**x^y**

//Need to add exception to be thrown when negative base is used with non-integer powers //and -integer powers as those are not defined in the set of real numbers

//Can just use an absolute tolerance for non integer exponents (I initially forgot how exponents work but solution of c=a^b where 0<b<1 always has 1<c<a )

//check overlap with nth power and negative integer exponents (may be better to implement an nth root algorithm )

public class powFuncs{

public static double getPow (double x double y) {

//check if exponent is negative. If negative find solution using positive //exponent and return 1 divided by that solution

if (y<0){

return 1/getPow(x (-1)\*y)

}

y\_int=getIntPart(y);

y\_fract=getFractPart(y y\_int);

if (y\_fract==0){

return intPow(x yint)

//non integer powers can be decomposed into the product of their integer and //fractional powers

}else{

return intPow(x y\_int)\*fractPow(x y\_fract)

}

}

public static int getIntPart(double y){

y\_int=(int) y; //need to confirm casting does not round

return y\_int

}

public static double getFractPart(double y int y\_int){

y\_fract=y-y\_int;

return y\_fract

}

public static double intPow (double x int y){

//This method performs integer exponentiation using the method of //exponentiation by squaring (recursive version)

if ( y==0){

return 1;

}else if((y%2)==1){

return x\*intPow(x,y-1)

}else{

return x\*intPow(x,y/2)

}

}

public static double fractPow (double x double y\_fract){

//Use algorithm base on identity x^y\_fract=e^(y\_fract\*log(x))

//tolerance set to 1\*10^(-12)

eps=1\*10^(-12)

//find taylor expansion approximation of e^(x\*log(y\_fract)) with tolerance eps

approx=0;//taylor expansion approximation

taylor\_n=1; //first term in expansion is 1

n=0

while (taylor\_n>=epsilon){

approx=approx+taylor\_n;

++n

taylor\_n=1/factorial(n)\*intPow(y\_fract\*log(x) n);

}

return taylor\_n

}

}

---------------------------------------------------------------------------------------------------------------------------

**nth^root**

//Use modularity and use getIntPart and getFracPart in powFuncs

//Using absolute tolerance but if we want to work on really small numbers a relative tolerance //may be better

public class rootFunc{

public static double getRoot(double x int n) {

//Check if x is positive

if (x<0) {

//return error message

}

//check if root is positive and integer

n\_int=getIntPart(n);

n\_fract=getFractPart(n n\_fract);

if (n<0){

//return error message

} else if (n\_fract != 0){

//return error message

}

}

//make guess x0

if (x>=1){

x0=x/n;

} else{ //roots of number x less than 1 are larger than x

xo=x\*n;

}

//set tolerance to 1\*10^(-12)

eps=1\*10^(-12);

x\_k=x0;

delta\_xk=x0;

count=0;

while ( abs(delta\_xk)<eps) {

++count

delta\_xk=1/n\*(x/getPow(x\_k (n-1)) - x\_k;

x\_k=x\_k+delta\_xk;

}

return x\_k

}

---------------------------------------------------------------------------------------------------------------------------

**log x**

//Priority is natural logarithm as above functions need it

//Need to extend to include log base 10

public class rootFunc{

public static double getNatLog(double x) {

//use borchardt's algorithm

a0=(1+x)/2;

b0=getRoot(x 2);

ak=a0;

bk=b0;

//set tolerance to 1\*10^(-12)

eps=1\*10^(-12);

while (abs(ak-bk)<eps){

ak=ak+bk/2;

bk=getRoot( (ak\*bk) 2)

}

log\_x=2\*(x-1)/(ak+bk);

}

}