Survey of Scientific Computing

(SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 03

Loops, Conditionals, Modulus

Session Goals

- Introduce the **bool** data type and **logical operators**
- Use the **if()** statement for ***conditional*** code execution
- Learn about the **while()** loop
- Appreciate the **%** “modulus” (remainder) operator
- Generate a list of **perfect numbers**
- Create an algorithm to find **square roots**
- Consider one approach to handling very large integers
- Write code to **factor** any **quadratic** with *integer* coefficients
- Use **Simpson’s Rule** to calculate area under a polynomial

Logical Operators

- A variable of type **bool** (Boolean) can store only **true** or **false** values. The default value for a **bool** is **false**
- Use the **&&** operator to calculate a Boolean **AND**
  - (A && B) == **true** only if both A <u>and</u> B are **true**
- Use the **||** operator to calculate a Boolean **OR**
  - (A || B) == **true** if either A <u>or</u> B are **true**
- Use the **!** operator to calculate a Boolean **NOT**
  - If A == **true**, then !A == **false**
  - If A == **false**, then !A == **true**

if() Statement

- An **if**() statement identifies which code block (scope) to run based upon the value of a **Boolean expression**
- The expression (the *condition*) between the parenthesis **must evaluate** to either a true or false value
- If the condition is **true**, then the scope <u>immediately following</u> the if() statement is executed
- If the condition is **false**, and there is an **else** clause, then the scope immediately following the **else** statement is executed
- Every **if**() statements does not need to have an **else** clause

Two types of **if**() Statements

An if() <u>without</u> an else

```
// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

An if() <u>with</u> an else

```
// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.
```

while() Loop

- A **while**() loop executes all the statements within its scope as long as loop conditional remains **true**

The loop conditional

```
double epsilon{ 1e-14 };

while (abs(estimateSquared - x) > epsilon) {
    if (estimateSquared > x) {
        highEnd = estimate;
    }
    else {
        lowEnd = estimate;
    }
    estimate = (highEnd + lowEnd) / 2;
    estimateSquared = pow(estimate, 2);
}
```

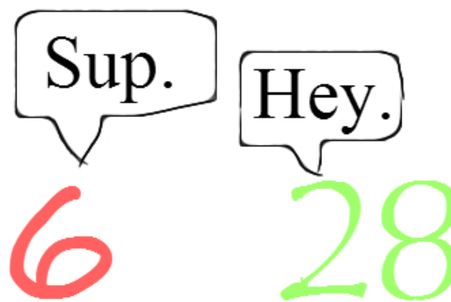
The Modulus (%) Operator

- The "**mod**" operator (**%**) returns the integer **remainder** of an implicit division operation, e.g. **37 % 5 = 2**
- Use double equals operator (**==**) when testing for **equality**

```
int sumOfFactors{ 1 };
for (int factor{ 2 }; factor < n;++factor) {
    if (n % factor == 0) {
        sumOfFactors += factor;
    }
}
```

Perfect Numbers

- Write a program to calculate and display all the perfect numbers **n** $(n \in \mathbb{Z}^+)$ between **2** and **10,000**
- An integer **n** is **perfect** when the sum of *almost all* of its divisors (including **1**, but *<u>not</u> including* **n** itself) is equal to **n**
- Example: **6 = 1 + 2 + 3**



Perfect Numbers

Number	Positive Factors	Sum of all factors excluding itself
1	1	0
2	1, 2	1
3	1, 3	1
4	1, 2, 4	3
5	1, 5	1
$\boxed{6}$	1, 2, 3, 6	$\boxed{6}$ Perfect!
7	1, 7	1
8	1, 2, 4, 8	7
9	1, 3, 9	4
10	1, 2, 5, 10	8
11	1, 11	1
12	1, 2, 3, 4, 6, 12	16

Edit Lab 1 – Perfect Numbers

perfect-numbers.cpp

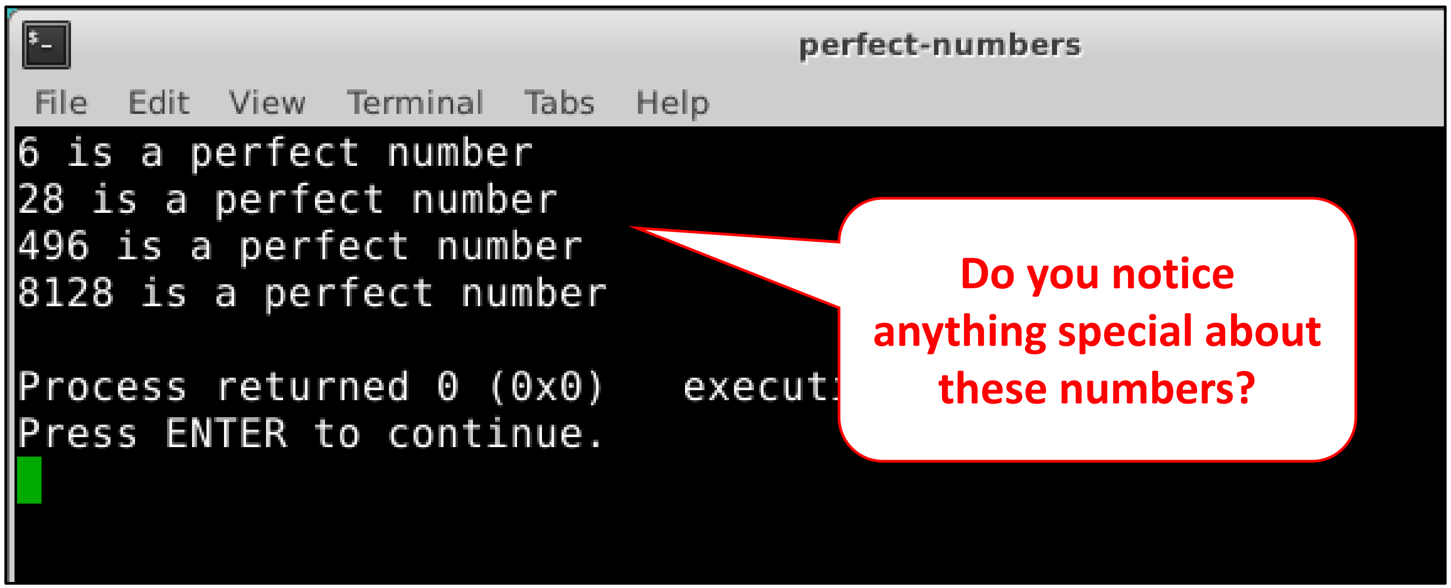
```
// perfect-numbers.cpp

#include "stdafx.h"

using namespace std;

int main()
{
    for (int n{ 2 }; n < 10000; ++n) {
        int sumOfFactors{ 1 };

        // Insert your code here

        if (sumOfFactors == n)
            cout << n << " is a perfect number"
                 << endl;
    }
    return 0;
}
```

Run Lab 1 – Perfect Numbers

perfect-numbers.cpp

```
// perfect-numbers.cpp

#include "stdafx.h"

using namespace std;

int main()
{
    for (int n{ 2 }; n < 10000; ++n) {
        int sumOfFactors{ 1 };

        for (int factor{ 2 }; factor < n; factor++)
            if (n % factor == 0)
                sumOfFactors += factor;

