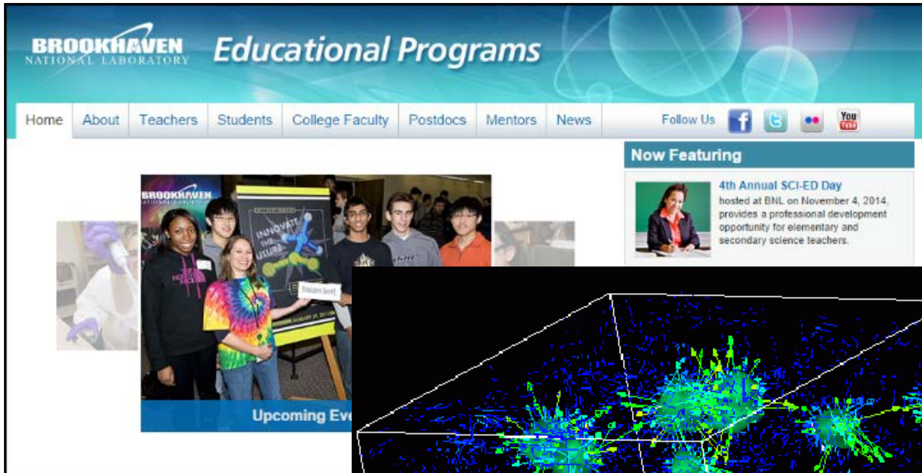
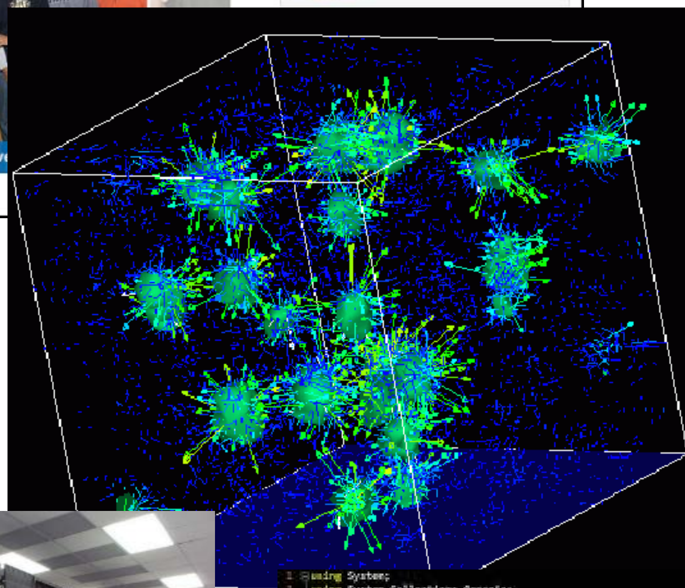
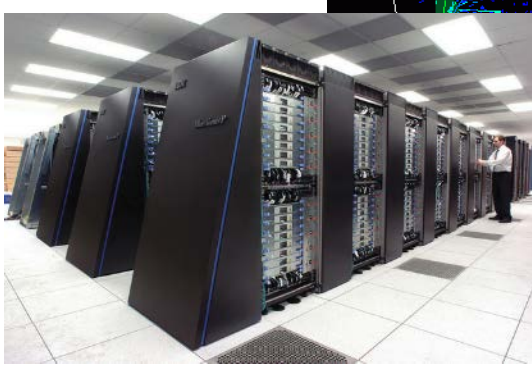






Survey of Scientific Computing
(SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 06
Statistics,
Euler Line

Session Goals

- Generate **Hero ability scores** in role-playing games
- Create and call **functions** in C++
- Calculate the **mean**, **variance**, and **standard deviation** of a sequence of numbers
- How to request "random" integers within a given range
- Develop a **computational mathematics experiment** that uncovers a **magic number** hidden in all uniform random number distributions
- Draw **Euler's Line**

Generating Hero Ability Values

- In most role-playing games, heroes have abilities such as strength, dexterity, intelligence, charism, etc.
- Initial abilities are often measured in ranges like **3 – 18**
- At the beginning of the game, players ***roll dice*** to determine the initial values for each ability
- The higher the value, the more likely the player will succeed while adventuring

	ABILITY SCORE	ABILITY MODIFIER	TEMP SCORE	TEMP MODIFIER
STR				
DEX				
CON				
INT				
WIS				
CHA				

Generating Hero Ability Values

- Two ways of rolling for initial abilities between **3 and 18**

1. Roll a **20**-sided die just <u>once</u> (**1d20**), but *reroll* if face value is 1, 2, 19, or 20
2. Roll a **6**-sided die <u>three times</u> (**3d6**), summing the value of each roll

- Using the **1d20** method is faster than **3d6**, especially when having to roll for six separate abilities

	ABILITY SCORE	ABILITY MODIFIER	TEMP SCORE	TEMP MODIFIER
STR	17			
DEX	11			
CON	15			
INT	5			
WIS	7			
CHA	3			

Functions

- You can declare and define your own custom functions
  - Function names use **CamelCase #1** – the *first* letter is Capitalized
  - A function is essentially a custom scope (a group of statements)
- Functions <u>receive</u> value via a **parameter list**
  - A function can get inbound values passed to it from somewhere else in the source code
  - Each parameter has a data type and variable identifier
- Functions <u>output</u> a value via the **return** statement
  - A function can only return one intrinsic data type
  - The **return** statement is usually at the end of the function

Defining a Function

Return Type

Function name

Inbound Parameter List

```
double CalcStdDev1d20(double totalRolls, double mean)
{
    double sum{};
    for (double rollNumber{}; rollNumber <= totalRolls;
        int roll = distribution1d20(generator);
        sum = sum + pow(roll - mean, 2);
    }
    double variance = sum / totalRolls;
    double stdDev = sqrt(variance);
    return stdDev;
}
```

Functions **open** a scope

Functions **close** a scope

A function ends at **return** and sends the value back to the *calling* function

Calling a Function

```
int main()
{
    double totalRolls = 1000000;

    cout.imbue(locale(""));

    cout << "Total number of dice r
            precision(0) << fixe
            << endl;

    double mean1d20 = CalcMean1d20(totalRolls);
    double mean3d6 = CalcMean3d6(totalRolls);

    distribution1d20.reset();
    distribution1d6.reset();
```

Return value stored in a local variable

Don't need types in front of the parameters when **calling** a function

Generating Hero Ability Values

- Two ways of rolling for initial abilities between **3 and 18**:

1. Roll a **20**-sided die just <u>once</u> (**1d20**), but *reroll* if face value is 1, 2, 19, or 20
2. Roll a **6**-sided die <u>three times</u> (**3d6**), summing the value of each roll

