スマートフォンのモーションセンサを利用した
個人認証アプリケーションの開発

情 11-0170　高坂 賢佑

目 次

図 目 次

はじめに

スマートフォンが徐々に普及しつつある現在，スマートフォンの個人認証方法は画面上に表示されるソフトウェアキーボードのテンキーを用いたパスコード認証が大部分を占めている．しかし，この認証方法は画面ロックを解除するたびに画面に表示されたソフトウェアキーボードを目で見て指でタッチして操作する必要が有るため，ユーザにとって煩雑な作業である．また，パスコード認証では，あらかじめ決められた文字種の中から１つずつ選択したものを元にパスコードを構築していくという性質上，パターン数が限られ，自由度が限定されてしまう．そこで，パスコード認証が抱える認証の煩雑さを解消し，かつ，自由度が高くより直感的に個人認証を行えるアプリケーションを開発する．このアプリケーションには，一般的なスマートフォンに搭載されている加速度センサとジャイロセンサを用いることとする．

第1章　使用するセンサについて

1.1　加速度センサ

加速度センサとは，X 軸，Y 軸，Z 軸の基準軸に対して直線運動の加速度をそれぞれ検出し，値として取り出すことのできるセンサである．ここでいう加速度とは，端末における，単位時間あたりの速度の変化率のことを指し，図 1.1 における直線で示した矢印の方向が正の値，逆が負の値をとる．

1.2　ジャイロセンサ

ジャイロセンサとは，X 軸，Y 軸，Z 軸の基準軸に対して回転運動の角速度をそれぞれ検出し，値として取り出すことのできるセンサである．ここでいう角速度とは，端末における，単位時間あたりの回転角のことを指し，図 1.1 における橙色で示した回転の方向が正の値，逆が負の値をとる．

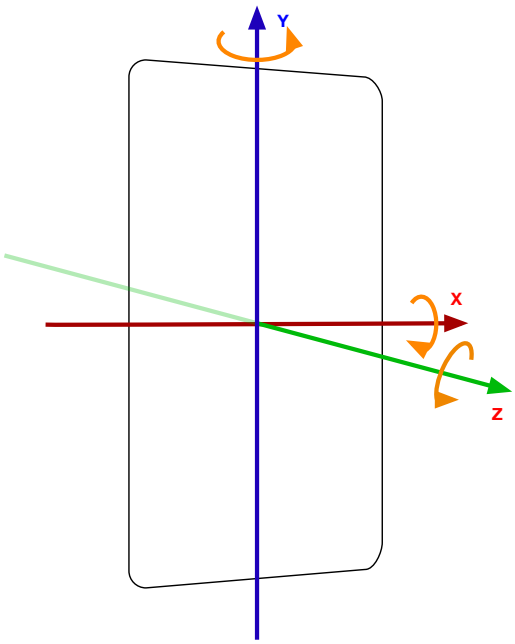

図 1.1: モーションセンサの座標系イメージ

第2章　先行研究

2.1　坂本の研究

坂本の研究では，ユーザが入力したモーションの数値化に加速度センサを用い，あらかじめ保存しておいた複数のジェスチャパターンのデータと認証時に入力したデータをパターンマッチング方式のアルゴリズムを用いて比較することで個人認証を行った．

しかし，このプログラムは扱うジェスチャによって認証率が高いものと低いものの二分化の傾向が見られるという問題点があった．

2.2　兎澤の研究

兎澤の研究では，モーションの数値化に加速度センサとジャイロセンサを用い，認証システムの中核に相関関数を用いたシステムを開発し，ユーザに三回入力させたモーションの平均値データと認証時に入力したデータの類似性を調べることで個人認証を行った．これにより，坂本の研究で指摘されていた成功率の二分化や，立体的な動きへの対応を可能にし，モーションの対応幅を広げることができた．

しかし，全体的な認証成功率が低く，特に手首のスナップを用いるような動きの小さいモーションに対して認証率が特に低く出るなど，対応できるモーションに限りがあるという問題点が指摘されていた．

第3章　本研究のシステム

3.1　本研究の概要

本研究では，兎澤の研究で挙げられていた，全体的な認証成功率の低さや対応できるモーションに限りがあるという点を改善することを目標とする．具体的には，より幅広いモーション，特に手首のスナップを用いるような比較的動きの小さいモーションに対しての，個人認証の全体的な認証成功率の向上を目指し，より実用レベルに近いアプリケーションの開発を行う．

3.2　システムの概要

本研究では，先行研究をもとに Android デバイス上で動作するアプリケーションとしてシステムを構築する．システムの動作フローを図 3.1 に示す．

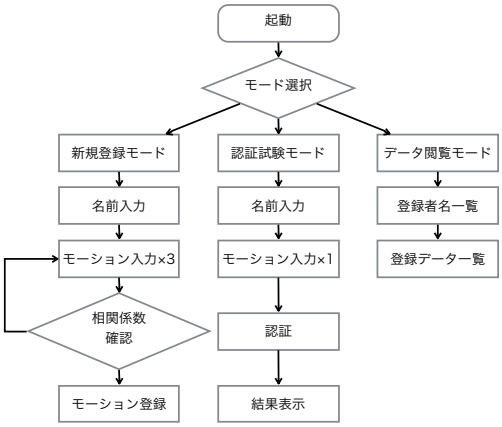

図 3.1: システム動作フロー図

アプリケーション起動時は，図 3.2 のような起動画面が表示される．ここで Start ボタンを押すことで，図 3.3 のようなモード選択ダイアログが表示される．

