MATH 282 Assignment #1

Errors in computing and using iterative methods to solve problems

**Due by 10pm on Monday, October 3, 2022**

**Total marks: 49**

Late assignments submitted by 10pm on Tuesday, October 4, 2022 will receive 75% of the mark that they would have received. Any assignments submitted after that will receive a mark of 0 and no feedback will be provided.

Assignment #1 can be completed in groups of 1-3 students of your choosing. If you are working in a group, you must email me ([wangxi@saskpolytech.ca](mailto:wangxi@saskpolytech.ca)) by 10pm on Monday, September 26, 2022 with the names of your group members. Any plagiarism or other academic misconduct will result in a mark of 0 for all offending parties and possible further consequences.

# Submission Instructions

If you have a paper copy (handwritten) of any of your answers, submit it to Michael or to the drop box in room 242.

* Your names and CST numbers must be on the pages
* Answers should be clearly labelled with the question number
* Answers must be legible

For submitting files:

* Create a folder named ***[polytechusername][polytechusername]*math282a1**, where ***[polytechusername]*** are the SaskPolytech usernames of your group members.
* Create subfolders for each question named **q1**, **q2**, etc.
* Place the files for each question in the subfolder for that question.
* For programming questions:
  + Use the package **math282a1q#** for all source code, where **q#** is the question number
  + Include only the source code (**.java** files)
  + Make sure that all files have proper comments and follow the CST programming style guide
  + Ensure that all source code files are submitted (so the program can be compiled and run from the files submitted)
* When done, create a compressed folder named   
  ***[polytechusername][polytechusername]*math282a1.zip**
* Submit the compressed (zipped) file to the Assignment #1 Dropbox Folder in the MATH282 online course material for MATH 282 (under Assessments > Dropbox).

1. *[7 marks]* Copy the files from the **Assignments\CoinDemo** folder on the MATH 282 OneDrive folder and run **BalanceRegister.java**. You will see that this program asks for the amount in a cash register (such as **2.5**) and then letters for the coins in the cash register (such as **tqq** for a toonie and 2 quarters), and then checks if the coins balance with the amount entered. The company developing this program plans to add a device to count the coins and send a signal for each coin instead of typing the letters for the coins. However, first they have encountered a problem that they need you to fix – sometimes when an amount is entered that should balance exactly with the coins entered, an error occurs (for instance, **0.30** does not match **ddd** for 3 dimes).

Test the letters for each coin separately (try several combinations with ***only*** dimes, then several with ***only*** nickels, etc. – do not test more than 10 of any coin). Then answer the following questions:

* 1. Which coins sometimes had problems? List one specific example of a problem (for instance, **0.30** does not match **ddd** for 3 dimes). *[2 marks]*
  2. Which coins had no problems? *[1 mark]*
  3. Explain why the coins in part (a) had problems, while the coins in part (b) never had problems. *[2 marks]*
  4. Determine and implement one way to fix the problems. Your solution should not cause any changes in how the user interacts with the program (for instance, they should still enter their amounts in dollars and cents, enter letters for each coin, and see the same output and messages.) *[2 marks]*

Submit your answers for parts (a)-(c), and the updated source code for part (d).

1. *[8 marks]* Recall that floating-point numbers in the computer are represented by a *mantissa* multiplied by 2 to the power of some *exponent* (that is, *mantissa* × 2*exponent*) where both the mantissa and exponent are represented by a certain number of bits.

We know that floating-point numbers can only store a limited range of values (as demonstrated by the factorial methods). One way to determine the *smallest* number that the computer can store in a floating-point number is by repeated testing. Start with the value 1 and then repeatedly divide the value by 2 (since computers are binary machines, dividing by two will produce the most accurate result possible) as long as the value is greater than 0. When the value becomes 0, then the *previous* value (the last value before the value became 0) is the smallest number that the computer can represent.

We also know that floating-point numbers only have a certain amount of precision. One way to measure that precision is with *machine epsilon*, the smallest number that makes a difference when added to 1. In other words, machine epsilon is the smallest number ε such that 1 + ε > 1. (ε is the Greek letter epsilon.)

Machine epsilon can also be calculated by repeated testing using a starting value of ε = 1, testing to see if 1 + ε > 1, and dividing ε by two. When it is no longer true that 1 + ε > 1, the *previous* value of ε is the value for machine epsilon.