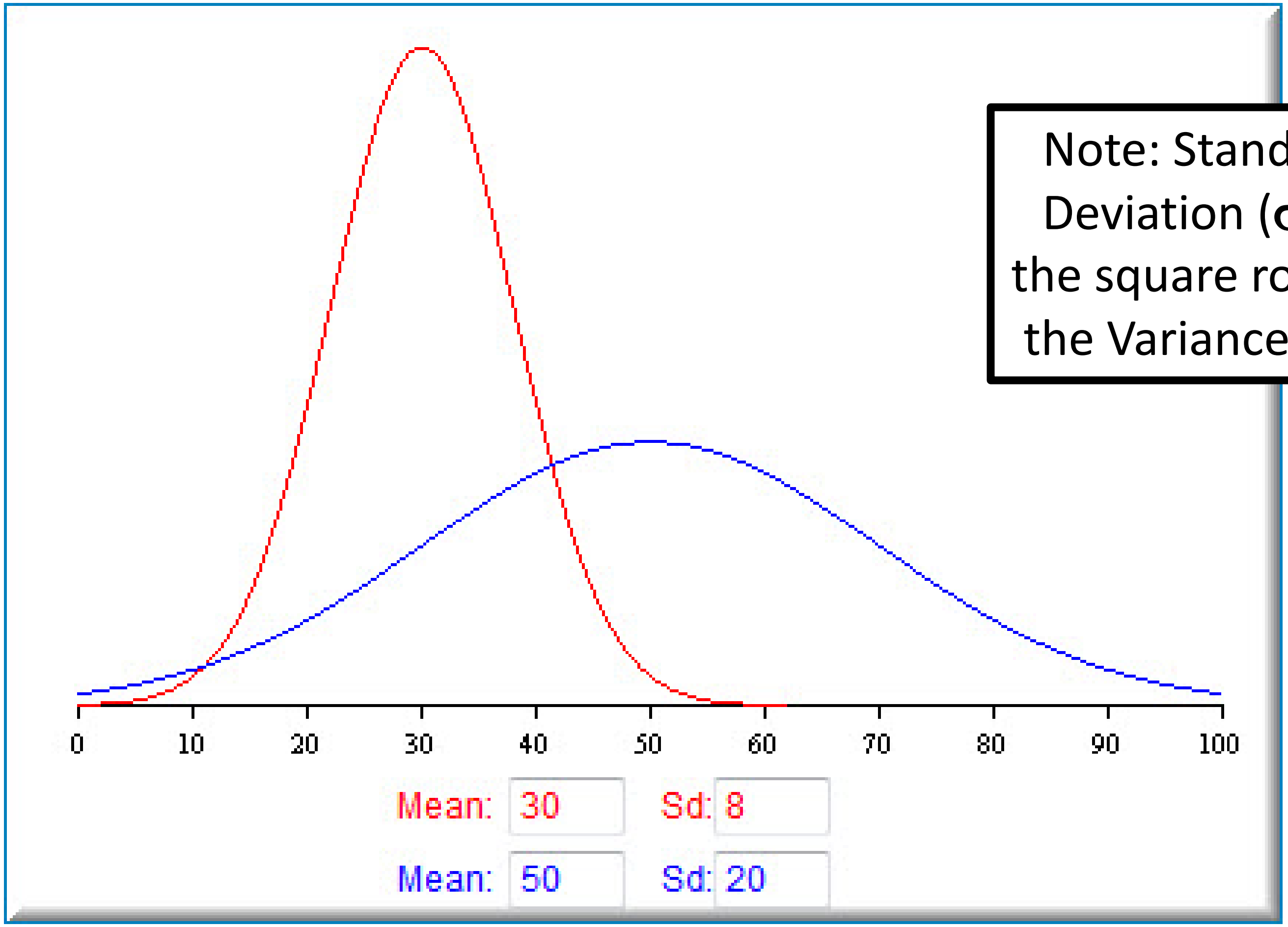
- **Which method would you want to use? Why?**

	ABILITY SCORE	ABILITY MODIFIER	TEMP SCORE	TEMP MODIFIER
STR	17			
DEX	11			
CON	15			
INT	5			
WIS	7			
CHA	3			

Mean vs. Variance

- Imagine two different classes take the same test
  - 1st period students score between 50 and 100 with μ = 75
  - 2nd period students score between 70 and 80 with μ = 75
- **Variance** $(\boldsymbol{\sigma}^2)$ is the average "distance" between each number in a set and the mean $(\boldsymbol{\mu})$ of that set
  - 1st period students have a greater **variance** in scores than 2nd period
  - Variance is a measure of **central tendency** – on average how close around the mean do all the numbers fall?
  - For <u>every</u> data point, we sum the ***square*** of the difference between the number and the mean. Then we divide that sum by the total number of data points

Mean vs. Standard Deviation



Note: Standard Deviation (σ) is the square root of the Variance (σ^2)

Mean, Variance, Standard Deviation

Mean

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Mean, Variance, Standard Deviation

Mean

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$x = \{2, 9, 11, 5, 6\} \therefore n = 5$$

$$\sum_{i=1}^{n} x_i = (2 + 9 + 11 + 5 + 6) = 33$$

$$\mu = \frac{33}{5} = 6.6$$

Mean, Variance, Standard Deviation

Variance

Mean

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \quad \text{Where} \quad \mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$x = \{2, 9, 11, 5, 6\} \therefore n = 5, \mu = 6.6$$

$$\sum_{i=1}^{n}(x_i - \mu)^2 = (2 - 6.6)^2 + (9 - 6.6)^2 + (11 - 6.6)^2 + \cdots = 49.2$$

$$\sigma^2 = \frac{49.2}{5} = 9.84$$

Mean, Variance, Standard Deviation

Variance

Mean

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \quad \text{Where} \quad \mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma = \sqrt{\sigma^2} = \sqrt{\frac{\sum(x_i - \mu)^2}{N}}$$

Standard Deviation

$$x = \{2, 9, 11, 5, 6\}$$

$$\sigma^2 = 9.84$$

$$\sigma = \sqrt{9.84} \cong 3.13$$

Pseudorandom Numbers

We will all use 2016 to verify your code with mine

Generator initialized to seed

```
seed_seq seed{ 2016 };
default_random_engine generator{ seed };
uniform_int_distribution<int> distribution1d20(3, 18);
uniform_int_distribution<int> distribution1d6(1, 6);
```

Emits random integers using a uniform distribution

Sets low & high range, both inclusive

Pseudorandom Numbers

```
double CalcStdDev3d6(double totalRolls, double mean)
{
    double sum{};
    for (double rollNumber{}; rollNumber <= totalRolls; ++rollNumber)
    {
        int roll = distribution1d6(generator)
                 + distribution1d6(generator)
                 + distribution1d6(generator);
        sum = sum + pow(roll - mean, 2);
    }
    double variance = sum / totalRolls;
    double stdDev = sqrt(variance);
    return stdDev;
}
```

Calls the **distribution1d6()** function three times. Each call moves the generator to the *next* state

Pseudorandom Numbers

```
cout << "Total number of dice rolls: "
    << setprecision(0) << fixed << totalRolls
    << endl << endl;

double mean1d20 = CalcMean1d20(totalRolls);
double mean3d6 = CalcMean3d6(totalRolls);

distribution1d20.reset();
distribution1d6.reset();

double stdDev1d20 = CalcStdDev1d20(totalRolls, mean1d20);
double stdDev3d6 = CalcStdDev3d6(totalRolls, mean3d6);
```

.reset() returns the distribution back to the original seed value

Open Lab 1 – Hero Abilities

- Update a program to generate **1,000,000** hero ability scores, comparing the *mean* and *standard deviation* of the 1d20 versus the 3d6 dice roll methods
- In particular, write the missing code to correctly calculate the **CalcStdDev1d20** function
- Use **pow**(x, y) to calculate x^y
- **Which dice roll method would you want to use to generate your hero's abilities?**

Edit Lab 1 – Hero Abilities

main.cpp

```cpp
#include "stdafx.h"

using namespace std;

seed_seq seed{ 2016 };
default_random_engine generator{ seed };
uniform_int_distribution<> distribution1d20(3, 18);
uniform_int_distribution<> distribution1d6(1, 6);

double CalcMean1d20(double totalRolls)
{
    double sum{};
    for (double rollNumber{}; rollNumber <= totalRolls; ++rollNumber)
    {
        int roll = distribution1d20(generator);
        sum = sum + roll;
    }
    double mean = sum / totalRolls;
    return mean;
}

double CalcStdDev1d20(double totalRolls, double mean)
{
    double sum{};
    for (double rollNumber{}; rollNumber <= totalRolls; ++rollNumber)
    {
        // Insert the correct code here

    }
    double variance = sum / totalRolls;
    double stdDev = sqrt(variance);
    return stdDev;
}
```