ユーザはまず，新規登録モードにおいて個人認証に用いる鍵情報となるモーションを登録する．新規登録モードでは，ユーザに同一のモーションを三回入力させ，入力した三回のモーションが同一のモーションであると確認できた場合にそのモーションの平均値をユーザのモーションデータとして登録する．

個人認証時には，事前に新規登録モードにおいてモーションデータを登録しておいたユーザ名を入

力させ，該当ユーザが登録されていると確認できた場合にのみ，ユーザにモーションを一回入力させる．入力されたモーションデータと指定したユーザ名で登録されたモーションデータ間の相関を取ることで個人認証を行う．

また，新規登録モードにおいて登録したユーザ名および登録データに関しては，データ閲覧モードにおいて閲覧することが可能である．

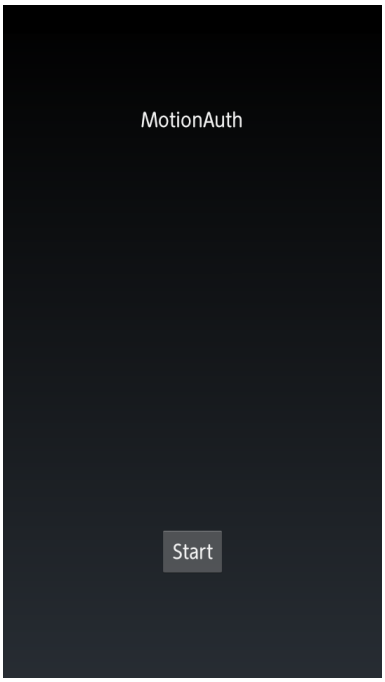

図 3.2: スタート画面

図 3.3: モード選択ダイアログ

3.2.1　新規登録モード

新規登録モードでは，まず図 3.4 の画面にて登録したいユーザの名前を入力させる．この画面にて OK ボタンを押した際に，テキストフィールドが空であれば，図 3.5 のような通知を表示してユーザに名前の再入力を促す．テキストフィールドが空でなければ次の図 3.6 の画面にてモーションの登録を行わせる．OK ボタンを押した際の名前の入力値チェックを行うコードをソースコード 3.1 に示す．

図 3.4: ユーザ名入力画面

図 3.5: エラー通知

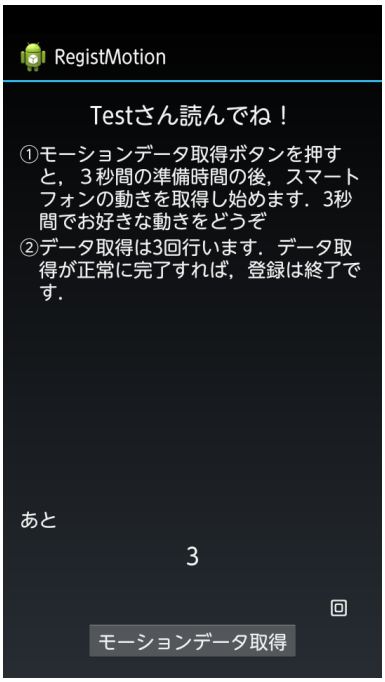

図 3.6: モーション登録画面

ソース 3.1: 入力値チェック

```
1  package com.example.motionauth.Registration;
2
3  import android.app.Activity;
4  import android.content.Context;
5  import android.content.Intent;
6  import android.os.Bundle;
7  import android.text.Editable;
8  import android.text.TextWatcher;
9  import android.util.Log;
10 import android.view.KeyEvent;
11 import android.view.View;
12 import android.view.Window;
13 import android.view.inputmethod.InputMethodManager;
14 import android.widget.Button;
15 import android.widget.EditText;
16 import android.widget.Toast;
17 import com.example.motionauth.R;
18 import com.example.motionauth.Utility.LogUtil;
```

```
19
20
21  /**
22   * ユーザに名前を入力させる
23   *
24   * @author Kensuke Kousaka
25   */
26  public class RegistNameInput extends Activity {
27      // ユーザが入力した文字列（名前）を格納する
28      public static String name;
29
30
31      @Override
32      protected void onCreate (Bundle savedInstanceState) {
33          super.onCreate(savedInstanceState);
34
35          LogUtil.log(Log.INFO);
36
37          // タイトルバーの非表示
38          requestWindowFeature(Window.FEATURE_NO_TITLE);
39          setContentView(R.layout.activity_regist_name_input);
40
41          name = "";
42
43          nameInput();
44      }
45
46
47      /**
48       * ユーザの名前入力を受け付ける処理
49       */
50      private void nameInput () {
```

```
51          LogUtil.log(Log.INFO);
52
53          final EditText nameInput = (EditText) findViewById(R.id.
                nameInputEditText);
54
55          nameInput.addTextChangedListener(new TextWatcher() {
56              // 変更前
57              public void beforeTextChanged (CharSequence s, int start,
                    int count, int after) {
58              }
59
60              // 変更直前
61              public void onTextChanged (CharSequence s, int start, int
                    before, int count) {
62              }
63
64              // 変更後
65              public void afterTextChanged (Editable s) {
66                  // ユーザの入力した名前をnameに格納
67                  if (nameInput.getText() != null) name = nameInput.
                        getText().toString().trim();
68              }
69          });
70
71          nameInput.setOnKeyListener(new View.OnKeyListener() {
72              @Override
73              public boolean onKey (View v, int keyCode, KeyEvent event) {
74                  if (event.getAction() == KeyEvent.ACTION_DOWN && keyCode
                        == KeyEvent.KEYCODE_ENTER) {
75                      // ソフトウェアキーボードの
                            Enterキーを押した時，ソフトウェアキーボードを閉じる
```

```
76                      InputMethodManager inputMethodManager = (
                            InputMethodManager) RegistNameInput.this.
                            getSystemService(Context.INPUT_METHOD_SERVICE);
77                      inputMethodManager.hideSoftInputFromWindow(v.
                            getWindowToken(), 0);
78
79                      return true;
80                  }
81                  return false;
82              }
83          });
84
85          // OKボタンを押した時に，次のアクティビティに移動
86          final Button ok = (Button) findViewById(R.id.okButton);
87
88          ok.setOnClickListener(new View.OnClickListener() {
89              @Override
90              public void onClick (View v) {
91                  LogUtil.log(Log.DEBUG, "Click ok button");
92                  // nameが入力されているかの確認
93                  if (name.length() == 0) {
94                      Toast.makeText(RegistNameInput.this, "名前が入力されていません", Toast.LENGTH_LONG).show();
95                  }
96                  else {
97                      RegistNameInput.this.moveActivity("com.example.
                            motionauth", "com.example.motionauth.Registration
                            .RegistMotion", true);
98                  }
99              }
100         });
101     }
```

```
102
103
104     /**
105      * アクティビティを移動する
106      *
107      * @param pkgName 移動先のパッケージ名
108      * @param actName 移動先のアクティビティ名
109      * @param flg     戻るキーを押した際にこのアクティビティを表示させるかどうか
110      */
111     private void moveActivity (String pkgName, String actName, boolean flg) {
112         LogUtil.log(Log.INFO);
113         Intent intent = new Intent();
114
115         intent.setClassName(pkgName, actName);
116
117         if (flg) intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_NEW_TASK);
118
119         startActivityForResult(intent, 0);
120     }
121
122 }
```

