Multitemplate Motion Recognition

James Schuback
Kiwi Wearables
June 2015

Table of Contents:

1 – Introduction

This document outlines the theory behind, and implementation of, the multitemplate motion recognition algorithm developed at Kiwi Wearables, Inc. The goal of the software is to provide this extended functionality while maintaining a portable code structure similar to the *MotionAnalyzer* Java class.

Some definitions will help clarify the discussion:

- *Template*: short for "motion template", a template is a recorded set of raw sensor data corresponding to a particular motion, such as a hand wave, sports motion (e.g. basketball shot) or any other movement gesture that can be uniquely identified by the sensors. The sensors consist of the accelerometer, gyroscope, and potentially magnetometer; each sensor outputs 3D measurement values. The data is typically stored in a matrix object, where one dimension is the number of sensor dimensions (e.g. accelerometer + gyroscope X,Y,Z = 6) and the other dimension is the length of the template, determined by the data sample rate times the time length. Lengths are typically determined by repeating the motion, examining the data visually, and performing manual segmentation.
- *User Data Buffer:* a queue-like data structure that stores the motion data (recorded sensor readings) of a user's device. An example implementation is an array of queues, one for each axis of an accelerometer and gyroscope. The size of each queue should be the length of the longest template one is attempting to recognize. For example, at a sample rate of 100Hz, and a max template length of 1.5 seconds, the buffer length (i.e. length of each queue) should be 150 data points.
- *Recognition:* the act of recognition determines examining the buffer in real-time, i.e. after every new data point is received, and making a decision on two factors: 1) how likely is it that any motion was recognized (via thresholding a raw algorithm value), and 2) which motion template best matches the user's data. The approach taken in this work returns information pertaining to both, on every update of the user buffer, i.e. in time 1/(sample rate).

The algorithm developed in this document is built to simultaneously test the match of user data to multiple pre-made templates. A small collection of external software is required to transform the pre-made templates into necessary data structures for use in Java code – this setup and importing process is described in detail for Android apps. The Java class *MultitemplateAnalyze* contains the full implementation. It relies on a numerical optimization library called *JOptimizer*. Instances of the analysis method can be created as background threads in Android via the *AsyncTask* class.

2 – Problem Definition and Setup

In this section we create precise definitions for various data structures as well as phrase the multitemplate recognition problem as an optimization problem. This phrasing will naturally lead into the software infrastructure, explained later in this document.

To begin, consider the task of recognizing a 1-D signal, i.e. an array of numbers of length *m* (imagine this representing an accelerometer axis). Say we have *n* of these different templates, and we concatenate them into a matrix structure. Note that the template signals need not have the same length; in this case we will zero-pad the shorter templates so that the total length equals that of the longest vector. The following image displays this data object for 5 sample templates:

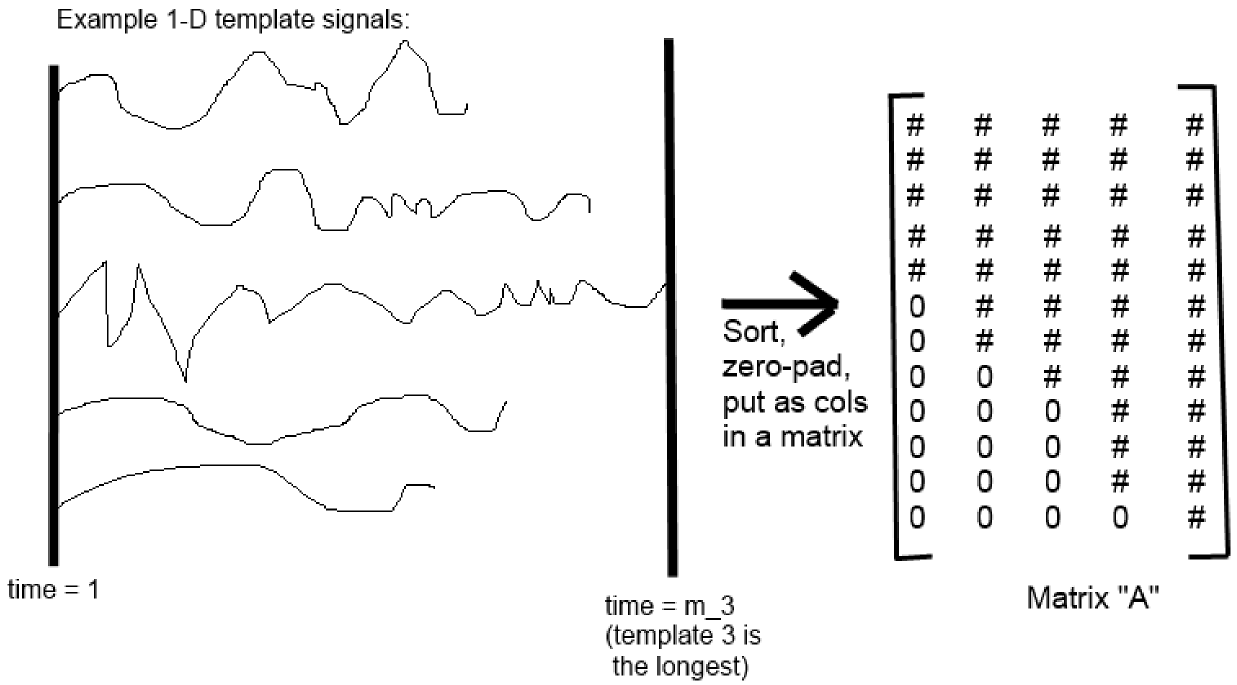

Example 1-D template signals:
Sort, zero-pad, put as cols in a matrix
time = 1
time = m_3 (template 3 is the longest)
Matrix "A"

Figure 1: Template Matrix Construction (1-D example)

Now given a vector of user data – i.e. a buffer of the last T time samples – we can phrase the multitemplate recognition problem as follows: find the column, or combination of columns, in matrix A, that best matches the user data. Here, we can take T to be the length of the longest template. The way to choose column combinations of the matrix A is to right-multiply it by a vector **x**, of length equal to the number of templates – which is 5 in this case.

