static stuff. Now to set things in motion...

First, we'll need to think about the velocities and accelerations of the objects. Since we know the forces now, we can calculate the accelerations.

Once we know the accelerations, we can calculate the change in velocity after a time step of delta t.

Calculating Trajectories:

Here's one simple way to approximate the motion of the objects.
Here, we assume a constant velocity during each time step:

```
void evolve(body *bodies, int nbodies, double delta_t){
  for (int i=0; i<nbodies; i++) {
    for (int j=0; j<3; j++){

      double acceleration_j =
             bodies[i].f_vec[j] / bodies[i].mass;

      bodies[i].s_vec[j] += bodies[i].v_vec[j]*delta_t;

      bodies[i].v_vec[j] += acceleration_j * delta_t;
    }
  }
}
```

$a_j = F_j/m$

$x_j^{new} = x_j + v_j \Delta t$

$v_j^{new} = v_j + a_j \Delta t$

x,y,z

A Better Approximation:

Here's a better approximation. In this version, we only assume a constant acceleration during each time step:

```
void evolve(body *bodies, int nbodies, double delta_t){
  for (int i=0; i<nbodies; i++) {
    for (int j=0; j<3; j++){

      double acceleration_j =
        bodies[i].f_vec[j] / bodies[i].mass;

      bodies[i].s_vec[j] +=
    (bodies[i].v_vec[j]*delta_t+0.5*acceleration_j*delta_t*delta_t);

      bodies[i].v_vec[j] += acceleration_j * delta_t;
    }
  }
}
```

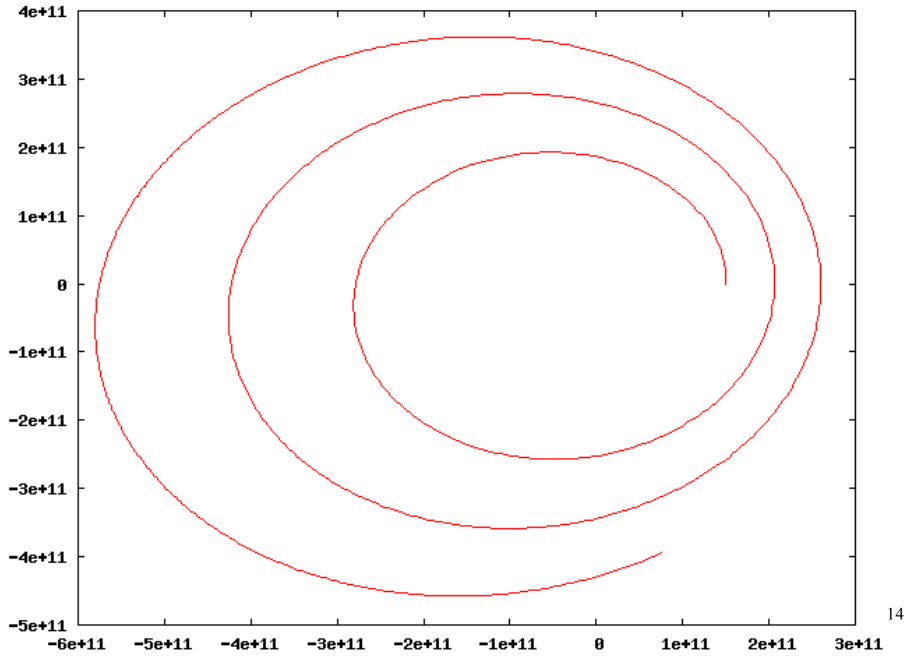
$a_j = F_j/m$

$x_j^{new} = x_j + v_j \Delta t + \frac{1}{2} a_j \Delta t^2$

$v_j^{new} = v_j + a_j \Delta t$

x,y,z

Orbits:



14

A good way to check the sanity of your program is to write out x,y,z values for one of the bodies into a file, and then plot them with gnuplot. (The output file should just have three columns of numbers.)

Seen above is the orbit of an object in the x-y plane. If you look at a plot like this and see something obviously non-physical (sharp corners, straight lines, etc.) then there's probably something wrong with your program.

Once you have a data file like this, you can plot it in gnuplot by just typing a command like:

plot “file.dat” with lines

Part 1: Introduction to C++ Classes

We've dealt a lot with structures now. Structures are a tried-and-true feature of the C programming language.

C++ offers an additional construct called a “class”, which is like a souped-up structure. Let's take a look at how it's used.

A Review of Structures:

In C, we can define structures that pack a lot of related information into a single variable. We can then create functions that operate on our newly-defined variable types.

```
typedef struct{
  double re, im;
} Complex;

double magnitude(Complex z) {
  return sqrt( z.re*z.re + z.im*z.im );
}

void conjugate(Complex *z) {
  z->im = -z->im;
}
```

Classes in C++:

```
class Complex{
public:
  double re,im;
  double magnitude() const {
   return sqrt(re*re+im*im);
   }
  void conjugate() {
   im = -im;
   }
};

int main () {
  Complex c;
  c.re = 1.0;
  c.im = 2.0;

  printf ("Magnitude is %lf\n", c.magnitude() );
  c.conjugate();
  printf ("Im(c) = %lf\n", c.im );
}
```

Variables in this class.

Functions (called "methods") can be included in the definition of a class.

An instance of a class is called an "object".

17

Like structures, classes can contain variables, but they can also contain dedicated functions, called "methods" that operate on the class's data.

We define new variables of a particular class just as we'd define a variable using structs: just type the name of the class followed by the variable name.

Each instance of the class is called an "object". The idea is that objects are sort of "smart variables" that are able to do things on their own and interact with each other.

