Change tables from fiugure to Table later on

Figure 1 - take-away box conaining bullet point version of specifications.

Figure 2 - flow-chart outlining the step structure

Figure 3 - table of all methods

Figure 4 - BPM maths

Figure 5 - BPM formula adjustment due to per100

Figure 6 - timeline for data window

Figure 7 - table containing criticisim of 3 cases.

Figure 8 - Take-away box with goals for testing.

1) Determine optimal setting

2) Determine quality of BPM code

3) Graph effect of changing iter\_windowITALICS length

4) Graph effect of changing min\_spacingITALICS

5) Graph effect of changing data window length

Figure 9 - Set thing

Figure 10 - Testing Environment Specifications

1) The testing environment has to be in a seperate file.

2) The testing environment has to emulate the moving window when it goes through test data.

3) The testing environment needs to be able to export results.

4) The testing environment needs to pass settings to services.py

5) The testing environment needs to receive information about the evaluation of each data window it passed to services.py.

Figure 11 - Testing Environment Diagram - INFORMATION ABOUT EVALUATION details needed

Figure 12 - Information passed from services.py to local.py

self.data = data

1) window length in time as well as data points

2) window density (data points per second)

3) BPM code setting (methods, benchmarks)

4) Maximum, minimum, average and variance of voltstamps

5) Percentage of iter\_windowsITALICS used

6) BPM

7) Beat times and voltstamps

Figure 13 - PROs and CONs of different f\_hat(x)

TABLE Index | Measure | PROs | CONs

1|

Measure: |Number of beats in window - actual number of beats in window|

PROs: Computationally inexpensive. Little manual work.

CONs: Tells us nothing about the spacing of beats.

2|

Measure: |BPM - actual BPM|

PROs: Measures number of beats and recognises uneven spacing.

CONs: Systematic error from error adjustment is carried through. Fluctuations in actual BPM either have to be ignored or entered manually. The real beats do not have to be detected because it is only the spacing of them that matters. Different 'violations' carry different weights.

3|

Measure: avg(|ti - actual ti|)

PROs: Would give most accurate measure if beats found by code are very close to actual beats.

CONs: Infeasible because it requires manual input of actual this. The number of actual and false beats may not be the same, leading to scoring complications.

Figure 14 - Error bounds for BPM as measure

TABLE Index | Issue | Error | Explanation

1|

Issue: Systematic Error from error adjustment

Error Bound: Copy from 1st table HELP

Explanation: Result was taken from before.

Issue: Fluctuations in the actual BPM

Error Bound: HELP calculations 2.

Explanation: In reality the heart rate changes constantly. The 'actual' BPM used for testing is an average, so it is not entirely accurate. Variations were less than one beat per 5 seconds.

Figure 15 - Variations in impact on quality measure

TABLE Index | Scenario | Impact in case of dense iteration of data windows | Impact in case of sparse iteration of data windows

1|

Scenario: Ommission of beat

Sparse: If the ommitted beat is outside of t1 and tn HELP (where t1 and tn are as in Figure 6ITALICS) it has no impact on the result. If it is inside it causes an error of HELP.

Dense: Since the moving window only changes slightly in each iteration, this beat will be ommitted from a number (approximately equal to the number of data points in a each window) of data windows. We see that the average error in these windows is HELP.

2|

Scenario: Shift of one beat

Sparse: Inside t1 and tn HELP there is no change, outside the change is HELP.

Dense: The average change is HELP

3|

Scenario: Addition of one beat

Sparse: HELP

Dense: Bounded by inside sparse

Figure 16 - Tested methods. Justifying why others were removed.

Figure 17 - Flow Chart

Figure 18 - Annotated flow chart

Figure 19 - min\_spacing graph

Figure 20 - iter\_window\_len graph

Figure 21 - window\_length graph

Figure 22 - Improvements

TABLE Index | Improvement | Difficulty of implementation | impact

1|

Improvement: Fix BPM error adjustment

Difficulty: Not difficult. Alternative methods have already been suggested in the Error Detection and AdjustmentITALICS section.

Impact: High. The error bounds for systematic error exceeded 10% of the BPM.

2|

Improvement: Record the actual time of heart beats so that more accurate measures of quality can be implemented.

Difficulty: Depending on the amount of available data (or the availability relevant established devices), this could be very time consuming.

Impact: High. As the results displayed in Testing ResultsITALICS suggest, a more accurate measure of quality is required.

3|

Improvement: Allow the selection of methods to vary depending on the data. Currently only benchmarks are adjusted as a function of average voltstamps.

