# 2 - Write programs to calculate numeric quantities

Suppose we have some formula or procedure to calculate a value. If this procedure involves repeated calculations, we know that we can’t go on calculating forever, and round-off error may accumulate as we do repeated calculations. Thus, we need to limit the calculations to achieving a desired method. Following this procedure is called an “iterative method” for calculating a value (iterative is just a fancy word for looping or repeating).

## General algorithm for iterative methods

Any iterative method will have the following form:

Start with an initial approximation

While the approximation is not “close enough”

Find a better approximation

Return the last approximation

What we need to do to calculate specific values is determine:

1. What the initial approximation will be
2. What does it mean to be “close enough”?
3. How do we find a better approximation

Also, in order for such a method to work, the approximations must converge, or get closer and closer to the correct answer.

## Example 1: Exponential Function

We have already seen an example of an iterative method when we calculated the special number *e* (Euler’s number).

Here we have an initial approximation (0 or 1), and a method of getting a better approximation (add the next term). So far, we just did a number of loops to determine when we were “close enough”, but we can do a calculation of the difference between successive values to determine the desired precision.

This process can be extended to the *exponential function*, which is *ex* for some desired power *x*. The formula for this is just a variation of the formula for *e*:

Note the summation notation symbol (the capital Greek letter Σ) is just a way of expressing the terms that we are adding in a formula; each term is *xk/k!* for a value of k varying from 0 to ∞.

So our initial approximation is 0 and then we add terms starting at *k* = 0; or our initial approximation is the first term, 1, and then we add terms starting at *k* = 1; or our initial approximation is 1 + *x*, and we add terms starting at *k* = 2. (We could go farther, but once we start calculating powers, it’s better to do that in a loop.)

Our method of getting a better approximation is to add a new term *xk*/*k*!. Because factorials grow faster than powers (at some point), the calculation is guaranteed to converge because we will be adding smaller and smaller numbers.

Finally, to check if we are “close enough”, we can check the difference between successive approximations (which is just the new term added) against our desired precision. This can be done using absolute error using the amount of error, or absolute relative error comparing the amount of error with the current calculated value.

See **“Iterative method for calculating exponential function.xlsx”** for an example of the calculations involved. Note that Excel has a built-in EXP function for calculating the exponential function, so we can test our value against the best possible result. When coding, we can start with a naïve version:

**public** **static** **double** myExponentialFunction( **double** x, **double** dPrecision )

{

**double** dResult = 0.0; // initial approximation

**double** dOldResult = 0.0;

**int** iCount = 0;

**do**

{

// keep track of the previous result

dOldResult = dResult;

// add next term x^k / k! for better approximation

dResult += Math.*pow*(x, iCount) / *dFact*(iCount);

iCount++;

} **while** ( Math.*abs*((dResult - dOldResult) / dResult) > dPrecision );

// "close enough" using relative error

**return** dResult; // last approximation

}

Note that if you just use the amount of error (dResult - dOldResult), the calculations will work until you try negative values of *x*, when the amount of the term added could be a negative number. Thus, you need to use the Math.*abs* function. The calculations will also have trouble with large values of *x* if you just use the amount of error (because it will be seeking precision to a certain decimal place rather than a relative percentage).

Testing the code with various values of *x* will show problems with large values of *x* (like 650, which will show up as Infinity) and negative values of *x* (like -40, which is quite far off the right answer).

Thus, the code can be improved in several ways. We can start off with a better approximation by adding the first two terms instead of starting at 0. For negative values of *x*, we can calculate 1/*e-x* since a power to a negative value is just 1 divided by *e* to the corresponding positive value. This will stop us from having to worry about negative terms (subtractions) being done in the calculations. Rather than calculating *xk* and *k*! each time through the loop (which can become very large numbers), we can just note that each term is *x* / *k* times the previous term (multiplying by *x* increased the top part to *xk* and dividing each term by *k* increases the size of the factorial in the bottom part). Finally, we don’t need to track the old result to get the error, since the difference is always just the term being added, so we can just look at the size of that term.