The mathematical quantity **Ax** will be a linear combination of templates then, and we want to choose x so that this combination matches up closely to the user data vector, which we'll call **b**. A way to measure the closeness of the vector A*x to the vector b is to take the norm of their difference, namely **||Ax – b||**.

Therefore the goal is to intelligently choose x so that we match Ax as closely as possible to b. Mathematically, we write this as an optimization problem:

Minimize (with respect to variable x): ||Ax – b||

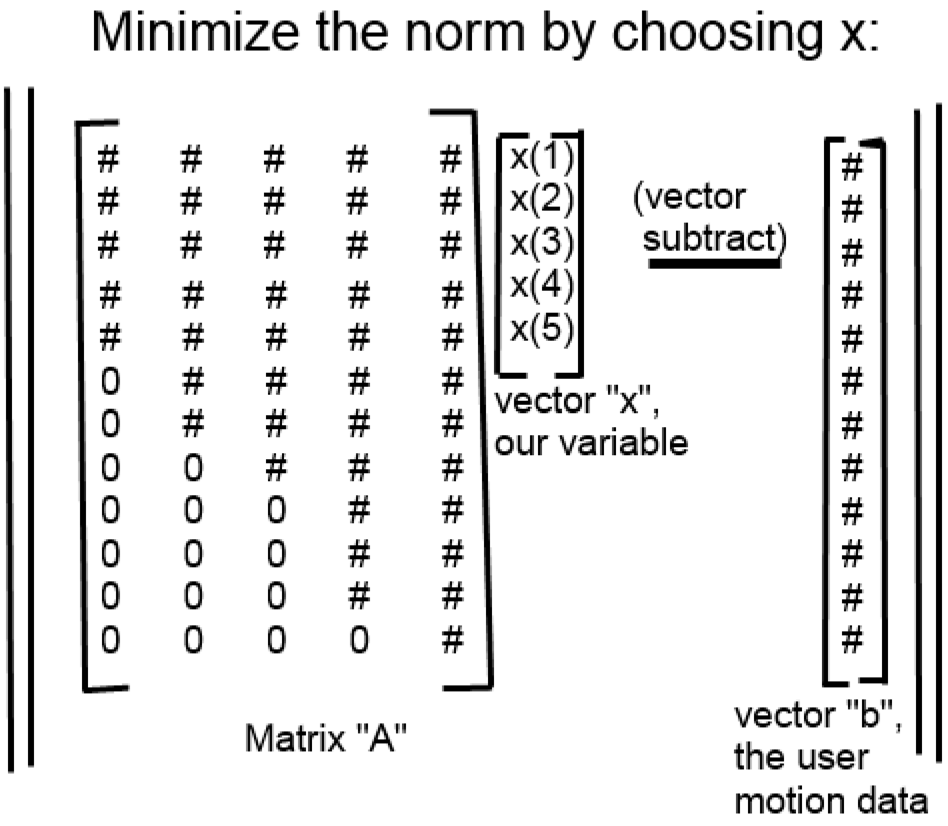


Figure 2: Norm Minimization Mathematics

At the highest level, this is how the algorithm works: every time we get a new piece of user data, we update the b vector by adding it to the end and removing the first like a queue. Then we solve for the vector x. If the value of the norm, after thresholding, indicates a match, we conclude that we recognized a motion. Then we look at which component of x has the largest value, indicating which column has the largest positive correlation with user's data. So then if x(3) is the largest, we conclude that we just recognized template #3 (in sorted length order).

However, there are a multiple problems with the vanilla approach as described. The rest of section will relate to defining these problems and addressing them by modifying the algorithm.

1) *What distance metric to use?*

Mathematically, a norm is a metric used to define the "distance" between 2 vectors. Of course, there are different ways of specifying distance. The most common, in engineering applications, is the 2-norm, i.e. Euclidean distance. This takes the sum of the squared differences of the two vectors' elements. Other norms measure things like the sum of absolute value differences, the max difference, etc.

Through testing, the 2-norm was determined to be a good metric and so we stick with that. It has the additional advantage of making the optimization problem easier to input into existing numerical optimization software; there is also real-time computation savings. The next sections will explain more in depth.

2) *Negative x values do not make sense for our application.*

We are interested in finding which templates *positively* correlate with the user's data. A negative correlation does not have any obvious physical meaning with respect to gesture recognition, and can hinder the predicted match probabilities of other motions.

What we would like, ideally, is that the vector x represent a *probability distribution* over the space of possible templates. So if x(4) = 0.8, then there's an 80% probability of the 4th motion being the one we just saw. To do this, we modify our problem into a *constrained optimization problem*. Essentially this means keeping the same objective function – we want to find a solution x that fits the user's data well – but we want to impose conditions on x that make the solution more meaningful to us.

