Robin Karlose
40089313

Problem 4

SOEN 6011
29th July 2019

GIT Repository Links

GIT Hub personal page

https://github.com/robinkarlose

GitHub repository for Project (Exact location may change)

https://github.com/robinkarlose/SOEN6011Project

Introductory Notes about the Java Implementation of the Beta Function

For input , I have used a simple textual user interface.

I have included some very basic but helpful error messages, with possible solution(s) to resolve them.

Debugger

Since I carried out the implementation of the function from scratch , I have used an IDE - the IntelliJ IDE.

IntelliJ is a fantastic IDE with a very good debugger and other solid features.

It has the following features:-

- Breakpoints and watchpoints for Java
- Support for multiple simultaneous debugging sessions
- Frames , variables and watches views in the debugger User Interface
- Runtime Evaluation of expressions , etc

Advantages

- Easy to evaluate individual terms of series and check for accuracy and see whats going wrong and where the approximations are slowly causing the result to become inaccurate
- Evaluation of mathematical expressions are easier
- Extremely customized break point properties:conditions , pass count , and so on

Disadvantages

- Steep Learning curve for first time users
- While the UI for the debugger is excellent brimming with features , it can get a little overwhelming and inefficient to use

Qualities of the Program

Correctness of code

I computed the Beta function using 2 methods -

1.Stirling's Approximation of the Gamma function and

2.Calculating the natural log of the Gamma function the Lanczos formula for the Stieltjes function

The Beta Function can be extremely inaccurate if you use an approximation of the power function for real numbers (as I needed to use in method 1). So method 1 is producing fairly inaccurate results. Method 2 is far more accurate and is producing a near correct calculation of the Beta function

Efficiency of code

The code is relatively efficient as most of my methods use single loops which leads to lower values of time complexity

Maintainability of code

I divided the implementation of the code into multiple classes and methods and it is easy to organize and build upon , hence highly maintainable

Robustness of code

I tried to make the code more robust by keeping provision for error messages with some tips on how to solve them

Usability of code

My code is easy to use with easy to execute and understand for the user with a simple textual user interface

Tool Used - Checkstyle

Checkstyle is a solid platform independent tool to check the source code.

It has some of the following features:-

- Naming conventions of attributes and methods
- The use of imports, and scope modifiers

- The number of function parameters , etc

Advantages

- Checkstlye is portable across IDEs - this is one of its biggest strengths as it is the same whether you use Eclipse or IntelliJ
- Checkstyle has better external tooling. It's much easier to integrate checkstyle with your external tools since it was really designed as a standalone framework
- It grants you the ability to create your own rules and standards

Disadvantages

- It can be a pain to install and the plugin itself can cause instability in the original IDE installation
- In some rare cases, forces some unnecessary conventions which don't affect the code much and a lot of time is wasted in re-organizing the code perfectly again

Test Cases

Test Case 1
Function — Power.powr(double, int)
Input— Power.powr(5.0, 3)
Expected — 125.0
Result — Pass
The above test case is tied to **FR3**

Test Case 2
Function — Power.nat_log(int)
Input— Power.powr(20.0)
Expected — 2.995
Result — Pass
The above test case is tied to **FR4**

Test Case 3
Function — Power.powr(double, double, double)
Input— Power.powr(16.0,0.75,0.001)
Expected — 8.0
Result — Pass
The above test case is tied to **FR3**

Test Case 4
Function — Power.epowrx(double)
Input— Power.epowrx(3.0)
Expected — 20.085
Result — Pass
The above test case is tied to **FR3**

Test Case 5
Function — BetaFuncStirling.CompBetaS(double , double)
Input— BetaFuncStirling.CompBetaS(3.0,4.0)
Expected — 0.016
Result — Pass , with a very low delta

The above test case is tied to **FA1, FR1 , FR2**

Test Case 6

Function — BetaFuncLnGamma.lngamma(double)

Input— BetaFuncLnGamma.lngamma(3.5)

Expected — 1.2009

Result — Pass , super accurate

The above test case is tied to **FA1**

Test Case 7

Function — BetaFuncLnGamma.beta(double,double)

Input— BetaFuncLnGamma.beta(3.0,4.0)

Expected — 0.0168

Result — Pass , super accurate

The above test case is tied to **FA1 , FR1 , FR2**

Test Case 8

Function — BetaFuncLnGamma.beta(double,double)

Input— BetaFuncLnGamma.beta(-3.0,-4.0)

Expected — $-\infty$

Result — Fail as expected, because it violates FA1 and FA2 , but commented out in the JUnit code

The above test case is tied to **FA1,FA2**