

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 11-170 高坂 賢佑

目次

第 1 章	使用するセンサについて	6
1.1	加速度センサ	6
1.2	ジャイロセンサ	6
第 2 章	先行研究	7
2.1	坂本の研究	7
2.2	兎澤の研究	7
第 3 章	本研究のシステム	8
3.1	本研究の概要	8
3.2	システムの概要	8
3.2.1	新規登録モード	9
3.2.2	認証試験モード	12
3.2.3	データ閲覧モード	15
3.3	先行研究からの改善点	15
3.3.1	モーションデータの増幅機能	15
3.3.2	フーリエ変換を用いたローパスフィルタ	19
3.3.3	モーション取得時のズレを修正する機能	21
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	21
第 4 章	実験と考察	23
4.1	実験方法	23
4.2	実験結果	23
4.3	考察	23
4.4	課題	23
第 5 章	おわりに	24

付 録 A ソースコード	27
A.1 スレ修正の判断部分	27
A.2 スレ修正アルゴリズム	29

目 次

1.1	モーションセンサの座標系イメージ	6
3.1	システム動作フロー図	8
3.2	スタート画面	9
3.3	モード選択ダイアログ	9
3.4	ユーザ名入力画面	10
3.5	エラー通知	10
3.6	モーション登録画面	10
3.7	処理待ちダイアログ	12
3.8	登録完了ダイアログ	12
3.9	登録失敗ダイアログ	12
3.10	ユーザ名入力画面	13
3.11	認証試験画面	13
3.12	エラー通知	13
3.13	認証成功ダイアログ	15
3.14	認証失敗ダイアログ	15
3.15	ユーザ名一覧画面	16
3.16	モーションデータ一覧画面	16
3.17	新規登録モードメニュー画面	18
3.18	増幅器設定ダイアログ	18
3.19	増幅前のデータ	18
3.20	増幅後のデータ	18
3.21	ローパスフィルタ前のデータ	21
3.22	ローパスフィルタ後のデータ	21
3.23	ズレ修正前のデータ	22

3.24 ズレ修正後のデータ	22
--------------------------	----

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あるため，ユーザにとって煩雑な作業である．また，あらかじめ決められた文字種の中から一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．

第1章 使用するセンサについて

本アプリケーションには2種類のモーションセンサを利用する。

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。

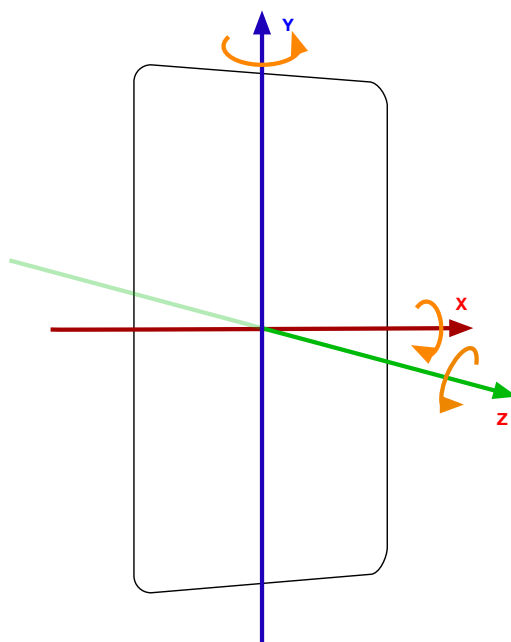


図 1.1: モーションセンサの座標系イメージ

第2章 先行研究

2.1 坂本の研究

坂本の研究[1]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[2]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3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．

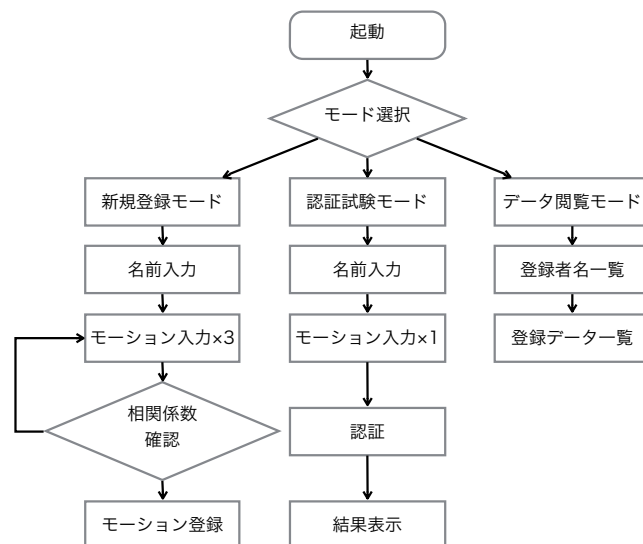


図 3.1: システム動作フロー図

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ユーザ名と共に登録する．このモードでは，ユーザに同一のモーションを 3 回入力させ，入力された 3 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。

認証試験モードでは、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1回入力させる。この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。