        if (sumOfFactors == n)
            cout << n << " is a perfect number"
                 << endl;
    }
    return 0;
}
```

Check Lab 1 – Perfect Numbers



Bonus points: Given a perfect number **n**, what is the **sum** of the **reciprocals** of all of its divisors (including **1** *and* **n**) ?

Perfect Numbers

Euclid–Euler theorem

$$n = 2^{(p-1)}(2^p - 1)\ is\ perfect$$

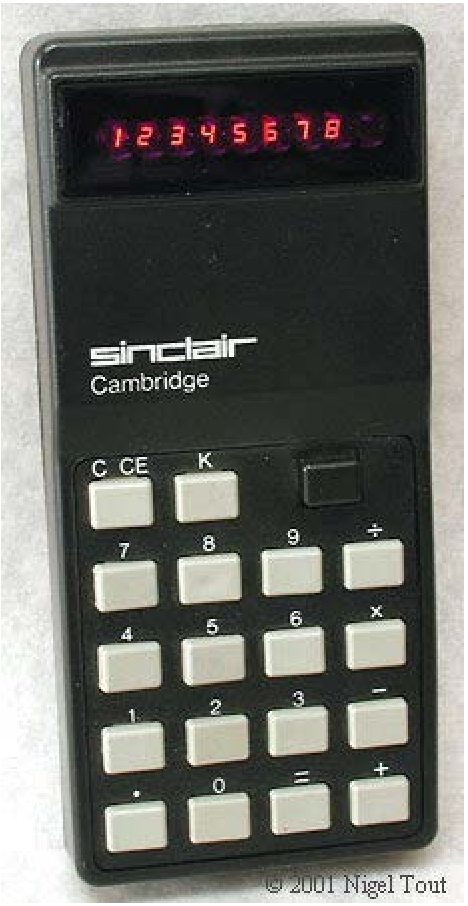
if and only if $\{p, (2^p - 1)\} \in primes$

p	2^p-1	n
2	3	6
3	7	28
5	31	496
7	127	8,128
11	2,047	~~2,096,128~~
13	8,191	33,550,336
17	131,071	8,589,869,056

2047 = 23 x 89

Old School Square Roots

- My first calculator back in 1977 could only add, subtract, multiply, and divide
- As a 6^{th} grader, I had heard of "Square Roots" and I knew that $\sqrt{25} = 5$.
- But what is $\sqrt{1977}$?
- How can we find the square root of a number using only the ***elementary*** (**+, -, *, /**) operations?
- Newton had solved that **313** years before me!

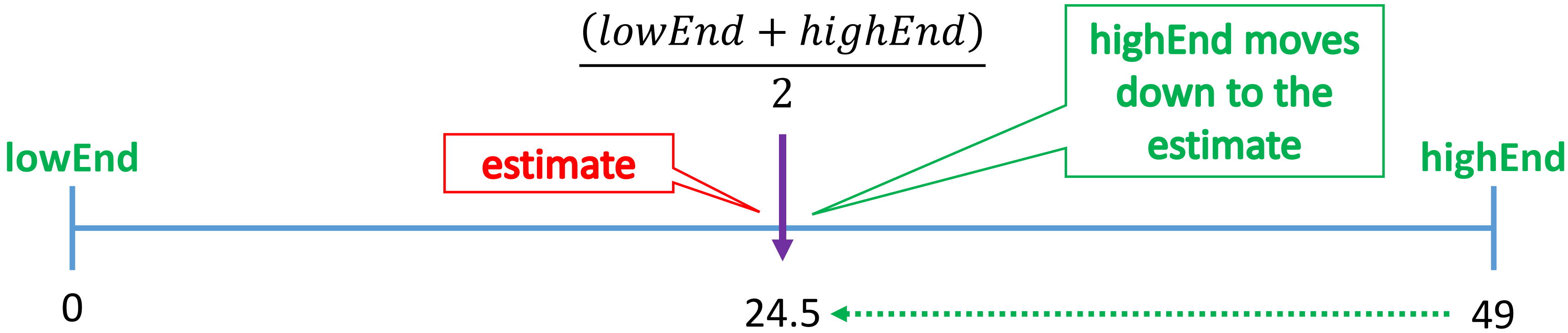


Old School Square Roots

- Newton's method for calculating the square root of any real **x** involves keeping track of a "low end" and "high end" bracket for the actual root
  - We start with the lowEnd = 0 and highEnd = **x**
  - The process **brackets inward** by keeping $\boxed{\text{lowEnd} \leq \sqrt{x} \leq \text{highEnd}}$
- During each loop iteration, our *estimate* is the **mean** of the current lowEnd & the highEnd values
  - Then if the $estimate^2$ > **x**, set highEnd = *estimate*
  - Alternatively, if the $estimate^2$ < **x**, set lowEnd = *estimate*
- Stop when the $|(estimate^2 - x)| \leq \varepsilon$

Newton's Method for $\sqrt{49}$

$\varepsilon = .0003$



$24.5^2 = 600.25$

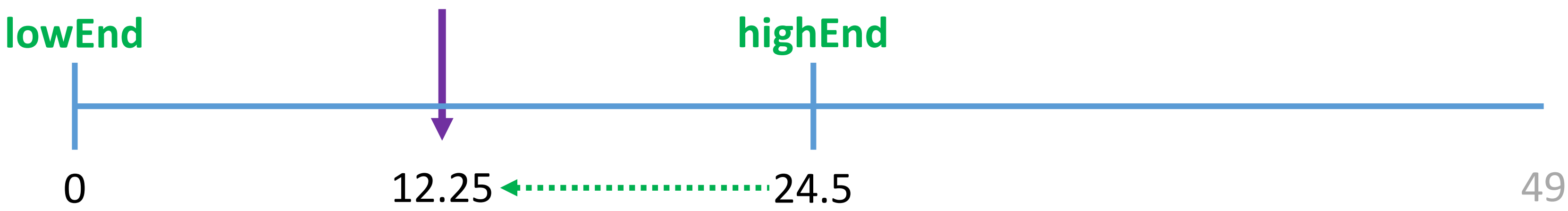
$600.25 > 49$

The estimate is too high!

Newton's Method for $\sqrt{49}$

$\varepsilon = .0003$

$$\frac{(lowEnd + highEnd)}{2}$$



$$12.25^2 = 150.0625$$

$$150.0625 > 49$$

The estimate is too high!

Newton's Method for $\sqrt{49}$

$\varepsilon = .0003$

$$\frac{(lowEnd + highEnd)}{2}$$

$$6.125^2 = 37.515$$

$$37.515 < 49$$

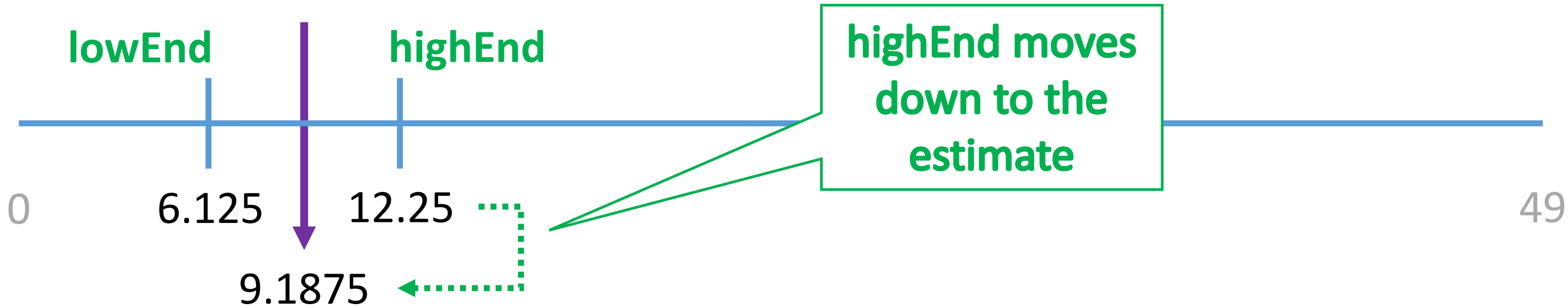
The estimate is too <u>low</u>!

lowEnd moves up to the estimate

Newton's Method for $\sqrt{49}$

$\varepsilon = .0003$

$$\frac{(lowEnd + highEnd)}{2}$$



$9.1875^2 = 84.410$

$84.410 > 49$

The estimate is too high!

Newton's Method for $\sqrt{49}$

$\varepsilon = .0003$

$$\frac{(lowEnd + highEnd)}{2}$$

lowEnd highEnd

0 6.125 9.1875 49

7.65625

$\boxed{7.65625}^2 = 58.618$

$58.618 > 49$

With each iteration we are closing in on the $\sqrt{49} = 7$

The estimate is too high!

Isaac Newton (1642-1726)

Newton's Method for $\sqrt{49}$

$\varepsilon = .0003$

lowEnd	highEnd	estimate	estimate^2	error	result
0.00000000	49.00000000	24.50000000	600.25000000	551.25000000	Too high
0.00000000	24.50000000	12.25000000	150.06250000	101.06250000	Too high
0.00000000	12.25000000	6.12500000	37.51562500	-11.48437500	Too low
6.12500000	12.25000000	9.18750000	84.41015625	35.41015625	Too high
6.12500000	9.18750000	7.65625000	58.61816406	9.61816406	Too high
6.12500000	7.65625000	6.89062500	47.48071289	-1.51928711	Too low
6.89062500	7.65625000	7.27343750	52.90289307	3.90289307	Too high
6.89062500	7.27343750	7.08203125	50.15516663	1.15516663	Too high
6.89062500	7.08203125	6.98632813	48.80878067	-0.19121933	Too low
6.98632813	7.08203125	7.03417969	49.47968388	0.47968388	Too high
6.98632813	7.03417969	7.01025391	49.14365983	0.14365983	Too high
6.98632813	7.01025391	6.99829102	48.97607714	-0.02392286	Too low
6.99829102	7.01025391	7.00427246	49.05983271	0.05983271	Too high
6.99829102	7.00427246	7.00128174	49.01794598	0.01794598	Too high
6.99829102	7.00128174	6.99978638	48.99700932	-0.00299068	Too low
6.99978638	7.00128174	7.00053406	49.00747709	0.00747709	Too high
6.99978638	7.00053406	7.00016022	49.00224307	0.00224307	Too high
6.99978638	7.00016022	6.99997330	48.99962616	-0.00037384	Too low
6.99997330	7.00016022	7.00006676	49.00093461	0.00093461	Too high
6.99997330	7.00006676	7.00002003	49.00028038	$\boxed{0.00028038}$	Too high

Open Lab 2 – Newton's Square Root

- Write a program to calculate the square root of a given **double** **x**, using only **elementary (+, -, *, /)** operations
- Use Newton's method to display the value of $\sqrt{\mathbf{168923}}$
- Use $\varepsilon = 1 \times 10^{-14}$
- Your current $estimate = \frac{(highEnd + lowEnd)}{2}$
- If your current $(estimate)^2 > x$ then your current $highEnd$ value must come **down** to $estimate$
- If your current $(estimate)^2 < x$ then your current $lowEnd$ value must come **up** to $estimate$

Edit Lab 2 - Newton’s Square Root

newton-sqrt.cpp