Edit Lab 1 – Hero Abilities

$$roll = x_i \qquad totalRolls = n \qquad mean = \mu$$

```cpp
double CalcStdDev1d20(double totalRolls, double mean)
{
    double sum{};
    for (double rollNumber{}; rollNumber <= totalRolls; ++rollNumber)
    {
        int roll = distribution1d20(generator);
        sum = sum + pow(roll - mean, 2);
    }
    double variance = sum / totalRolls;
    double stdDev = sqrt(variance);
    return stdDev;
}
```

$$sum = \sum (x_i - \mu)^2$$

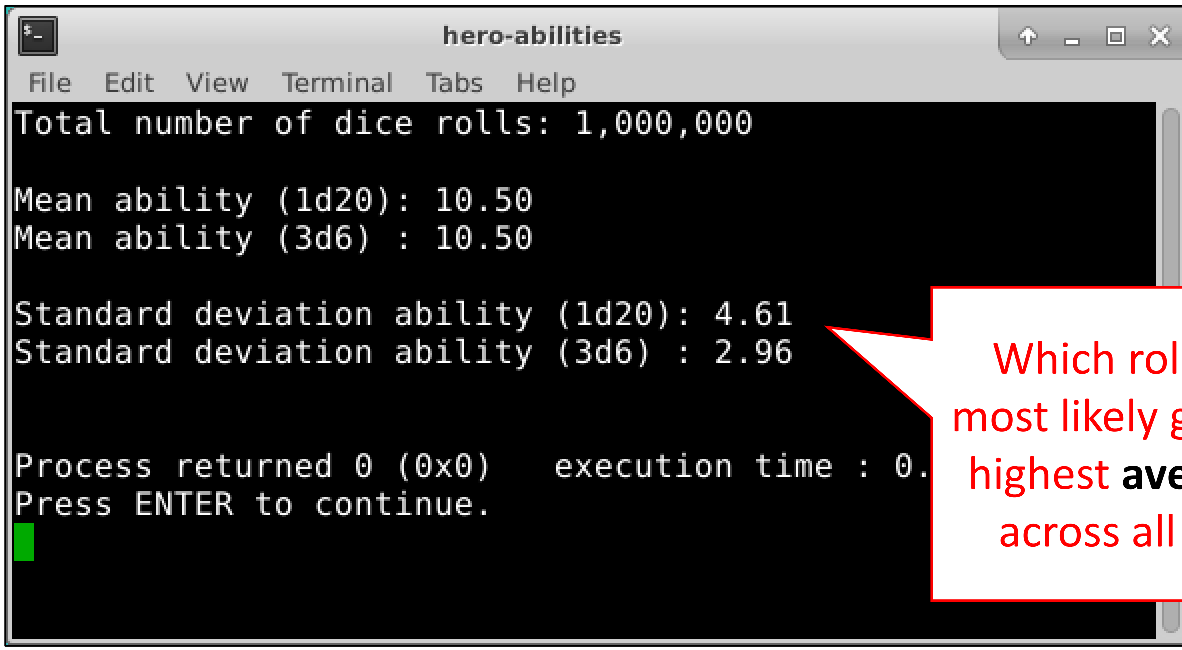
$$variance = \frac{sum}{n} = \sigma^2 \qquad stdDev = \sqrt{\sigma^2}$$

$$\boxed{\sigma^2 = \frac{1}{n}\sum_{i=i}^{n}(x_i - \mu)^2 \;\; where \;\; \mu = \frac{1}{n}\sum_{i=1}^{n} x_i}$$

Run Lab 1 – Hero Abilities

```
double CalcStdDev1d20(double totalRolls, double mean)
{
    double sum{};
    for (double rollNumber{}; rollNumber <= totalRolls; ++rollNumber)
    {
        int roll = distribution1d20(generator);
        sum = sum + pow(roll - mean, 2);
    }
    double variance = sum / totalRolls;
    double stdDev = sqrt(variance);
    return stdDev;
}
```

Check Lab 1 – Hero Abilities



Which roll type will most likely give you the highest **average** score across all abilities?

Variance of Uniform Distributions

- Given an existing program that can:
  - Generate 15 sets of **random size** between 1 million & 2 million items
  - Within each set, every item is a random integer chosen within a range between a **lower limit** and an **upper limit**
  - The **lower limit** is a random number between **0 and 1000**
  - The **upper limit** is **2x** that set's lower limit **+** another random number between 0 and 1000
  - Calculate the mean (μ) and variance (σ^2) for each set

$$\sigma^2 = \frac{1}{n}\sum_{i=i}^{n}(x_i - \mu)^2 \;\; where \;\; \mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Variance of Uniform Distributions

- Your assignment is to discover a magic number hidden in **all** uniform random number distributions
  - Calculate and display this “constant” for each set:

$$\textit{magicNumber} = \frac{(\boldsymbol{upperLimit} - \boldsymbol{lowerLimit})^{\mathbf{2}}}{variance}$$

  - Is this number the same for ALL uniform distributions?
- Can we use this value to test if dice are loaded?

Edit Lab 2 – Variance of Uniform Distributions

main.cpp

```
        int lowerLimit = distLimits(generator);
        int upperLimit = 2 * lowerLimit + distLimits(generator);

        uniform_int_distribution<> distRange(lowerLimit, upperLimit);

        double sum{};
        for (int n{}; n < setSize; ++n)
            sum += distRange(generator);
        double mean = sum / setSize;

        distRange.reset();

        sum = 0;
        for (int n{}; n < setSize; ++n)
            sum += pow(distRange(generator) - mean, 2);
        double variance = sum / setSize;

        double magicNumber = 0;

        cout << right << fixed
             << setw(5) << setNumber
             << setw(7) << lowerLimit
             << setw(7) << upperLimit
             << setw(12) << setSize
             << setw(12) << setprecision(3) << mean
             << setw(13) << setprecision(3) << variance
             << setw(8) << setprecision(0) << magicNumber
             << endl;
    }
```

Fix this formula

Run Lab 2 – Variance of Uniform Distributions

main.cpp

```cpp
        int lowerLimit = distLimits(generator);
        int upperLimit = 2 * lowerLimit + distLimits(generator);

        uniform_int_distribution<> distRange(lowerLimit, upperLimit);

        double sum{};
        for (int n{}; n < setSize; ++n)
            sum += distRange(generator);
        double mean = sum / setSize;

        distRange.reset();

        sum = 0;
        for (int n{}; n < setSize; ++n)
            sum += pow(distRange(generator) - mean, 2);
        double variance = sum / setSize;

        double magicNumber = pow(upperLimit - lowerLimit,2) / variance;

        cout << right << fixed
             << setw(5) << setNumber
             << setw(7) << lowerLimit
             << setw(7) << upperLimit
             << setw(12) << setSize
             << setw(12) << setprecision(3) << mean
             << setw(13) << setprecision(3) << variance
             << setw(8) << setprecision(0) << magicNumber
             << endl;
    }
```