**public** **static** **double** myImprovedExp( **double** x, **double** dPrecision )

{

**double** dResult = 1.0 + x; // initial approximation - first two terms

**double** dTerm = x; // the current term of the series being added

**int** iCount = 1;

**if** (x < 0) // for negative x, calculate 1/e^x to avoid subtractions

{

**return** 1.0 / *myImprovedExp*( -x, dPrecision );

}

**do**

{

iCount++;

// new term is just old term \* x / iCount

dTerm = dTerm \* x / iCount;

dResult += dTerm; // add the new term to get a better approximation

} **while** ( Math.*abs*(dTerm / dResult) > dPrecision );

// use relative error (error is just last term, divide by result)

**return** dResult;

}

This improved version will handle large values of *x* and negative values of *x* – at least, as well as a **double** can!

## Example 2: Sine function

Any infinite series can be treated in the same way as the exponential function. Let’s examine *sin*(*x*) for some angle *x* in radians.

First, a reminder of what radians are: they are a measure of an angle by the number of radiuses that the angle would cut out of the circumference of a circle around the angle. So instead of 360 degrees in a circle, we have the full circumference of the circle, which is π \* *diameter* = π \* 2 \* *radius* = 2π radians.

Similarly:

180° = π radians

90° = π / 2 radians

45° = π / 4 radians

30° = π / 6 radians

And in general: *angle*° = *angle* \* π / 180 radians

For an angle *x* in radians, the sine of the angle is:

https://wiki.ubc.ca/images/math/4/4/7/447a79826774707026bbefcd76962d3a.png

Thus we have an initial approximation (the first term, *x*), a way of getting a better approximation (adding the next term), and a method of checking if we are close enough (using successive approximations).

See **“Iterative method for calculating sine function.xlsx”** for an example of the calculations involved. Note that Excel has a built-in SIN function for calculating the sine function, so we can test our value against the best possible result. When coding, we can start with a naïve version:

**public** **static** **double** myNaiveSin( **double** dAngle, **double** dPrecision )

{

**double** dResult = dAngle; // first approximation is x^1/1! = x

**int** iCount = 0;

**double** dOldResult;

**do**

{

dOldResult = dResult;

iCount++;

// add another term

dResult += Math.*pow*(-1.0, iCount) \* Math.*pow*(dAngle, 2 \* iCount + 1)

/ *dFact*(2 \* iCount + 1);

} **while** (Math.*abs*(dResult - dOldResult) > dPrecision); // "close enough"

**return** dResult;

}

The naïve version will not handle large angles (like 3600° = 20π radians) because the values used in calculating the new term are initially too big, and round-off error prevents us from getting the desired better results.

We can then improve that version in a couple of ways. First, instead of trying to take the sin of a large number which just represents several times around the circle (for instance, 20π is just 10 times around the circle, ending up at an angle of 0!), we can take the remainder of the number divided by 2π. This will leave us with a correct angle that never goes around the circle, and stays in a range (0 to 2π) that can be calculated.

Second, rather than calculating the powers and factorials of large numbers, we can just use the relationship between terms: the new term is just -1 \* *x*2 / (2*k* \* (2*k* + 1)) – in other words, we multiply by -1 to change the sign, multiply by *x*2 to increase the numerator, and divide by the two new factors to increase the denominator.

Third, the difference between successive terms is just the value of the last term, so we don’t need to track the old result and subtract it from the new result. We can just look at the amount of the last term.

See **“IterativeMethodsExamples.java”** for examples of both the exponential function and the sin function, or **“SinFunction.java”** for examples of the sin function alone.

**public** **static** **double** myImprovedSin( **double** dAngle, **double** dPrecision )

{

dAngle = dAngle % (2 \* Math.***PI***); // convert angle to between 0 and 2\*pi radians

**double** dResult = dAngle; // first approximation is x^1/1! = x

**double** dTerm = dAngle;

**int** iCount = 0;

**do**

{

iCount++;

// new term = -1 \* old term \* x \* x / (2k(2k+1))

dTerm = -1.0 \* dTerm \* dAngle \* dAngle / ((2.0 \* iCount) \* (2.0 \* iCount + 1));

dResult += dTerm;

} **while** (dResult != 0.0 && Math.*abs*(dTerm / dResult) > dPrecision);

// while the term we are adding affects the precision, continue

**return** dResult; // return last approximation

}

## Example 3: Finding square roots

A third example of an iterative method is finding square roots of numbers. For a given value *x*, we seek to within some desired precision.