```
// newton-sqrt.cpp

#include "stdafx.h"

using namespace std;

int main()
{
    double x{ 168923 };

    double lowEnd{};
    double highEnd{ x };

    double estimate = (highEnd + lowEnd) / 2;
    double estimateSquared = pow(estimate, 2);

    double epsilon{ 1e-14 };

    while (abs(estimateSquared - x) > epsilon) {
        if (estimateSquared > x)
            highEnd =
        else
            lowEnd =

        estimate = (highEnd + lowEnd) / 2;
        estimateSquared = pow(estimate, 2);
    }

    cout << "Estimated Square Root of "
         << x << " = " << fixed
         << setprecision(14) << estimate
         << endl;

    return 0;
}

```

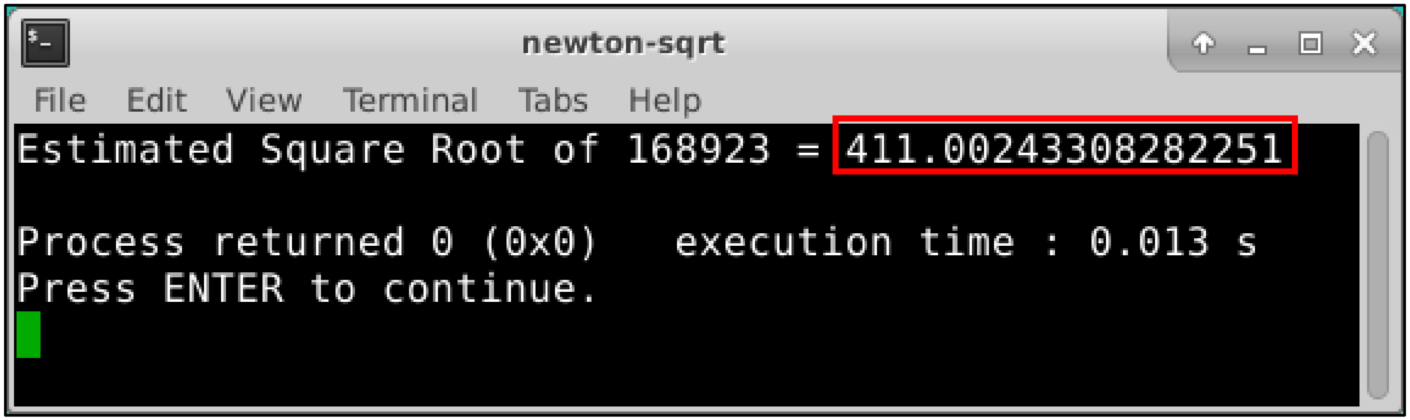
Fix these two lines of code

Run Lab 2 - Newton's Square Root

newton-sqrt.cpp

```cpp
// newton-sqrt.cpp

#include "stdafx.h"

using namespace std;

int main()
{
    double x{ 168923 };

    double lowEnd{};
    double highEnd{ x };

    double estimate = (highEnd + lowEnd) / 2;
    double estimateSquared = pow(estimate, 2);

    double epsilon{ 1e-14 };

    while (abs(estimateSquared - x) > epsilon) {
        if (estimateSquared > x)
            highEnd = estimate;
        else
            lowEnd = estimate;

        estimate = (highEnd + lowEnd) / 2;
        estimateSquared = pow(estimate, 2);
    }

    cout << "Estimated Square Root of "
         << x << " = " << fixed
         << setprecision(14) << estimate
         << endl;

    return 0;
}
```

Check Lab 2 – Newton's Square Root



Roots of Googol

- Create a program to calculate the **integer square root** of an number with **100** random digits
  - This is bigger than a **googol** (1×10^{100})
  - The program <u>can still</u> use **Newton's method** to calculate $\sim\lfloor\sqrt{x}\rfloor$
- Specifically, the code must compute the **mean** of two very large **Base 10** numbers represented as **vectors** of digits
  - The code will implement *column wise* addition and multiplication, just as you learned in grade school
  - The challenge is there is an **add()** & **multiply()** function available for big integers, but there is no **divide()** function ☺
  - The mean of a & b = (a+b)/2 = (a+b)*5/10 but where the "/10" is achieved by shifting all digits one position **to the right**!

Treating Large Integers as vector<int>

b	21,985
a	6,443

pos	00	01	02	03	04	05	06	07	08
vector<int> b	2	1	9	8	5				
vector<int> a	6	4	4	3					
Reverse b	5	8	9	1	2				
Reverse a	3	4	4	6					
Add	8	12	13	7	2				
Ripple	8	2	4	8	2				
Reverse	2	8	4	2	8				
Sum	28,428								
Multiply									
3 x 5,8,9,1,2	15	24	27	3	6				
4 x 5,8,9,1,2		20	32	36	4	8			
4 x 5,8,9,1,2			20	32	36	4	8		
6 x 5,8,9,1,2				30	48	54	6	12	
Add	15	44	79	101	94	66	14	12	
Ripple	5	5	3	9	4	6	1	4	1
Reverse	1	4	1	6	5	9	3	5	5
Product	141,649,355								

vectors are normalized so **b** is longer than **a**

vectors are reversed to easily align places

Treating Large Integers as vector<int>

b	**21,985**
a	**6,443**

pos	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**
vector<int> b	2	1	9	8	5				
vector<int> a	6	4	4	3					
Reverse b	5	8	9	1	2				
Reverse a	3	4	4	6					
Add	8	12	13	7	2				
Ripple	8	2	4	8	2				
Reverse	2	8	4	2	8				
Sum	**28,428**								
Multiply									
3 x 5,8,9,1,2	15	24	27	3	6				
4 x 5,8,9,1,2		20	32	36	4	8			
4 x 5,8,9,1,2			20	32	36	4	8		
6 x 5,8,9,1,2				30	48	54	6	12	
Add	15	44	79	101	94	66	14	12	
Ripple	5	5	3	9	4	6	1	4	1
Reverse	1	4	1	6	5	9	3	5	5
Product	**141,649,355**								

Columns are added straight downward

Carries are rippled forward

Final answer is reversed

Treating Large Integers as vector<int>

b	**21,985**
a	**6,443**

pos	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**
vector<int> b	2	1	9	8	5				
vector<int> a	6	4	4	3					
Reverse b	5	8	9	1	2				
Reverse a	3	4	4	6					
Add	8	12	13	7	2				
Ripple	8	2	4	8	2				
Reverse	2	8	4	2	8				
Sum	**28,428**								
Multiply									
3 x 5,8,9,1,2	15	24	27	3	6				
4 x 5,8,9,1,2		20	32	36	4	8			
4 x 5,8,9,1,2			20	32	36	4	8		
6 x 5,8,9,1,2				30	48	54	6	12	
Add	15	44	79	101	94	66	14	12	
Ripple	5	5	3	9	4	6	1	4	1
Reverse	1	4	1	6	5	9	3	5	5
Product	**141,649,355**								

Each element in **a** is multiplied individually by each element in **b**

Carries are rippled forward

Final answer is reversed

Open Lab 3 – Big Integer Square Root

bigint-sqrt.cpp

```
// bigint-sqrt.cpp

#include "stdafx.h"

using namespace std;

vector<int> five{ 5 };

vector<int>* getDigits(const string& s)
{
    size_t len = s.size();
    vector<int>* digits = new vector<int>(len);
    for (size_t i{}; i < len;++i)
        digits->at(i) = s.at(len - i - 1) - '0';
    return digits;
}

string makeString(const vector<int>* digits)
{
    string s{};
    for (size_t i{ digits->size() }; i > 0;--i)
        s += digits->at(i - 1) + '0';
    while (s.size() > 1 && s.at(0) == '0')
        s.erase(0, 1);
    return s;
}
```

This function creates a vector<int> from the digits of a string

This function creates a string from a vector<int> of digits

View Lab 3 – Big Integer Square Root

```
int main()
{
    seed_seq seed{ 2016 };
    default_random_engine generator{ seed };
    uniform_int_distribution<int> distribution(0, 9);

    string s = "1";
    for (int i{}; i < 99;++i)
        s += distribution(generator) + '0';

    cout << "The Integer Square Root of "
        << endl << endl
        << s << endl << endl
        << "is" << endl << endl;

    cout << intSqrt(getDigits(s))
        << endl << endl;