データ閲覧モードでは、新規登録モードにおいて登録したユーザ名およびモーションデータをリスト形式で閲覧することが出来る。

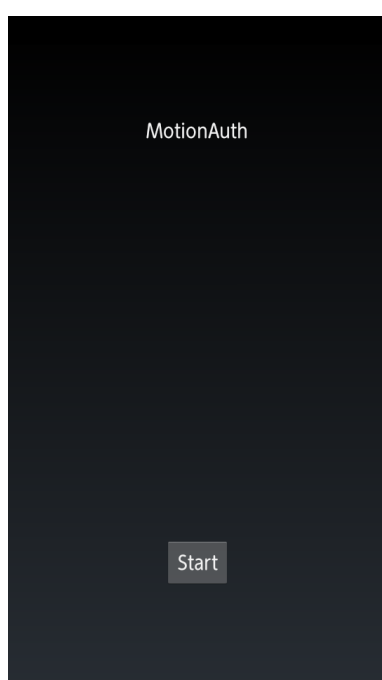


図 3.2: スタート画面

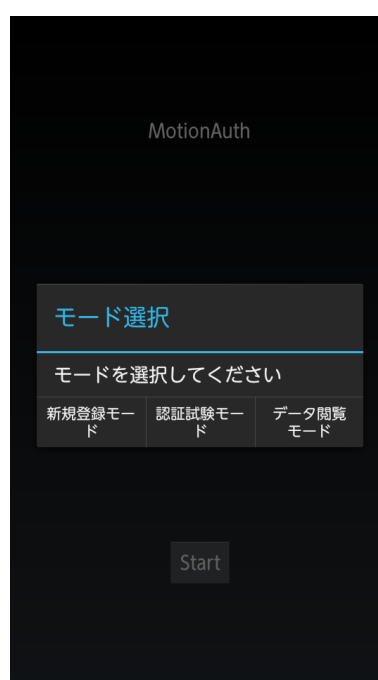


図 3.3: モード選択ダイアログ

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。この画面において 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。

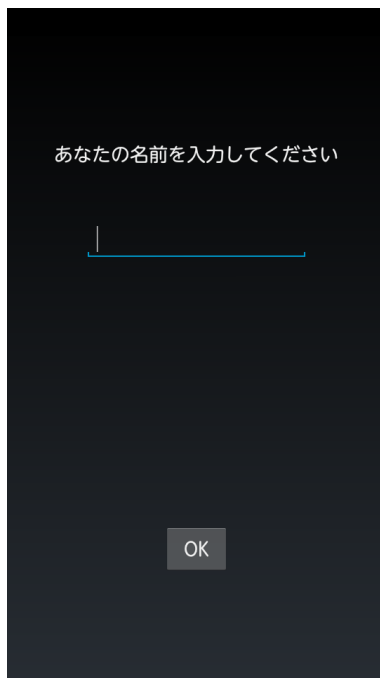


図 3.4: ユーザ名入力画面

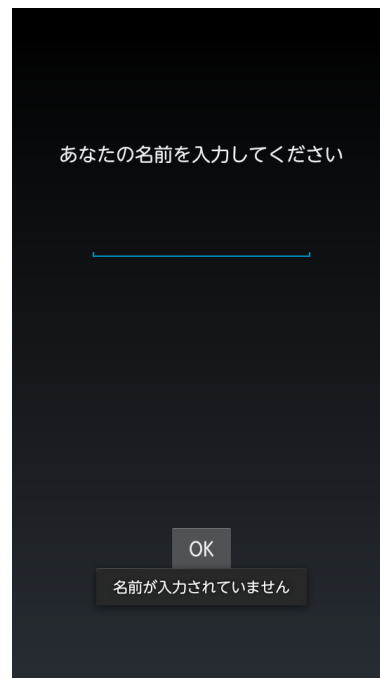


図 3.5: エラー通知

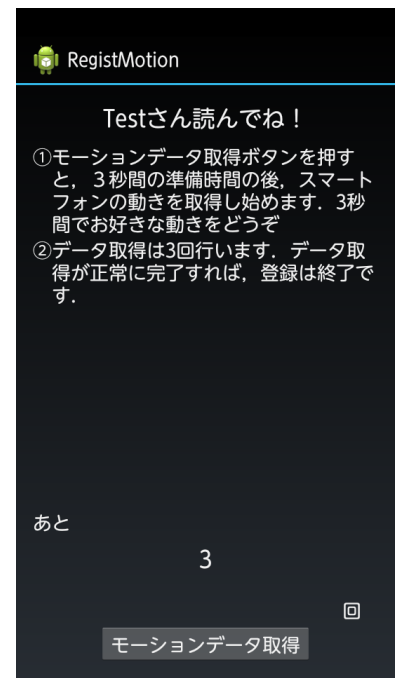


図 3.6: モーション登録画面

ソース 3.1: 入力値チェック

```

1 public class RegistNameInput extends Activity {
2     // ユーザが入力した文字列（名前）を格納する
3     public static String name;
4     ...
5     private void nameInput () {
6         final EditText nameInput = (EditText) findViewById (R.id.
7             nameInputEditText);
8
9         nameInput.addTextChangedListener(new TextWatcher() {
10             ...
11             public void afterTextChanged (Editable s) {
12                 // ユーザの入力した名前をnameに格納
13                 if (nameInput.getText() != null) name = nameInput.
14                     getText().toString().trim();
15             }
16         });
17     }
18     ...
19     // OKボタンを押した時に、次のアクティビティに移動