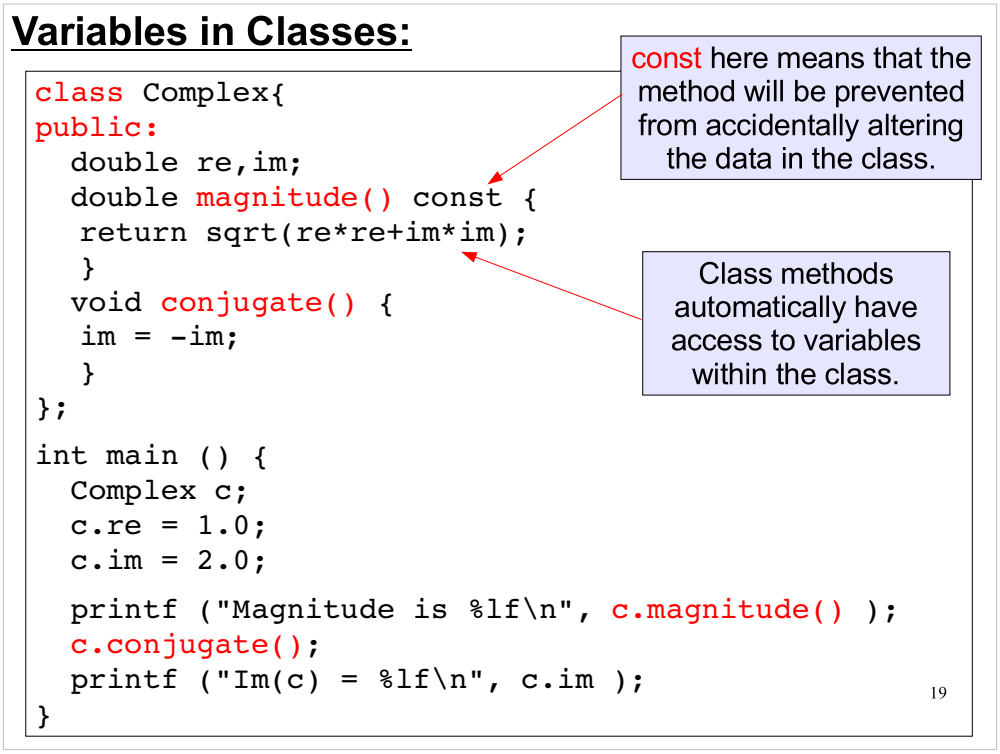
Invoking Methods:

```
class Complex{
public:
  double re,im;
  double magnitude() const {
   return sqrt(re*re+im*im);
   }
  void conjugate() {
   im = -im;
   }
};

int main () {
  Complex c;
  c.re = 1.0;
  c.im = 2.0;

  printf ("Magnitude is %lf\n", c.magnitude() );
  c.conjugate();
  printf ("Im(c) = %lf\n", c.im );
}
```

Class methods are invoked like this.

The methods of a class are invoked in much the same way we'd use the elements of a struct.

Variables in Classes:



The methods in a class don't need to use “.” or “->” to get to the variables within the class. They just use the variable name.

Scope of Variables and Methods:

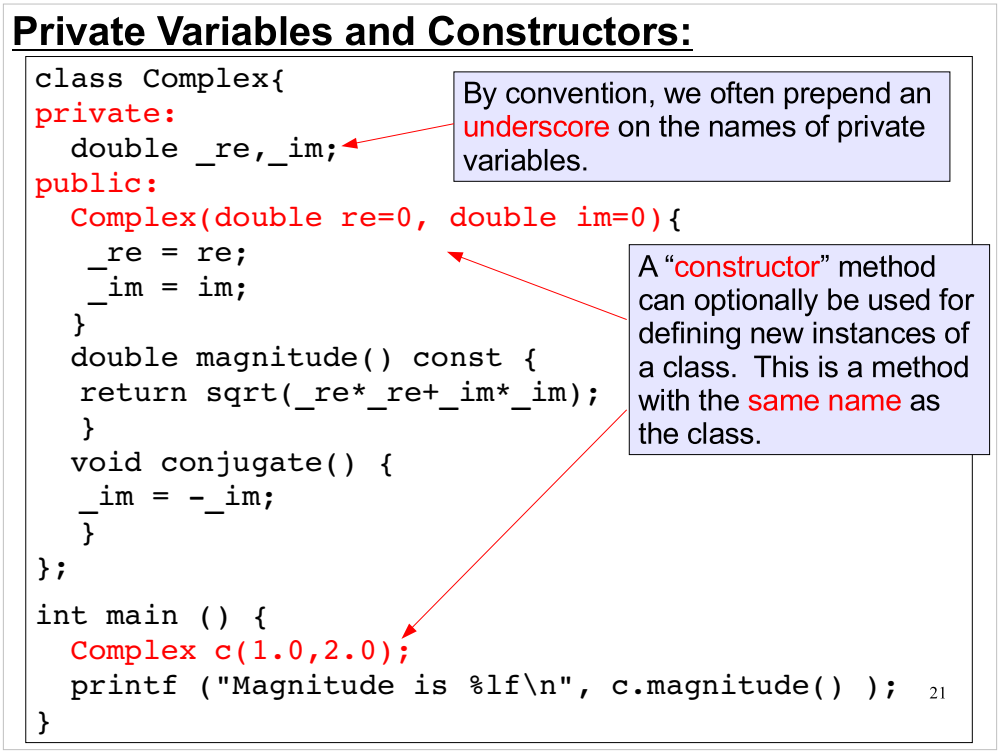
```
class Complex{
public:
  double re,im;
  double magnitude() const {
   return sqrt(re*re+im*im);
   }
  void conjugate() {
   im = -im;
   }
};

int main () {
  Complex c;
  c.re = 1.0;
  c.im = 2.0;

  printf ("Magnitude is %lf\n", c.magnitude() );
  c.conjugate();
  printf ("Im(c) = %lf\n", c.im );
}
```

Methods and variables within a class can either be "public" or "private". Private components can only be accessed by methods within the class. Default is "private".

Methods and variable that are "public" can be used outside the class (in "main", for example). In the slide above, everything in the class is set to "public".

What happens if we make some things private?

Private Variables and Constructors:



In this example, we've made _re and _im "private" variables. This means that parts of our program outside the class can't get to these variables. They're only available inside the class.

Constructor methods can be arbitrarily complicated. We can have the constructor do anything we want to initialize our variables. We might, for example, have a histogram class with a constructor that automatically initializes to zero all of the bins of a new histogram.

Accessing and Setting Private Variables:

```
class Complex{
private:
  double _re,_im;
public:
  Complex(double re=0, double im=0){
    _re = re;
    _im = im;
  }
  double re() const { return(_re); }
  double im() const { return(_im); }
  void re( double rval ) { _re = rval; }
  void im( double ival ) { _im = ival; }
};
int main () {
  Complex c(1.0,2.0);
  printf ("Re(c) = %lf\n", c.re() );
  c.re(3.0);
  printf ("Re(c) = %lf\n", c.re() );
}
```

Accessor and setter methods:

Example of function overloading in C++.

In the example above, we have a problem if we make _re and _im private. Now we can no longer set them directly in "main" or elsewhere.