Check Lab 2 – Variance of Uniform Distributions

uniform-variance

File Edit View Terminal Tabs Help

```
Set #  Lower  Upper   Set Size       Mean     Variance   Magic
    1    689  2,316  1,620,041  1,502.833  220,857.039  11.986
    2    696  1,927  1,266,716  1,311.233  126,397.952  11.989
    3    649  1,382  1,478,220  1,015.444   44,859.287  11.977
    4    869  2,333  1,853,400  1,601.290  179,153.351  11.963
    5    404  1,340  1,503,737    872.097   73,188.823  11.970
    6     54    166  1,948,452    109.961    1,064.990  11.779
    7    615  1,958  1,320,808  1,286.224  150,687.341  11.969
    8    525  2,032  1,372,102  1,278.145  189,437.533  11.988
    9    503  1,903  1,059,304  1,203.810  163,723.989  11.971
   10    177  1,336  1,769,803    756.601  111,999.179  11.994
   11    183    905  1,132,902    544.112   43,569.989  11.964
   12    820  2,216  1,020,461  1,517.969  162,862.874  11.966
   13    753  1,726  1,137,399  1,239.296   79,058.507  11.975
   14    416  1,599  1,860,276  1,007.888  116,835.220  11.978
   15    238  1,099  1,861,754    668.469   61,906.115  11.975

Process returned 0 (0x0)   execution time : 3.236 s
Press ENTER to continue.
```

- Every set had a different lower and upper limit, size, mean, and variance… yet the magic number was ~12 for all of them!
- Why would Mother Nature pick the value 12 for this magic number? What is so special about 12? Why not pick a nice even 10?
- Boundless natural curiosity is what makes a good scientist…

Variance of Uniform Distributions

$$\boxed{\sigma^2 = \frac{1}{n}\sum_{i=i}^{n}(x_i - \mu)^2 \;\; where \;\; \mu = \frac{1}{n}\sum_{i=1}^{n} x_i}$$

The *expected* value ($\mathbb{E}$) of a random variable $\boldsymbol{X}$ is its <u>mean</u> value (μ)

$$\mathbb{E}(X) = \mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Variance (σ^2) is the mean difference *squared* between every $\boldsymbol{X}$ and its $\mathbb{E}(X)$

$$\sigma^2 = \mathbb{E}\left(X - \mathbb{E}(X)\right)^2$$

The *expected* value ($\mathbb{E}$) returns a **constant** value

$$\mathbb{E}(X) = \mu$$

The *expected* value ($\mathbb{E}$) of a **constant** value returns that same value

$$\mathbb{E}(\mu) = \mu$$

$$\mathbb{E}\big(\mathbb{E}(X)\big) = \mathbb{E}(X)$$

$$\mathbb{E}\Big(\mathbb{E}\big(\mathbb{E}(X)\big)\Big) = \mathbb{E}(X)$$

Variance of Uniform Distributions

$$\boxed{\sigma^2 = \frac{1}{n}\sum_{i=i}^{n}(x_i - \mu)^2 \;\; where \;\; \mu = \frac{1}{n}\sum_{i=1}^{n} x_i}$$

$$\mu = \mathbb{E}(X) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma^2 = \mathbb{E}\left(X - \mathbb{E}(X)\right)^2$$

$$\mathbb{E}(\mu) = \mu$$

$$\mathbb{E}\big(\mathbb{E}(X)\big) = \mathbb{E}(X)$$

$$\sigma^2 = \mathbb{E}\left(X - \mathbb{E}(X)\right)^2$$

FOIL

$$\sigma^2 = \boxed{\mathbb{E}}[\boxed{X^2} - \boxed{2X\mathbb{E}(X)} + \boxed{\mathbb{E}(X)^2}]$$

Note: $\mathbb{E}(x)$ is a distributive liner operator

$$\sigma^2 = \mathbb{E}(X^2) - \mathbb{E}\big(2X\mathbb{E}(X)\big) + \mathbb{E}(\mathbb{E}(X)^2)$$

$$\sigma^2 = \mathbb{E}(X^2) - 2\mathbb{E}(X)\mathbb{E}(X) + \mathbb{E}(X)^2$$

$$\sigma^2 = \mathbb{E}(X^2) - 2\mathbb{E}(X)^2 + \mathbb{E}(X)^2$$

$$\sigma^2 = \mathbb{E}(X^2) - \mathbb{E}(X)^2$$

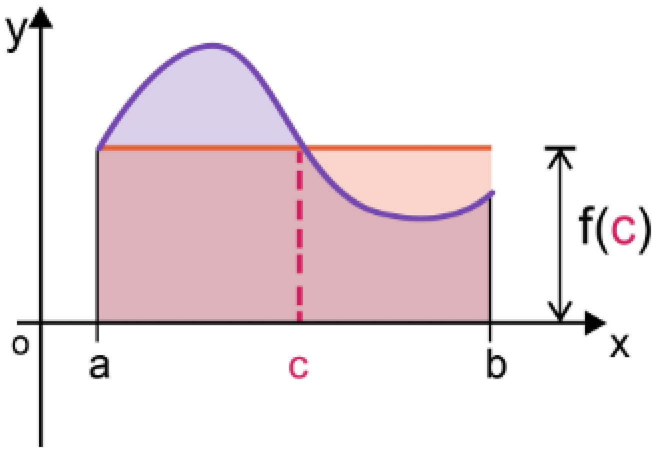
$$\sigma^2 = \mathbb{E}(X^2) - \mu^2$$

$$\sigma^2 = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) - \mu^2$$

Faster because only <u>one</u> subtraction is required!

Variance of Uniform Distributions

$f(c)$ = the average value of the function



Random Variable (Uniform Distribution)

Discrete: $\mathbb{E}(X) = \frac{1}{n}\sum_{i=1}^{n} x_i$

Continuous: $\mathbb{E}(X) = \frac{1}{(b-a)}\int_a^b x\,dx$

Mean Value Theorem (*Integrals*)

$$Area_{red} = Area_{curve}$$

$$Area_{red} = f(c) \times (b-a)$$

$$Area_{curve} = \int_a^b f(x)\,dx$$

$$f(c) \times (b-a) = \int_a^b f(x)\,dx$$

$$f(c) = \frac{1}{(b-a)}\int_a^b f(x)\,dx$$

$$f(c) = \mu = \mathbb{E}(X)$$

Variance of Uniform Distributions

Moment Generating Function

$$\mathbb{E}(X) = \frac{1}{(b-a)} \int_a^b x\, dx$$

$$\mathbb{E}(X^2) = \frac{1}{(b-a)} \int_a^b x^2\, dx$$

$$\sigma^2 = \mathbb{E}(X^2) - \mu^2$$

$$\mu = \int_a^b x \frac{1}{b-a} dx = \frac{1}{b-a}\left(\frac{x^2}{2} \Big|_a^b\right) = \frac{b^2-a^2}{2(b-a)}$$

$$= \frac{(b-a)(b+a)}{2(b-a)} = \frac{b+a}{2}$$

```
Mean ability (1d20): 10.50
Mean ability (3d6) : 10.50
```

$$\frac{(18+3)}{2} = 10.5$$

Lab 1 Results

Variance of Uniform Distributions

Moment Generating Function

$$\mathbb{E}(X) = \frac{1}{(b-a)}\int_a^b x\,dx$$

$$\mathbb{E}(X^2) = \frac{1}{(b-a)}\int_a^b x^2\,dx$$

$$\sigma^2 = \mathbb{E}(X^2) - \mu^2$$

$$\mu = \int_a^b x\frac{1}{b-a}dx = \frac{1}{b-a}\left(\frac{x^2}{2}\ \Big|_a^b\ \right) = \frac{b^2-a^2}{2(b-a)}$$

$$= \frac{(b-a)(b+a)}{2(b-a)} = \frac{b+a}{2}$$

$$E(X^2) = \int_a^b x^2\frac{1}{b-a}dx = \frac{1}{(b-a)}\left(\frac{x^3}{3}\ \Big|_a^b\ \right)$$

$$= \frac{b^3-a^3}{3(b-a)} = \frac{(b-a)(b^2+ab+a^2)}{3(b-a)} = \frac{b^2+ab+a^2}{3}$$