Mathematically, we write the following:

Minimize (with respect to variable x): ||Ax – b||

Subject to: x(i) >= 0, and sum(x(i)) == 1, i ranges from 1, 2, …, numTemplates

What types of variables are nonnegative and sum to 1? Probability distributions of course. We now can interpret the optimal solution of x of the above problem as we did above: x(3) = 0.74 means the third template has a 74% match.

One might wonder how these additional constraints change the optimal solution. Is the norm still minimized? Of course, we are still finding the best fit possible, but with respect to our new definition of what x "means".

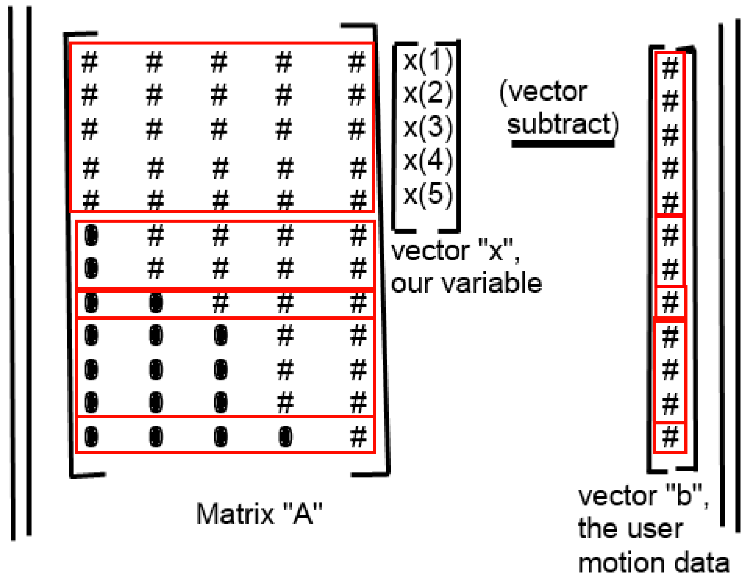
Constrained optimization is handled in the proposed frameworks, which will be discussed in the next sections.

3) *How do we compare to the shorter templates even though the buffer is of full length?*

This is where the problem gets a little messier. Note that in the current Ax – b format, the shorter columns (templates) in matrix A are augmented with zeros. That means that we are in fact extrapolating fake (zero) values for these templates and comparing it to the user's data. This has no correspondence to true physical template information.

What we do to overcome this is *sub-block* the A matrix and the b vector into common-length sequences, and minimize the sum of norms of all the sub-blocked problems. The following visual graphic will clarify:

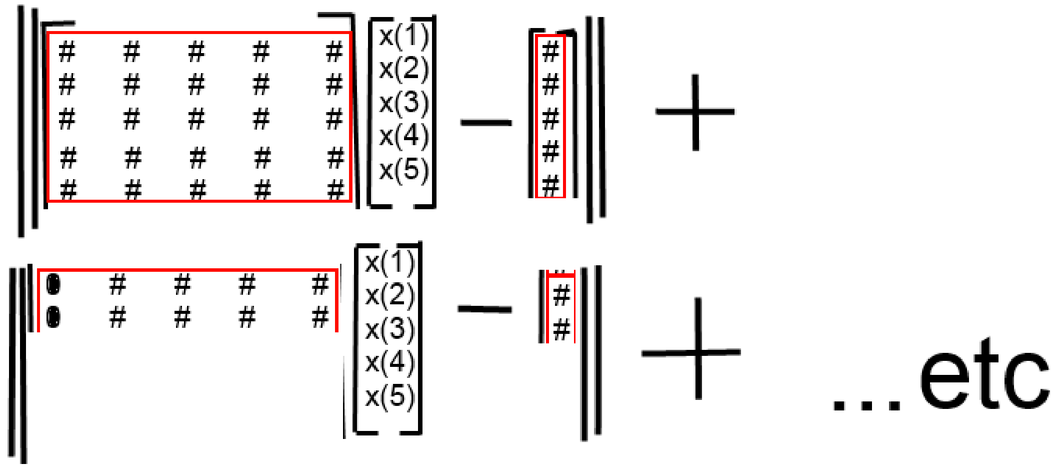
Sub-blocking: a visual explanation



- Group all equal-length rows together into sub-matrices.
- Divide the user input vector into sub-vectors of the same (row) lengths
- We will NOT be using 0's as the placeholders; we will figure out what to use below:

- In this example we have 5 sub-blocks, for 5 templates. This is because all templates are of different lengths. Note that we could have less than 5 sub-blocks using this same methodology.

- Now: minimize the sum of norms of the sub-block matches:



- KEY POINT: in the second term with sub-block 2, we only care to see the match between templates 2,3,4,5 with that portion of the user data.

- Why? Because template 1 (in sorted order) isn't even that long, so we don't want to give it "points" (additional x(1) weighting) compared to the rest of the data. That would create false positives.

Figure 3: A visual explanation of sub-blocking

The main advantage of sub-blocking different length templates is this: we can put weights in front of each of the norm terms, so we can favor comparisons for the *common larger portions* of the motions. Consider having a max template length of 60, with sub-block lengths of 35, 15, 5, 2, 2, 1. Clearly the bulk of the motion to be recognized is in the first sub-block, so we could assign that norm term a weighting of 35/60 = 0.5833. Similarly, the second-last sub-block (length 2) would have a weighting of 2/60 = 0.0333.