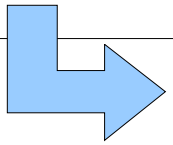
We need "setter" and "accessor" methods to get and set these values for us.

But wait. We have two methods called "re" and two called "im". What's up with that? In C++, you can have two (or more) functions with the same name, as long as they have a different calling syntax. In the case above, one "re" function takes no arguments and returns a double. The other "re" function returns nothing, but takes a double as an argument.

C++ figures out which function to call based on how you use it in your program. This is called "function overloading".

A “print” Method:

```
class Complex{
...
public:
  double magnitude () {
   return( sqrt(_re*_re+_im*_im) );
  }
  void print () {
   printf( “%lf+%lfi, mag=%lf\n”, _re, _im,
           magnitude() );
  }
...
};
int main () {
  Complex c(1.0,2.0);
  c.print();
}
```



```
1.0+2.0i, mag=2.24
```


It's often useful to have a “print” method, that knows how to print out your variables.

An Object-Oriented Histogram Library:

The new version of our histogram library supports object-oriented syntax:

```
int main(){
  h1 hist1(100,0.,100.,
           "Uniform,weighted w/ X");
  hist1.labels("X-value","# of entries");
  h1 hist2(150,0.,1000.,"Normal, weights=1");
  hist2.labels("X-value","# of entries");
  for (int i=0; i<10000; i++) {
    double tmp = randu(0.,100.);
    hist1.fill(tmp,tmp);
    hist2.fill(randn(600.,100.),1.0);
  }
  hist1.errors(true);
  hist1.plot();
}
```


Feel free to use this, or not.

Object-Oriented Programming:

Until now, we've been concentrating on a programming style called "procedural programming", in which we pass data from function to function as we step through the jobs our program needs to do

In C++ one often follows an object-oriented design model, in which classes are designed to encapsulate both data and the operations that are used with those data

Object-oriented programming was once the dominant programming model, but has recently fallen into some disfavor. At Carnegie-Mellon University, for example, object-oriented programming has now been entirely eliminated from the introductory CS curriculum.

You won't be required to program in this model, but you should be aware of the syntax, just in case you see similar usage in a problem solution.

Part 2: Searching

We've been creating a lot of arrays lately. What happens when we want to search for a particular element in an array?

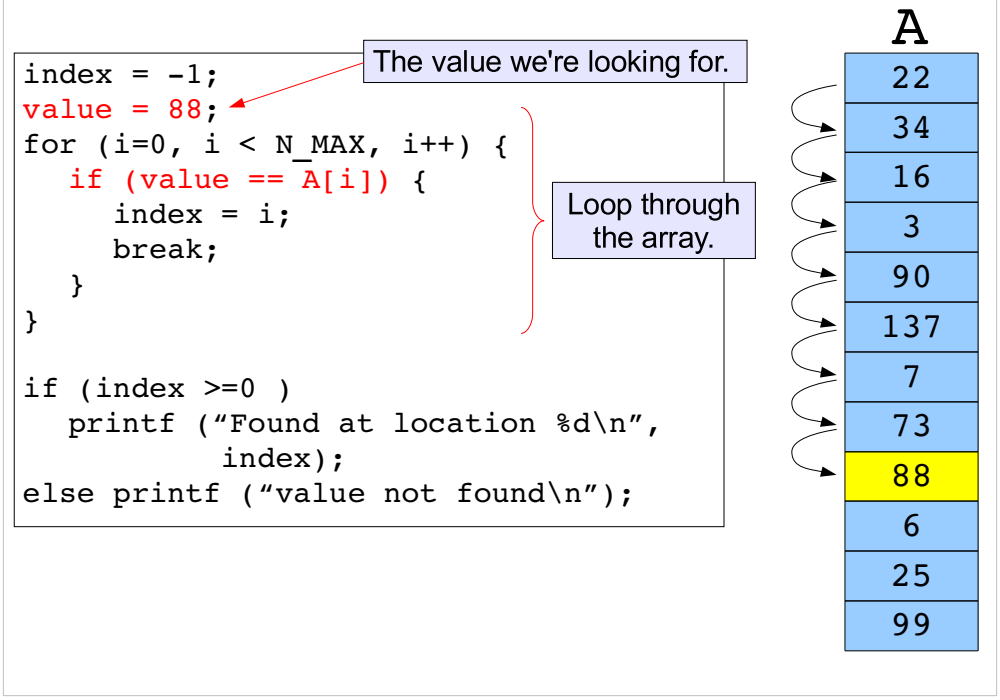
Search Methods:

We'll discuss two simple approaches to searching for a particular value within a collection of data:

1) a linear search
In a linear search, we star at the beginning of an array and move down the line of elements looking for matches.

2) a binary search
If we have an ordered list of data (either ascending or descending), this method provides a MUCH faster search for a particular value. Binary searches are sometimes called “bisection”.

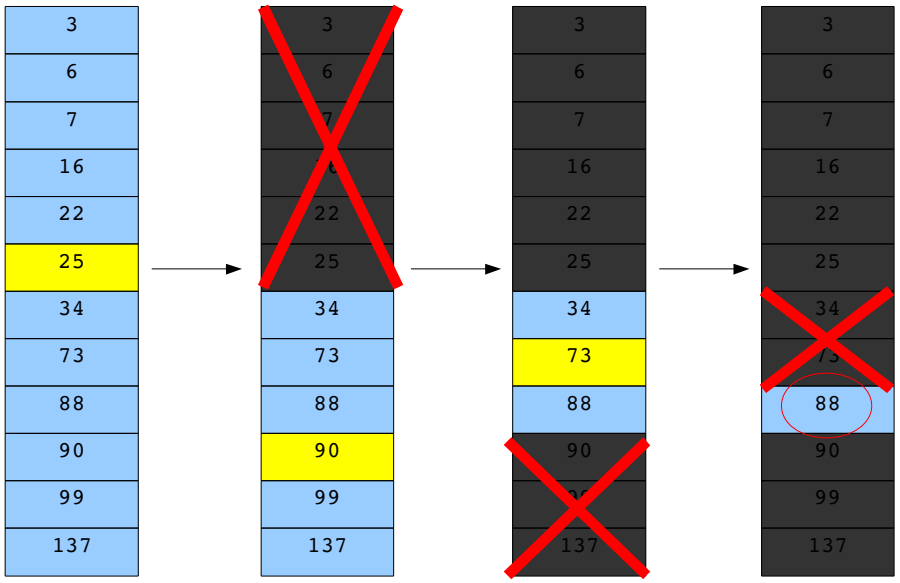
Linear Search:



A linear search just starts at the top of the array and works its way down until it finds what it's looking for.

