KKTPM CALCULATOR 1.01

A Java Library for Karush Kuhn Tucker Proximity Measure

Abstract

Since its proposal in 2015, KKTPM has generated a lot of interest as a metric of how close a solution is from being an optimum. KKTPM Calculator is a Java library intended for practitioners and researcher interested in using KKTPM without going deep into its proof and implementation details.

Haitham Seada

seadahai@msu.edu

Contents

Introduction

In 2015, Deb and Abouhawwash proposed Karush Kuhn Tucker Proximity Measure (KKTPM). A metric that can measure how close a point is from being "an optimum". The smaller the metric the closer the point. Their metric is applicable to both single objective and multiobjective optimization problems. In a single objective problem the metric shows how close a point is from being a "local optimum", while in multiobjective problems the metric shows how close a point is from being a "local Pareto point". Exact calculations of KKTPM for each point requires solving a whole optimization problem, which is extremely time consuming. In order to alleviate this problem, the authors of the original work again proposed several approximations to the true KKTPM, namely Direct KKTPM, Projected KKTPM, Adjusted KKTPM and Approximate KKTPM. Approximate KKTPM is simply the average of the former three and is what we call throughout this tutorial simply "KKTPM". All there approximations can be calculated by solving a system of linear equations. No optimization is required anymore. Moreover, Deb and Abouhawwash were able to show that Approximate KKTPM is reliable and can be used in place of the exact one. KKTPM Calculator provides an easy interface through which practitioners can make use of the new metric without delving into the intricacies of proofs and calculations. KKTPM Calculator can be used to calculate all the four approximations and the set of Lagrange multipliers at a specific point. It also supports XML-based input and symbolic mathematical function evaluation among other features.

License

Installation

Installing KKTPM Calculator is just like installing any other Java library. You need to follow the following steps if you are not using any specific IDE (which is highly unlikely).

1. Download the library (*.zip* file)
2. Extract the *.jar* file
3. Add it to your CLASSPATH.
4. Enjoy!

NetBeans instructions

If you are using some IDE (NetBeans, Eclipse etc.) your life will definitely be easier, and you can install KKTPM Calculator with just a few clicks. A set of instructions specific to NetBeans user are listed below. For other IDEs similar set of instructions exist.

1. Download the library (*.zip* file)
2. Extract the *.jar* file
3. Open NetBeans
4. Open the *Properties* panel of your project
5. Click *Libraries* (on the left)

6. Click *Add Jar* (on the right)
7. Browse for your *.jar* file
8. Enjoy!

How-To

Throughout this guide we will use the sample constrained single objective test problem expressed in Equations **Error! Reference source not found.** and (1) for simplicity. KKTPM Calculator however can be used with any number of objectives.

$$\begin{aligned} min.\ f &= {x_1}^2 \\ s.t.\ x_1 &\geq 0.5 \end{aligned} \tag{1}$$

Before proceeding we need to express all our constraints a *less-than-or-equal* format. This converts Equation (1) to:

$$g = 0.5 - x_1 \leq 0 \tag{2}$$

Since KKTPM calculations require gradient information, the user of the library should provide the following formulations as well:

$$\begin{aligned} \frac{df}{dx_1} &= 2x_1 \\ \frac{dg}{dx_1} &= -1 \end{aligned} \tag{3}$$

```
1.  // Raw input data
2.  double[] x = new double[]{1};
3.  double[] f = new double[]{1};
4.  double[] z = null;
5.  double[] g = new double[]{-0.5};
6.  double[][] jacobianF = {{2}}; // Two functions & three variables
7.  double[][] jacobianG = {{-1}}; // Four constraints & three variables
8.  // Calculations
9.  double kktpmDirect = KKTPMCalculator.getDirectKKTPM(x,f,z,g,jacobianF,jacobianG);
10. double kktpmAdjusted = KKTPMCalculator.getAdjustedKKTPM(x,f,z,g,jacobianF,jacobianG);
11. double kktpmProjected = KKTPMCalculator.getProjectedKKTPM(x,f,z,g,jacobianF,jacobianG);
12. double kktpm = KKTPMCalculator.getKKTPM(x, f, z, g, jacobianF, jacobianG);
13. // Display results
14. System.out.format("%12s  = %10.6f%n", "Direct KKTPM", kktpmDirect);
15. System.out.format("%12s  = %10.6f%n", "Adj. KKTPM", kktpmAdjusted);
16. System.out.format("%12s  = %10.6f%n", "Proj. KKTPM", kktpmProjected);
17. System.out.format("%12s  = %10.6f%n", "KKTPM", kktpm);
```

Code Listing 1 *Calculate KKTPM using raw input data*

Having these formulations in hand there are two ways to calculate KKTPM for a specific point:

1. Using exact values
2. Using symbolic formulations

Using Exact Values

By the term "exact values" we refer to the actual values of variables, objectives, constraints and gradients at a specific point. In this approach, it is the user's responsibility to do these evaluations for a specific point before conducting the KKTPM calculations. This is the most primitive/basic way of using KKTPM Calculator. Listing 1 shows how to use this approach with our sample problem. Lines 2 to 7 sets the *x* vector (variables), *f* vector (objective function(s)), *z* (ideal point), *g* (constraint(s)), *jacobianF* (objective function(s) gradient(s)), *jacobianG* (constraint(s) gradient(s)), respectively. Notice that the ideal point for single objective problems is *null*. For multiobjective problems, *z* will be a vector (array) equal in size to the *f* vector. The output of this code is shown in Listing 2.

```
Direct KKTPM  =   0.246914
  Adj. KKTPM  =   0.444444
 Proj. KKTPM  =   0.404938
       KKTPM  =   0.365432
```

Code Listing 2 *Output*

```
1.  // Create an optimization problem object
2.  double[] x = new double[]{1};
3.  OptimizationProblem problem = new OptimizationProblem();
4.  for (int i = 0; i < x.length; i++) {
5.      problem.addVariable("x" + (i + 1), x[i]);
6.  }
7.  problem.addObjective("x1^2");
8.  problem.addConstraint("0.5-x1");
9.  problem.setObjectivePartialDerivative(0, "x1", "2*x1");
10. problem.setConstraintPartialDerivative(0, "x1", "-1");
11. // Caclulations
12. double kktpmDirect = KKTPMCalculator.getDirectKKTPM(problem, null);
13. double kktpmAdjusted = KKTPMCalculator.getAdjustedKKTPM(problem, null);
14. double kktpmProjected = KKTPMCalculator.getProjectedKKTPM(problem, null);
15. double kktpm = KKTPMCalculator.getKKTPM(problem, null);
16. // Display results
17. System.out.format("%12s  = %10.6f%n", "Direct KKTPM", kktpmDirect);
18. System.out.format("%12s  = %10.6f%n", "Adj. KKTPM", kktpmAdjusted);
19. System.out.format("%12s  = %10.6f%n", "Proj. KKTPM", kktpmProjected);
20. System.out.format("%12s  = %10.6f%n", "KKTPM", kktpm);
```

Code Listing 3 *Calculate KKTPM using an OptimizationProblem object*

```java
// Set variables
VariablesManager vm = new VariablesManager();
vm.setVector("x_$11", new double[]{10,20,30,40,50});
vm.set("_asd", 10.2);
vm.set("i", 5);
// Parse expression (create parse tree)
AbstractNode parseTree = MathEpressionParser.parse(
        "23/(_asd + 10.23)*15 -  10/88*99-x_$11[i*2-9]/50 "
                + "+ sum{k,2,4,x_$11[k]+100*  tan(0.2)"
                + "-100*sin(0.2)/cos(0.2)}", vm); // 95.43693098384728
// Print parse tree
System.out.println("Whole expression: " + parseTree.toString());
// Evaluate parse tree and print
System.out.println("Value = " + parseTree.evaluate());
```

Code Listing 4 *Using the math expression parser (code)*

Using Symbolic Formulations

The other way of using KKTPM Calculator is to allow users to insert the actual formulas instead of the exact values of their evaluations. This enables the user of the library to define his problem only once and change only the values of the *x* vector (the point under investigation) as necessary.

KKTPM Calculator includes a mathematical expression parser. This parser provides a great deal of flexibility enabling users to provide formulations of their objectives, constraints and gradients easily and in virtually any mathematically acceptable format. The parser supports the following features:

1. Floating-point numbers/calculations
2. Addition and subtraction operators
3. Multiplication and division operators
4. Modulo operator (division remainder)
5. Negation
6. Redundant positive signs
7. Summation operator (∑)
8. Product operator (∏)
9. Constants
10. Trigonometric functions
11. Floor and ceiling operators
12. C-type identifiers (for variable names)
13. Simple variables (name-value pairs)
14. Array variables (a value for each index)
15. Indexing offset of One (instead of Zero)
16. Using mathematical expressions as array indices
17. Handling arbitrary spacing

18. Nested parentheses
19. Ensuring parentheses, brackets and braces balancing
20. C-like operator precedence
21. Meaningful and informative error messages
22. Extensibility

```
Whole expression: (((((23.0 / ((_asd + 10.23))) * 15.0) - ((10.0 / 88.0) *
99.0)) - (x_$11[((i * 2.0) - 9.0)] / 50.0)) + Sum{k=[2.0,4.0],((x_$11[k] +
(100.0 * tan(0.2))) - ((100.0 * sin(0.2)) / cos(0.2)))})

Value = 95.43693098384728
```