```

```
17         final Button ok = (Button) findViewById(R.id.okButton);
18
19         ok.setOnClickListener(new View.OnClickListener() {
20             @Override
21             public void onClick (View v) {
22                 // nameが入力されているかの確認
23                 if (name.length() == 0) {
24                     Toast.makeText(RegistNameInput.this, "名前が入力されてい
                        ません", Toast.LENGTH_LONG).show();
25                 }
26                 else {
27                     RegistNameInput.this.moveActivity("com.example.
                        motionauth", "com.example.motionauth.Registration
                        .RegistMotion", true);
28                 }
29             }
30         });
31     }
32 }
```

モーション登録画面においてモーションデータ取得ボタンを押すと、3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3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1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1回分として取得する。

モーションデータの取得が3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、データの加工が行われる。データの振れ幅を確認し、あらかじめ設定しておいた増幅器の閾値と増幅量を元に、取得したデータの振れ幅が閾値よりも小さい場合はモーションの動きが小さいと判断してデータの増幅処理を行う。次にモーション取得時の手の細かなブレなどから生じうるデータに対する影響を取り除く、フーリエ変換を用いたローパスフィルタ処理を行う。ローパスフィルタ処理が終われば、取得した回数ごとのデータが同一のものであるかの確認を行い、確認されればモーション取得時に生じうる時間的なズレを必要に応じて修正する。ズレ修正の処理が終われば3回分のデータ間の相関係数を算出し、相関が認められた場合は図3.8のような登録完了ダイアログを表示する。このダイアログの

OK ボタンを押すことで、取得した3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。相関が認められなかった場合は、図 3.9 のような登録失敗ダイアログを表示する。このダイアログの OK ボタンを押すことで、プログラム内部のモーションデータ取得カウンタが初期化され、モーションデータの取り直しをさせる。

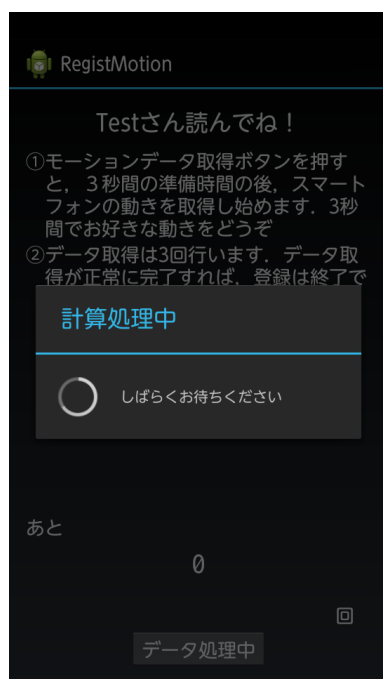


図 3.7: 処理待ちダイアログ

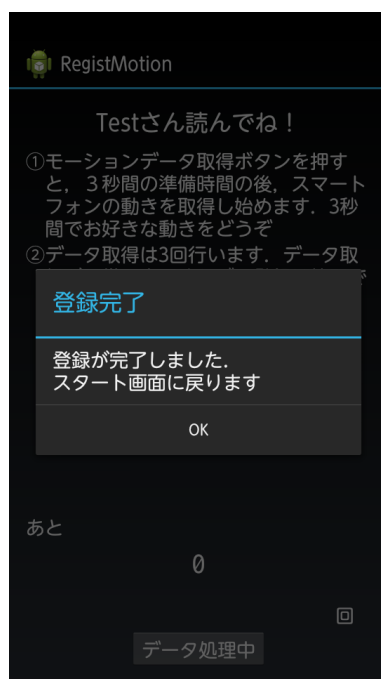


図 3.8: 登録完了ダイアログ

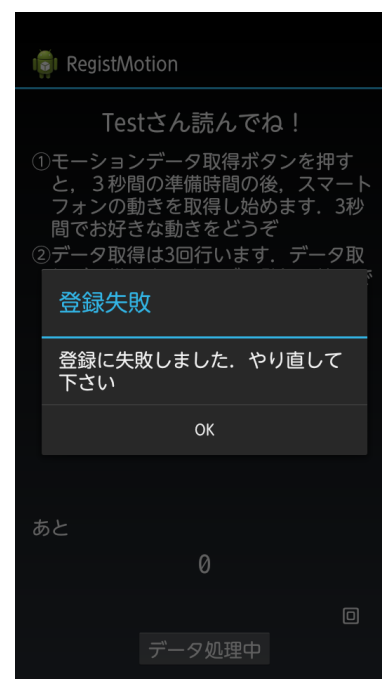


図 3.9: 登録失敗ダイアログ

3.2.2 認証試験モード

認証試験モードでは、まず図 3.10 の画面で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で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 の画面で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。指定されたユーザ名で既にモーションの登録がなされているかを確認するコードをソースコード 3.2 に示す。