Binary Search:

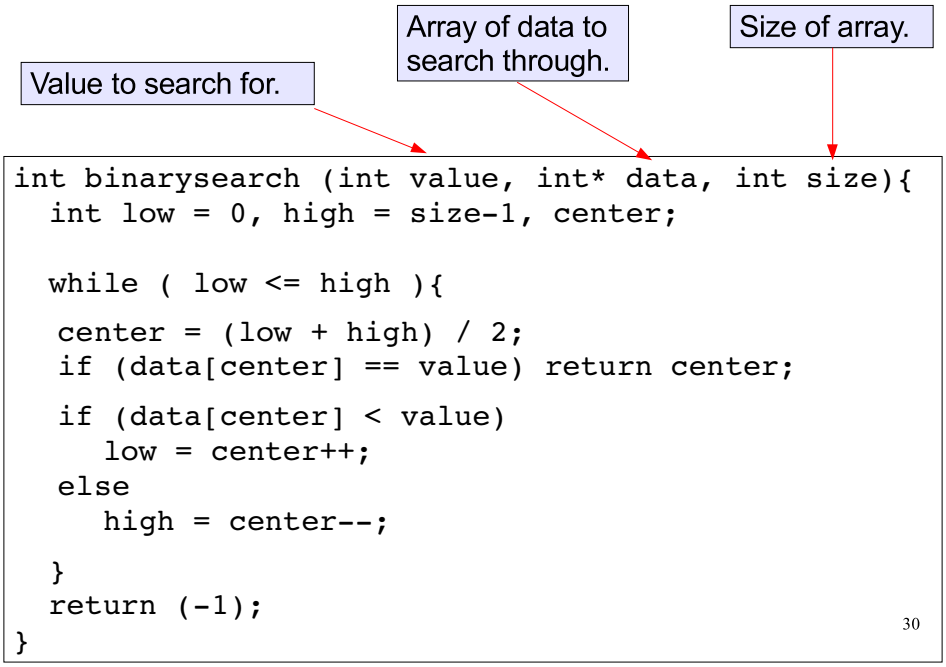
With a pre-sorted list, we can use a faster binary search. Start by picking a number in the middle of the array, then continue breaking the list in half each time:



Note that this only took three steps. As we'll see, binary searches can be very fast, even for large arrays.

Remember that binary searches only work for sorted lists, though.

A Binary Search Function:



Here's one way to write a simple binary search function. Let's step through how it works.

A Binary Search Function:

First, pick an index approximately in the middle of the current range:

```
int binarysearch (int value, int* data, int size){
  int low = 0, high = size-1, center;

  while ( low <= high ){
   center = (low + high) / 2;
   if (data[center] == value) return center;

   if (data[center] < value)
      low = center++;
   else
      high = center--;

  }
  return (-1);
}
```

Pick ~center of current range.

A Binary Search Function:

Maybe we get lucky, and the number we're looking for will be at the index we picked. If not, look at whether the number that's there is higher or lower than the number we're looking for, and adjust the range accordingly.

```
int binarysearch (int value, int* data, int size){
  int low = 0, high = size-1, center;

  while ( low <= high ){
   center = (low + high) / 2;
   if (data[center] == value) return center;
   if (data[center] < value)
      low = center++;
   else
      high = center--;
  }
  return (-1);
}
```

Found it!

Too low. Raise lower limit.

Too high. Lower upper limit.

A Binary Search Function:

Keep doing this until we either find the number or exhaust all of the possible array elements. If we don't find the number anywhere in the array, return “-1” to indicate that we've failed.

```
int binarysearch (int value, int* data, int size){
  int low = 0, high = size-1, center;

  while ( low <= high ){
   center = (low + high) / 2;
   if (data[center] == value) return center;
   if (data[center] < value)
      low = center++;
   else
      high = center--;
  }
  return (-1);
}
```

Number not found.

Speed of a Linear Search:

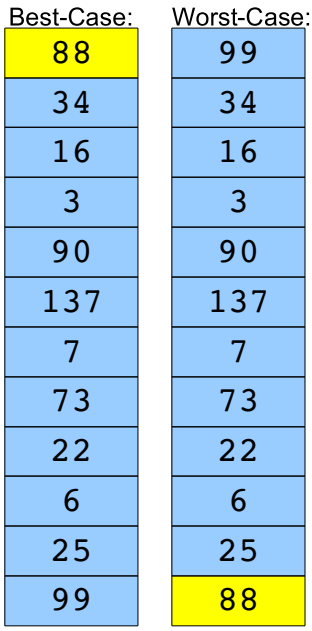
Let's assume we have an ordered array of N elements and we want to search for the location of a number that we know to be in the array.

How much work does this require?

For a linear search we make no distinction whether the number is high or low, we always start looking through the array from the start.

On average it takes us N/2 tries to find the number. Thus the work required is proportional to N. In Computer Science terms, this is what's called an O(N) algorithm.

If we double N, we double the work on average.



Speed of a Binary Search:

In the worst case the work done by the binary search is proportional to the number of times we can divide the array in half, before only one element remains.

if N = 128, we can cut the array in half only 7 times!
(128, 64, 32, 16, 8, 4, 2, 1 2^7 = 128)

If we double N, then we need only do 8 divisions instead of 7.
A relatively small increase in work.

The number of iterations required of a binary search algorithm is only proportional to $\log_2(N)$, where $\log_2(N)$ is the power to which you need to raise 2 to get N.

In Computer Science terms, this an $O(\log_2 N)$ algorithm.

This is very important when N is large. In that case, $\log_2(N) << N$. For example, if N = 4 billion, it would only take up to 32 steps to find any given number with a binary search. It could take up to 4 billion steps with a linear search.

Part 3: Sorting

So, we see that binary searches are fast, but they require pre-sorted data. How about routines for sorting?

Sorting Algorithms:

In order to take advantage of the binary search we need sorted data, this leads naturally to a discussion of sorting algorithms.