Variance of Uniform Distributions

Moment Generating Function

$$\mathbb{E}(X) = \frac{1}{(b-a)} \int_a^b x\, dx$$

$$\mathbb{E}(X^2) = \frac{1}{(b-a)} \int_a^b x^2\, dx$$

$$\boxed{\sigma^2 = \mathbb{E}(X^2) - \mu^2}$$

$$\mu = \int_a^b x \frac{1}{b-a} dx = \frac{1}{b-a}\left(\frac{x^2}{2} \Big|_a^b \right) = \frac{b^2 - a^2}{2(b-a)}$$

$$= \frac{(b-a)(b+a)}{2(b-a)} = \frac{b+a}{2}$$

$$E(X^2) = \int_a^b x^2 \frac{1}{b-a} dx = \frac{1}{(b-a)}\left(\frac{x^3}{3} \Big|_a^b \right)$$

$$= \frac{b^3 - a^3}{3(b-a)} = \frac{(b-a)\left(b^2 + ab + a^2\right)}{3(b-a)} = \frac{b^2 + ab + a^2}{3}$$

$$\boxed{\sigma^2 = E\left(X^2\right) - \mu^2} = \frac{b^2 + ab + a^2}{3} - \left(\frac{b+a}{2}\right)^2 = \frac{4b^2 + 4ab + 4a^2 - 3b^2 - 6ab - 3a^2}{12}$$

$$= \frac{b^2 - 2ab + a^2}{12} = \boxed{\frac{(b-a)^2}{12}}$$

$$\boxed{12 = \frac{(upperLimit - lowerLimit)^2}{variance}}$$

Understanding Probability

- The probability of a continuous random variable having an exact value is **zero!!**
  - We can never measure continuous distributions exactly
  - The measurement changes as we increase magnification/precision
- Accumulating statistics may enable accurate trend prediction
  - Single events may happen in a **non-deterministic** manner
  - Yet the aggregate *ensemble* behavior might be **deterministic**
  - A paradox? Think Lotto: small odds for all, yet someone always wins
- Scientific *observables* are governed by averages
  - **Statistical Mechanics**: temperate = average kinetic energy
  - Quantum Mechanics: Shape of electron orbitals

What Should Students Learn?

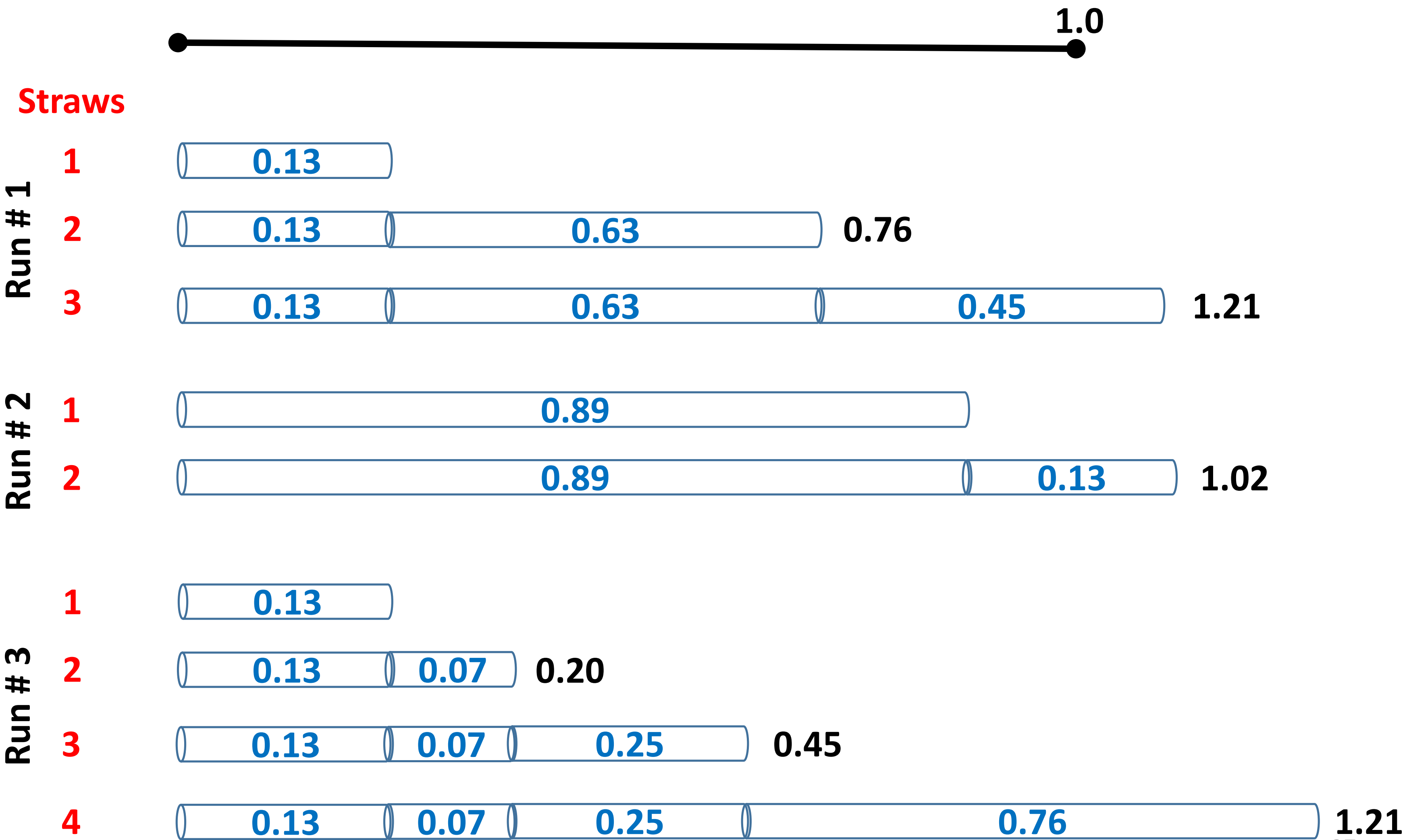
The Probability and Statistics Used Most Often at BNL:

- Uniform, Normal, Exponential, Logarithmic
- Student's ***t***
- Bernoulli, Binomial, Beta
- Poisson, Erlang, Pareto
- Mean, Mode, Median
- Skew, Variance, Std. Dev
- Moments of Distribution
- Conditional Probability
- Bayesian Inference
- Gamma, Chi-Squared
- Maxwell-Boltzman
- Regression Techniques
- Confidence Intervals
- Hypothesis Testing
- Time Series Analysis
- ANOVA
- Stochastic Processes
- Markov Chains
- Clustering Algorithms

Random Straws

- Write a program to perform **ten million runs** of an experiment that places a varying number of straws ***end-to-end*** each run
- In each run, start with a single straw of **random** length between $0 \le n < 1$
- Then enter a loop that keeps adding additional straws of **random** length ($0 \le n < 1$) until the <u>total</u> length is > 1
- Find the **mean** number of straws added before the total length exceeds 1, across *all* million runs of the experiment

Random Straws



Edit Lab 3 – Random Straws

main.cpp

```cpp
#include "stdafx.h"

using namespace std;

seed_seq seed{ 2016 };
default_random_engine generator{ seed };
uniform_real_distribution<double> distribution(0.0, 1.0);

int main()
{
    double maxIterations = 10000000;
    double totalStraws = 0;

    cout.imbue(locale(""));
    cout << fixed << setprecision(0);
    cout << "Performing " << maxIterations
         << " experiments.." << endl;

    for (double iteration = 0; iteration < maxIterations; iteration++)
    {
        // Insert your code here

    }
    double meanStrawsPerUnitLength = totalStraws / maxIterations;

    cout << setprecision(6) << endl
         << "Mean straws per iteration: "
         << meanStrawsPerUnitLength << endl;