* 1. For both the float and double data types, write a Java program that will calculate (using the techniques described above) and display both the smallest positive number the computer can represent and machine epsilon. Hand in your source code and any other required files for this question (when executed, it should show all of the requested values).
  2. Which part of the floating-point number is primarily responsible for determining what the smallest number is – the mantissa or the exponent? Explain.
  3. Which part of the floating-point number is primarily responsible for determining what machine epsilon is – the mantissa or the exponent? Explain.
  4. The technique given to calculate machine epsilon is essentially an iterative method. Identify the initial approximation used, how a better approximation is calculated, and how we can tell when we are “close enough” (or in this case, have gone too far).

1. *[3 marks]* One way to calculate the cosine of an angle expressed in radians is the following formula:

If you calculate cos(0.1) using only the first 4 terms of this formula, what types of errors would be involved in your calculation? Explain.

1. *[10 marks]* This question examines how error propagates. Ben is applying weed control on the farm: *v* is the velocity of the tractor, *t* is the time in his favorite field, *r* is the rate at which he is applying product, and *w* is the width of his sprayer implement.
   1. *[2 marks]* Computer the relative error in each of the following values:
   2. *[2 marks]* Convert (by using the appropriate scale factor) *v* to units of *m*/*s* (meters per second), and convert *t* to units of *s* (seconds). Computer the relative error of the converted values of *v* and *t*. When a scale factor is applied, does the relative error change?
   3. *[2 marks]* Compute the field area, *a*, covered and express the maximum possible error (as both amount of error and relative error). Use *v* and *t* from part (b).
   4. *[2 marks]* A PID controller controls the rate of application, *r*. Calculate the amount of product, *p*, required for this field and the maximum possible error, both the absolute amount of error and relative error. Use *a* from part (c).
   5. *[1 mark]* What is the minimum amount of product (in kg) with which Ben can fill his Sprayer tank in order to ensure that he can complete his field without stopping to refill?
   6. *[1 mark]* If Ben starts with the amount in (e), what is the maximum amount of leftover product (in kg) at the end of the field?

*In practice, an experienced farmer can make small adjustments to the target product application rate such that a Sprayer tank is exactly empty at the end of the field. To do so, one must have a measure (via a sensor with its own source of error) of how much product is remaining in the Sprayer tank. A PID controller can also be enhanced to automatically make such similar small adjustments. This is an example application of Precision Farming.*

1. *[9 marks]* The Babylonian method of calculating square roots requires an initial value for its calculations. In class, we just used 1 as the initial value; however, there are methods for calculating a rough estimate of the square root which can then be used as the starting point. See the description for calculating an initial estimate of the square root here, which includes several methods for getting a good estimate of the starting point: <https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Initial_estimate>

One quick method to estimate the square root of a positive number *S* is as follows: Let *D* be the integer part of the base-10 logarithm of *S*. Then the rough estimation (not as good as the Wikipedia method, but simpler) is 10D/2.

* 1. Implement a method to calculate the rough estimate for the square root of a number using double values. You can use the quick process above or a better process described by Wikipedia, or another process if you find one. *[2 marks]*
  2. Write a method to calculate square roots using the Babylonian method using double values, but modify it to accept an initial value to use for the first guess and to display the number of loops required by the method. *[4 marks]*
  3. Using the modified Babylonian method, calculate the square root of 4.9 × 10*i* for every value of *i* from -10 to 10. Use a precision of 0.00001. Do one run using a starting value of 1 and one run using a starting value provided by the rough estimate. Display the answers calculated by each method, along with the answer provided by the built-in square root method in Java. *[2 marks]*
  4. Based on the results of the two starting values, comment on the efficiency. For example, if there are fewer loops in the square root calculation with the rough estimate, how does that compare to the extra calculations required to find the rough estimate? *[1 mark]*

1. *[12 marks]* One way to find the zeros or roots of a function is the method of bisection, as covered in class.
   1. Consider the following function *p*(*x*). What is the maximum number of zeros for this function? How can you tell?
   2. In order for the method of bisection to work, you need two starting values – one where the value of the function is positive and one where the value of the function is negative. Find a set of starting values for as many zeros of *p*(*x*) as you can identify. List the starting points.
   3. Add error-checking to the code for the method of bisection for finding zeros discussed in class to guarantee that it is passed in valid starting points (where the function is +/-).
   4. Use the code to find each of the zeros of *p*(*x*) from parts (a)-(b).
   5. Use your bisection method to find the birth rate *x* in a certain population, where the birthrate *x* is determined by the following function. (Note that is the exponential function – use Math.exp(value) to implement in Java.)

Submit your answers for parts (a) and (b), and your code for parts (c)-(e). When run, your code should show the answers for parts (d) and (e).