We'll consider two algorithms:

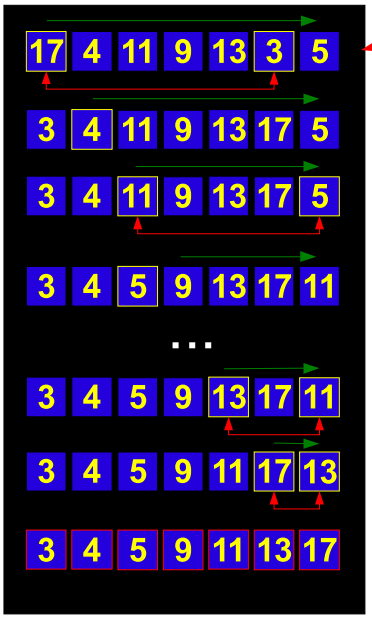
1) The very simple and intuitive Selection Sort

2) The clever Quicksort algorithm

There are many other sorting algorithms, some optimized for particular data set characteristics.

You may never need to do anything more than choose between a slow but simple sort and some kind of optimized sort in your programs. But this is a rich topic to explore if sorting times become an important limiting factor in your work.

Selection Sort:



Scan forward from position (1) swap smallest number into (1)

Scan from (2), swap smallest number into (2)

Scan from (3), swap smallest number into (3)

.

.

.

Scan from (N-1), swap smallest number into (N-1)

All sorted.

Implementing a Selection Sort:

As you can see, a selection sort is really easy to write:

```
const int num=7;
int a[num] = {17, 4, 11, 9, 13, 3, 5};

for (i=0; i<num ; i++) {
   for (j=i+1; j<num; j++)
      if (a[j] < a[i])
         swap (&a[j], &a[i]);
}
...
void swap (int *i, int *j) {
   int tmp;
   tmp = *i;
   *i = *j;
   *j = tmp;
}
```

For each element starting at the beginning...

Search through remaining elements i+1 to num-1

If we find a smaller element, swap the two

"swap" function

The Problem with Selection Sorts:

The problem with the selection sort algorithm is its pair of nested loops.

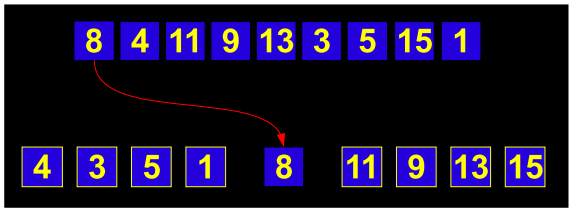
```
const int num=7;
int a[num] = {17, 4, 11, 9, 13, 3, 5};

for (i=0; i<num ; i++) {
   for (j=i+1; j<num; j++)
      if (a[j] < a[i])
         swap (&a[j], &a[i]);
}
```

The time to go through a single loop is proportional to N (or "O(N)"). Here we have a loop within a loop, so the time is proportional to N*N.

Computer scientists would say that this is an $O(N^2)$ algorithm, making it very slow for large values of N.

The Quicksort Algorithm:

A better sorting algorithm is the one called “Quicksort”. It works like this:



1. Start with a value, say the first one.

2. Split the list into elements less than the value and elements greater than the value.

3. Reapply this procedure to each of the two “satellite” lists.

4. Repeat until all lists have one element left.

After each step, we're scanning lists of half the original size.
This translates into a huge reduction in the work needed to sort the list.

A Quicksort is a $O(N\log_2 N)$ algorithm.

For $N=10^6$, compare: $N^2 = 10^{12}$, $N\log_2 N \sim 2\times10^7$, about 50,000 times less.

So, how do we write a program to do a quicksort?

The “qsort” Function:

The “qsort” function in the standard C library implements a Quicksort:

man qsort

```
QSORT(3)              Linux Programmer's Manual              QSORT(3)

NAME
       qsort - sorts an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int(*compar)(const void *, const void *));

DESCRIPTION
       The  qsort()  function sorts an array with nmemb elements of size size.
       The base argument points to the start of the array.

       The contents of the array are sorted in ascending order according to  a
       comparison  function  pointed  to  by  compar, which is called with two
       arguments that point to the objects being compared.

       The comparison function must return an integer less than, equal to,  or
       greater  than  zero  if  the first argument is considered to be respec-
       tively less than, equal to, or greater than the second.  If two members
       compare as equal, their order in the sorted array is undefined. ...
```


Fortunately, we don't have to write our own function.
The “qsort” function is in the standard C library.

qsort Syntax:

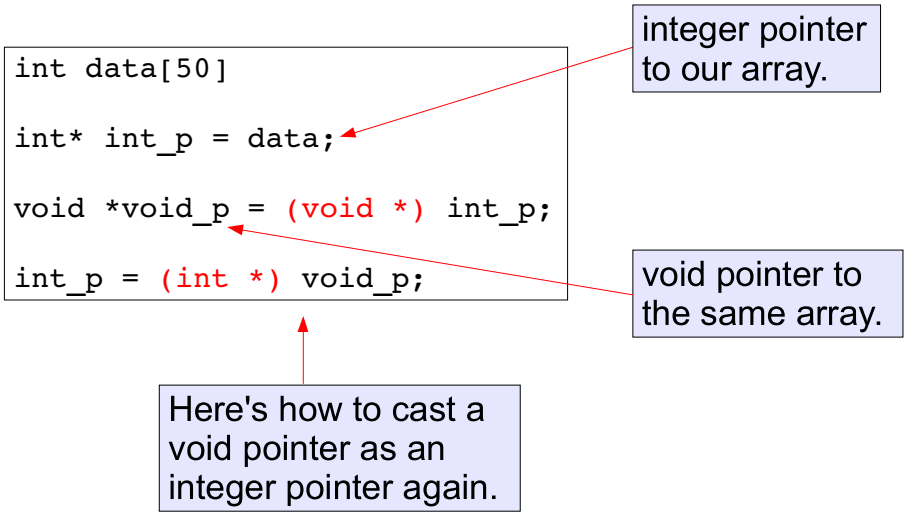
```
void qsort(void *base,
           size_t nmemb,
           size_t size,
           int(*compar)(const void *, const void *));
```

- `void *base` is a generic memory location. It's like a pointer without a specific data type. In this case, it points to the beginning of the array we want to sort.
- `size_t nmemb` is the number of elements in the array. For now, assume size_t is just the same as int.
- `size_t size` contains the size of each element of the array. Qsort can operate on any array (double, int or some complicated struct), so it needs to know how big each element of the array is.
- `int (*compar)(const void *, const void *)` This is a function pointer, pointing to a function that can compare two values to see which is "bigger". We can write this function any way we want, to suit our own definition of "bigger".

Qsort has a slightly complicated calling syntax. We'll look at some examples of how to use it soon.