モーションデータ取得ボタンを押すと，三秒間のインターバルを挟んだ後に，モーションデータの取得を三秒間行う．この際，画面の下部にそれぞれの経過秒数を表示し，ヴァイブレーションにて一秒毎の時間経過を知らせるようにしている．

モーションデータの取得は加速度データ，ジャイロデータそれぞれを 0.03 秒ごとにセンサから取得し，X・Y・Z 軸のそれぞれから 100 個ずつ，計 600 個を一回分として取得する．

モーションデータの取得が三回行われると，図 3.7 のような計算処理待ちダイアログが表示される．そして三回分のモーションが同一のモーションであると確認された場合，図 3.8 のような登録完了ダイアログが表示され，OK ボタンを押すことで取得した三回分のデータの平均値データを保存し，起動画

面に移動する．また，三回分のモーションが同一のモーションでないと判断された場合，図 3.9 のような登録失敗ダイアログが表示され，OK ボタンを押すことでプログラム内部のカウンタが初期化されてデータの取り直しをさせる．

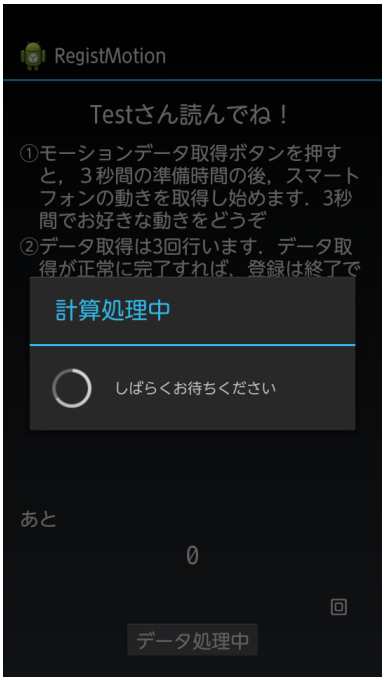

図 3.7: 処理待ちダイアログ

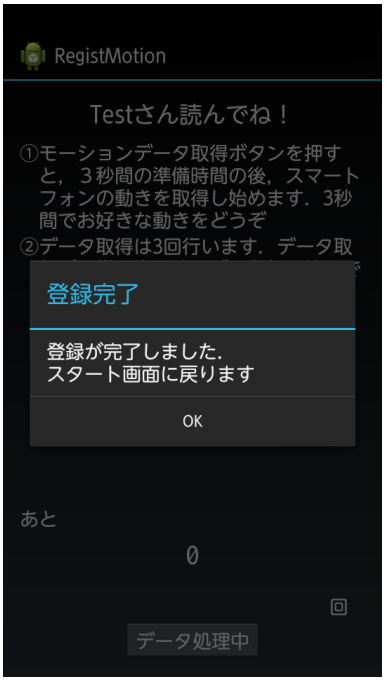

図 3.8: 登録完了ダイアログ

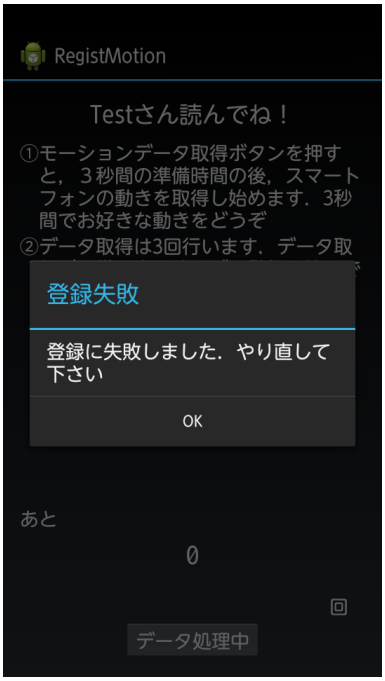

図 3.9: 登録失敗ダイアログ

3.2.2　認証試験モード

認証試験モードでは，図 3.10 の画面にて，新規登録モードであらかじめ登録されたユーザ名を入力させる．指定されたユーザ名にて既にモーションの登録がなされていることが確認できた場合にのみ，図 3.11 の画面にて個人認証を行わせる．モーションの登録がなされていなかった場合，図 3.12 のようなダイアログを表示する．