    return 0;
}
```

This code creates a “number” with 100 random digits

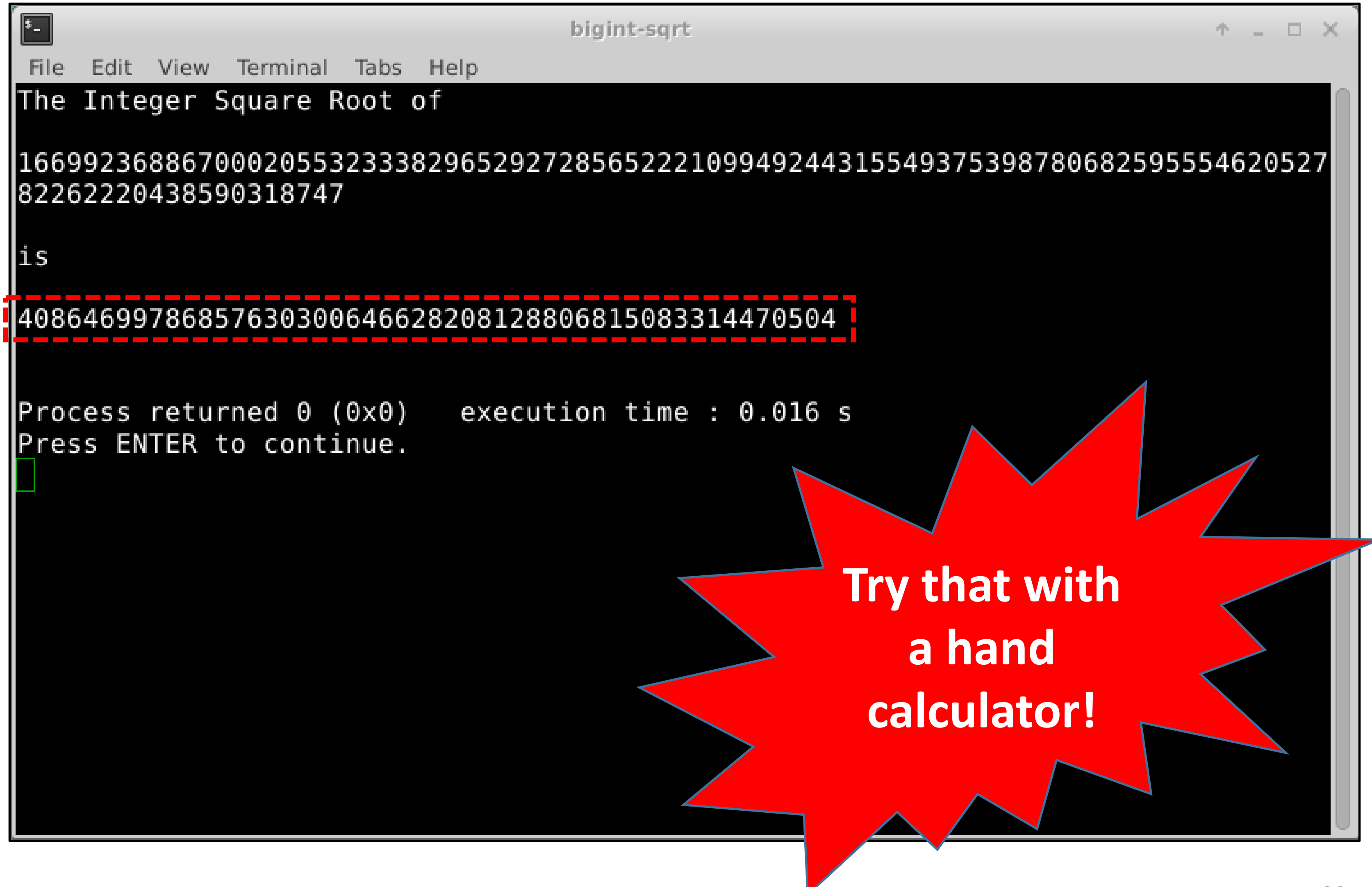
Run Lab 3 – Big Integer Square Root

```
string intSqrt(vector<int>* x)
{
    vector<int>* lowEnd = new vector<int>{ 0 };
    vector<int>* highEnd = x;

    vector<int>* lastEstimate = new vector<int>{ 0 };

    vector<int>* estimate = average(lowEnd, highEnd);

    while (!isEqual(lastEstimate, estimate))
    {
        vector<int>* estimateSquared = multiply(estimate, estimate);

        if (isGreater(estimateSquared, x))
            highEnd = estimate;
        else
            lowEnd = estimate;

        lastEstimate = estimate;
        estimate = average(lowEnd, highEnd);
        delete estimateSquared;
    }

    string s = makeString(estimate);
    delete estimate;
    delete highEnd;
    delete lowEnd;

    return s;
}
```

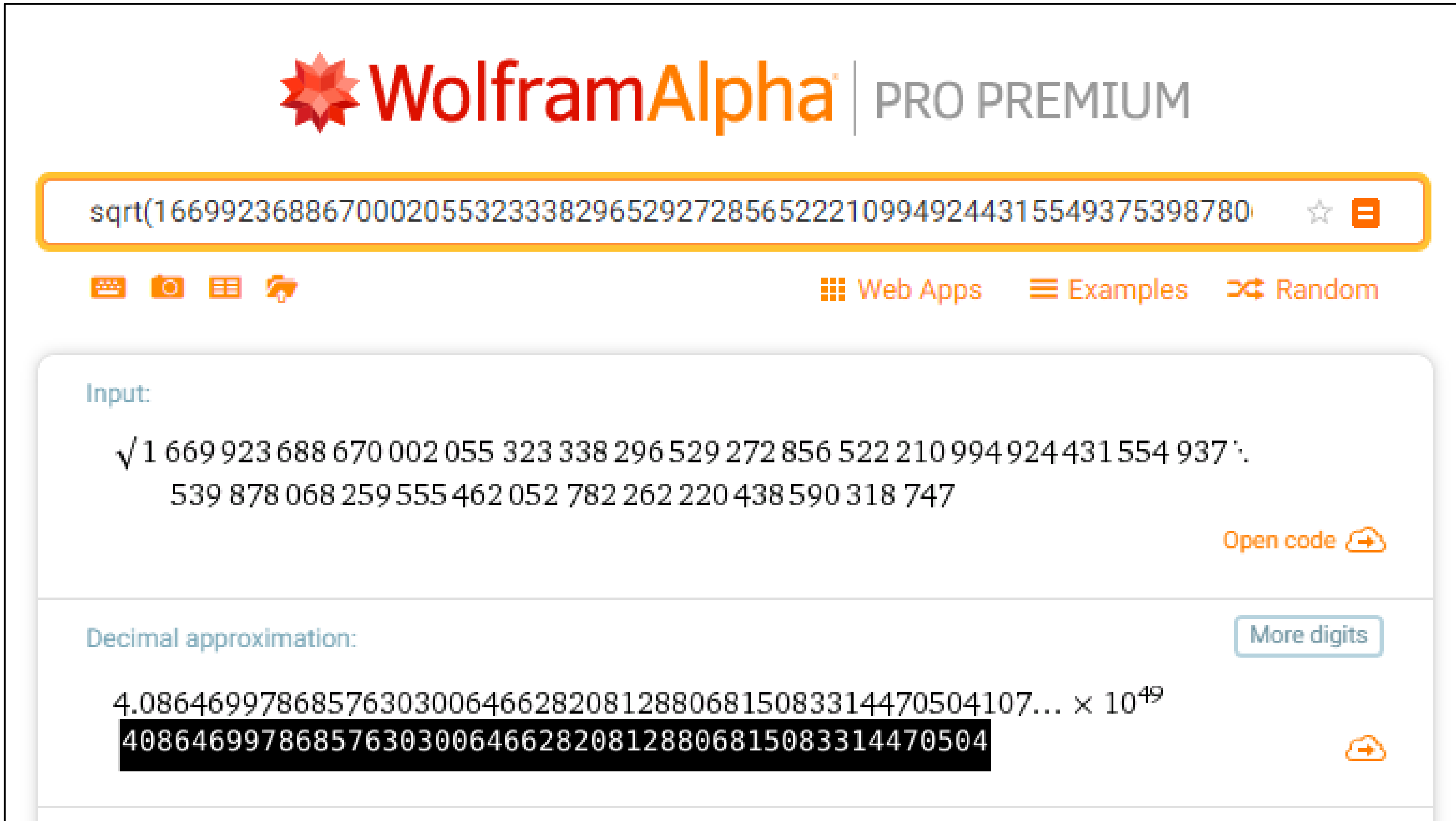
We can use the same algorithm!

```
vector<int>* average(vector<int>* x, vector<int>* y)
{
    vector<int>* z = multiply(&five, add(x, y));
    z->erase(z->begin());
    return z;
}
```

Check Lab 3 – Big Integer Square Root



Check Lab 3 – Big Integer Square Root



Representing a Quadratic Polynomial

- The Fundamental Theorem of Algebra shows a polynomial of degree **2** will have exactly **2** roots
  - $\boldsymbol{Jx^2 + Kx + L = 0}$, the roots can be unique or repeated
  - Either one (or both) of the roots can be a real or a complex number
- Assume we have factored a quadratic polynomial

$$(ax + b)(cx + d) = 0$$

$$(ac)x^2 + (ad + bc)x + (bd) = 0$$

$$\boldsymbol{J}x^2 + \boldsymbol{K}x + \boldsymbol{L} = 0$$

$$\boldsymbol{J} = (ac), \qquad \boldsymbol{K} = (ad + bc), \qquad \boldsymbol{L} = (bd)$$

Factoring a Quadratic Polynomial

$$\boldsymbol{J}x^2 + \boldsymbol{K}x + \boldsymbol{L} = 0$$

$$\boldsymbol{J} = (ac), \qquad \boldsymbol{K} = (ad + bc), \qquad \boldsymbol{L} = (bd)$$

- To factor J we need to try every integer $\boldsymbol{a}$ where 1 ≤ $\boldsymbol{a}$ ≤ J
  - If J % $\boldsymbol{a}$ == 0 (no remainder) then set $\boldsymbol{c}$ = J / $\boldsymbol{a}$
- To factor L we need to try every integer **b** where 1 ≤ **b** ≤ L
  - If L % $\boldsymbol{b}$ == 0 (no remainder) then set $\boldsymbol{d}$ = L / $\boldsymbol{b}$
- If $(\boldsymbol{ad} + \boldsymbol{bc}) = K$ then we have found a factorization!
  - If they do not equal K, then we have to keep trying more factors

Open Lab 4 – Factor Quadratic Polynomial

- Write a C++ console application to display **only** (but all) correct factorizations of a given quadratic polynomial

$$\boldsymbol{J}x^2 + \boldsymbol{K}x + \boldsymbol{L} = 0$$

- You may assume in all cases $\{J, K, L\} \in \mathbb{Z}^+$
- Please factor **this quadratic**:

$$\mathbf{115425}\boldsymbol{x}^{\mathbf{2}} + \mathbf{3254121}\boldsymbol{x} + \mathbf{379020}$$

Edit Lab 4 – Factor Quadratic Polynomial

factor-quadratic.cpp