Void * Pointers:

Here's an example showing how to convert between `void *` pointers and pointers of other types, using typecasts:



Since qsort uses void * pointers, we need to know how to work with them. The examples above show how to cast other types as void *, and vice versa.

Comparison Functions for Qsort:

Qsort comparison functions return "an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second."

```
int compare_int(void *a, void *b) {
  int x = *(int *)a;
  int y = *(int *)b;
  if (x > y) return 1;
  if (x < y) return -1;
  return 0;
}

int compare_float(void *a, void *b) {
  float x = *(float *)a;
  float y = *(float *)b;
  if (x > y) return 1;
  if (x < y) return -1;
  return 0;
}
```


We can write our comparison function any way we want. For example, we might have an array of histograms and want to sort them by the number of counts they contain. We could write a comparison function to do this.

Sorting Arrays with qsort:

Finally, here's an example showing how to use qsort to sort arrays:

```
int data[50];
float fdata[50];

qsort( (void *)data, 50, sizeof(int),
compare_int );

qsort( (void *)fdata, 50, sizeof(float),
compare_float );
```


Here are some real examples of qsort usage. Even though the syntax sounds complicated, it's not so bad when you actually start using it.

Creating a Sort-order List:

If you have an array of big data structures (e.g., histograms), it may take a lot of time to actually move them around in memory while sorting them. Usually, we really don't care how the items are arranged in memory, we just need to know which comes first, which comes second, etc..

In this case, we might just want to create a list of indices, sorted appropriately. The array indices are small (just ints), so it doesn't much time to move them arround in memory.

```
int order[MAX]
some_big_struct data[MAX]
```

After sorting the indexes, use `data[order[i]]` to retrieve elements in sorted order.

Part 4: Interpreting Experimental Uncertainties

When we do an experiment and report a result in the form “x +- sigma”, what does that really tell us?

Example: Neutron Decay:

Left to themselves, neutrons are unstable and decay into protons, electrons and neutrinos. The mean lifetime of a free neutron is predicted to be about 886 seconds (about 15 minutes).

$$n^0 \longrightarrow p^+ + e^- + \bar{\nu}_e$$

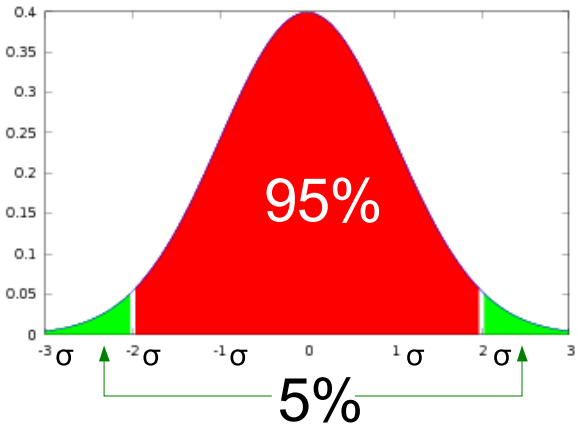
What if we do an experiment to measure the lifetime of a free neutron, and we come up with a result of 926 +/- 20 seconds.

Is our result consistent with the theoretical prediction?

Note that the theoretical result differs from our result by 2σ.

Probability of a 2σ Deviation:

When we cite a result like “x +/- σ”, we're saying that we believe that if our experiment were repeated many times, the results would be distributed like the graph below. In particular, we expect that only about 5% of the results will be more than 2σ from the mean.

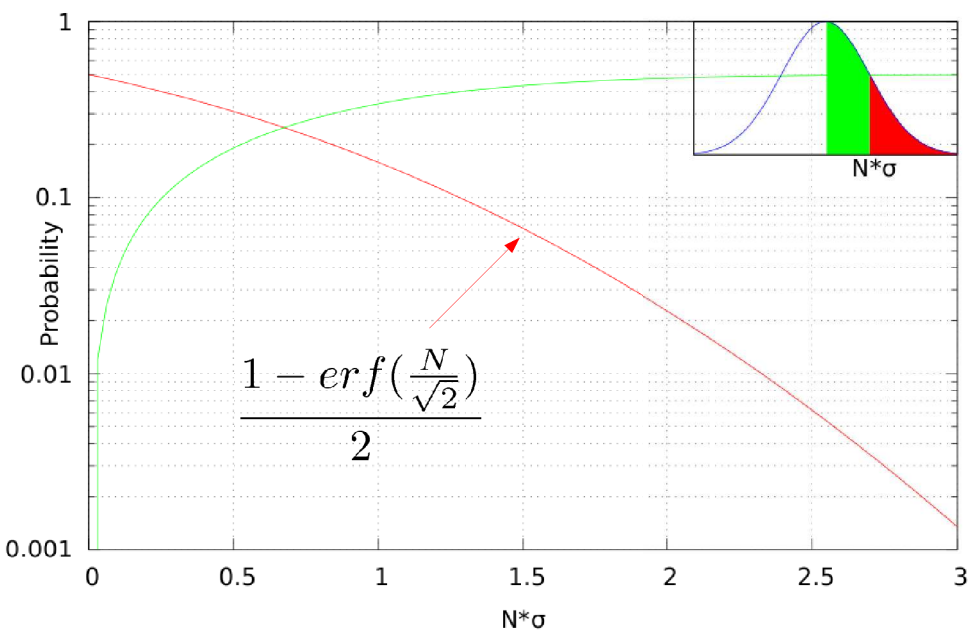


In our neutron decay example, our measured value differed from theory by 2σ.

If theory and experiment were both correct, we'd expect to see a deviation this large about 5% of the time. So, it isn't out of the question that both theory and our experiment are correct.

The agreement in this case isn't great, but it doesn't provide any compelling reason to throw out the theory.

Probability of a Given Deviation:



If we want to do this kind of comparison a lot, a graph like the one above might be useful. In red, it shows the total area between N*sigma and infinity. In green, it shows the area between zero and N*sigma.

Graphs like this can be generated using the “erf” function (the “error function”) which is related to the integral of the Gaussian function.

When Can We Trust Numerical Probabilities?

Only when:

• The quantity of interest has been correctly measured.
(no important systematic biases).

• Size of error has been correctly calculated.
Incorrect errors are disastrous for determining significance of experiment. Recall a 2σ deviation happens ~5% of the time.