```

Write code to perform each step of the experiment

Run Lab 3 – Random Straws

main.cpp

```
#include "stdafx.h"

using namespace std;

seed_seq seed{ 2016 };
default_random_engine generator{ seed };
uniform_real_distribution<double> distribution(0.0, 1.0);

int main()
{
    double maxIterations = 10000000;
    double totalStraws = 0;

    cout.imbue(locale(""));
    cout << fixed << setprecision(0);
    cout << "Performing " << maxIterations
         << " experiments.." << endl;

    for (double iteration = 0; iteration < maxIterations; iteration++)
    {
        double straws = 1;
        double length = distribution(generator);
        while (length <= 1.0)
        {
            straws++;
            length += distribution(generator);
        }
        totalStraws += straws;
    }
```

Check Lab 3 – Random Straws

random-straws

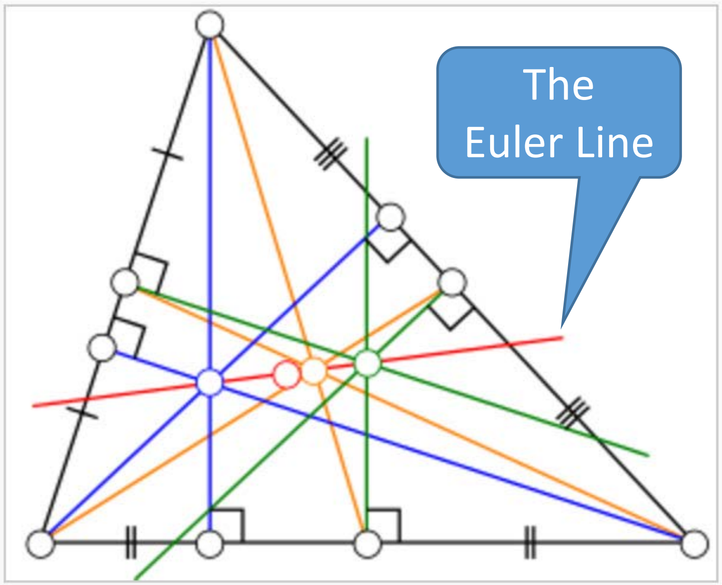
File Edit View Terminal Tabs Help

```
Performing 10,000,000 experiments..

Mean straws per iteration: 2.718281
Base of natural logarithm: 2.718282

Process returned 0 (0x0)   execution time : 5.789 s
Press ENTER to continue.
```

$$e = \lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x = 2.718281828459\ldots$$

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

A Geometry Gem the Greeks Overlooked

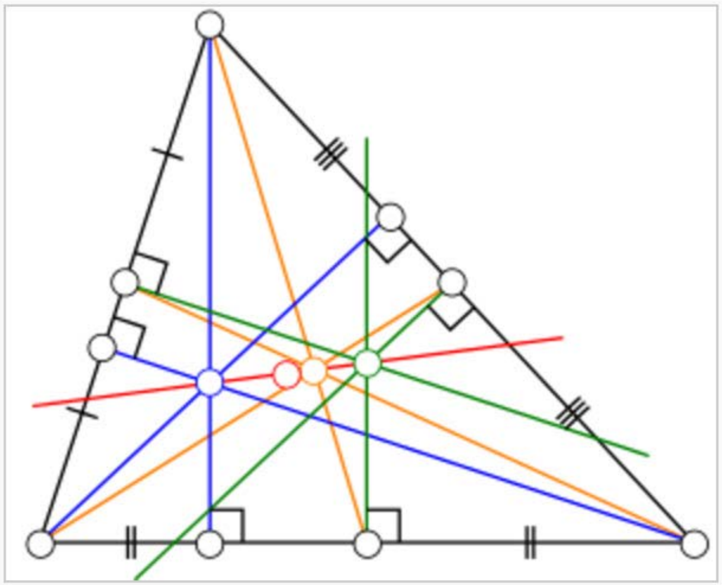


Euler's line (red) is a straight line through the centroid (orange), orthocenter (blue), circumcenter (green) and center of the nine-point circle (red).

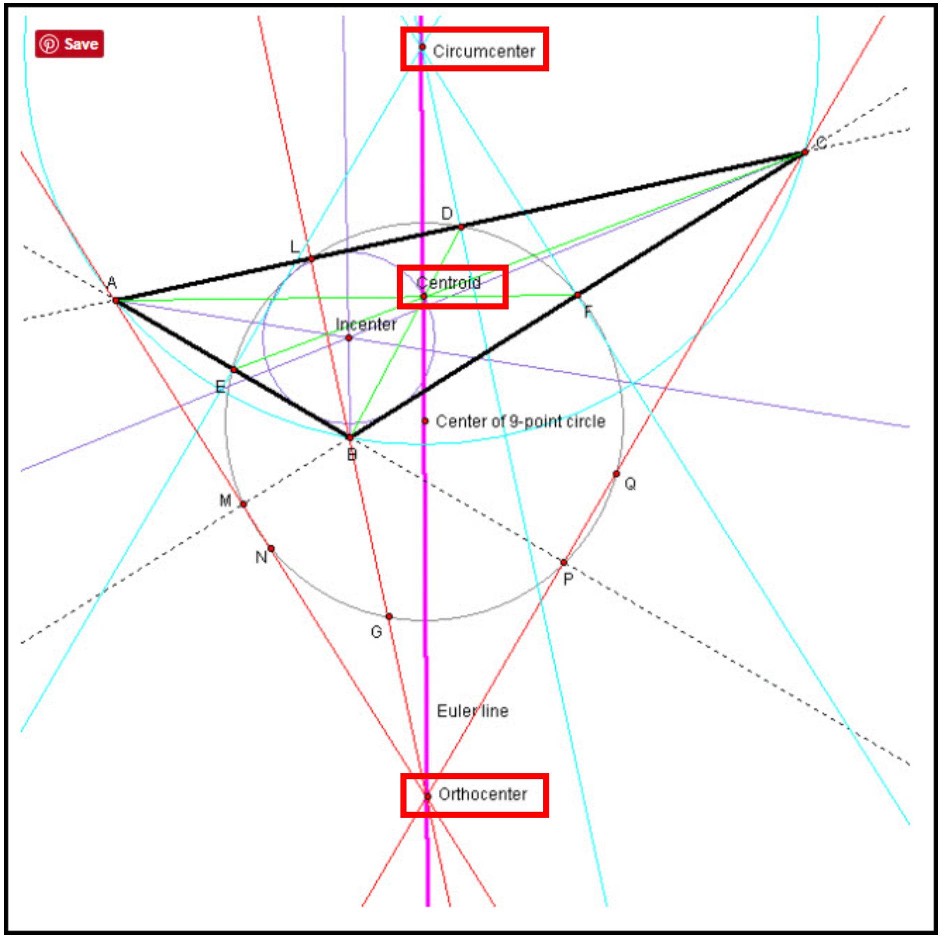
- **Centroid** = center of mass
- **Circumcenter** = intersection of ⊥ *side* bisectors
- **Orthocenter** = intersection of the altitudes

Euler was the first to realize and prove those **three** points are **always colinear** for <u>any</u> given triangle!