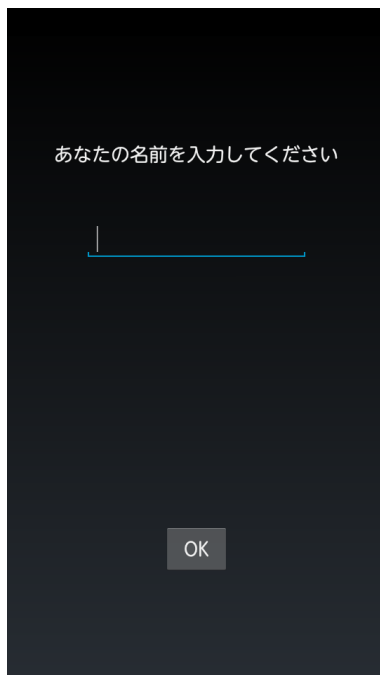


図 3.10: ユーザ名入力画面

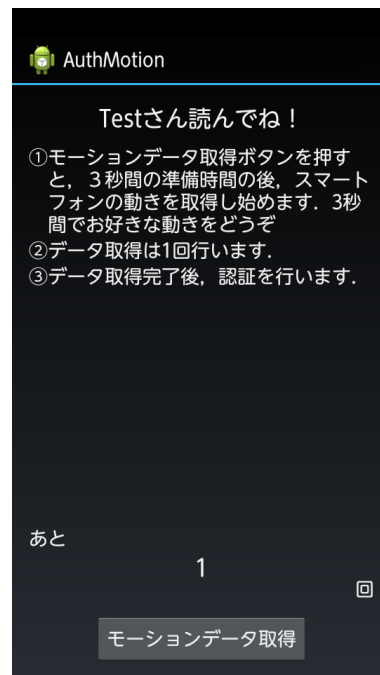


図 3.11: 認証試験画面

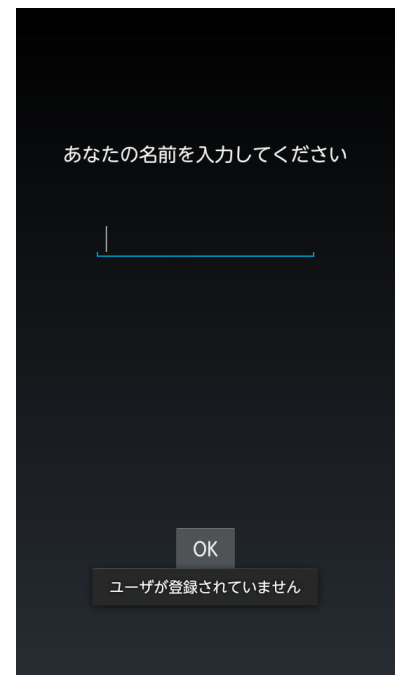


図 3.12: エラー通知

ソース 3.2: ユーザ確認

```

1 public class AuthNameInput extends Activity {
2     // ユーザが入力した文字列（名前）を格納する
3     public static String name;
4     ...
5     private void nameInput () {
6         final EditText nameInput = (EditText) findViewById(R.id.
            nameInputEditText);
7
8         nameInput.addTextChangedListener(new TextWatcher() {
9             ...
10            public void afterTextChanged (Editable s) {
11                // ユーザの入力した名前をnameに格納
12                if (nameInput.getText() != null) name = nameInput.
                    getText().toString().trim();
13            }
14        });
15        ...
16        // OKボタンを押した際に、次のアクティビティに移動

```

```
17         final Button ok = (Button) findViewById(R.id.okButton);
18
19         ok.setOnClickListener(new View.OnClickListener() {
20             @Override
21             public void onClick (View v) {
22                 // 指定したユーザが存在するかどうかを確認する .
23                 if (AuthNameInput.this.checkUserExists()) {
24                     AuthNameInput.this.moveActivity("com.example .
25                         motionauth", "com.example.motionauth .
26                         Authentication.AuthMotion", true);
27                 }
28             }
29         });
30     });
31 }
32
33 private boolean checkUserExists () {
34     Context mContext = AuthNameInput.this.getApplicationContext();
35     SharedPreferences preferences = mContext.getSharedPreferences("
36         UserList", Context.MODE_PRIVATE);
37
38     return preferences.contains(name);
39 }
```

認証試験モードでは新規登録モードと異なり、1 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、新規登録モードにおいて登録されたデータに増幅処理が施されていれば、取得したデータに対しても増幅処理を行う。そして、フーリエ変換を用いたローパスフィルタ処理を行い、新規登録モードにおいて登録されたデータとの相関係数を算出し個人認証を行う。個人認証に成功すれば、図 3.13 のような認証成功ダイアログを表示する。このダイアログの OK ボタンを押すことで、起

動画面へと移動する．個人認証に失敗すれば，図 3.14 のような認証失敗ダイアログを表示する．このダイアログの OK ボタンを押すことでプログラム内部のモーションデータ取得カウンタが初期化され，再度個人認証を行えるようにする．