The problem would then look like:

Minimize 0.5833*norm(sub-block 1) + 0.25*norm(sub-block 2) + … + 0.017*norm(sub-block 6)

Subject to (constraints on x)

Of course, this particular weighting approach favors the beginning of the motions over the end of the motions. In practice this is a perfectly acceptable way to approach the problem of differentiating between vastly different motions at once: e.g. a bicep curl versus a bench press. For motions that involve the same start but slightly different endings, this weighting approach can be modified.

We'll call these weights the *alpha weights*.

One question remains unanswered: what should be put in place of zeros? What is the problem with using zeros in the first place? The problem is that it is "fake data", i.e. numbers close to zero in the user data buffer will match up with these placeholders, giving unfair weight advantages to shorter templates in that situation.

Assuming our template (and user input) data is normalize to be between [-1, +1] (which we will do), having this unknown placeholder data at zero – halfway between the data range extremes – might not be so bad. In initial testing, it is not, though we see that different motions have rather different threshold values.

The other approach we can use is to fill these placeholder spots with the *mean value* of the rest of the templates/columns at that row, where the row represents the time slice. If the user's data better matches any of the other templates, the scores will reflect this, since the average of a few templates will never be closer than the actual matching template.

While there is an opportunity for false positives, they become more likely only when the motions are very similar. If this is the case, then it means we are likely using the algorithm as a way to store redundant templates for increased detection accuracy. Thus these are not really "false positives", unless we care deeply about the accuracy of detecting the difference between bicep curl 1 and bicep curl 2.

An ideal usage of the algorithm then is as follows: for each motion, have multiple recorded templates of that same motion. This reflects the power of the algorithm: the speed to compare 15+ templates in real-time, aided by data redundancy from multiple templates.

4) *How do we combine information from multiple axes?*

To do this, we can increase the summation dimension to include the (say, 6) axes. Extending this approach to the alpha weights, the full problem can then look like this:

```
Minimize (with respect to x): F(x), defined as follows:
    For every axis j,
        For every sub-block k,
            F = F + alpha(j,k)*norm(M(j,k)x – b(j,k))
Subject to:
    Sum(x) == 1, x >= 0 [for all elements in x]
```

The index *j* represents the axis, i.e. j = 1,2,…,6 for accelerometer x,y,z and gyroscope x,y,z. So we've taken our template recognition task from 1-D to 6-D. The matrix **M(j,k)** represents the kth sub-block for the jth axis. So referring to the above sub-block examples, imagine having 6 setups like that, for accX, accY, …, gyroZ.

5) *How can we custom-tune parameter weights in the future?*

Naturally, we have a lot of freedom with the alpha(j,k) weights. Imposing that they sum to 1 (over j,k) is a useful normalization technique to keep the thresholding results in a reasonable and predictable range. The initial implementation of alpha(j,k) weights sets the axis weights equally, but weights sub-blocks based on length (more length == more weight).

The first obvious change to make would be manually tuning the axis weights, similar to previous algorithms. A later section will describe the software that sets up the algorithm's static parameters, and in this section it will be clear how to change the alphaWeights.

3 – Optimization Background

The multitemplate algorithm relies on numerical optimization methods to solve for the template match likelihoods. While a review of this topic is outside the scope of this document, here is the “view from thirty thousand feet” that one should understand.

Every optimization problem is of the following form. There is a variable, x, which can be a scalar, vector, or matrix (vector in our case). We have an *objective function F(x)* that we want to minimize by choosing x. As a simple scalar example is this: minimize the function $F(x) = (x - 5)^2 - 2$. From calculus or visual inspection, we know that the *optimal solution* is $x^* = 5$. This is the value of the variable that minimizes the function. The *optimal objective value* is the value of the objective function at x^*. In this example, $F(x^*) = -2$.

The same definitions hold for higher dimensions. The variable x can be a vector of length 1000, and the function F(x) can be a quadratic in the form $F(x) = x'Px - q'x - r$ where P is a matrix, q is a vector of length 1000, and r is a scalar, and the prime (‘) operation denotes transpose. What’s important is that the objective function returns a *single scalar number* no matter the size of the input. That way the algorithms – which are iterative – always have a notion of what “less” means.

The other important concept is that of *constraints*. Basically, what values is the variable x allowed to take on, while trying to minimize the objective function? Other names for this concept are the *problem domain* or the *feasible set of solutions*.

In order to interpret x as a probability distribution, we put constraints on it: all the elements (of the vector) must sum to 1, and all the elements must be nonnegative. Within this problem domain / feasible set, we seek to minimize the norms discussed above. As an example, minimizing $F(x) = (x-5)^2 - 2$ subject to x >= 6 has the optimal solution $x^* = 6$, with $F(x^*) = -1$. To see this, one can plot the function, since the variable is in 1-D. Note that compared to the above, having constraints on the variable shifts the solution.

The general form of an optimization problem always looks like this:

Minimize F(x)

Subject to $G_i(x)$ <= 0, i = 1,…,m

$H_i(x)$ == 0, i = 1,…,p

There are two types of possible constraints: inequality constraints and equality constraints. In our problem, F(x) = sum(j)(sum(k)(alpha(j,k)norm(M(j,k)x – b(j,k))), we have the (vectorized) inequality constraint x >= 0, and the equality constraint 1’x == 1, where 1’ is a row vector of 1’s the same length of x (and that inner product is equal to the sum of all elements of x).

Can we solve all types of optimization problems - globally?

No. Certain combinatorial problems are NP hard; certain problems will have large numbers of local extrema and saddle points.