Code Listing 5 *Using the math expression parser (output)*

```
1.  // Create an optimization problem object
2.  double[] z = null;
3.  double[] x = new double[]{1};
4.  double[] f = new double[]{1};
5.  double[] g = new double[]{-0.5};
6.  OptimizationProblem problem = new OptimizationProblem();
7.  for (int i = 0; i < x.length; i++) {
8.      problem.addVariable("x" + (i + 1), x[i]);
9.  }
10. problem.addObjective("x1^2");
11. problem.addConstraint("0.5-x1");
12. problem.setObjectivePartialDerivative(0, "x1", "2*x1");
13. problem.setConstraintPartialDerivative(0, "x1", "-1");
14. // Calculate Lagrange multipliers
15. double[] u = KKTPMCalculator.getLagrangeMultiplers(problem, z);
16. // Caclulations (using the calculated Lagrange multipliers)
17. double kktpmDirect = KKTPMCalculator.getDirectKKTPM(f, g, u);
18. double kktpmAdjusted = KKTPMCalculator.getAdjustedKKTPM(f, g, u);
19. double kktpmProjected = KKTPMCalculator.getProjectedKKTPM(f, g, u, kktpmDirect);
20. double kktpm = KKTPMCalculator.getKKTPM(f, g, u);
21. // Display results
22. System.out.format("%12s  = %10.6f%n", "Direct KKTPM", kktpmDirect);
23. System.out.format("%12s  = %10.6f%n", "Adj. KKTPM", kktpmAdjusted);
24. System.out.format("%12s  = %10.6f%n", "Proj. KKTPM", kktpmProjected);
25. System.out.format("%12s  = %10.6f%n", "KKTPM", kktpm);
```

Code Listing 6 *Calculate Lagrange multipliers independently*

A user can take advantage of the mathematical expression parser to create a generic `OptimizationProblem` object. Listing 3 shows how to do that. This code generates the exact same output shown in Listing 2.

A detailed descriptions of the mathematical expression parser is out of the scope of this tutorial, however in order to get a taste of its power and flexibility Listing 4 shows how to parse and evaluate the expression in (4):

$$\frac{23(15)}{\_asd + 10.23} - \frac{10(99)}{88} - \frac{x\_\$11[2i-9]}{50} + \sum_{k=2}^{4} x\_\$11[k] + 100\tan(0.2) - \frac{\sin(0.2)}{\cos(0.2)} \qquad (4)$$

Notice that *i*, *_asd*, *x_$11* are all valid C-type identifiers. The first two are simple variables while the last one is an array variable. *k* is also a simple variable however since it is just used as a counter inside the summation operator, it does not need to be explicitly declared (like *i*, *_asd* and *x_$11*). The output of this code snippet is shown in Listing 5.

Lagrange Multipliers

As part of KKTPM computations, Lagrange multipliers are also calculated. Lagrange multipliers are used in calculating the three variations of KKTPM, direct, projected and adjusted. In order to avoid repeating Lagrange multipliers calculations, KKTPM Calculator enables its users to calculate Lagrange multipliers independently, then using them for further calculations. Listing 6 shows a sample code doing so. Again the output of this code snippet is the exact same output shown in Listing 2.

XML Input

Validation

KKTPM Calculator supports reading an optimization problem from an XML file. The format of the XML file must follow the schema found here. Listing 7 shows an XML input file representing the same single objective sample problem we are dealing with here. For problems having more than one objective, Listings 8 and 9 show how the famous BNH bi-objective problem (shown in Equations (5) to (10)) can be represented in the required XML format.

$$min.\ f_1 = 4{x_1}^2 + 4{x_2}^2 \qquad (5)$$
$$min.\ f_2 = (x_1 - 5)^2 + (x_2 - 5)^2 \qquad (6)$$
$$\text{s.t.}\ (x_1 - 5)^2 + x_2^2 \leq 25 \qquad (7)$$
$$(x_1 - 8)^2 + (x_2 + 3)^2 \geq 7.7 \qquad (8)$$
$$0 \leq x_1 \leq 5 \qquad (9)$$
$$0 \leq x_2 \leq 5 \qquad (10)$$

```xml
<?xml version="1.0" encoding="UTF-8"?>
<problem xmlns="http://www.coin-laboratory.com/xml/">
    <!-- Variables -->
    <variables>
        <variable>x</variable>
    </variables>
    <!-- Objectives -->
    <objectives>
        <objective>
            <function>
                x^2
            </function>
            <gradient>
                <derivative var="x">
                    2 * x
                </derivative>
            </gradient>
        </objective>
    </objectives>
    <!-- Constraints -->
    <constraints>
        <constraint>
            <function>
                0.5 - x
            </function>
            <gradient>
                <derivative var="x">
                    -1
                </derivative>
            </gradient>
        </constraint>
    </constraints>
</problem>
```