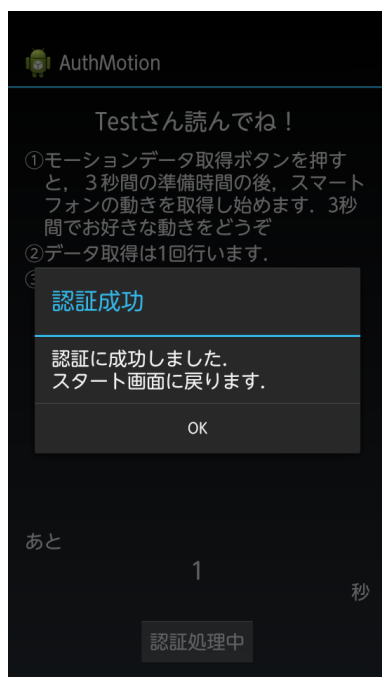


図 3.13: 認証成功ダイアログ

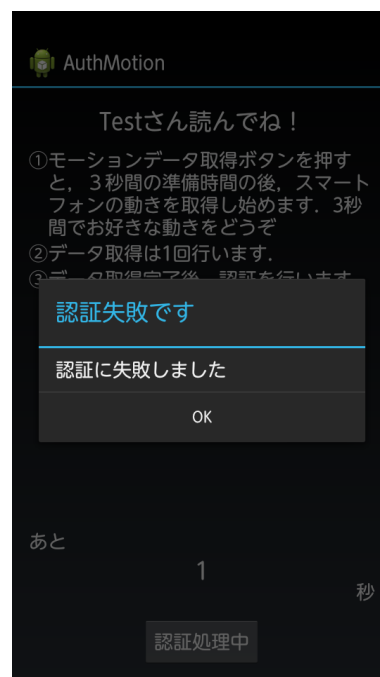


図 3.14: 認証失敗ダイアログ

3.2.3 データ閲覧モード

データ閲覧モードでは，新規登録モードにおい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．

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 3.3 に示す．

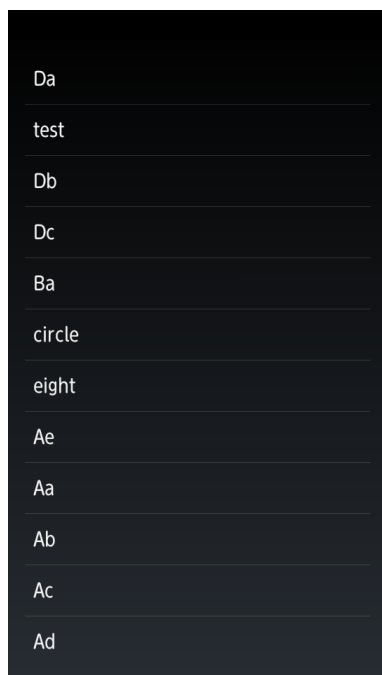


図 3.15: ユーザ名一覧画面

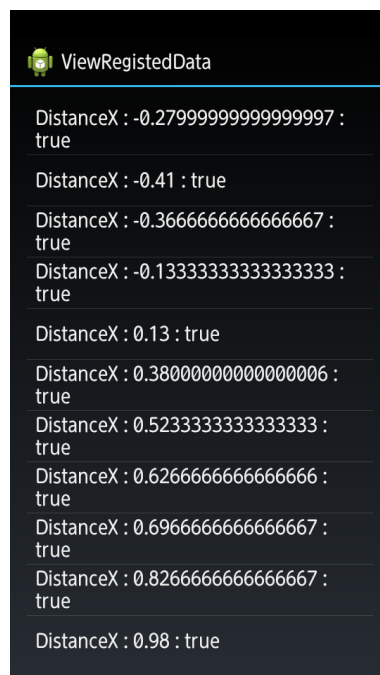


図 3.16: モーションデータ一覧画面

ソース 3.3: モーションデータ増幅機能

```

1 public double[][][] Amplify (double[][][] data, double ampValue) {
2     if (ampValue != 0.0) {
3         for (int i = 0; i < data.length; i++) {
4             for (int j = 0; j < data[i].length; j++) {
5                 for (int k = 0; k < data[i][j].length; k++) {
6                     data[i][j][k] *= ampValue;
7                 }
8             }
9         }
10    }
11    return data;
12 }

```

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 3.4 に示す。

ソース 3.4: データレンジチェック

```

1 private boolean isRangeCheck = false;
2

```

```
3 public boolean CheckValueRange (double[][][] data, double
   checkRangeValue) {
4     double[][] max = new double[data.length][data[0].length];
5     double[][] min = new double[data.length][data[0].length];
6
7     for (int i = 0; i < data.length; i++) {
8         for (int j = 0; j < data[i].length; j++) {
9             max[i][j] = 0;
10            min[i][j] = 0;
11        }
12    }
13
14    double range;
15    for (int i = 0; i < data.length; i++) {
16        for (int j = 0; j < data[i].length; j++) {
17            for (int k = 0; k < data[i][j].length; k++) {
18                if (data[i][j][k] > max[i][j]) {
19                    max[i][j] = data[i][j][k];
20                }
21                else if (data[i][j][k] < min[i][j]) {
22                    min[i][j] = data[i][j][k];
23                }
24            }
25        }
26    }
27
28    for (int i = 0; i < max.length; i++) {
29        for (int j = 0; j < max[i].length; j++) {
30            range = max[i][j] - min[i][j];
31            if (range < checkRangeValue) isRangeCheck = true;
32        }
33    }
```

```

34
35     return isRangeCheck;
36 }