But there exist a class of optimization problems, called *convex optimization problems*, which can be solved in polynomial time using efficient numerical algorithms. Convex Optimization is a field of applied math becoming increasingly popular in engineering applications over the last few decades. The leading reference on this topic is Boyd and Vandenberghe's *Convex Optimization* (2004).

It turns out that the problem we have developed does indeed fall into this category; specifically, it is a *quadratic program*, which is a type of convex optimization problem. That means that there exists efficient software to solve our problem in polynomial time. In fact, due to the problem's small size, and the time-series nature of solutions being similar, we can repeatedly solve the multitemplate matching problem in sub-10ms times for around 15 6-axis templates with data lengths up to 60 points per template.

The software used for algorithm development and debugging was CVX for MATLAB (http://cvxr.com/cvx/), while the real-time Java implementation was done with JOptimizer (http://joptimizer.com/). In the following sections we will examine the full software flow of setting up a multitemplate analyzer in an application.

4 – Data Preconditioning

The main form of data formatting performed is range normalization, so that all accelerometer and gyroscope values – of both the templates and the real-time user data – are between [-1, +1]. To do this, something like the following is done in preparation software for the templates (example in MATLAB):

```
for i=1:numTemplates
      T{i}(:,1:3) = T{i}(:,1:3) / ACC_MAX;
      T{i}(:,4:6) = T{i}(:,4:6) / GYRO_MAX;
end
```

This section may be updated in the future to include additional pre-processing.

5 – Test and Setup Software – Matlab

The purpose of the Matlab software suite is to set up and test the algorithm in a simple, high-level format. The two relevant files are *OrganizeData.m* and *model_test.m*. The former is run once in the workspace, and then the latter can be run to test the multitemplate matching against a long sequence of user data (warning: full test may take up to 2 hours without modifying various parameters). The software is available at:

https://github.com/kiwiwearables/MultiTemplate-Android-Test/tree/master/MultiTemplate_MATLAB

Setting up the problem as a quadratic program

As mentioned in the above section, the algorithm requires use of algorithms that solve quadratic programs, a subset of convex optimization programs. While the full mathematical derivation is somewhat long and tedious, there are a few key points that will allow one to re-derive it for themselves. First, let's examine the form of a quadratic program, with variable vector x of length n:

Minimize 0.5*x'Px + q'x + r

Subject to Gx <= h

Ax == b

Source: http://www.joptimizer.com/quadraticProgramming.html

Therefore to use our program with JOptimizer eventually in Java, it will be useful to phrase the problem in this format, and test that we have done it correctly with Matlab. Our actual problem, if we break the objective function into individual terms (since the min(A(x)+B(x)) is min(A(x)) + min(B(x)) for convex problems), looks like:

Minimize norm(M(j,k)x – b(j,k))

Subject to x >= 0

1'x == 1

We need to convert this to the form above. Then we can sum up the terms over all j,k indices and remain in that form since quadratic functions are homogeneous.

Note first that the optimal solution of norm(y) is the same as norm(y)^2, i.e. they are monotonic with respect to one another. Note also that we are using the 2-norm, so that for a vector y, norm(y)^2 = y'y, i.e. the inner product of the vector with itself, i.e. the sum of the squares of the elements. Therefore,

norm(Mx – b)^2 = (Mx – b)'(Mx – b)' = x'M'Mx – 2b'Mx + b'b

So clearly we can set P = 2M'M, q' = -2b'M, r = b'b. Note that we will omit r, because shifting by a constant does not affect the optimal solution x*. As well, we can multiply everything by the appropriate alpha(j,k) constant (which is absorbed into P and q).

Now in summing up over all our norm objectives, we have simple formulas for creating P and q for use with standard QP solvers:

P = 2*sum(sum(alpha(j,k)*M(j,k)'*M(j,k)))

q = -2*sum(sum(alpha(j,k)*M(j,k)'*b(j,k)))

Where we have taken the transpose of the q' expression, and we understand that the sum operations run over all axes j, and all sub-blocks k.

Next, we need to organize the constraints. The condition x >= 0, written as Gx <= h, can be done as follows: take G to be –I, the negative identity matrix, and h to be 0 (as a vector). Then every row of Gx <= h is equivalent to –x(i) <= 0, i.e. x(i) >= 0 for every element in x.

Finally, the equality constraint is already in the required form. For Ax == b, take the matrix A to be 1', which is a row of 1's the length of x. Then b is just the scalar 1. Again, this constraint is that the sum of all elements in x is equal to 1.