The Euler Line

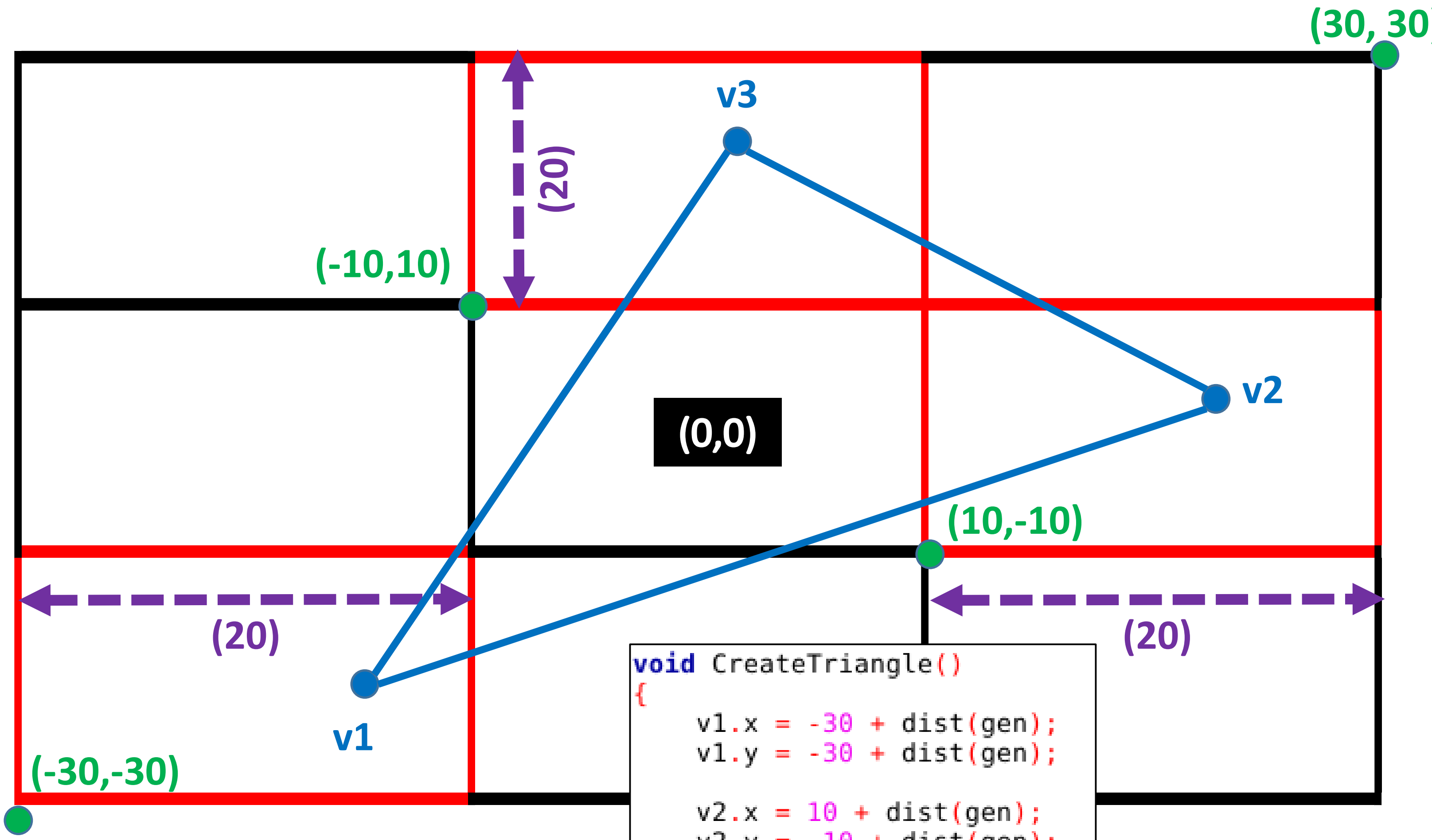


Euler's line (red) is a straight line through the centroid (orange), orthocenter (blue), circumcenter (green) and center of the nine-point circle (red).



```
seed_seq seed{ 2016 };
default_random_engine gen{ seed };
uniform_int_distribution<int> dist(0, 20);
```

The Euler Line




```
void CreateTriangle()
{
    v1.x = -30 + dist(gen);
    v1.y = -30 + dist(gen);

    v2.x = 10 + dist(gen);
    v2.y = -10 + dist(gen);

    v3.x = -10 + dist(gen);
    v3.y = 10 + dist(gen);
}
```

Open Lab 4 – Euler Line

```
void eventHandler(SimpleScreen& ss, ALLEGRO_EVENT& ev)
{
    if (ev.type == ALLEGRO_EVENT_KEY_CHAR) {
        if (ev.keyboard.keycode == ALLEGRO_KEY_N) {
            CreateTriangle();
            ss.Clear();
            ss.Redraw();
        }
    }
}

int main()
{
    SimpleScreen ss(draw, eventHandler);

    ss.SetWorldRect(-30, -30, 30, 30);

    CreateTriangle();

    ss.HandleEvents();

    return 0;
}
```

Press the N key to draw a new random triangle

View Lab 4 – Euler Line

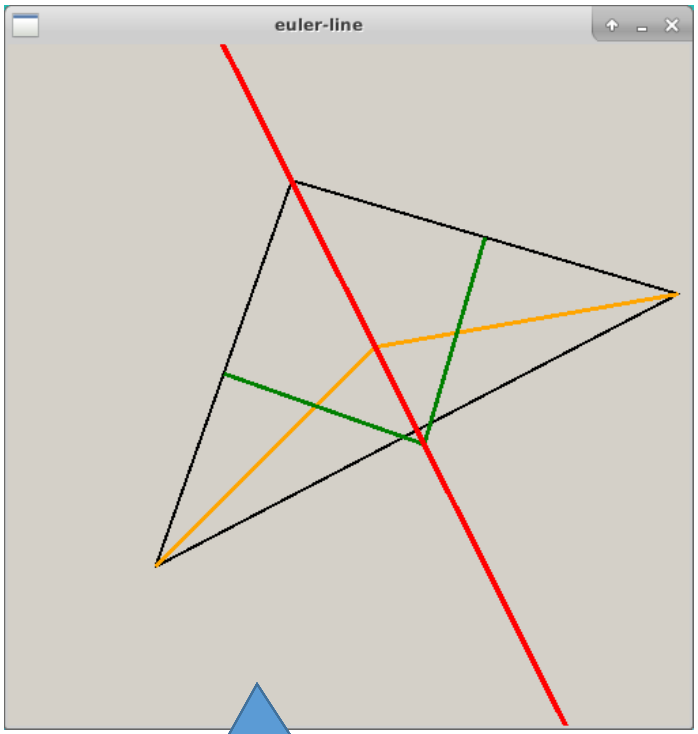
```
void draw(SimpleScreen& ss)
{
    // Connect the three vertices
    PointSet ps;
    ps.add(&v1);
    ps.add(&v2);
    ps.add(&v3);
    ss.DrawLines(&ps, "black", 2, true);

    // Calculate the slope of each side
    double slope12 = (v2.y - v1.y) / (v2.x - v1.x);
    double slope13 = (v3.y - v1.y) / (v3.x - v1.x);
    double slope23 = (v3.y - v2.y) / (v3.x - v2.x);

    // Calculate perpendicular slopes of each side
    slope12 = -1 / slope12;
    slope13 = -1 / slope13;
    slope23 = -1 / slope23;

    // Calculate the centroid
    Point2D centroid((v1.x + v2.x + v3.x) / 3,
                     (v1.y + v2.y + v3.y) / 3);

    // Connect vertices to centroid
    ss.DrawLine(v1, centroid, "orange", 3);
    ss.DrawLine(v2, centroid, "orange", 3);
    ss.DrawLine(v3, centroid, "orange", 3);
```