```

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．

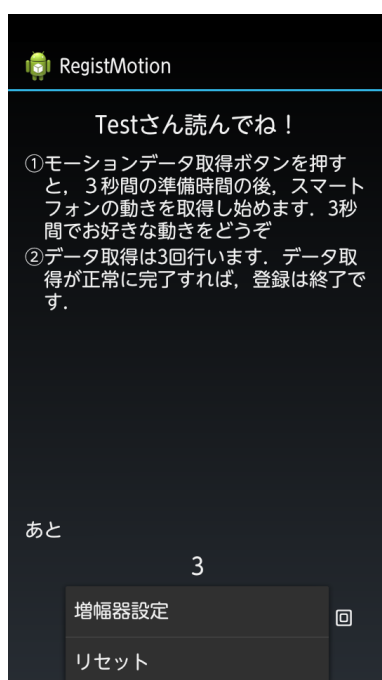


図 3.17: 新規登録モードメニュー画面

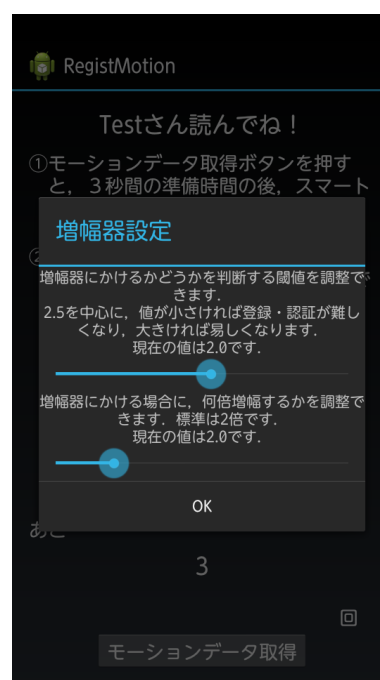


図 3.18: 増幅器設定ダイアログ

増幅器にかける前のデータをグラフ化したものを図 3.19 に，かけた後のデータをグラフ化したものを図 3.20 に示す．

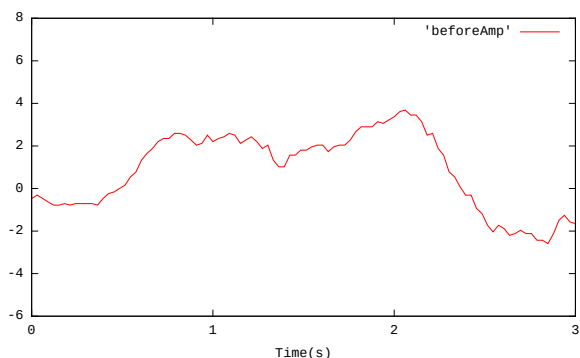


図 3.19: 増幅前のデータ

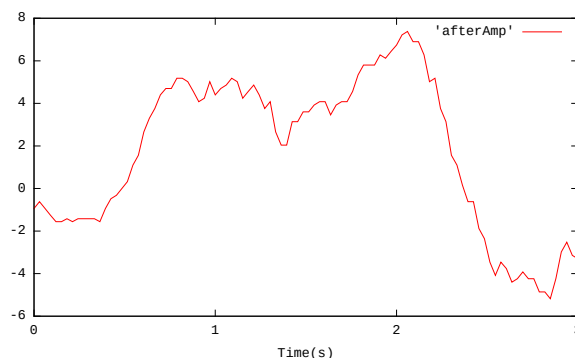


図 3.20: 増幅後のデータ

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。フーリエ変換を実装する際には、CERNのColt Project[3]で開発されたJavaによる科学技術計算用ライブラリであるColt[4]をマルチスレッド化したParallel Colt[5]に含まれている、JTransforms[6]を用いて実装している。この処理をソースコード3.5に示す。

ソース 3.5: ローパスフィルタ処理

```
1 public double[][][] LowpassFilter (double[][][] data, String dataName) {
2     DoubleFFT_1D realfft = new DoubleFFT_1D(data[0][0].length);
3
4     // フーリエ変換を実行
5     for (double[][] i : data) {
6         for (double[] j : i) {
7             realfft.realForward(j);
8         }
9     }
10
11     // 実数部, 虚数部それぞれを入れる配列
12     double[][][] real = new double[data.length][data[0].length][data
13         [0][0].length];
14     double[][][] imaginary = new double[data.length][data[0].length][
15         data[0][0].length];
16
17     int countReal = 0;
18     int countImaginary = 0;
19
20     // 実数部と虚数部に分解
21     for (int i = 0; i < data.length; i++) {
```

```
20     for (int j = 0; j < data[i].length; j++) {
21         for (int k = 0; k < data[i][j].length; k++) {
22             if (k % 2 == 0) {
23                 real[i][j][countReal] = data[i][j][k];
24                 countReal++;
25                 if (countReal == 99) countReal = 0;
26             }
27             else {
28                 imaginary[i][j][countImaginary] = data[i][j][k];
29                 countImaginary++;
30                 if (countImaginary == 99) countImaginary = 0;
31             }
32         }
33     }
34 }
35
36 // ローパスフィルタ処理
37 for (int i = 0; i < data.length; i++) {
38     for (int j = 0; j < data[i].length; j++) {
39         for (int k = 0; k < data[i][j].length; k++) {
40             if (k > 30) data[i][j][k] = 0;
41         }
42     }
43 }
44
45 // 逆フーリエ変換を実行
46 for (double[][] i : data) {
47     for (double[] j : i) {
48         realfft.realInverse(j, true);
49     }
50 }
51
```

```

52     return data;
53 }