Difficulty: Implementing this improvement may be challenging as it would require the code to distinguish between different data sets in order to select an appropriate method.

Impact: Small. Theoretically data from all users should be somewhat similar (even though there are differences in the duration of beats).

4|

Improvement: Decrease run time of the testing algorithm.

Difficulty: Some improvements could be easily implemented. The dense iteration for example, computed the BPM for each window even though only one data point changed.

Impact: High. A faster algorithm would allow for the testing of more combinations of methods and constants.

5|

Improvement: Refrain from using similar data and window length.

Difficulty: Depending on the run time of the algorithm, this could be simple to implement.

Impact: Small. An effective quality measure should work for any window size greater than two seconds.

Calculation 1 - Error for my bad error adjustment

Calculation 2 - Error for BPM as measurement function

Appendix 1 - Data Used

Appendix 2 - Phase 1 Results

Appendix 3 - Phase 2 a and b Results

Appendix 4 - Phase 3 a,b and c Results

Filip Frahm

s1232367

System Level Integration Practical

Group A: UberVest

INTRODUCTION (Level 1 heading)

With recent news about NSA surveillance, large scale data collection seems to have a negative connotation in our society. While fear for ones' privacy is justified,

large scale data collection can be of tremendous benefit. An application of it that stands out is the imrpovement of our society's health. At the University of

Edinburgh, researcher Andy Sims is using bioinformatics and genetic data to predict which cancer drugs are most appropriate for a breat cancer patient[3].

In San Francisco, Google-backed company Calico[2] gained access to Ancestry.com's genetic database with the intention of using genetic and family tree related data

to improve the human lifespan.[1]

The goal for our project was to create a vest that would 1)have as many health related sensors as possible and 2)process and present that data to the user. We believe

that such a vest would not only be attractive for personal use but that it could, similarly to the examples above, also advance medical research by providing data.

In the duration of the course, a temperature sensor and an electrocardiogram(ECG) were installed to a vest. Further, a website and mobile application were created to

process and present the data. While a respiratory sensor has not yet been integrated to the vest some research regarding future applications of such a sensor has been

completed.

My role in the project was to process data. Data processing included the conversion of data from the ECG to a beats per minute(BPM) value. Additionally, it

was my task to explore potential uses of a respiratory sensor (RESpeck). ADD STUFF ON RESpeck HERE

ECG DATA PROCESSING (Level 1 heading)

SPECIFICATIONS (level 2 heading)

The requirements for the ECG data processing code (BPM code) were to take ECG data from the server and to then return a BPM. Data passed from the server would be in a

python dictionary with unix timestamps as keys and integers representing voltage (voltstamps) as values. Voltstamps theoretically range from zero to 1024 but only

range from zero to around 360 in practice. In available data, data points were typically between 0.001 and 0.05 seconds apart. In order to be able to provide a live BPM

feed, the BPM code was to be applied to a moving window with length between zero and ten seconds. Of course this moving window always consists of consecutive data

points.

Figure 1

METHOD (level 2 heading)

The BPM code can be divided into two major parts. The first and more computationally expensive one parses through a given window of data and attempts to identify the

timestamps and voltstamps of heart beats. In the process, more information about the data is collected. The second part of the code uses the previously collected

information to predict how many beats would occur over 60 seconds.

BEAT DETECTION AND STEP STRUCTURE (level 3 heading)

In early stages of the project the approach to finding a beat detecting method was somewhat manual. A method that would theoretically detect beats would be chosen and

then tested against available data. This led to some issues. Firstly, little data was available for testin and the manual approach allowed little flexibility.

As a result, methods would frequently work for small samples, but not be flexible enough to work for larger samples. Secondly, the varying density of datapoints,

requires more versatile methods. A combination of different methods was required to attain a good output.

To avoid the above-listed issues, a computational approach was pursued. Instead of choosing methods manually, a fixed structure (the step structure) was created. The

step structure itself remains the same, but it allows for the choosing of different combinations of methods and benchmarks within it. To determine the optimal

combination of methods and benchmarks a testing environment was created (detailed in TestingITALICS). The computational approach doesn't just try more options and

parse more data than the manual approach, it also easily adapts to new data.

The step structure code starts with the window of data as described before and ends with a dictionary of suspected beats. It does so in four steps outlined in

Figure 2ITALICS. The individual methods(or conditions) for each step are described in Figure 3ITALICS.

Figure 2

Figure 3

Note that for all the step one conditions any of the following five logical statements can be applied to the condition:

1) Whether the condition is met or not has no impact on the selection of the iter\_windowITALICS.