```cpp
// factor-quadratic.cpp

#include "stdafx.h"

using namespace std;

int main()
{
    int J{ 115425 };
    int K{ 3254121 };
    int L{ 379020 };

    cout << "Given the quadratic:" << endl
         << J << "x^2 + " << K << "x + " << L
         << " = 0" << endl << endl
         << "The factors are:"
         << endl << endl;

    // TODO:  Insert your code here


    return 0;
}
```

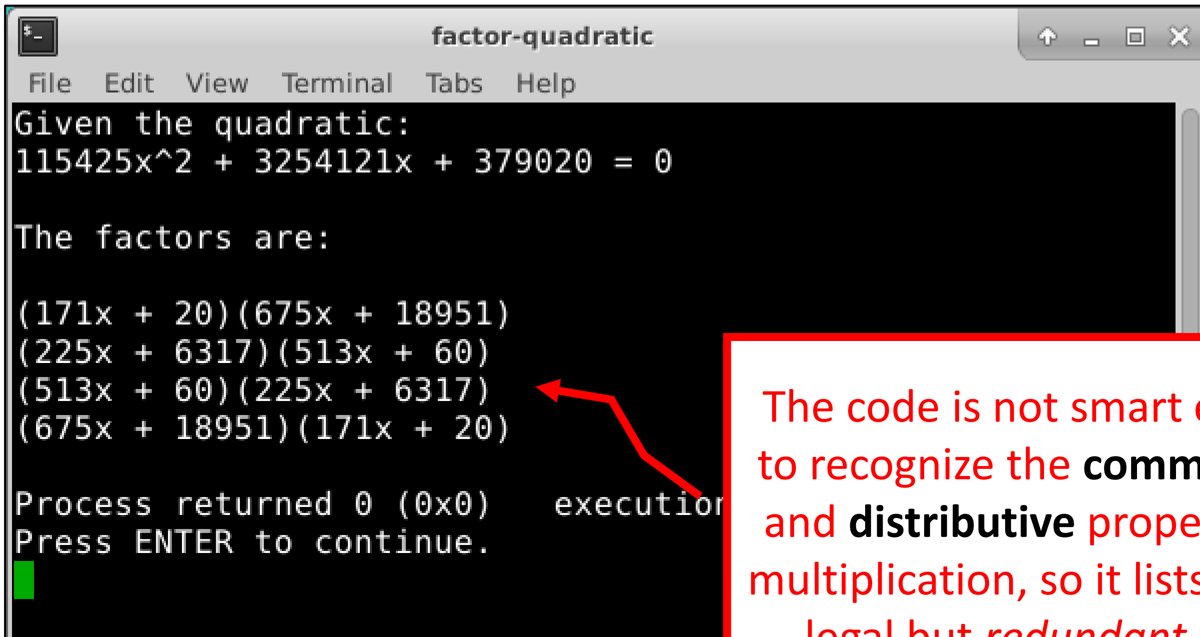
Run Lab 4 - Factor Quadratic Polynomial

factor-quadratic.cpp

```cpp
// factor-quadratic.cpp

#include "stdafx.h"

using namespace std;

int main()
{
    int J{ 115425 };
    int K{ 3254121 };
    int L{ 379020 };

    cout << "Given the quadratic:" << endl
         << J << "x^2 + " << K << "x + " << L
         << " = 0" << endl << endl
         << "The factors are:"
         << endl << endl;

    for (int a{ 1 }; a <= J; ++a) {
        if (J % a == 0) {
            int c = J / a;
            for (int b{ 1 }; b <= L; ++b) {
                if (L % b == 0) {
                    int d = L / b;
                    if (a*d + b*c == K) {
                        cout << "(" << a << "x + " << b << ")"
                             << "(" << c << "x + " << d << ")"
                             << endl;
                    }
                }
            }
        }
    }

