# 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.