図 3.10: ユーザ名入力画面

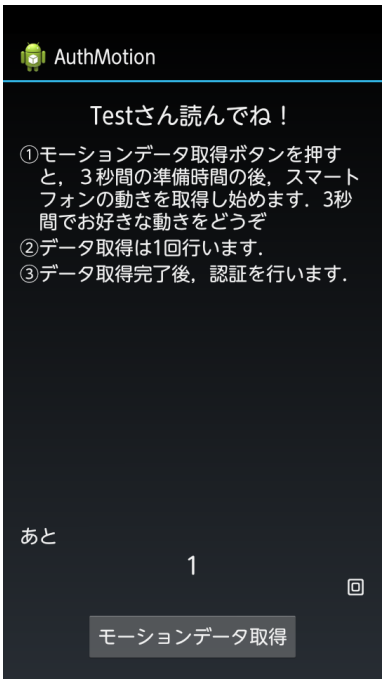

図 3.11: 認証試験画面

図 3.12: エラー通知

認証試験モードでは新規登録モードとは異なり，一回分のみのモーションデータ取得を行わせる．モーションデータの取得後，新規登録モードにて登録されたモーションデータを読み出し，これと新たに入力させたモーションとの相関を取ることで個人認証を行う．認証に成功すれば図 3.13 のような認証成功ダイアログを表示し，OK ボタンを押すことで起動画面へ移動する．認証に失敗すれば図 3.14 のような認証失敗ダイアログを表示し，OK ボタンを押すことで内部のカウンタを初期化して再度認証を行えるようにする．

3.2.3　データ閲覧モード

データ閲覧モードでは，新規登録モードにて登録されたユーザ名を一覧で見ることが出来る．また，その中からユーザ名を選択するとそのユーザが登録したモーションデータを見ることが出来る．このモードを選択すると，まず図 3.15 のようなユーザ名の一覧が表示される．ここでデータを閲覧したいユーザ名を選択することで，図 3.16 のようなモーションデータ一覧が表示される．