    return 0;
}
```

Check Lab 4 – Factor Quadratic Polynomial

$$\mathbf{115425}x^2 + \mathbf{3254121}x + \mathbf{379020}$$



The code is not smart enough to recognize the **commutative** and **distributive** properties of multiplication, so it lists several legal but *redundant* roots

Edit Lab 4 – Factor Quadratic Polynomial

- What happens with a ***prime*** polynomial such as:

$$\mathbf{2}x^2 + \mathbf{14}x + \mathbf{3} \text{ ?}$$

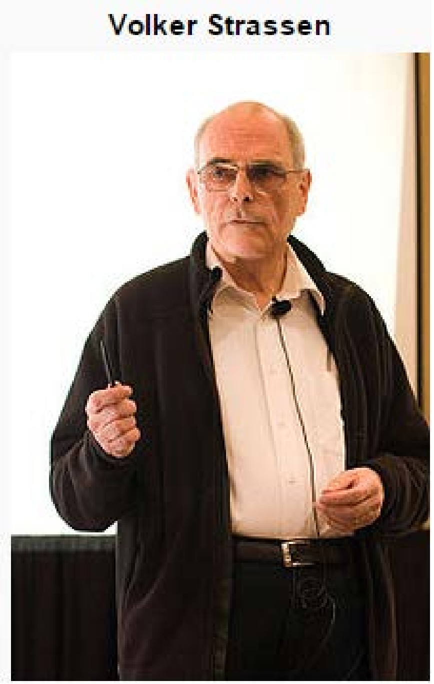
- The code as currently written can handle only ***positive*** coefficients - how could we strengthen the code to process ***negative*** coefficients?
- How could we avoid displaying simple commutative interchanges of the previously found factors?

Strassen's Method

- Multiplication is repeated addition, so a computer is **much faster at adding** two numbers than *multiplying* them
- From our first days in Algebra we are taught that you can only add "**like**" terms (those terms where each variable and exponent are the same)
- Hence we are taught that the **FOIL** method of expanding the product of two monomials requires **four (4)** multiplications: **first**, **outside**, **inside**, **last**:

$$(ax + b)(cx + d) = (ac)x^2 + (ad + bc)x + (bd)$$

Strassen's Method



- Volker Strassen showed in 1969 that you only need **three** (3) multiplications

$$(3x + 5)(7x + 9)$$

$$3 \ * 7 = 21$$

$$5 \ * 9 = 45$$

$$8 \ * 16 = 128$$

- The solution is then:

$$(21)x^2 + (\mathbf{128} \ - \mathbf{45} \ - \mathbf{21})x + (45)$$

$$21x^2 + 62x + 45$$

Strassen's Method

- We break the rules by **adding** the 3 + 5 = **8** and 7 + 9 = **16**, even though they are <u>not</u> like terms!

$$(3x + 5)(7x + 9)$$

$$3 \ * 7 = 21$$

$$5 \ * 9 = 45$$

$$\mathbf{8} \ * \mathbf{16} = 128$$

- Essentially **we *trade* <u>one</u> multiplication for <u>two</u> subtractions**

$$(21)x^2 + (128 \ - 45 \ - 21)x + (45)$$

$$21x^2 + 62x + 45$$

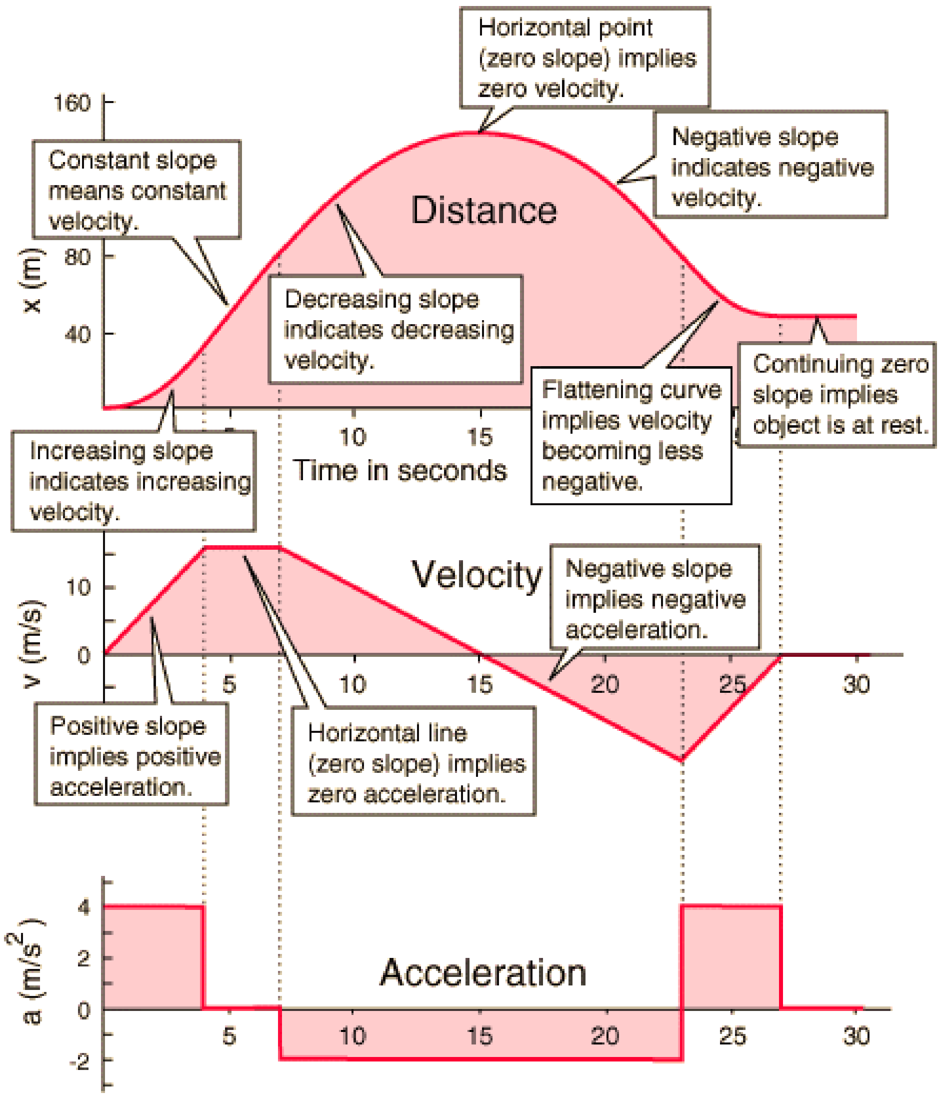
Strassen's Method

- When we cover matrix multiplication, you will find that the naïve approach requires $\mathbf{N^3}$ operations, so a 3 x 3 matrix multiply requires <u>27</u> multiplications
- Volker Strassen showed in his 1969 paper that the exponent is less than 3. In fact further improvements on Strassen's method has brought this down to $\mathbf{N^{2.375477}}$
- **Why is this important?** Because if you have really large matrices (think about solving **1,000** equations with **1,000** unknowns) the difference adds up *quickly*
- With **N = 1000**, Strassen's method is **74x faster!** (not just merely 74% faster) than the naïve approach to matrix multiplications

Why do we need integrals?

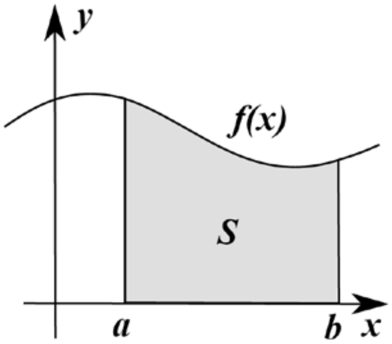
- How to calculate the ***total*** change in a variable X
  - When variable X *depends* on the changes in variable Y…
  - … and variable Y *depends* on the changes in variable Z…
  - … and variable Z is <u>constantly</u> changing…
- Think about an accelerating car and the total distance it will travel in a given number of seconds
  - The total distance *depends* on the velocity of the car…
  - … the velocity of the car *depends* on the acceleration
  - … and the acceleration is <u>constantly</u> changing

Why do we need integrals?



Why do we need integrals?

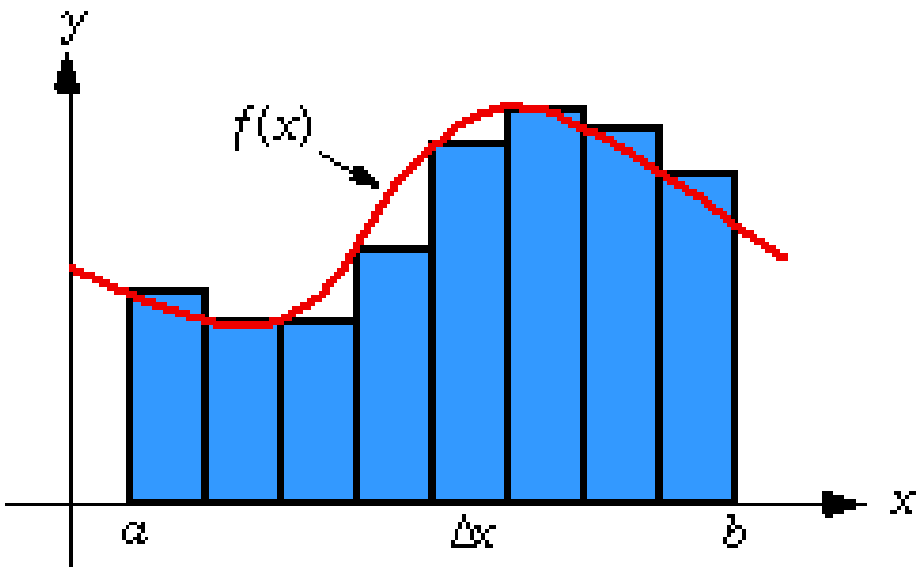
- The **integral** of a function can be defined as the **area under a curve** f(x) within the region [a,b]



- There are ways to often determine exactly the value of the integral of **f(x)** which we would write $\mathbf{F(x) = \int_a^b f(x)}$
- However, sometimes it is not possible to find an analytic expression for **F(x)** – so we use ***numerical* integration**

Riemann Sums

- One way we can integrate f(x) is to divide the area under the curve into strips (**intervals**) and sum the area of each strip
- This estimate may not be totally accurate because we might have **gaps** between the true value of f(x) and the top of a strip

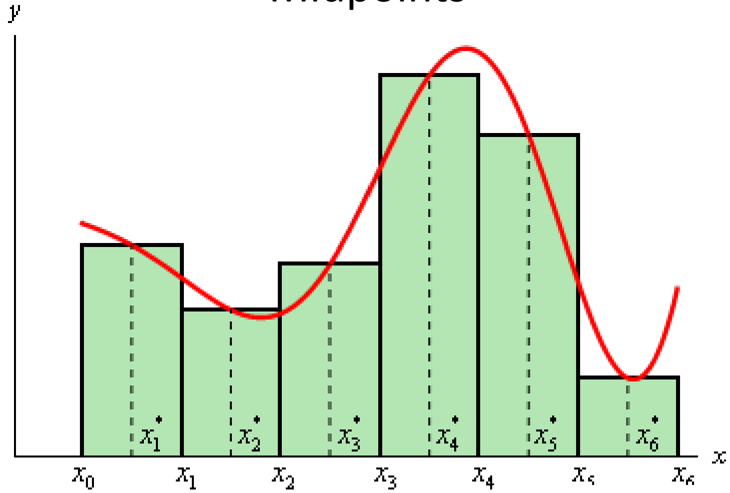


Riemann Sums

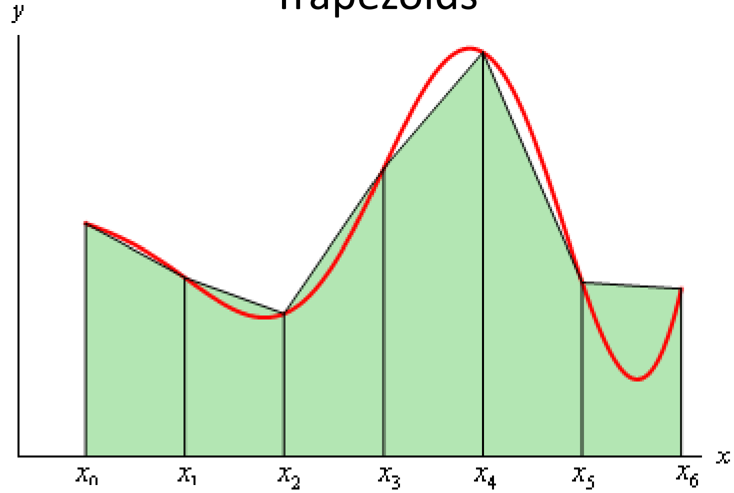
- The width of each strip is $\boldsymbol{\Delta x} = \frac{(\boldsymbol{b}-\boldsymbol{a})}{\#\ \boldsymbol{of\ intervals}}$
- We can minimize the gaps by increasing the number of intervals, which makes the $\boldsymbol{\Delta x}$ ***smaller***
- There are different strategies for determine the shape and height of each strip
  - Left-hand Rule, Right-hand Rule, Midpoint Rule
  - Fit Trapezoids
  - Fit Parabolas (Simpson's Rule)
- Depending upon the particular shape of $\boldsymbol{f(x)}$, one method might be more accurate than the others

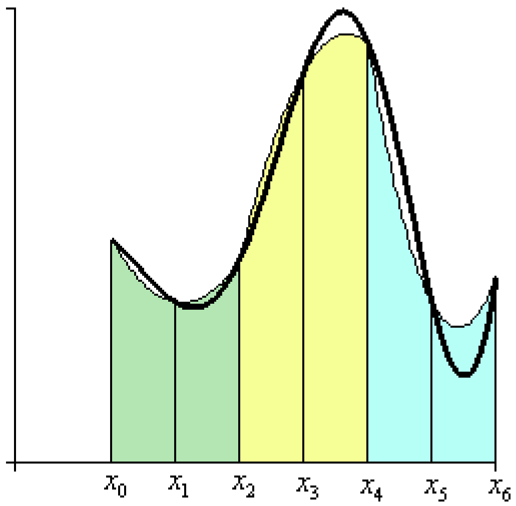
Riemann Sums

Midpoints



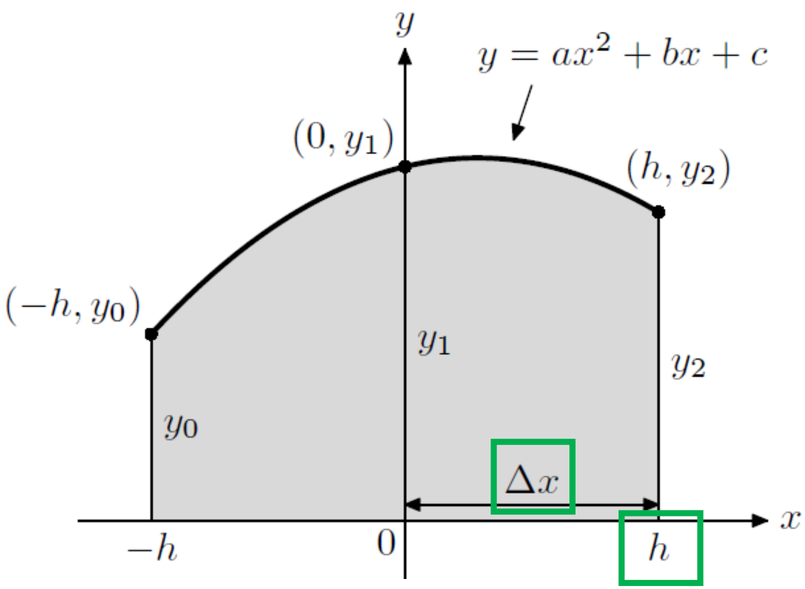
Trapezoids





Parabolas
(Simpson's Rule)

Simpson's Rule is more accurate!



$$A = \int_{-h}^{h} (ax^2 + bx + c)\ dx$$
$$= \left(\frac{ax^3}{3} + \frac{bx^2}{2} + cx \right) \Bigg|_{-h}^{h}$$
$$= \frac{2ah^3}{3} + 2ch$$
$$= \frac{h}{3} \boxed{(2ah^2 + 6c)}$$

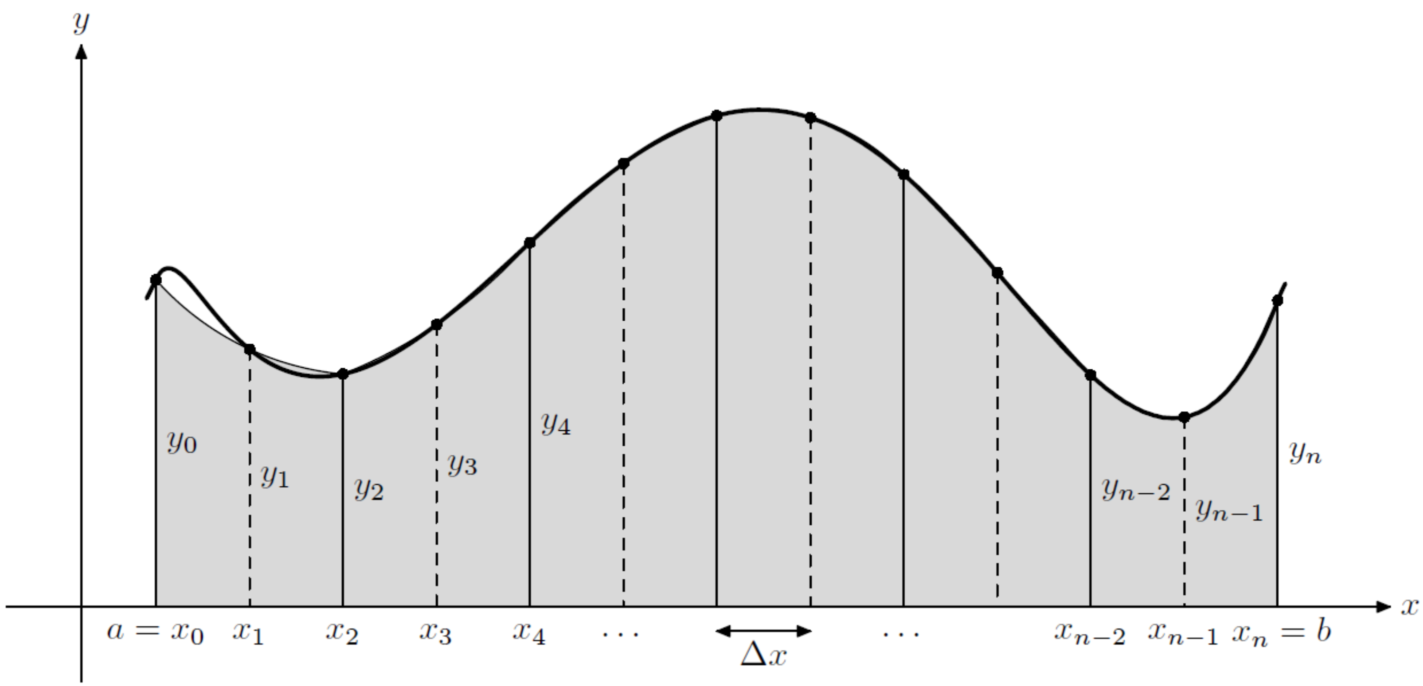
$$A = \frac{h}{3} \boxed{(y_0 + 4y_1 + y_2)} = \boxed{\frac{\boxed{\Delta x}}{3} (y_0 + 4y_1 + y_2)}$$

$$y_0 = ah^2 - bh + c$$
$$y_1 = c$$
$$y_2 = ah^2 + bh + c$$

$$\boxed{y_0 + 4y_1 + y_2} = (ah^2 - bh + c) + 4c + (ah^2 + bh + c) = \boxed{2ah^2 + 6c}$$

Simpson's Rule is more accurate!

$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \ldots, \quad y_n = f(x_n).$$



$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_{n-1} + y_n\right)$$

Simpson's Rule is more accurate!

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_{n-1} + y_n\right)$$

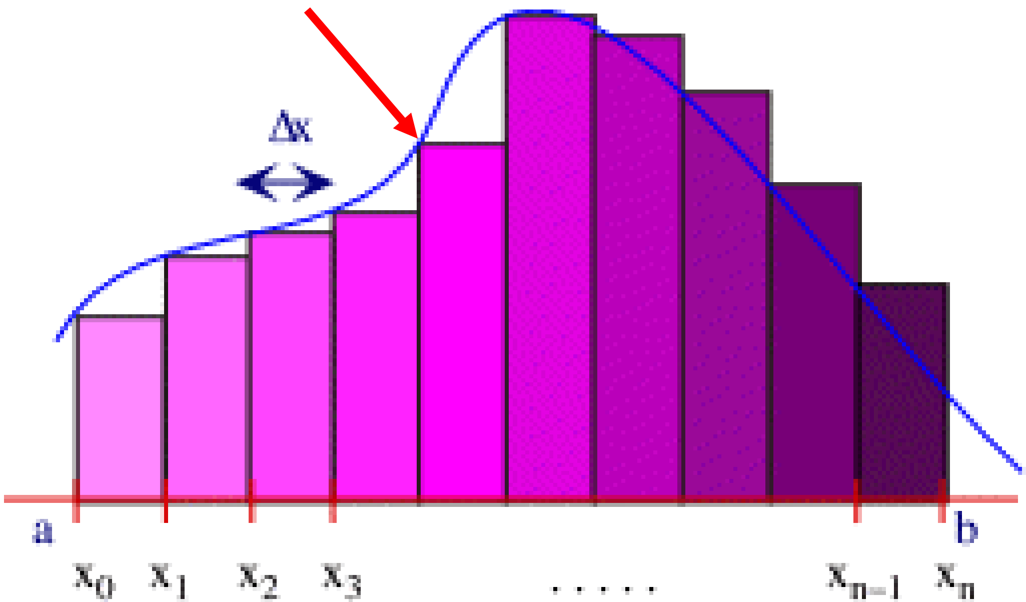
Point	y0	y1	y2	y3	y4	y5	y6
Coeff	1	4	2	4	2	4	1

The first and last point have coefficient = 1
Every point with an odd index has coefficient = 4
Every point with an even index has coefficient = 2