Note that the square root of *x* is always between 1 and *x*. Thus, we have an upper boundary and a lower boundary for the value to be found. We can guess halfway between the lower and upper boundaries, and then compare the guess2 to *x*.

* If the guess is actually the square root of *x*, guess2 will be exactly equal to x.
* If the guess is too high, guess2 will be greater than *x*, and thus the guess will be a new upper boundary for the square root.
* If the guess is too low, guess2 will be less than *x*, and thus the guess will be a new lower boundary for the square root.

If the guess wasn’t correct, we thus have a new lower boundary or a new upper boundary for the square root, and we can guess halfway between the updated lower and upper boundaries and compare the guess to *x*2 again. Thus, we have an initial approximation for , a way of calculating a better guess for , and by comparing the lower and upper boundaries, we have a way of checking if we are “close enough”. Those are all of the elements necessary for an iterative method. This method is called the method of bisection.

See the **Bisection** sheet of **“Square root examples.xlsx”** for a partial example of the calculations involved. (This is only a partial example – which cases doesn’t it handle? What needs to be updated in the example to handle those cases?)

Here is a *partial* algorithm for the method of bisection for finding square roots (which cases doesn’t it handle? What needs to be updated to handle this case?):

**PARTIAL algorithm for finding square roots by bisection:**

Given x, precision

guess ß 0

if x < 0

indicate error

else if x > 0

lower ß 1

upper ß x

keepGoing ß true

while keepGoing

guess ß (lower + upper) / 2

test ß guess \* guess

if test is equal to x

keepGoing ß false

else if test < x

lower ß guess

else

upper ß guess

if (upper – lower) <= precision

keepGoing ß false

return guess

See also the code in **“IterativeMethodsExamples.java”** for an example.

So far, we have been creating iterative methods for values where Excel and Java already had functions or methods to calculate the result. These have been good initial examples of iterative methods because they allowed us to test our results against the Excel function/Java method. But can we find cases where there is no existing method? Well, consider that in Java 1.8, there is no square root method for the BigDecimal class. If you need a square root, you have to write a method on your own. Let’s try that.

See **“BigDecimalSquareRoot.java”** for the code; for example:

**public** **static** BigDecimal bdSqrt(BigDecimal bdValue, BigDecimal bdPrecision)

{

**final** BigDecimal BIG\_DECIMAL\_TWO = **new** BigDecimal("2");

BigDecimal bdGuess = BigDecimal.***ZERO***;

**if** (bdValue.compareTo(BigDecimal.***ZERO***) < 0)

{

**throw** **new** ArithmeticException("Cannot find square root of a negative number");

}

**else** **if** (bdValue.compareTo(BigDecimal.***ZERO***) > 0)

{

MathContext mcPrecisionDigits = **new** MathContext(bdPrecision.scale()

+ bdPrecision.precision() + bdValue.precision());

BigDecimal bdLower = BigDecimal.***ONE***;

BigDecimal bdUpper = bdValue;

**boolean** bKeepGoing = **true**;

**while** (bKeepGoing)

{

bdGuess = bdUpper.add(bdLower).divide(BIG\_DECIMAL\_TWO);

BigDecimal bdTest = bdGuess.multiply(bdGuess);

**if** (bdTest.compareTo(bdValue) == 0)

{

bKeepGoing = **false**;

}

**else** **if** (bdTest.compareTo(bdValue) < 0)

{

bdLower = bdGuess;

}

**else**

{

bdUpper = bdGuess;

}

// check if we are "close enough" so we can stop

BigDecimal bdError = bdUpper.subtract(bdLower); // error

bdError = bdError.divide(bdGuess, mcPrecisionDigits); // relative error

bdError = bdError.abs(); // absolute relative error

**if** (bdError.compareTo(bdPrecision) < 0)

{

bKeepGoing = **false**;

}

}

}

**return** bdGuess;

}

Compare the results with an online source which provides an answer with multiple digits, such as Wolfram Alpha.