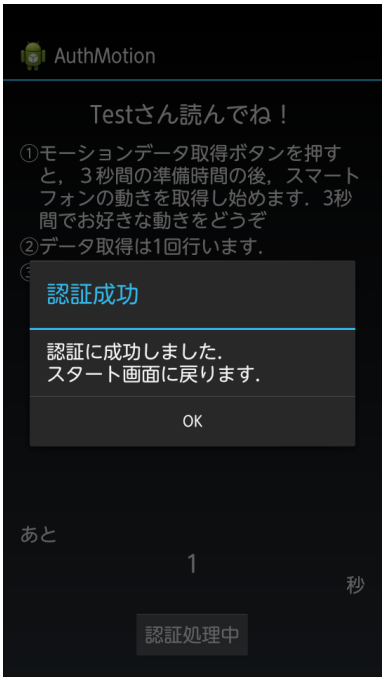

図 3.13: 認証成功ダイアログ

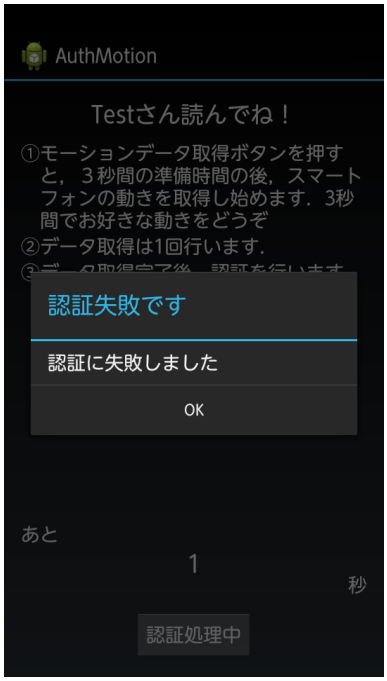

図 3.14: 認証失敗ダイアログ

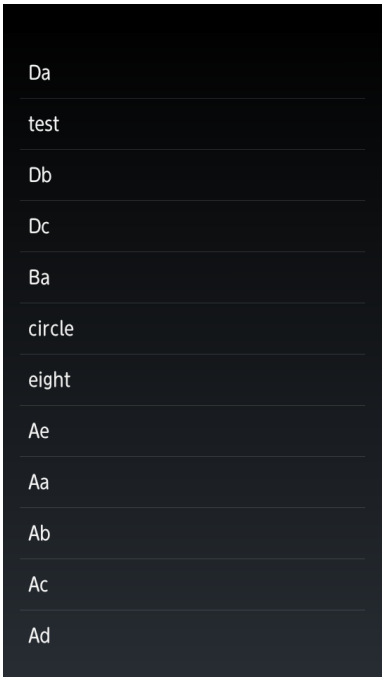

図 3.15: ユーザ名一覧画面

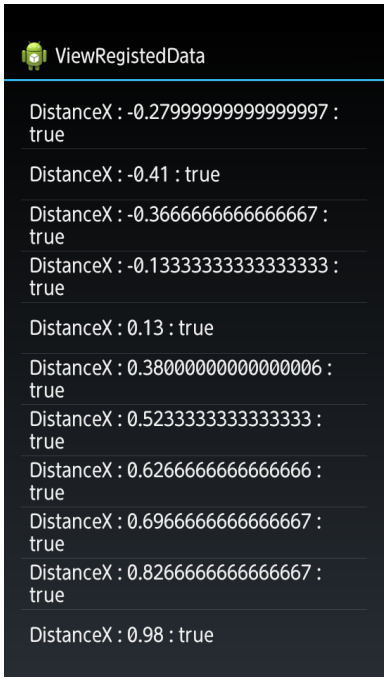

図 3.16: モーションデータ一覧画面

3.3　先行研究からの改善点

先行研究において指摘されていた，手首のスナップを用いるような比較的動きの小さなモーションにおいて，認証率が低く出てしまうという課題を解決するために導入した機能をここに挙げる．

3.3.1　モーション取得時のズレを修正する機能

新規登録モードにおいてモーションを登録する際，三回分のモーションデータの入力が必要になるが，この回数ごとにモーションの時間的なズレが生じた場合，データ登録や認証時に影響を与えてしまう可能性があるため，このズレを修正する処理を三回分のデータ取得後に行うようにしている．

三回分のデータ取得後，まずはこのデータ間の相関係数を算出し，回数ごとに全く別のモーションが入力されていないかの確認を行う．そしてある程度以上の相関が見られるが，それでも相関係数が低く出てしまった場合に，ズレ修正後の相関係数を確認しつつ，データの最大値に合わせるパターン，データの最小値に合わせるパターン，データの中央値に合わせるパターンの最大 3 つの方法で相関係数の向上を試みる．この部分をソースコード 3.2 に示す．

ソース 3.2: ズレ修正前の相関係数チェック

```
1  // measureCorrelation用の平均値データを作成
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 100; j++) {
4          averageDistance[i][j] = (distance[0][i][j] + distance[1][i][j] +
               distance[2][i][j]) / 3;
5          averageAngle[i][j] = (angle[0][i][j] + angle[1][i][j] + angle
              [2][i][j]) / 3;
6      }
7  }
8
9  //region 同一のモーションであるかの確認をし，必要に応じてズレ修正を行う
10 Enum.MEASURE measure = mCorrelation.measureCorrelation(distance, angle,
       averageDistance, averageAngle);
11
12 LogUtil.log(Log.DEBUG, "After measure correlation");
13 LogUtil.log(Log.DEBUG, "measure = " + String.valueOf(measure));
14
15
```

```
16 if (Enum.MEASURE.BAD == measure) {
17     // 相関係数が0.4以下
18     return false;
19 }
20 else if (Enum.MEASURE.INCORRECT == measure) {
21     // 相関係数が0.4よりも高く，0.6以下の場合
22     // ズレ修正を行う
23     int time = 0;
24     Enum.MODE mode = Enum.MODE.MAX;
25
26     double[][][] originalDistance = distance;
27     double[][][] originalAngle = angle;
28
29     while (true) {
30         switch (time) {
31             case 0:
32                 mode = Enum.MODE.MAX;
33                 break;
34             case 1:
35                 mode = Enum.MODE.MIN;
36                 break;
37             case 2:
38                 mode = Enum.MODE.MEDIAN;
39                 break;
40         }
41
42         distance = mCorrectDeviation.correctDeviation(originalDistance,
               mode);
43         angle = mCorrectDeviation.correctDeviation(originalAngle, mode);
44
45         for (int i = 0; i < 3; i++) {
46             for (int j = 0; j < 100; j++) {
```

```
47                  averageDistance[i][j] = (distance[0][i][j] + distance
                        [1][i][j] + distance[2][i][j]) / 3;
48                  averageAngle[i][j] = (angle[0][i][j] + angle[1][i][j] +
                        angle[2][i][j]) / 3;
49              }
50          }
51
52          Enum.MEASURE tmp = mCorrelation.measureCorrelation(distance,
                angle, averageDistance, averageAngle);
53
54          if (tmp == Enum.MEASURE.PERFECT || tmp == Enum.MEASURE.CORRECT)
                {
55              break;
56          }
57          else if (time == 2) {
58              // 相関係数が低いまま, アラートなどを出す？
59              distance = originalDistance;
60              angle = originalAngle;
61              break;
62          }
63
64          time++;
65      }
66  }
67  else if (Enum.MEASURE.PERFECT == measure || Enum.MEASURE.CORRECT ==
        measure) {
68      // PERFECTなら, 何もしない
69  }
70  else {
71      // なにかがおかしい
72      return false;
73  }
```