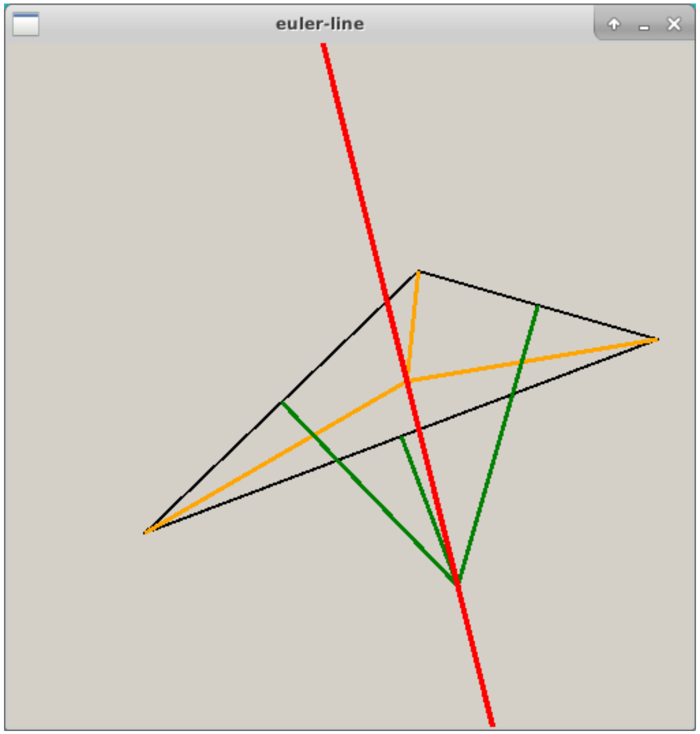
```
    // Calculate side bisector points
    Point2D bis12((v1.x + v2.x) / 2, (v1.y + v2.y) / 2);
    Point2D bis13((v1.x + v3.x) / 2, (v1.y + v3.y) / 2);
    Point2D bis23((v2.x + v3.x) / 2, (v2.y + v3.y) / 2);

    // Calculate y-intercept of each perpendicular side bisector
    double yint12 = bis12.y - slope12*bis12.x;
    double yint13 = bis13.y - slope13*bis13.x;
    double yint23 = bis23.y - slope23*bis23.x;

    // Calculate the circumcenter
    double ccx = (yint13 - yint12) / (slope12 - slope13);
    double ccy = slope12*ccx + yint12;
    Point2D cc(ccx, ccy);

    // Connect the side bisectors to the circumcenter
    ss.DrawLine(bis12, cc, "green", 3);
    ss.DrawLine(bis13, cc, "green", 3);
    ss.DrawLine(bis23, cc, "green", 3);
```

Note: This code only draws the line connecting the centroid and circumcenter

```
    // Calculate the point-slope of the Euler line
    // connecting the centroid with the circumcenter
    double slope_el = (centroid.y - cc.y) / (centroid.x - cc.x);
    double yint_el = cc.y - slope_el*cc.x;

    // Draw the Euler line
    Point2D el1(-30, -30 * slope_el + yint_el);
    Point2D el2(30, 30 * slope_el + yint_el);
    ss.DrawLine(el1, el2, "red", 4);
}
```

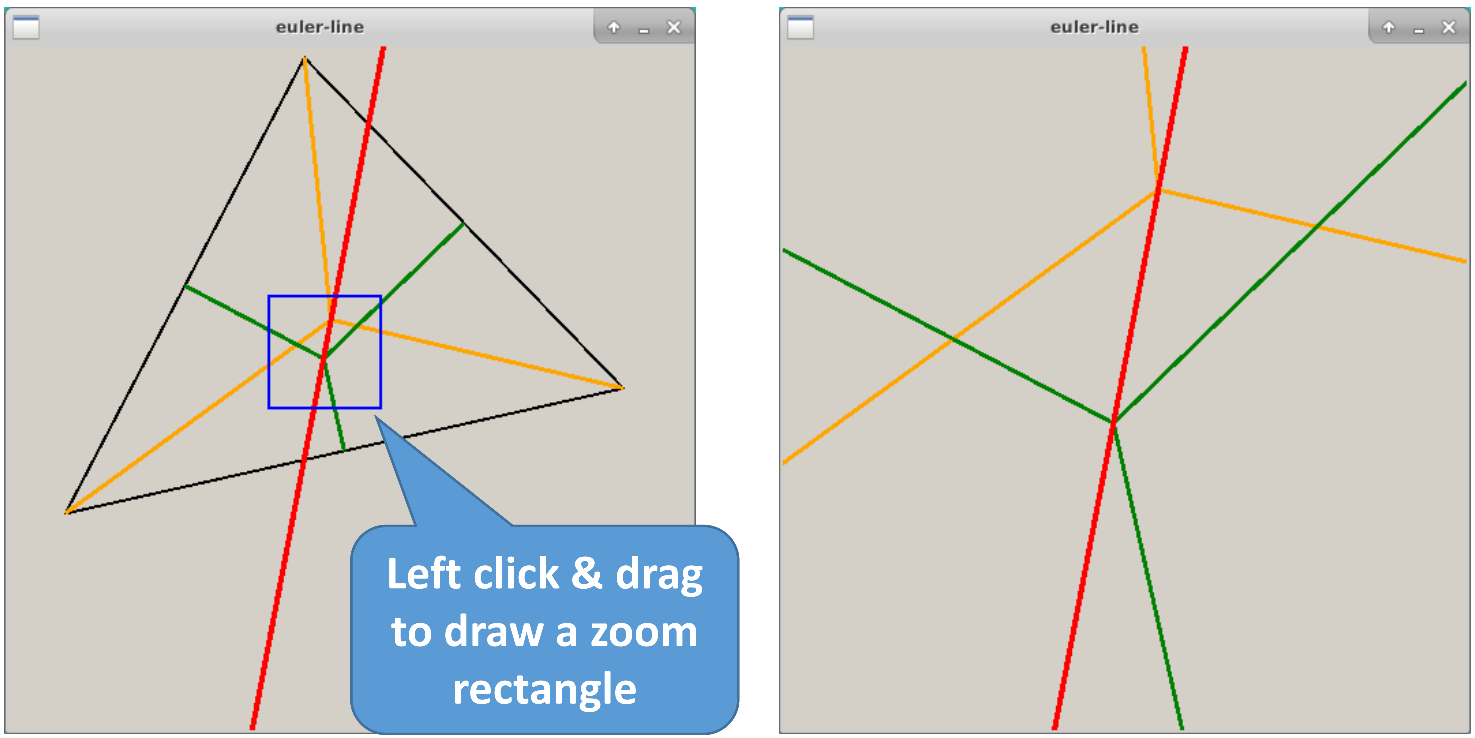
Run Lab 4 – Euler Line





Press the N key to draw a new random triangle

Check Lab 4 – Euler Line



Edit Lab 4 – Euler Line

- Recall Euler proved the **orthocenter** (the intersection of the altitudes) is also on that *same* **red** line
- The **orthocenter** is similar to the circumcenter, except instead of using the midpoint of each side, **we use the vertex opposite each side**, to find the point-slope form of each altitude
  - Use v1 and the negative reciprocal of slope v2v3
  - Use v2 and the negative reciprocal of slope v1v3
- Add code to **Lab 4** to calculate and draw the **orthocenter**, to visually confirm it falls on the same line formed from the centroid and circumcenter – **for all triangles**!

Now you know…

- How to calculate **Mean**, **Variance**, **Standard Deviation**
- Variance is just another average: it is the average distance between each data point and the mean $\boldsymbol{\mu}$ of the entire set
- Create and call **custom functions** to organize code into named scopes having specific purposes
- A seemingly random process hides a different universal constant: $\boldsymbol{e}$
- Some statistics are only meaningful after generating a **very large** sample set – everything in scientific computing is **big**