```
double simpsons(double a, double b)
{
    const double dx{ (b - a) / intervals };
    double sum{ f(a) + f(b) };
    a += dx;
    for (int i{ 1 };i < intervals;++i, a += dx)
        sum += f(a)*(2 * (i % 2 + 1));
    return (dx / 3) * sum;
}
```

Open Lab 5 – Simpson's Rule

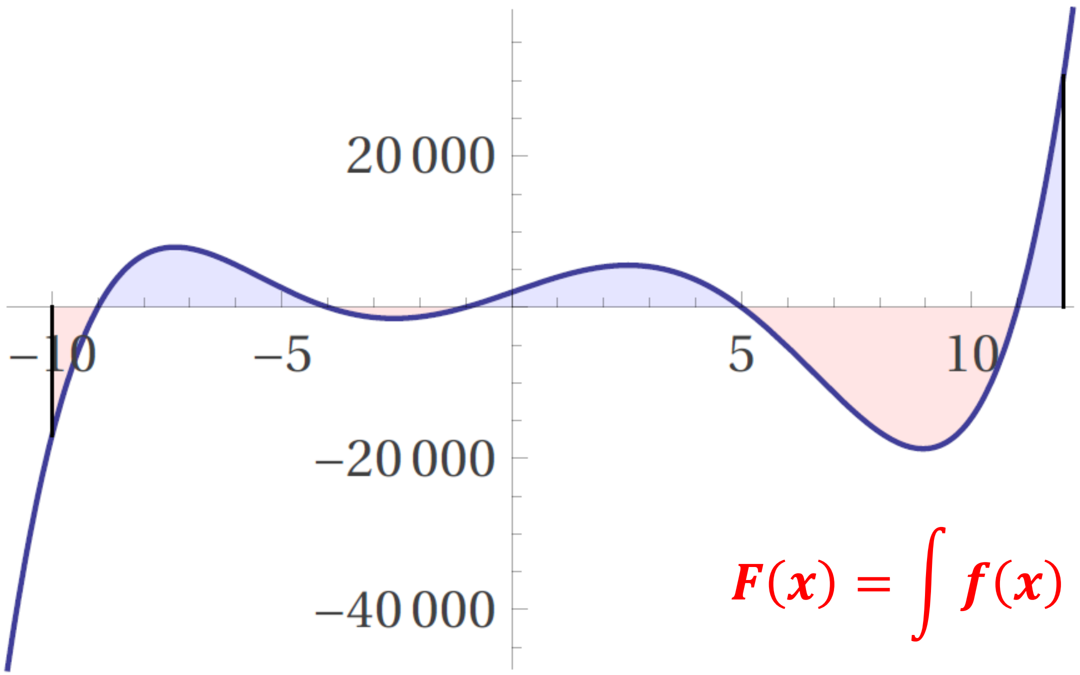
- Compare the percent error in the estimate of the integral provided by the **left-hand rule** vs. **Simpson's rule**



View Lab 5 – Simpson's Rule

$$y = f(x) = x^5 - 2x^4 - 120x^3 + 22x^2 + 2119x + 1980$$

$$= (x+9)(x+4)(x+1)(x-5)(x-11)$$



$$\boldsymbol{F(x) = \int f(x) = ?}$$

View Lab 5 – Simpson's Rule

- Perform ***numerical*** integration with respect to x using a <u>million</u> intervals on the following polynomial over the domain $[\mathbf{-10, 12}]$:

$$F(x) = \int_{-10}^{12} (x^5 - 2x^4 - 120x^3 + 22x^2 + 2119x + 1980)\,dx$$

$$F(x) = \frac{x^6}{6} - \frac{2x^5}{5} - 30x^4 + \frac{22x^3}{3} + \frac{2119x^2}{2} + 1980x$$

$$F(x) = \left[\frac{-174744}{5} - \frac{-43550}{3}\right] = \frac{-306482}{15} = \mathbf{-20432.13333}$$

Run Lab 5 - Simpson's Rule

simpsons-rule.cpp