相関係数を算出した結果，ズレ修正が必要であると判断された場合，ソースコード 3.3 を用いて修正を行う．

ソース 3.3: ズレ修正アルゴリズム

```
1  public class CorrectDeviation {
2
3      /**
4       * 取得回数ごとのデータのズレを時間的なズレを修正する
5       *
6       * @param data 修正するdouble型の3次元配列データ
7       * @return newData ズレ修正後のdouble型の3次元配列データ
8       */
9      public double[][][] correctDeviation (double[][][] data, Enum.MODE
           mode) {
10         LogUtil.log(Log.INFO);
11
12         double[][][] newData = new double[3][3][100];
13
14         // 試行回ごとの代表値の出ている時間を抽出
15         // 変数は．桁揃え，計算後のdistance，angleを利用
16
17         // 回数・XYZを配列で
18         double value[][] = new double[3][3];
19
20         // 代表値の出ている時間，回数，XYZ
21         int count[][] = new int[3][3];
22
23         // とりあえず，変数にXYZそれぞれの一個目の値を放り込む
24         for (int i = 0; i < 3; i++) {
25             for (int j = 0; j < 3; j++) {
26                 value[i][j] = data[i][j][0];
27             }
28         }
```

```
29
30
31          // 代表値が出ている場所を取得する
32          switch (mode) {
33              case MAX:
34                  for (int i = 0; i < 3; i++) {
35                      for (int j = 0; j < 3; j++) {
36                          for (int k = 0; k < 100; k++) {
37                              if (value[i][j] < data[i][j][k]) {
38                                  value[i][j] = data[i][j][k];
39                                  count[i][j] = k;
40                              }
41                          }
42                      }
43                  }
44                  break;
45              case MIN:
46                  for (int i = 0; i < 3; i++) {
47                      for (int j = 0; j < 3; j++) {
48                          for (int k = 0; k < 100; k++) {
49                              if (value[i][j] > data[i][j][k]) {
50                                  value[i][j] = data[i][j][k];
51                                  count[i][j] = k;
52                              }
53                          }
54                      }
55                  }
56                  break;
57              case MEDIAN:
58                  // キーが自動ソートされる
                        TreeMapを用いる．データと順番を紐付けしたものを作成し，中央値
```

```
59                  for (int i = 0; i < 3; i++) {
60                      for (int j = 0; j < 3; j++) {
61                          TreeMap<Double, Integer> treeMap = new TreeMap
                                <>();
62
63                          for (int k = 0; k < 100; k++) {
64                              treeMap.put(data[i][j][k], k);
65                          }
66
67                          int loopCount = 0;
68                          for (Integer initCount : treeMap.values()) {
69                              if (loopCount == 49) {
70                                  count[i][j] = initCount;
71                              }
72
73                              loopCount++;
74                          }
75                      }
76                  }
77                  break;
78              }
79              // 1回目のデータの代表値が出た場所と，2回目・3回目のデータの代表
                    値が出た場所の差をとる
80              // とったら，その差だけデータをずらす（ずらしてはみ出たデータは
                    空いたとこに入れる）
81
82              // sample
83              // 一回目：50，二回目：30 → 右に20ずらす 一回目−二回目=+20
84              // 一回目：30，二回目：50 → 左に20ずらす 一回目−二回目=−20
85
86              // ずらす移動量を計算（XYZそれぞれ）
87              int lagData[][] = new int[2][3];
```

```
88
89
90          ArrayList<ArrayList<ArrayList<Double>>> tmpData = new ArrayList
                <>();
91
92          ArrayList<ArrayList<Double>> tmpData1 = new ArrayList<>();
93          ArrayList<Double> tmpData1X = new ArrayList<>();
94          ArrayList<Double> tmpData1Y = new ArrayList<>();
95          ArrayList<Double> tmpData1Z = new ArrayList<>();
96
97          tmpData1.add(tmpData1X);
98          tmpData1.add(tmpData1Y);
99          tmpData1.add(tmpData1Z);
100
101         ArrayList<ArrayList<Double>> tmpData2 = new ArrayList<>();
102         ArrayList<Double> tmpData2X = new ArrayList<>();
103         ArrayList<Double> tmpData2Y = new ArrayList<>();
104         ArrayList<Double> tmpData2Z = new ArrayList<>();
105
106         tmpData2.add(tmpData2X);
107         tmpData2.add(tmpData2Y);
108         tmpData2.add(tmpData2Z);
109
110         tmpData.add(tmpData1);
111         tmpData.add(tmpData2);
112
113         for (int i = 0; i < 2; i++) {
114             for (int j = 0; j < 3; j++) {
115                 for (int k = 0; k < 100; k++) {
116                     tmpData.get(i).get(j).add(data[i][j][k]);
117                 }
118             }
```

```
119         }
120
121
122         // どれだけズレているかを計算する
123         for (int i = 0; i < 3; i++) {
124             lagData[0][i] = count[0][i] - count[1][i];
125             LogUtil.log(Log.DEBUG, "lagData[0]" + "[" + i + "]" + ": " +
                    lagData[0][i]);
126
127             lagData[1][i] = count[0][i] - count[2][i];
128             LogUtil.log(Log.DEBUG, "lagData[1]" + "[" + i + "]" + ": " +
                    lagData[1][i]);
129         }
130
131
132         // 実際にリストの要素をずらしていく
133
134         for (int i = 0; i < 2; i++) {
135             for (int j = 0; j < 3; j++) {
136                 Collections.rotate(tmpData.get(i).get(j), lagData[i][j
                        ]);
137             }
138         }
139
140
141         // これでずらせたはずだ. 成果はリストに入っているはずなので, ず
                らした後のデータを専用の配列に入れよう
142         // 1回目のデータに関しては基準となるデータなのでそのまま入れる
143         for (int i = 0; i < 3; i++) {
144             for (int j = 0; j < 100; j++) {
145                 newData[0][i][j] = data[0][i][j];
146             }
```

```
147             }
148
149             // ずらしたデータを入れる
150             for (int i = 0; i < 2; i++) {
151                 for (int j = 0; j < 3; j++) {
152                     for (int k = 0; k < 100; k++) {
153                         newData[i + 1][j][k] = tmpData.get(i).get(j).get(k);
154                     }
155                 }
156             }
157
158             return newData;
159         }
160 }
```