```

ローパスフィルタにかける前のデータをグラフ化したものを図 3.21 に、かけた後のデータをグラフ化したものを図 3.22 に示す。

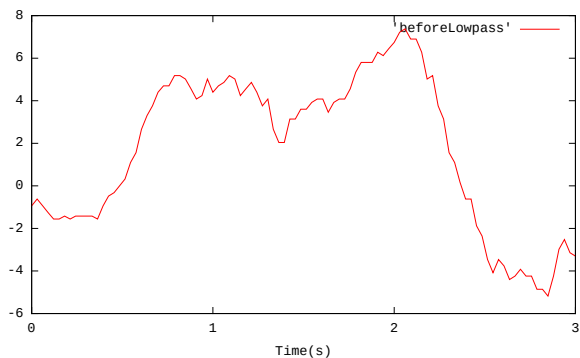


図 3.21: ローパスフィルタ前のデータ

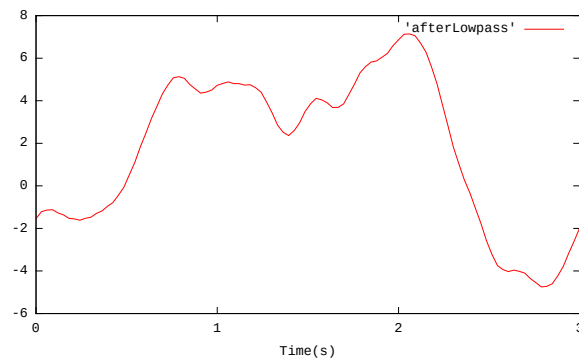


図 3.22: ローパスフィルタ後のデータ

3.3.3 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、3 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 3 回分のデータ取得後に行うようにしている。

3 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三つの方法で相関係数の向上を試みる。この部分のコードを付録のソースコード A.1 に示す。

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、付録のソースコード A.2 に示したズレ修正アルゴリズムを用いて修正を行う。

ズレ修正を行う前のデータをグラフ化したものを図 3.23 に、修正を行った後のデータをグラフ化したものを図 3.24 に示す。

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、1 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り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。

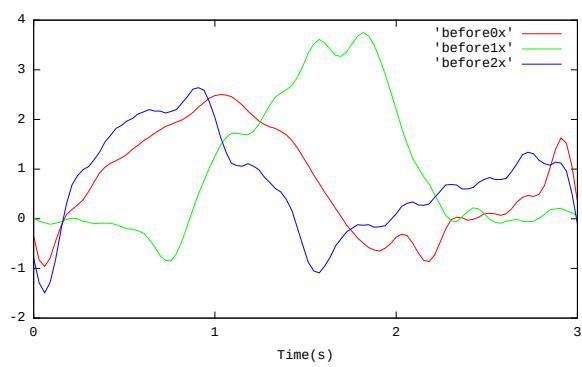


図 3.23: ズレ修正前のデータ

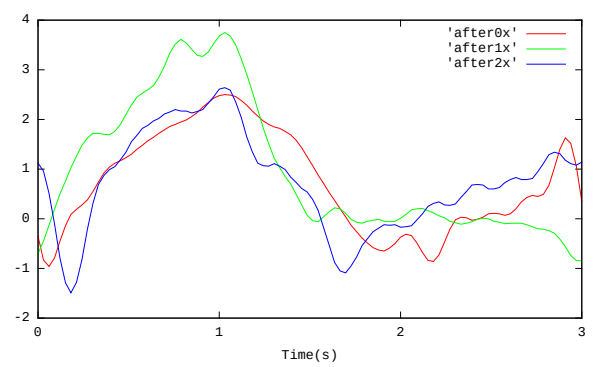


図 3.24: ズレ修正後のデータ

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10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．

参考文献，参考 URL 等

- [1] 坂本 翔，ユーザの直感的な入力をとらえるための 3 軸加速度センサによるジェスチャ認識の研究，2009 年度公立はこだて未来大学卒業論文．
- [2] 兎澤星伸，三軸加速度センサ及び三軸ジャイロセンサを用いた認証アプリケーションの開発，2012 年度卒業研究．
- [3] Colt Project，<https://dst.lbl.gov/ACSSoftware/colt/>，2014 年 12 月 27 日確認．
- [4] Colt，<https://github.com/carlsonp/Colt>，2014 年 12 月 27 日確認．
- [5] Parallel Colt，<https://sites.google.com/site/piotrwendykier/software/parallelcolt>，2014 年 12 月 27 日確認．
- [6] JTransforms，<https://sites.google.com/site/piotrwendykier/software/jtransforms>，2014 年 12 月 27 日確認．