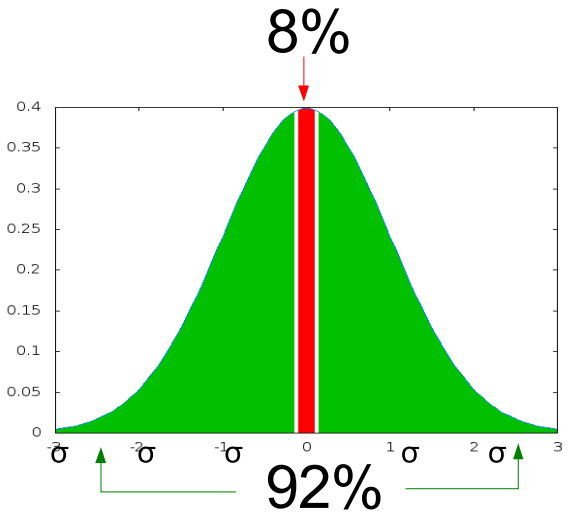
But if we underestimate errors by a factor of two, the same result implies a 4σ deviation. The probability for such a result is only 6E-5. It's very unlikely we'd ever observe this if the theory is correct. This is a reminder that experimental results are meaningless without uncertainties.

• The form of the experiment's uncertainties are adequately modeled by a Gaussian.
The Central Limit Theorem makes this common, but it's not always true.

Deviations that are too Small:

What if our neutron result had been 888 +/- 20 seconds , compared to the prediction of 886 seconds? These only differ by 0.1σ. We'd expect this to happen only about 8% of the time.

When results agree too closely, we need to think about their validity.

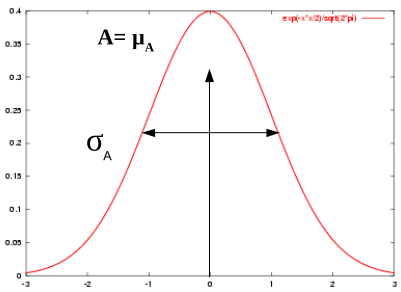


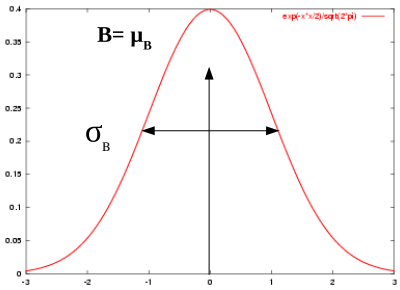
- Are we unusually lucky?
- Did we overestimate σ?
- Did we look at the prediction and cheat on the experiment?

The third possibility is something you should really think about when looking at published results. Is this result too good to be true?

Errors in Linear Combinations of Variables:

Assume that we measure two values, A and B, and are interested in their sum, C = A+B.





If we know the errors in A and B, what's the error in C?

Let's assume A and B fluctuate randomly and independently (we say that they're uncorrelated).

Propagation of Errors:

If we have a function of several variables, f(x,y,z...), we can use propagation of errors to find the error in f, given the errors in x, y, z....

$$\sigma^2_{f(x,y,z\ldots)} = \sigma^2_x(\frac{\partial f}{\partial x})^2 + \sigma^2_y(\frac{\partial f}{\partial y})^2 + \sigma^2_z(\frac{\partial f}{\partial z})^2 \ldots$$

Applying this to C(A,B) = A + B, we get:

$$\sigma^2_C = \sigma^2_A + \sigma^2_B$$

We say that the uncertainties add in quadrature, meaning that we add the squares instead of just adding the numbers directly.

For example, if $\sigma_A = 3$ and $\sigma_B = 4$, we'd have $\sigma_C = 5$, since $5^2 = 3^2 + 4^2$.

Errors in Averages:

We can apply this same technique to other combinations of variables. Consider the average, $\overline{A}$, of a bunch of N measurements, A_1 through A_N.

$$\overline{A} = \frac{1}{N}\sum_{i=1}^{N} A_i = \frac{1}{N}(A_1 + A_2 + A_3 ...)$$

By propagation of errors, the error in $\overline{A}$ should be given by the sum:

$$\sigma_{\overline{A}}^2 = \sigma_1^2 \frac{1}{N^2} + \sigma_2^2 \frac{1}{N^2} + \sigma_3^2 \frac{1}{N^2} ...$$

Leading to the result that the error in the average is less than the individual errors (all assumed equal here) by a factor of √N:

$$\sigma_{\overline{A}} = \frac{\sigma}{\sqrt{N}}$$

This tells us (as we knew intuitively) that we'll get a more precise value by averaging several measurements.

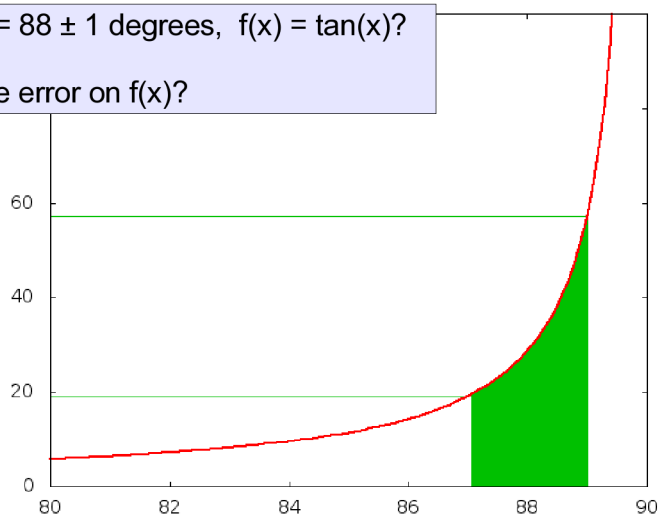
Errors and Non-linear Functions:

What if f(x,y,z,) is very nonlinear in one or more of the variables, and the individual errors, though normally distributed, are not small?

In this case the error on f may no longer be symmetric.

What if x = 88 ± 1 degrees, f(x) = tan(x)?

What is the error on f(x)?



Asymmetrical Errors:

From propagation of errors, we'd say that the error in f(x) = tan(x) was:

$$\sigma_f^2 = (1 + tan^2(x))^2 \sigma_x^2$$

For x = 88 +/- 1 degrees, that would give us σ_f = 14.3, and we might say that our value for f was tan(88) +/- 14.3, or f = 28.6 +/- 14.3.