3.3.2　フーリエ変換を用いたローパスフィルタ

モーションデータを取得している際の，細かな手の震えなどによるデータに対する影響を取り除き，モーションデータとしての純度を高めるために，フーリエ変換を用いたローパスフィルタ処理を行っている．

フーリエ変換を用いて時間軸で表されるモーションデータを周波数領域に変換することで，モーション中の細かな手の震えなどのデータが高周波成分として現れる．そしてこの高周波成分を取り除いた上で元の時間軸のデータに戻すローパスフィルタ処理を行うことで，細かな手の震えなどによるデータに対する影響を取り除いている．この処理をソースコード 3.4 に示す．

ソース 3.4: ローパスフィルタ処理

```
1 /**
2  *
      double型三次元配列の入力データに対し，フーリエ変換を用いてローパスフィルタリ
3  *
4  * @param data      データ平滑化を行うdouble型三次元配列データ
5  * @param dataName アウトプット用，データ種別
6  * @return フーリエ変換によるローパスフィルタリングにより滑らかになった
```

```
     double型三次元配列データ
 */
public double[][][] LowpassFilter (double[][][] data, String dataName) {
    LogUtil.log(Log.INFO);

    DoubleFFT_1D realfft = new DoubleFFT_1D(data[0][0].length);

    // フーリエ変換（ForwardDFT）の実行
    for (double[][] i : data) {
        for (double[] j : i) {
            realfft.realForward(j);
        }
    }


    // 実数部, 虚数部それぞれを入れる配列
    double[][][] real = new double[data.length][data[0].length][data
       [0][0].length];
    double[][][] imaginary = new double[data.length][data[0].length][
       data[0][0].length];

    int countReal = 0;
    int countImaginary = 0;

    // 実数部と虚数部に分解
    for (int i = 0; i < data.length; i++) {
        for (int j = 0; j < data[i].length; j++) {
            for (int k = 0; k < data[i][j].length; k++) {
                if (k % 2 == 0) {
                    real[i][j][countReal] = data[i][j][k];
                    countReal++;
                    if (countReal == 99) countReal = 0;
```

```
36                  }
37                  else {
38                      imaginary[i][j][countImaginary] = data[i][j][k];
39                      countImaginary++;
40                      if (countImaginary == 99) countImaginary = 0;
41                  }
42              }
43          }
44      }
45
46      mManageData.writeDoubleThreeArrayData("ResultFFT", "rFFT" + dataName
            , RegistNameInput.name, real);
47      mManageData.writeDoubleThreeArrayData("ResultFFT", "iFFT" + dataName
            , RegistNameInput.name, imaginary);
48
49      // パワースペクトルを求めるために，実数部（k），虚数部（k + 1）それ
            ぞれを二乗して加算し，平方根を取り，絶対値を求める
50      double[][][] power = new double[data.length][data[0].length][data
            [0][0].length / 2];
51
52      for (int i = 0; i < data.length; i++) {
53          for (int j = 0; j < data[i].length; j++) {
54              for (int k = 0; k < data[i][j].length / 2; k++) {
55                  power[i][j][k] = Math.sqrt(Math.pow(real[i][j][k], 2) +
                        Math.pow(imaginary[i][j][k], 2));
56              }
57          }
58      }
59
60      mManageData.writeDoubleThreeArrayData("ResultFFT", "powerFFT" +
            dataName, RegistNameInput.name, power);
61
```

```
62      // ローパスフィルタ処理
63      for (int i = 0; i < data.length; i++) {
64          for (int j = 0; j < data[i].length; j++) {
65              for (int k = 0; k < data[i][j].length; k++) {
66                  if (k > 30) data[i][j][k] = 0;
67              }
68          }
69      }
70
71      for (double[][] i : data) {
72          for (double[] j : i) {
73              realfft.realInverse(j, true);
74          }
75      }
76
77      mManageData.writeDoubleThreeArrayData("AfterFFT", dataName,
            RegistNameInput.name, data);
78
79      return data;
80  }
```

3.3.3　モーションデータの増幅機能

手首を中心とするような動きの小さいモーションにおける個人認証成功率を向上させるために，モーションデータの増幅機能を実装した．これにより，比較的動きの小さなモーションであってもデータを増幅して用いることができ，比較的動きの大きなモーションと比べて遜色なく用いることが出来るようになった．この処理をソースコード 3.5 に示す．

ソース 3.5: モーションデータ増幅機能

```
1   /**
2    * 与えられたデータを増幅させる
3    *
4    * @param data 増幅させるdouble型三次元配列データ
5    * @param ampValue どれだけデータを増幅させるか
```

```
6   * @return 増幅後のdouble型三次元配列データ
7   */
8  public double[][][] Amplify (double[][][] data, double ampValue) {
9      LogUtil.log(Log.INFO);
10
11     if (ampValue != 0.0) {
12         for (int i = 0; i < data.length; i++) {
13             for (int j = 0; j < data[i].length; j++) {
14                 for (int k = 0; k < data[i][j].length; k++) {
15                     data[i][j][k] *= ampValue;
16                 }
17             }
18         }
19     }
20     return data;
21 }
```