Can we make the square root code more efficient? Yes. For instance, instead of using the method of bisection, we can use the Babylonian method. The Babylonian method works as follows:

Again, we start with a value of *x* to find the square root of, and a desired precision to achieve for our result. Let’s call our desired result *a*, so a = .

Then make an initial guess. For maximum efficiency, we could use an educated guess, but it is fine to start with 1. So *a*0 = 1.

Then we need a method of calculating a better approximation. The Babylonian method uses the formula

or in other words,

The sequence of values *a*0, *a*1, *a*2, … will converge to . Or in other words, the sequence of *new guesses* will get closer and closer to .

The algorithm for this method is as follows: SQRT(2)=1.414 SQRT(3)=1.732

**Algorithm for finding square roots using the Babylonian method:**

Given x, precision

guess ß 0

if x < 0

indicate error

else if x > 0

guess ß 1

keepGoing ß true

while keepGoing

oldGuess ß guess

guess ß (oldGuess + x / oldGuess) / 2

if (|guess – oldGuess| <= precision)

keepGoing ß false

return guess

Coding the algorithm is left as an exercise for the students.

How does the speed of the Babylonian method compare to the speed of the bisection method? See the **Babylonian** sheet of **“Square root examples.xlsx”** for a demonstration of the method, showing that it converges much faster than the method of bisection.

How does the Babylonian method work? Well, it’s not essential to know how it works – if you follow the algorithm and use the formula, you will get the right answer. But it’s always good to know why a formula is working, isn’t it?

The formula essentially says to make our guess halfway between *old guess* and (*x* / *old guess*). What do you get if you multiply *old guess* \* (*x* / *old guess*)? You get *x*. In other words, our two boundaries, *old guess* and (*x* / *old guess*), are factors of *x* – two numbers that when multiplied together equal *x*. Averaging out these factors produces a new number between the two values which is closer to the correct answer. Then we use that as the new better value as the *old guess* and continue the process.

Some questions that you should look at to make sure you fully understand the method of bisection and the Babylonian method:

* If you change the precision from 0.01 to 0.001 or to 0.000001, how will the answer be changed? How will the number of loops needed to calculate the answer change?
* What happen if you start with an initial guess of a negative number for *a*0 in the Babylonian method?
* What happens if you try to find the square root of a negative number without the safeguards that our algorithm has put into place? How should you deal with the square root of negative numbers?
* Could you alter the algorithms to find the cube root of a number?
  + Notes: The cube root *a* of a number *x* is the number where *a*3 = *x*. The cube root of a number *x* is always between 1 and *x* (or -1 and -*x*).

## Example 4: Finding the zeros (or roots) of a function

Now we will look at an example of an iterative method that can solve problems even when we don’t have a built-in method to check our answers; we will look for the zeros (or roots) of a function.

*Note: “zeros” is a term more commonly used in Great Britain, while “roots” is more common in North America. However, the term “roots” is easy to confuse with the square roots that we have just studied, so I will use the term “zeros”, which we will see is more descriptive anyway!*

A rough definition of a function is a rule that maps a set of values *x* to corresponding values of *f*(*x*). To be more precise, one value of *x* is mapped to exactly one value in *f*(*x*) and never to any other value of *f*(*x*) (although multiple values of *x* can map to the same value in *f*(*x*)). We will examine functions involving real numbers.

How can functions be expressed? First, they can often be expressed as a formula or equation. For instance:

*f*(*x*) = 2*x* + 1

which may also be written as *y* = 2*x* + 1, since we often graph the function values on the *y*-axis.

Formulas can also be more complex and can be written with different letters, such as:

*f*(*x*) = sin(*x*)

*g*(*x*) = *xx* - - 1

*p*(*x*) = *x*2 - 2

We can also show some function values in a table – for instance:

|  |  |
| --- | --- |
| *x* | *f*(*x*) = 2*x* + 1 |
| -1 | -1 |
| -0.5 | 0 |
| 0 | 1 |
| 0.5 | 2 |
| 1 | 3 |

The limitation of tables is that for real numbers, we cannot write out all possible values, so a table doesn’t provide a complete picture of how the function works.