2) If the condition is met, the iter\_windowITALICS will be selected regardless of the results of other methods.

3) If the condition is not met, the iter\_windowITALICS will be selected regardless of the results of other methods.

4) Unless 2) or 3) occurs, this condition must not be met for the iter\_windowITALICS to be selected.

5) Unless 2) or 3) occurs, this condition must be met for the iter\_windowITALICS to be selected.

As a result of testing (detailed in TestingITALICS), the best choice of benchmarks and methods emerged to be HELP.

BEATS PER MINUTE (level 3 heading)

The idea of the BPM is not to project how many beats a user will have in one minute, but rather to let the user know how many beats in a minute he would have at the

current rate. This makes the generation of a BPM value simple (since the beats have already been identified). We simply divide 60 seconds by the average time between

beats.

Simplifying,

Figure 4

ERROR DETECTION AND ADJUSTMENT (level 4 heading)

Since the ECG is easily affected by movement, the BPM code needs to adjust to unusable data. It does so in two ways. Firstly, it flags data windows with extremely high

variance (variance > 60). Secondly, it discards iter\_windowsITALICS that contain zero voltstamps. In practice zero voltstamps only occur as a symptom of extreme noise (the normal

range is 140 to 250). The percentage of iter\_windowsITALICS are used is then computed and used to adjust the BPM formula as described in Figure 5ITALICS below.

Figure 5

This adjustment is not correct. This method was chosen due to time constraints, but an alternative method is suggested implicitly in the criticism below. Given a

data window such as the one in Figure 6ITALICS below, the effect of the error adjustment depends on where the error occurs. The issue with the current method lies in

the fact that data outside of t1 and tn HELP is not considered for the BPM but it is considered in the error adjustment. Figure 7ITALICS discusses different scenarios.

Figure 6

Figure 7

TABLE Index | Scenario | Current Adjustment | Correct Adjustment | Error | Summary

1|

Scenario: Error occurs only in between t1 and tn HELP.

Current Adjustment: 100\*BPM¬old / percentage of windows used HELP

Correct Adjustment: BPM¬old\*(tn-t1)/(tn-t1-error\_duration) HELP

Error Bounds: HELP

Summary: Adujstment is not severe enough.

2|

Scenario: Error occurs only outside of t1 and tn HELP

Current Adjustment: 100\*BPM¬old / percentage of windows used HELP

Correct Adjustment: No adjustment

Error Bounds: HELP

Summary: Adjustment is too severe.

3|

Scenario: Error occurs inbetween t1 and tn HELP and outside of t1 and tn HELP

Current Adjustment: 100\*BPM¬old / percentage of windows used HELP

Correct Adjustment: BPM¬old\*(tn-t1)/(tn-t1-inside\_error\_duration) HELP

Error Bounds: weighted combination of 1 and 2 HELP so worst one.

Summary: Adjustment could be too severe or not sever enough.

In testing, data windows with high error rates (error > X% HELP) were discarded. Nonetheless, the above adjustment could cause significant error. Calculations for the

above error bounds are provided in the CalculationsITALICS seciont (Calculations 1ITALICS).

TESTING (level 3 heading)

The goals for testing were to determine the best possible benchmarks and combinations of methods (setting) for the step structure and to measure their effectiveness.

Further objectives were to graph the effect of changing the minimum spacing between beats, the length of iter\_windowsITALICS and most importantly the length of data

windows.

Figure 8

TESTING ENVIRONMENT (level 4 heading)

Goals two through five quickly follow from goal one, so this section will focus on reaching goal one. A more mathematical way of formulating the problem would be to

see all possible settings as a set (X HELP). Testing them is equivalent to mapping this set to another set(Q HELP), where each element is a measure of quality,

to then choose the best quality and map it back to X HELP. The idea behind the testing environment was to emulate this process. Of course X is far too large to test

every setting. The limitations of the testing environment and their impact is discussed in Testing ProcedureITALICS.

Figure 9

Since the BPM code is contained in a file (services.py) which is then compiled and used in the server, the testing environment had to be created in a seperate file

(local.py). Other specifications required for effective testing are described in Figure 10ITALICS. Figure 11ITALICS displays how information moves between local.py

services.py and a CSV file. Figure 12ITALICS lists the information that was passed from services.py to local.py.

Figure 10

Figure 11

Note that for testing purposes the order in which the code processes data windows is irrelevant as the BPM code draws no connection between consecutive data windows.