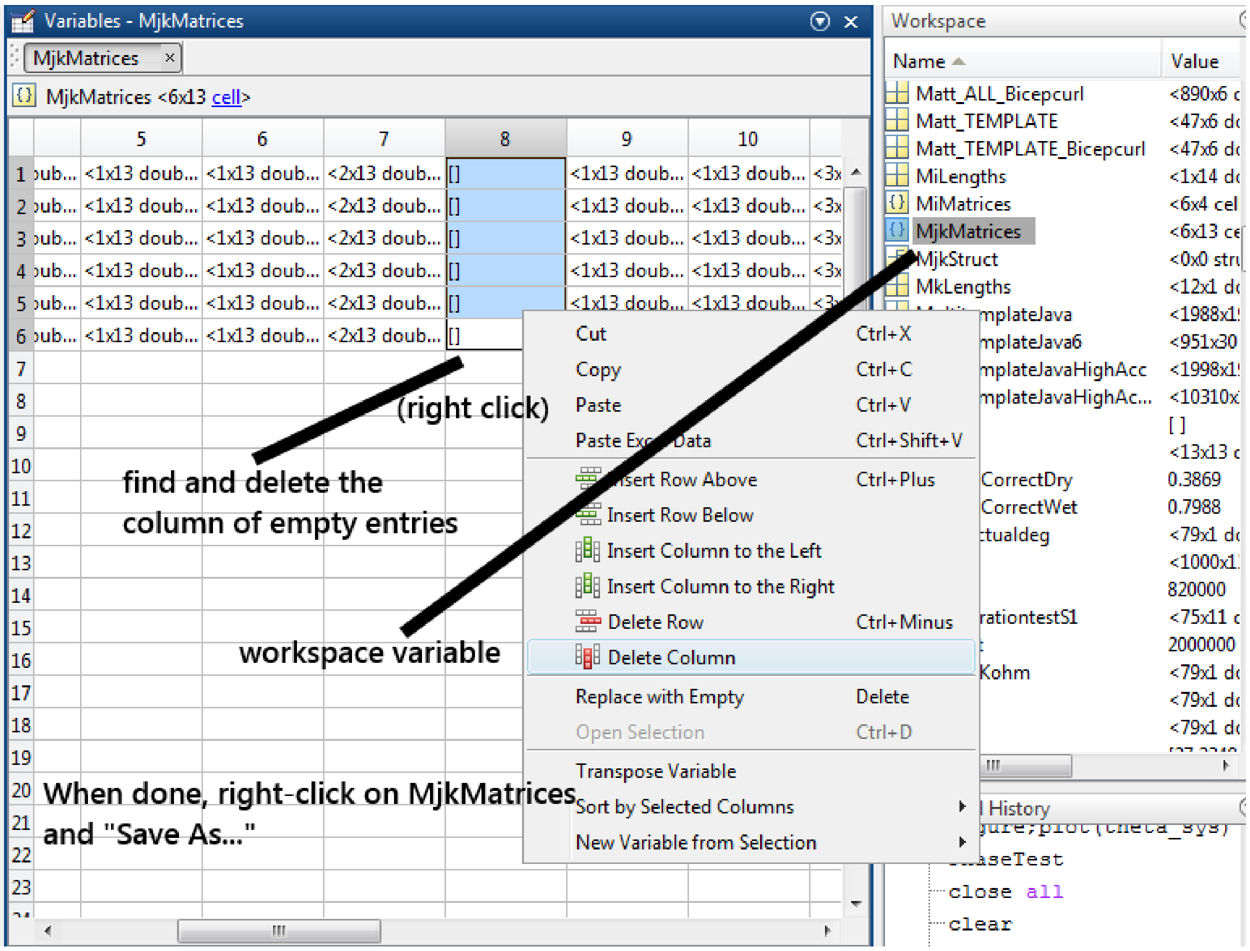
Now with all the pieces of the puzzle in place, one can make the following type of function call to the numerical optimization libraries:

(p*, x*) = QPsolve(P, q, r, G, h, A, b)

What will be returned is a scalar p*, the optimal value of the function over the feasible set. This is the value we will use for thresholding; if p* < threshold, then a motion has been detected. If a motion is detected, we can look at the optimal solution x*, and if x*(i) is the highest value, then motion i (in template-sorted order, see above sections) is the motion recognized.

Data Exporting from Matlab

The purpose of the script "OrganizeData.m" is to create, in Matlab, the Mjk matrices and the alphaWeights. Before exporting, one small step is needed. In the workspace variables (in the upper-right panel), we need to delete "empty columns" in alphaWeights and MjkMatrices. Empty columns occur when the number of sub-blocks is strictly less than the number of templates; this occurs if multiple templates have the same length. The image below shows an example of this process:



	5	6	7	8	9	10	
1	<1x13 doub...	<1x13 doub...	<2x13 doub...	[]	<1x13 doub...	<1x13 doub...	<3x
2	<1x13 doub...	<1x13 doub...	<2x13 doub...	[]	<1x13 doub...	<1x13 doub...	<3x
3	<1x13 doub...	<1x13 doub...	<2x13 doub...	[]	<1x13 doub...	<1x13 doub...	<3x
4	<1x13 doub...	<1x13 doub...	<2x13 doub...	[]	<1x13 doub...	<1x13 doub...	<3x
5	<1x13 doub...	<1x13 doub...	<2x13 doub...	[]	<1x13 doub...	<1x13 doub...	<3x
6	<1x13 doub...	<1x13 doub...	<2x13 doub...	[]			

Name	Value
Matt_ALL_Bicepcurl	<890x6 c
Matt_TEMPLATE	<47x6 dc
Matt_TEMPLATE_Bicepcurl	<47x6 dc
MiLengths	<1x14 dc
MiMatrices	<6x4 cel
MjkMatrices	<6x13 ce
MjkStruct	<0x0 stru
MkLengths	<12x1 dc
mplateJava	<1988x1!
mplateJava6	<951x30
mplateJavaHighAcc	<1998x1!
mplateJavaHighAc...	<10310x
	[ ]
	<13x13 c
CorrectDry	0.3869
CorrectWet	0.7988
:tualdeg	<79x1 dc
	<1000x1
	820000
rationtestS1	<75x11 c
	2000000
Kohm	<79x1 dc
	<79x1 dc
	<79x1 dc

(right click)

find and delete the column of empty entries

workspace variable

When done, right-click on MjkMatrices and "Save As..."