We can also show functions with a graph. For instance:

A “zero” or “root” of a function is a value of *x* where *f*(*x*) = 0. Note that this is the value of *x* at that point, and it is **not** *f*(0), the value of the function where *x* = 0.

So for example, for *f*(*x*) = 2*x* + 1, we can find its zero in several ways:

* In this case, it shows up on our table (it will not always do so, but for this function, we are lucky):

|  |  |
| --- | --- |
| *x* | *f*(*x*) = 2*x* + 1 |
| -0.5 | 0 |

So the zero of the function is *x* = -0.5.

* We can also find zeros on the graph – a zero is where the graph of the function touches the *x*-axis. So for instance, we can see that the zero of *f*(*x*) = 2*x* + 1 is -0.5 on the following graph (but note that graphs only give us an estimate or a picture of where the zero is, not an exact value):

*x*-axis

* In some cases, we can solve for the zero analytically. For instance, for *f*(*x*) = 2*x* + 1, we are looking for a value of *x* such that *f*(*x*) = 0, so

0 = 2*x* + 1

-1 = 2*x* *(add -1 to both sides)*

-1/2 = x *(divide both sides by 2)*

So *x* = -0.5 is the zero of the function

Similarly, the quadratic equation returns the two possible zeros of an equation of the form *f*(*x*) = *a* + *bx* + *cx*2.

However, in some cases, there is no analytical solution to the equation or the solution is not easy to find. Consider *g*(*x*) = *xx* - , for instance, or *f*(*x*) = *ex* - cos(*x*) - 1.

* We can also use computational approaches to solve for the zeros of a function. In particular, there are iterative methods that can be used, even for functions where there is no analytical solution to the equation.

Before we move on to the computational approaches, we should consider one special class of functions: *polynomial* functions.

A polynomial function *p*(*x*) is made up of sums of powers of x, so it will look like:

*p*(*x*) *= a0* + *a*1*x* + *a*2*x*2 + *a*3*x*3 + … + *anxn*

Note that *a0* is *a*0*x*0 since *x*0 = 1, and *a*1*x* is just *a*1*x*1, so those terms are also powers of x.

So this includes lines (like *f*(*x*) = 2*x* + 1) and parabolas (like *p*(x) = *x*2 - 2) and any other equation with higher powers of x as well.

The *degree* of a polynomial function *p*(*x*) is its highest power of *x*. So a line is of degree 1, a parabola is of degree 2, and so on.

For *p*(*x*), there are **at most** *d* zeros if *p*(*x*) is of degree *d*. (If we were to include complex numbers and repeated zeros, there would always be exactly *d* zeros.)

The quadratic equation solves for the zeros of a degree 2 equation, and there are procedures for calculating the zeros of a degree 3 or degree 4 equation. However, there is no general solution to finding the zeros of polynomials with higher degrees.

### Computational approaches to solving for the zeros of an equation

Suppose we have values *x*1 and *x*2 where *f*(*x*1) > 0 and *f*(*x*2) < 0. Then if the function is *continuous* (smoothly connected), *f*(*x*) must have a zero between *x*1 and *x*2. This is because the value of the function switches from positive to negative, so at some point the function must touch the *x*-axis. For example, consider *f*(*x*) = *ex* - cos(*x*) - 1.

*f*(0.5) = -0.228861291 and

*f*(1.0) = 1.177979523

Thus, there must be a zero of the function somewhere between *x*1 = 0.5 and *x*2 = 1.0.

To find the zero, we can use bisection. Given *x*1 and *x*2, we can try calculating *f*((*x*1 + *x*2) / 2), halfway between the two starting points. This value of the function will be either 0, in which case we have found our zero; or negative, in which case we have a new value of *x* that produces a negative *f*(*x*); or positive, in which case we have a new value of *x* that produces a positive *f*(*x*). We can keep on trying halfway between the values until we are within some desired precision.

Note that the function values *f*(*x*1) > 0 and *f*(*x*2) < 0, but that does NOT mean that the values of *x*1 and *x*2 will be positive and negative. They could both be positive (see the example above) or both be negative.

See **“Sample function for zero-finding by bisection.xlsx”** and **“Function examples.xlsx”** for examples of how to follow this process in Excel, as well as values of sample functions we can use for later calculation.