Code Listing 7 *Single objective XML sample input file*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<problem xmlns="http://www.coin-laboratory.com/xml/">
    <!-- Variables -->
    <variables>
        <variable>x1</variable>
        <variable>x2</variable>
    </variables>
    <!-- Objectives -->
    <objectives>
        <objective>
            <function>4*x1^2+4*x2^2</function>
            <gradient>
                <derivative var="x1">8*x1</derivative>
                <derivative var="x2">8*x2</derivative>
            </gradient>
        </objective>
        <objective>
            <function>(x1-5)^2+(x2-5)^2</function>
            <gradient>
                <derivative var="x1">2*(x1-5)</derivative>
                <derivative var="x2">2*(x2-5)</derivative>
            </gradient>
        </objective>
    </objectives>
    <!-- Constraints -->
    <constraints>
        <constraint>
            <function>((x1-5)^2+x2^2-25)/25</function>
            <gradient>
                <derivative var="x1"> 2*(x1-5)/25 </derivative>
                <derivative var="x2"> 2*x2/25 </derivative>
            </gradient>
        </constraint>
        <constraint>
            <function>(-(x1-8)^2-(x2+3)^2+7.7)/7.7</function>
            <gradient>
                <derivative var="x1"> -2*(x1-8)/7.7 </derivative>
                <derivative var="x2"> -2*(x2+3)/7.7 </derivative>
            </gradient>
        </constraint>
```

Code Listing 8 *Bi-objective XML sample input file (part 1)*

```
        <constraint>
            <function> -x1 </function>
            <gradient>
                <derivative var="x1"> -1 </derivative>
                <derivative var="x2"> 0 </derivative>
            </gradient>
        </constraint>
        <constraint>
            <function> -x2 </function>
            <gradient>
                <derivative var="x1"> 0 </derivative>
                <derivative var="x2"> -1 </derivative>
            </gradient>
        </constraint>
        <constraint>
            <function> x1-5 </function>
            <gradient>
                <derivative var="x1"> 1 </derivative>
                <derivative var="x2"> 0 </derivative>
            </gradient>
        </constraint>
        <constraint>
            <function> x2-3 </function>
            <gradient>
                <derivative var="x1"> 0 </derivative>
                <derivative var="x2"> 1 </derivative>
            </gradient>
        </constraint>
    </constraints>
</problem>
```

Code Listing 9 *Bi-objective XML sample input file (part 2)*

But before using the input XML file, it makes sense to validate its correctness against the schema. Listing 10 validates the XML file `my_problem.xml` against `problem.xsd`. In case the XML file does not follow the schema this code will throw a `SAXException`.

```
File xmlFile = new File(filePath);
try {
    // Validate
    XMLValidator.validate(xmlFile);
    // XML input file follows the schema
    System.out.println("\"" + xmlFile.getAbsoluteFile() + "\" is valid");
} catch (SAXException ex) {
    // XML input file does not follow the schema
    System.out.println("\"" + xmlFile.getAbsoluteFile() + "\" is NOT valid");
    System.out.println("Reason: " + ex.getLocalizedMessage());
}
```

Code Listing 10 *XML validation*

```
1.  // Load the problem
2.  OptimizationProblem problem = XMLParser.readXML(new File(filePath));
3.  // Display the loaded problem
4.  System.out.println(problem);
```

Code Listing 11 *Load an* `OptimizationProblem` *form an XML input file*

```
Variables{x=0.0} - Objectives{f(0) = (x ^ 2.0) {df(0)/dx=(2.0*x)}} -
Constraints{f(0) = (0.5 - x) {df(0)/dx=(0.0-1.0)}}
```

Code Listing 12 *Printing our* `OptimizationProblem` *object*

Wrapping Up

KKTPM Calculator provides its users with an easy-to-use and flexible interface for defining single and multiobjective optimization problems, using symbolic mathematical functions and calculate Karush Kuhn Tucker Proximity Measure (KKTPM). The best fit of this library is to be used along with other population based optimization code/library, however the XML input format proposed here should be applicable to any optimization context, whether it is population-based, point-based, heuristic or deterministic.

References

[1] Kalyanmoy Deb and Mohamed Abouhawwash. “An Optimality Theory Based Proximity Measure for Set Based Multi-Objective Optimization”, IEEE Transaction on Evolutionary Computations (2015)

[2] Kalyanmoy Deb and Mohamed Abouhawwash. “A Computationally Fast and Approximate Method for Karush-Kuhn-Tucker Proximity Measure”, COIN technical report 2015014 (2015)