Figure 12

MEASURING QUALITY (level 4 heading)

Referring back to the mathematical formulation of the problem, we need a function (f(x):X->Q HELP) that measures the quality of a particular setting. As already

suggested in the pseudo code of Figure 11ITALICS, a Monte Carlo style approach was used to determine this funciton. First, a measure of quality for a particular

setting and data window was defined (f\_hat(x) HELP). This measure of quality was then applied to as many data windows as possible (with the same setting) to generate

an expected value for the quality of the setting (f(x) HELP). Three measures were considered for f\_hat(x) HELP.

Figure 13

Measure one is far too inaccurate to be of use and measure three is too impractical. Measure two has significant issues, but is the only viable method. Figures

14ITALICS and 15 ITALICS show an attempt to quantify them. The calculations behind the error bounds are in Calculation 2ITALICS.

Figure 14

An issure not addressed in Figure 14ITALICS is that the spacing of beats does not always matter and different types deviation of the computed heart beats from the real

heart beats have different impacts on the quality measure. In order to save computational cost not all available data windows were evaluated in the testing. This has

further impacts on the quality measure. The results below were generated by considering a sparse iteration to be an iteration that uses each data point exactly once

(for example by dividing all available data into evenly spaced data windows). Dense iteration was considered to be an iteration that fixes window length and only

changes the window by one data points each iteration. Results for changes in the dense iteration are always presented as an average of the effect on all data windows

relevant to the described deviation. To generate the results in Figure 15ITALICS it was assumed that all heart beats are read correctly except the ones described in

each scenario.

Figure 15

TESTING PROCEDURE (level 4 heading)

In my experience, it took about 0.017 seconds for a data window containg 200 data points to be evaluated. 200 data points typically correspond to X seconds HELP,

which is large enough to generate a BPM, but not so large that the spacing of beats can be uneven without strongly influencing the BPM. Having set restricted run time

to 24 hours (personal choice) this allowing for roughly 5,000,000 data windows of size 200 to be evaluated. The data available, attached in Appendix 1ITALICS, consists

of 5.5 minutes of data from three different subjects. The data contains intentional movement to make testing more realistic. Other data not considered in the

experiment was collected using electrodes. This was deemed irrelevant as there was significantly less noise. The 5.5 minutes of data corresponds to X data points HELP

and therefore provide roughly X-600 data windows to be evaluated. Additionally, there were X HELP methods combinations to choose from along with X HELP benchmarks and

constants. This left a total of XXX HELP data windows to be evaluated. Obviously this number is far larger than our limitation of 5,000,000.

By selecting the sparse iteration method the number of data windows to be evaluated was cut down to XXX HELP. Further, the number of considered methods was reduced.

Instead of the methods from Figure 3ITALICS, only the methods from figure 16ITALICS were considered. For more reductions in computational cost, the flow chart in

Figure 17ITALICS was followed. This flow chart is clearly imperfect. Testing for the best methods and then using those results to test for constants and benchmarks

leaves a gap. This approach limnits results to methods that work well with the chosen benchmarks and constants, preventing potentially effective combinations from

being selected. I decided to test for methods first because I had a better feeling for which ranges would be acceptable for constants and benchmarks (from informal

testing), but I had very little idea about which methods would work most effectively.

Regarding the min\_spacingITALICS, window\_sizeITALICS and iter\_windowITALICS constants, I worked under the theoretical assumption that neither of them should have an

impact on the data while they are within some accpetable range.

Figure 17

This left XXX help data windows to be evaluated for phase 1 testing.

TESTING RESULTS (level 4 heading)

The results of phases one, two and three can be found in Appendix 2ITALICS, Appendix 3ITALICS and Appendix 4ITALICS respectively. The best setting was found to be X

HELP. Elimination occured as described in Figures 18ITALICS, 19ITALICS, 20ITALICS and 21ITALICS.

Figure 18

Figure 19

Figure 20

Figure 21

In order to better test the resulting setting, it was applied the available data using dense iteration. The result was X HELP.

METHOD EVALUATION AND IMPROVEMENTS (level 3 heading)

The phase 3 graphs from the previous section were slightly worrying as you would expect the accuracy of the BPM code to increase with window size. Instead, the

accuracy decreased. While this may be partially attributed to increased exposure to error data(the sparse method was used) it suggests that the measuring criteria and

testing procedure were flawed (the best solution is not generalisable). This notion is supported by the final test using dense iteration as BPM deviated twice as much.

Figure 22ITALICS provides suggestions for improvement.