See **“Algorithm For Finding Zeros Using Bisection.docx”** for the algorithm for this process.

To code this process, we need a general way of expressing functions. One way to do this is to have function objects. But some things (such as the method of bisection for finding zeros of functions) are common to all functions. How can we express this?

First, we can have an interface that says what all functions need to implement – see **Function\IFunction.java** for a starting file.

Then we can have an abstract class that implements the interface and contains all of the common methods for all functions – see **Function\ACFunction.java** for a starting file which contains a printTable method that can be used to print a table of values for any function that implements ACFunction. Note that the calculate method is left as an abstract method since its implementation will change from function to function to represent the specific calculations for that function.

Finally we can have specific functions that extend ACFunction and implement the calculate method for the specific function – see **Function\Function1.java** and **Function\Function2.java** for examples.

We can create any new function that we want by following the same templates that were used for Function1 and Function2. For instance, to implement *f*(*x*) = *ex* - cos(*x*) - 1, we could create **Function3.java** as follows:

/\*\*

\* Sample function implementing abstract function class<br>

\* Implements function f(x) = e^x - cos(x) - 1<br>

\*

\* **@author** MATH 282

\* **@created** September 25, 2019

\*/

**public** **class** Function3 **extends** ACFunction

{

/\*\*

\* Default constructor for the Function3 object<br>

\*/

**public** Function3()

{

}

/\*\*

\* Returns the value of the function f(x) = e^x - cos(x) - 1<br>

\* for a given x, implementing the abstract method from ACFunction<br>

\*

\* **@param** x Value to evaluate function at

\* **@return** Value of function f(x) = e^x - cos(x) - 1 at argument

\*/

**public** **double** calculate( **double** x )

{

**return** Math.*exp*(x) - Math.*cos*(x) - 1.0;

}

}

How do we add the zero-finding process to these classes? First, we add it to the interface so that all functions must implement it:

**public** **double** findZero( **double** xEvalPos, **double** xEvalNeg, **double** precision );

Next, we want to implement the findZero method. Where should it go? It applies to all functions, regardless of how they are calculated, so it should go in ACFunction. The code would be something like the following:

**public** **double** findZero( **double** xEvalPos, **double** xEvalNeg, **double** precision )

{

**double** guess = 0.0;

**boolean** keepGoing = **true**;

**while** (keepGoing)

{

guess = (xEvalPos + xEvalNeg) / 2.0;

**double** fAtGuess = **this**.calculate(guess);

**if** (fAtGuess == 0.0)

{

keepGoing = **false**;

}

**else** **if** (fAtGuess > 0.0)

{

xEvalPos = guess;

}

**else**

{

xEvalNeg = guess;

}

**if** (Math.*abs*(xEvalPos - xEvalNeg) <= precision)

{

keepGoing = **false**;

}

}

**return** guess;

}

Finally, we can create an instance of the function in the main method as follows:

IFunction f3 = **new** Function3();

System.***out***.println( "Table of function 3, f(x) = e^x - cos(x) - 1");

f3.printTable(-2.5, 2.5, 0.5);

System.***out***.println(f3.findZero(1.0, 0.5, 0.0000001));

Thus we have an iterative method for finding zeros of a function. We have a starting approximation (the range between *x*1 and *x*2), a method of getting a better approximation (checking the value halfway in between *x*1 and *x*2), and a method of checking if we are close enough (checking the distance between *x*1 and *x*2).

Note the limitation of the bisection method – we must start with two values *x*1 and *x*2 where *f*(*x*1) > 0 and *f*(*x*2) < 0. If we cannot find such values, we cannot apply the method of bisection.

See also **“oldNotes\Zero-finding MATH282-CST.doc”** for general notes on finding zeros and **“Last zero-finding example.xlsx”** for one last example (note the brief discussion of Excel’s Goal Seek feature which uses an iterative method to attempt to find a desired answer).

For more information on zeros of functions and methods for finding zeros (including bisection but also other methods), see the following:

* <https://en.wikipedia.org/wiki/Zero_of_a_function>
* <https://en.wikipedia.org/wiki/Root-finding_algorithm>