この増幅機能は，事前にデータの最大値と最小値の差を取り，得られた値があらかじめ設定された閾値を下回った場合にのみ機能するようにしている．この処理をソースコード 3.6 に示す．

ソース 3.6: データレンジチェック

```
1  private boolean isRangeCheck = false;
2
3
4  /**
5   * 全試行回数中，一回でもデータの幅が閾値よりも小さければtrueを返す
6   *
7   * @param data チェックするdouble型三次元配列データ
8   * @return 全試行回数中，一回でもデータの幅が閾値よりも小さければ
        true，そうでなければfalse
9   */
10 public boolean CheckValueRange (double[][][] data, double
       checkRangeValue) {
```

```
11      LogUtil.log(Log.INFO);
12
13      LogUtil.log(Log.DEBUG, "checkRangeValue" + checkRangeValue);
14
15      double[][] max = new double[data.length][data[0].length];
16      double[][] min = new double[data.length][data[0].length];
17
18      for (int i = 0; i < data.length; i++) {
19          for (int j = 0; j < data[i].length; j++) {
20              max[i][j] = 0;
21              min[i][j] = 0;
22          }
23      }
24
25      double range;
26      for (int i = 0; i < data.length; i++) {
27          for (int j = 0; j < data[i].length; j++) {
28              for (int k = 0; k < data[i][j].length; k++) {
29                  if (data[i][j][k] > max[i][j]) {
30                      max[i][j] = data[i][j][k];
31                  }
32                  else if (data[i][j][k] < min[i][j]) {
33                      min[i][j] = data[i][j][k];
34                  }
35              }
36          }
37      }
38
39      for (int i = 0; i < max.length; i++) {
40          for (int j = 0; j < max[i].length; j++) {
41              range = max[i][j] - min[i][j];
42              LogUtil.log(Log.DEBUG, "range = " + range);
```

```
43                  if (range < checkRangeValue) isRangeCheck = true;
44              }
45          }
46
47          return isRangeCheck;
48      }
```

データをどれだけ増幅させるかを決める値や，データの最大値と最小値の差がどれだけあれば増幅を行うかを決める閾値に関しては，新規登録モードにおいてメニューキーを押すことで表示される，図 3.17 のようなメニューアイテム内の増幅器設定を選択することで表示される，図 3.18 のような設定ダイアログより変更することが出来る．

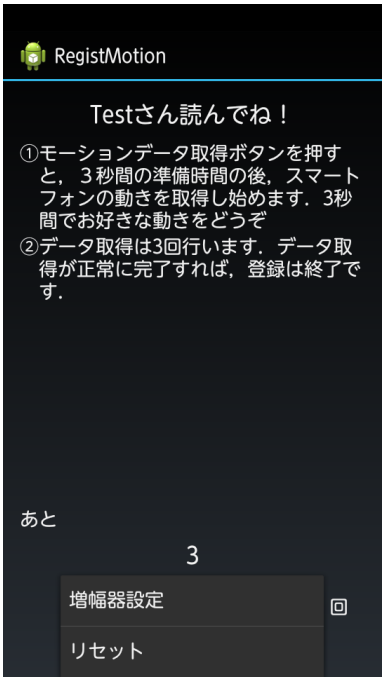

図 3.17: 新規登録モードメニュー画面

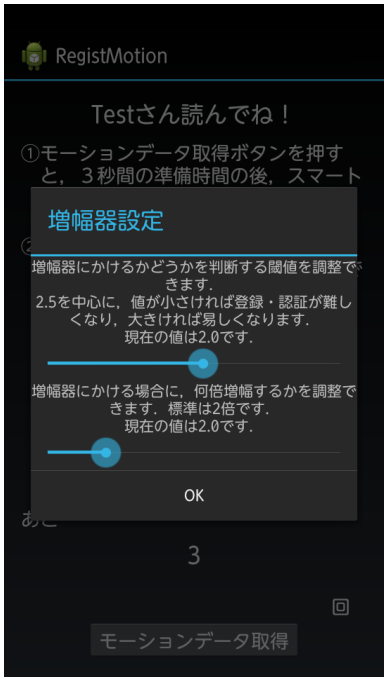

図 3.18: 増幅器設定ダイアログ

3.3.4　モーション取得時のインターバル及びヴァイブレーション機能

新規登録モード及び認証試験モードにおいて，モーション取得時の時間経過が把握しやすいように，一秒毎に端末をヴァイブレーションさせる機能を実装した．また，データ取得時のインターバルからデータ取得に移る際には通常よ長めのヴァイブレーションにすることで，データ取得開始のタイミングを明確に意識できるようにした．

第4章　実験と考察

4.1　実験方法

今回開発したシステムを用いて，以下に挙げるそれぞれのモーションを新規登録モードにてあらかじめ登録しておき，認証試験モードにて個人認証の成功率を検証する．認証はそれぞれのモーションにつき十回ずつ行い，この結果から個人認証の成功率を算出する．

4.2　実験結果

実験結果を表を用いて示す

4.3　考察

実験結果に対する考察

4.4　課題

実験結果より生じた課題

第5章　おわりに

謝辞

本研究のプログラム開発や実験，本論文の執筆にあたり，手厚い指導と様々な助言をしていただいた，関西大学総合情報学部セキュア情報システム研究室の小林孝史准教授に深く感謝いたします．また，研究テーマの選定をはじめ，日頃から有益なアドバイスを頂いた同研究室の皆様に感謝いたします．