Figure 4: Data formatting to prepare structures for Java environment

In the workspace, one can right-click on these variables and "Save As…" .mat files. These files can be used for importation into a Java application: save these variables as file names "mjkmatrices.mat" and "alphaweights.mat" into the application's *res/raw/* folder. The Android section will explain how these variables are brought into application.

Finally, one can test the full algorithm on the sample collected dataset (which happens to be weightlifting) via the script "model_test.m". To prepare running this script, do the following:

- load the MultiTemplate.mat file into your Matlab environment
- Run "OrganizeData.m"
- Run "model_test.m"

The test script is exceedingly slow; its purpose is to test correctness, so it uses a very slow high-level implementation within the CVX framework (i.e. it doesn't do things like take previous solutions – or even optimization problem setup – into account before trying on the next time slice). As a result, the following variables in that script may be helpful to alleviate testing time:

- "skip": this variable decides how many time slices to skip in time slices per analysis. E.g. skip = 1 analyzes every time slice; skip = 4 does every fourth, etc. The overall broad patterns in data are very similar for skip being 1 or 4.

- "fakeLim": the test data has overall ~10000 time slices. It may be convenient to set this lower to test just the first 1000 points for example (which would cover a set of 10 bicep curls in the test data).

The algorithm returns/creates two main variables in the workspace, corresponding exactly to (p*, x*) as described above. *optVals* is a vector of length ~10000 that stores the optimal values from the algorithm, i.e. p*(i). This is the value to examine for thresholding. The variable *Xs* is a Tx13 (T ~ 10000) matrix where every row contains the optimal solution x*(i) for the 13 test templates. To see which template was predicted at, say, time step 556, we would run (in Matlab's command window):

```
>> Xs(556,:)

ans =

  Columns 1 through 5


    0.0000    0.0000    0.0000    0.0000    0.0000


  Columns 6 through 10


    0.0000    0.0191    0.0000    0.0000    0.0000


  Columns 11 through 13


    0.6601    0.3208    0.0000
```

From this, we see that template #11 (in length-sorted order) is predicted with 66% confidence, by far the highest. However, recall that predictions are only useful at time *i* after thresholding on p*(i).

A final detail – to view results in length-sorted order, observe in "OrganizeData.m" the ordering of the T{} cell array, and then cross-reference with the variable "indicies". E.g.

```
indicies =

  Columns 1 through 9

    10     2     8     9     3    13     7    12    11

  Columns 10 through 13

     6     1     4     5
```

Therefore we see that sorted-length template #11 actually corresponds to the first index of T, which is Matt's bicepcurl. This is what we expect; the test data starts off by iterating over all test data of Matt doing bicepcurls.

6 – Application Software – Java & Android

This section will explain the relevant components of the Java class implementing the Multitemplate motion analysis algorithm. The class is organized in such a way that users can restructure the software to suit their app, e.g. replacing AsyncTask with their own background thread management classes.

The relevant files discussed are "MainActivity.java" and "MultitemplateAnalyze.java", available at:

https://github.com/kiwiwearables/MultiTemplate-Android-Test/tree/master/src/com/joptimizer

Algorithm Setup and Data Importing

The MultitemplateAnalyze class requires some initial setup before use in application. Its constructor requires 4 arguments:

```
mTA = new MultitemplateAnalyze(6, 13, tv, getApplicationContext(), true);
```

The first two arguments are number of axes (e.g. 6 for ax, ay, …, gz) and the number of templates, respectively. The last two arguments are for debugging – application context is for UI interaction, and the last argument is a debug flag.

```
// HOW TO ANALYZE REAL-TIME DATA:
	double[] newMotionData = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6}; //ax, ay, ..., gz
	mTA.new AnalyzeMultiTemplateTask().execute(newMotionData);
	double detectionVal = mTA.getOptVal(); //threshold on this value to detect if 
motion present
	double[] solution = mTA.getXstar(); //examine this vector to determine which 
motion detected
```

This implements the logic discussed throughout: feed a new point in time of motion data; this is stored in a (queue) buffer representing the bjk data. Then execute() on the AsyncTask class to run the analysis in a background thread. Then get the optimal value to threshold on, and finally, examine the solution vector to see which template was matched.

In the constructor, the exported data files (.mat format, discussed in previous section) from Matlab are stored in res/raw/. In the constructor, they are pulled in via the following:

```
// //////////// import matlab data ///////////////
	try {
		createMjkMatrices();
		// access via Mjk[j].get(k) for the appropriate matrix
		createAlphaWeights(numAxes);
		createPmatrix(numAxes, n);
		createUserTestData(); // NOTE - takes about 2 mins to load..
		if (tv != null)
			tv.setText(printMatrix(P));
	} catch (Exception e) {
		Toast.makeText(mContext, e.toString(), Toast.LENGTH_LONG).show();
```

```
        }
```

The functions themselves are rather straightforward to debug; the biggest issue will be file name consistency, and possibly Matlab variable name consistency. For example, the alphaWeights variable should actually be named 'alphaweights' (without the capital 'w'), which is used in this call in createAlphaWeights():

```
MLArray mlArray = reader.getMLArray("alphaweights"); //all lowercase var names
```

One final note – there is a createUserTestData() function that optionally imports a test Bjk matrix that can be exported from Matlab; in particular, it is the variable "B" obtained after running "model_test.m". This proved to be useful for debugging, as afterwards we can save optimal values and optimal solutions to a .csv file and compare against Matlab results.