```cpp
// simpsons-rule.cpp

#include "stdafx.h"
using namespace std;

const double a{-10};
const double b{12};
const int intervals = 1e6;

inline double f(double x)
{
    return (x+9)*(x+4)*(x+1)*(x-5)*(x-11);
}

int main()
{
    cout.imbue(locale(""));

    cout << "Integrating "
         << "x^5 - 2x^4 - 120x^3 + 22x^2 + 2199x +1980"
         << endl << " over [" << a << ", " << b << "]"
         << " using " << intervals << " intervals:"
         << endl << endl;

    double i1{-306482./15};
    cout << "Analytic (Exact): "
         << fixed << setprecision(14)
         << i1 << endl << endl;

    double i2{lefthand(a,b)};
    cout << "Left-hand Rule   : "
         << fixed << setprecision(14)
         << i2 << endl
         << scientific << setprecision(4)
         << "% Error = " << (i2-i1)/i1
         << endl << endl;

    double i3{simpsons(a,b)};
    cout << "Simpson's Rule   : "
         << fixed << setprecision(14)
         << i3 << endl
         << scientific << setprecision(4)
         << "% Error = " << (i3-i1)/i1
         << endl << endl;

    return 0;
}
```

```cpp
double lefthand(double a, double b)
{
    const double dx{(b-a)/intervals};
    double sum{};
    while(a<=b)
    {
        sum+=f(a);
        a+=dx;
    }
    return sum*dx;
}

double simpsons(double a, double b)
{
    const double dx{(b-a)/intervals};
    double sum{f(a)+f(b)};
    a+=dx;
    for(int i{1}; i<intervals; ++i,a+=dx)
        sum+=f(a)*(2*(i%2+1));
    return (dx/3)*sum;
}
```

Check Lab 5 – Simpson's Rule

simpsons-rule

File Edit View Terminal Tabs Help

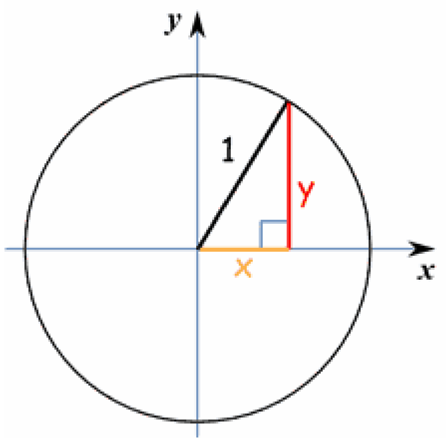
```
Integrating x^5 - 2x^4 - 120x^3 + 22x^2 + 2199x +1980
 over [-10, 12] using 1,000,000 intervals:

Analytic (Exact): -20,432.133333333333503

Left-hand Rule  : -20,432.65677902821335
% Error = 2.5619e-05

Simpson's Rule  : -20,432.13333371190674
% Error = 1.8528e-11


Process returned 0 (0x0)   execution time : 0.071 s
Press ENTER to continue.
```

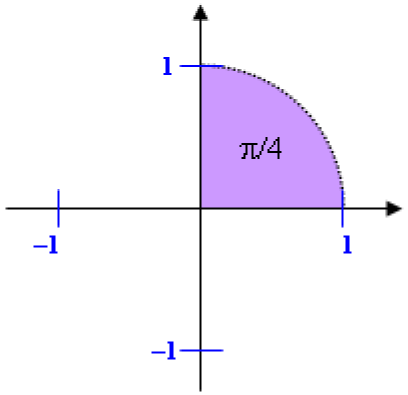
Open Lab 6 – Circle Area

- Specify in the code the correct $f(x)$, limits $[a, b]$, and exact analytic value for **the area of a unit circle:**

$$F(x) = 4\int_{0}^{1} \sqrt{1 - x^2}\,dx$$

Note: in C++ the constant $\mathbf{M\_PI} = \pi$





Edit Lab 6 – Circle Area

circle-area.cpp

```cpp
// circle-area.cpp

#include "stdafx.h"
using namespace std;

const double a{0};
const double b{-1};
const int intervals = 1e6;

inline double f(double x)
{
    return 0;
}

double lefthand(double a, double b)
{

double simpsons(double a, double b)
{

int main()
{
    cout.imbue(locale(""));

    cout << "Integrating "
         << "4 * sqrt(1-x^2)"
         << endl << " over [" << a << ", " << b << "]"
         << " using " << intervals << " intervals:"
         << endl << endl;

    double i1{0};
    cout << "Analytic (Exact): "
         << fixed << setprecision(14)
         << i1 << endl << endl;

```

Run Lab 6 – Circle Area

circle-area.cpp

```cpp
// circle-area.cpp

#include "stdafx.h"
using namespace std;

const double a{0};
const double b{1};
const int intervals = 1e6;

inline double f(double x)
{
    return sqrt(1-x*x);
}

double lefthand(double a, double b)
{

double simpsons(double a, double b)
{

int main()
{
    cout.imbue(locale(""));

    cout << "Integrating "
         << "4 * sqrt(1-x^2)"
         << endl << " over [" << a << ", " << b << "]"
         << " using " << intervals << " intervals:"
         << endl << endl;

    double i1{M_PI};
    cout << "Analytic (Exact): "
         << fixed << setprecision(14)
         << i1 << endl << endl;

```

Check Lab 6 – Circle Area

circle-area

File Edit View Terminal Tabs Help

```
Integrating 4 * sqrt(1-x^2)
 over [0, 1] using 1,000,000 intervals:

Analytic (Exact): 3.14159265358979

Left-hand Rule  : 3.14159465240248
% Error = 6.3624e-07

Simpson's Rule  : 3.14159265311902
% Error = -1.4985e-10


Process returned 0 (0x0)   execution time : 0.044 s
Press ENTER to continue.
```

Now you know…

- An **algorithm** is a recipe, often with loops, that changes inputs to outputs
- There are many **simple to state**, but hard to *solve*, open problems in **number theory**
- It is not known if there are any **odd** perfect numbers
- It is not known if there are *infinitely* many perfect numbers
- The **bool** data type to store true or false values
- Use the **if()** statement for ***conditional code execution***
- The **if()** statement introduces a scope {}, and can have an optional **else** {} scope
- The **while()** is like an **if()** statement that loops
- The **while()** loop a simplified **for()** loop

Now you know…

- The **%** operator returns the **remainder**
- Use **double** equals **==** operator to test for equality
- Use single equal **=** to define the value of a variable
- The **&&** operator performs a **logical AND** of two Boolean values
- Numerical Integration finds **the area under the curve** using **successively smaller** and smaller strips
- The strips can be sized according to the Left, Right, Trapezoid, or Midpoint rules
- Simpson's method is the more accurate due to fitting parabolas