付 録A ソースコード

A.1 ズレ修正の判断部分

```
1 Enum.MEASURE measure = mCorrelation.measureCorrelation(distance , angle ,
    averageDistance , averageAngle);
2
3 if (Enum.MEASURE.BAD == measure) {
4     // 相関係数が0.4以下
5     return false;
6 }
7 else if (Enum.MEASURE.INCORRECT == measure) {
8     // 相関係数が0.4よりも高く , 0.6以下の場合 , ズレ修正を行う
9     int time = 0;
10    Enum.MODE mode = Enum.MODE.MAX;
11
12    double[][][] originalDistance = distance;
13    double[][][] originalAngle = angle;
14
15    while (true) {
16        switch (time) {
17            case 0:
18                mode = Enum.MODE.MAX;
19                break;
20            case 1:
21                mode = Enum.MODE.MIN;
22                break;
23            case 2:
```

```
24         mode = Enum.MODE.MEDIAN;
25         break;
26     }
27
28     distance = mCorrectDeviation.correctDeviation(originalDistance ,
29         mode);
30     angle = mCorrectDeviation.correctDeviation(originalAngle , mode);
31
32     for (int i = 0; i < 3; i++) {
33         for (int j = 0; j < 100; j++) {
34             averageDistance[i][j] = (distance[0][i][j] + distance
35                 [1][i][j] + distance[2][i][j]) / 3;
36             averageAngle[i][j] = (angle[0][i][j] + angle[1][i][j] +
37                 angle[2][i][j]) / 3;
38         }
39     }
40
41     Enum.MEASURE tmp = mCorrelation.measureCorrelation(distance ,
42         angle , averageDistance , averageAngle);
43
44     if (tmp == Enum.MEASURE.PERFECT || tmp == Enum.MEASURE.CORRECT)
45     {
46         break;
47     }
48     else if (time == 2) {
49         // 相関係数が低いまま
50         distance = originalDistance;
51         angle = originalAngle;
52         break;
53     }
54
55     time++;
```

```
51     }
52 }
53 else if (Enum.MEASURE.PERFECT == measure || Enum.MEASURE.CORRECT ==
    measure) {
54     // 何もしない
55 }
56 else {
57     return false;
58 }
```

A.2 スレ修正アルゴリズム

[illegible]

```
21         value[i][j] = data[i][j][k];
22         count[i][j] = k;
23     }
24 }
25 }
26 }
27 break;
28 case MIN:
29     for (int i = 0; i < 3; i++) {
30         for (int j = 0; j < 3; j++) {
31             for (int k = 0; k < 100; k++) {
32                 if (value[i][j] > data[i][j][k]) {
33                     value[i][j] = data[i][j][k];
34                     count[i][j] = k;
35                 }
36             }
37         }
38     }
39     break;
40 case MEDIAN:
41     // キーが自動ソートされる
42     // TreeMapを用いる。データと順番を紐付けしたものを作成し、中央値の初
43
44     for (int i = 0; i < 3; i++) {
45         for (int j = 0; j < 3; j++) {
46             TreeMap<Double, Integer> treeMap = new TreeMap<>();
47
48             for (int k = 0; k < 100; k++) {
49                 treeMap.put(data[i][j][k], k);
50             }
51
52             int loopCount = 0;
```

```
51         for (Integer initCount : treeMap.values()) {
52             if (loopCount == 49) {
53                 count[i][j] = initCount;
54             }
55
56             loopCount++;
57         }
58     }
59 }
60 break;
61 }
62 // 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
63 // 出た場所の差をとる
64 // とったら、その差だけデータをずらす
65
66 // sample
67 // 一回目：50，二回目：30    右に20ずらす 一回目-二回目=+20
68 // 一回目：30，二回目：50    左に20ずらす 一回目-二回目=-20
69
70 // ずらす移動量を計算（XYZそれぞれ）
71 int lagData[][] = new int[2][3];
72
73 ArrayList<ArrayList<ArrayList<Double>>> tmpData = new ArrayList<>();
74
75 ArrayList<ArrayList<Double>> tmpData1 = new ArrayList<>();
76 ArrayList<Double> tmpData1X = new ArrayList<>();
77 ArrayList<Double> tmpData1Y = new ArrayList<>();
78 ArrayList<Double> tmpData1Z = new ArrayList<>();
79 tmpData1.add(tmpData1X);
80 tmpData1.add(tmpData1Y);
81 tmpData1.add(tmpData1Z);
```



```
82     ArrayList<ArrayList<Double>> tmpData2 = new ArrayList<>();
83     ArrayList<Double> tmpData2X = new ArrayList<>();
84     ArrayList<Double> tmpData2Y = new ArrayList<>();
85     ArrayList<Double> tmpData2Z = new ArrayList<>();
86     tmpData2.add(tmpData2X);
87     tmpData2.add(tmpData2Y);
88     tmpData2.add(tmpData2Z);
89
90     tmpData.add(tmpData1);
91     tmpData.add(tmpData2);
92
93     for (int i = 0; i < 2; i++) {
94         for (int j = 0; j < 3; j++) {
95             for (int k = 0; k < 100; k++) {
96                 tmpData.get(i).get(j).add(data[i][j][k]);
97             }
98         }
99     }
100
101     // どれだけズレているかを計算する
102     for (int i = 0; i < 3; i++) {
103         lagData[0][i] = count[0][i] - count[1][i];
104         lagData[1][i] = count[0][i] - count[2][i];
105     }
106
107     // 実際にリストの要素をずらしていく
108     for (int i = 0; i < 2; i++) {
109         for (int j = 0; j < 3; j++) {
110             Collections.rotate(tmpData.get(i).get(j), lagData[i][j]);
111         }
112     }
113
```

```
114 // ずらした後のデータを配列に入れる
115 // 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
116 for (int i = 0; i < 3; i++) {
117     for (int j = 0; j < 100; j++) {
118         newData[0][i][j] = data[0][i][j];
119     }
120 }
121
122 // ずらしたデータを入れる
123 for (int i = 0; i < 2; i++) {
124     for (int j = 0; j < 3; j++) {
125         for (int k = 0; k < 100; k++) {
126             newData[i + 1][j][k] = tmpData.get(i).get(j).get(k);
127         }
128     }
129 }
130
131 return newData;
132 }
```