7 – Real Time Performance

The setup described in the previous section will create the P matrix from the alphas and Mjks, but the bjk vectors must be computed in real-time based on user data. To do this, new data points are shifted into a queue (see above sub-section), and internally the queue is converted into a double[][] array for quick matrix computation.

All code needed to run the algorithm is implemented in an AsyncTask with a synchronized lock (so only one thread can ever read the data buffer at once). The following steps are taken:

- Buffer the new user data point

- Compute the "q" vector for the quadratic program (see previous section)

- Create a new object of the JOptimizer numerical library, i.e. solve(P,q)

- Set the initial point solution guess as the previous solution (*)

- Call optimize, get double[] sol and double optVal

* - this is where we get the tremendous speed of the algorithm. Iterative optimization on time series data has a great property: the solution at time t is very similar to the solution at time t-1. Stated differently, no matter the motion, the collected motion data at time t is going to be similar to time t-1, therefore we are solving a very similar optimization problem, and therefore our iterative method (implemented via JOptimizer) will require very few iterations.

In fact, it can require only 1. The class of algorithms underlying the solver, called *Interior Point Algorithms*, typically take 20-80 iterations to converge to a globally optimal solution. However, through testing, and thanks to the above time series property of the data, just 1 iteration will let us have an almost optimal solution. The figure below shows the accuracy trade-off (for the optimal value over a series of test data) this creates:

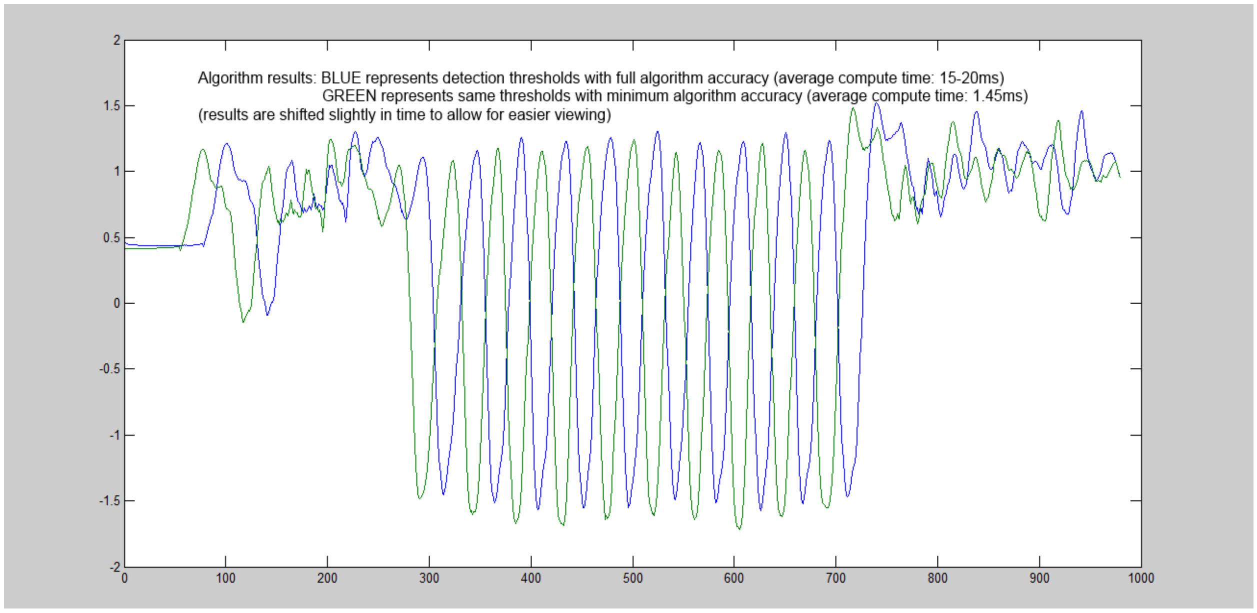


Figure 5: Optimal value comparisons over test data. The blue is Matlab (reference), and the green is the Java solver implementation with the minimum iterations (real-time). As we see, accuracy is within a few percent. Results are shifted in time for easier visual comparison.

Is the slight accuracy trade-off worth it?

For reference, we tested **13 templates** with a max length of **324 data points per template** (max length of 54 x 6 axis) and achieved average computation times of **2 milliseconds per solve**.

This leaves room to grow in various dimensions; given that the fastest sample rate encountered across devices so far is 100Hz, the algorithm has a max computational window of 10 milliseconds per new data point solve. Without much trouble, we could likely double the number of templates. Similarly, we could probably handle slightly longer templates, perhaps up to 600 points (100 points per axis).

The scaling will be dependent on the size of the data, i.e. (13,324) describes the data size. It is not clear at what rate the data will scale; the most taxing operation seems to be a matrix-vector multiply of the longer dimension (in computation of the q vector).

Still, with plenty of room to add/lengthen templates, there is much opportunity to use data redundancy (multiple recordings of the same motion) to smooth out sensor noise without over-fitting.

8 – Future Work

Integration into existing Kiwi applications is the first task being tackled. After this, it will be useful to be able to, from the Java code, set up the full algorithm from motion templates stored in the Kiwi cloud. This will involve rewriting “OrganizeData.m” into Java. A useful task after that will be benchmarking real-time performance, for number of templates as well as overall template length.

It is also suspected that pre-filtering all data, both template and user, with a low-pass filter will help improve performance.