But:

tan(88) ~ 29
tan(89 = 88+1) ~ 57
tan(87 = 88-1) ~ 19

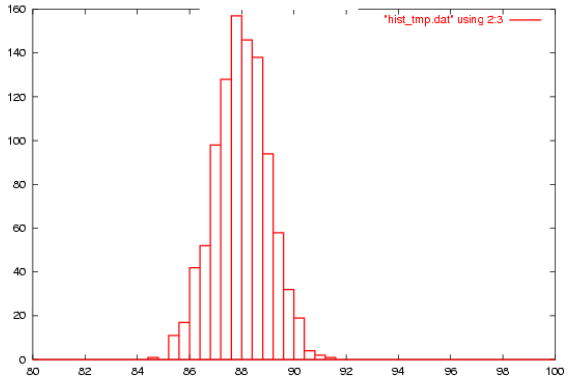
So f(x) = 29^{+29}_{-10} is more appropriate in this case!

Monte Carlo Estimation of Errors:

It is always possible to combine errors via a Monte Carlo approach.

This can be very useful for complex error propagations.

$x = 88 \pm 1$



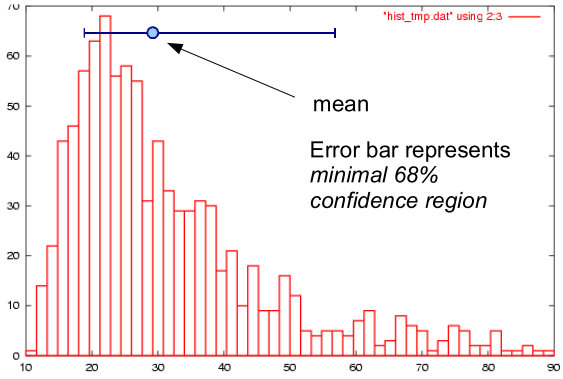
Generate a random distribution for each element, x, in our function

Then plot the distribution f(x), where x is drawn from the random sampling.

Monte Carlo Estimation of Errors:

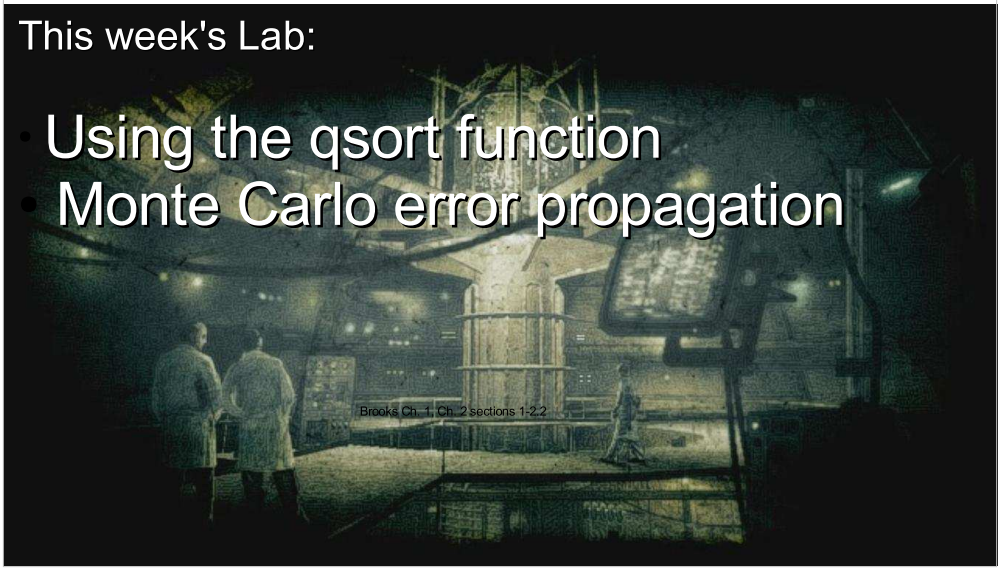
The Monte Carlo technique allows us to determine the actual uncertainty distribution on our dependent quantity. This technique is not limited to Gaussian uncertainties, but can be applied to any distribution.

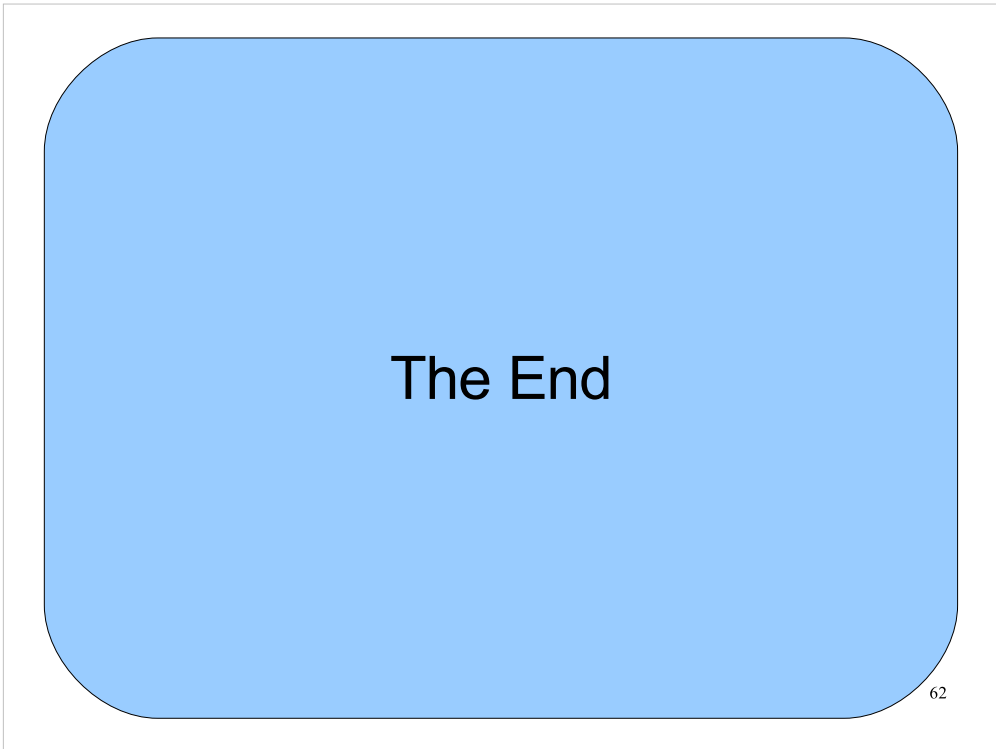
tan(x), where x is distributed according to 88 ± 1



Next Time:

- Fitting
- Chi-squared





Thanks!