### Mapping Scheme

#### Design

Mapping Scheme klassens har til funktion af løse to opgaver

1. Lagre brugerindstillinger for hvordan data fra en given sensor omdannes til et MidiSignal
2. At syntetisere et MidiSignal ud fra en given sensor måling.

##### Klassediagram

Herunder ses et klassediagram for MappingScheme og dets lagringsstructs[[1]](#footnote-1) samt for MidiSignal, som MappingScheme også benytter i sin map() –funktion. Bemærk at der for klassediagrammet herunder er udeladt set- og get-metoder.



##### Funktionsbeskrivelser

|  |  |
| --- | --- |
| **Funktion** | map(int data, MidiSignal & signal): void |
| **Parametre** | int data: sensordata i range 0-127 MidSignal & signal: en reference til et givent MidiSignal, der skal ændres |
| **Returværdi** | Ingen |
| **Beskrivelse** | Funktionen tilretter MidiSignalet, der refereres til fra parametren signal på baggrund af parametren ”data” samt de lagrede brugerindstillinger. |

|  |  |
| --- | --- |
| **Funktioner** | mapKey(int data, MidiSignal & signal): void mapVelocity(int data, MidiSignal & signal): void mapCCAbs(int data, MidiSignal & signal): void mapCCRel(int data, MidiSignal & signal): void |
| **Parametre** | int data: sensordata i range 0-127 MidSignal & signal: en reference til et givent MidiSignal, der skal ændres |
| **Returværdi** | Ingen |
| **Beskrivelse** | Funktionen tilretter MidiSignalet, der refereres til fra parametren signal på baggrund af parametren ”data” samt de lagrede brugerindstillinger. Disse funktioner kaldes fra funktionen map() afhængig af hvilken param\_, der er indstillet i MappingSchemet. |

|  |  |
| --- | --- |
| **Funktion** | quantizeDiatonic(int &): void |
| **Parametre** | int & dataIn |
| **Returværdi** | Ingen |
| **Beskrivelse** | Hjælpe funktion, der kvantiserer parametren dataIn, så den til slut genererede tone passer i den af brugeren valgte key\_.scale\_. |

|  |  |
| --- | --- |
| **Funktion** | noteStringToInt(string): int |
| **Parametre** | string note |
| **Returværdi** | int |
| **Beskrivelse** | Hjælpefunktion, der konverterer den af brugeren valgte grundtone key\_.root\_ til en tilsvarende int. |

#### Implementering

Herunder foreligger beskrivelser af implementeringsfremgangen for klassens funktioner.

Se bilag MappingScheme.h og MappingScheme.c for den endelige MappingScheme implementering.

##### map

map() har til ansvar at kalde den en underfunktion jf. den param\_, brugeren har indstillet.

##### mapKey

Programflowet i mapKey er som vist i nedenstående flowchart



Tilrettelse af data: Systemet kan generere toner fra oktaverne -2 til 9 = 12 oktaver. I hver oktav er der 12 toner. Dette giver mulighed for at generer i alt 120 forskellige toner. Derfor benyttes kun sensorData i intervallet 3-122.

quantizeDiatonic: beskrives i flg. afsnit.

Dataforskydning muliggør korrekt kvantisering ved forskellige grundtoner.

Sæt Midisignal: For at muliggøre brug af flere note inputs på samme polyfoniske MIDI instrument, skal den forrige tone for sensoren slukkes inden en ny igangsættes. Der tjekkes derfor først om tonen er forskellig fra den gamle inden den sættes. Er den ny, sættes kommandoen til ”note off”. Dermed kræves to gennemløb med samme modtagne data før tonen ændres. Desuden er der implementeret et static array af bools, noteOn, der kan sættes af mapVelocity, og som benyttes til at indikere hvorvidt tonen er slukket af mapVelocity. Er dette tilfældet, slukkes tonen også i mapKey.

##### quantizeDiatonic

Dur- eller mol-skalaer hører begge under kategoriseringen diatonisk.

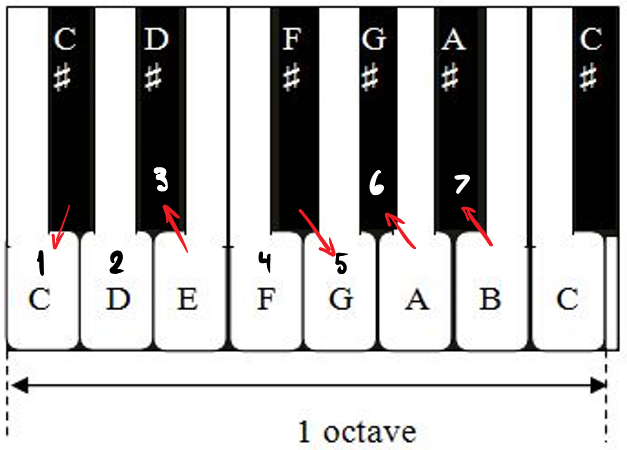
Når det indkomne data (som indeholder alle oktavens toner) skal kvantiseres, skiftes de toner, der ikke ligger i skalaen op- eller ned til en tone i skalaen. Kvantiseringen er designet således, at de mest brugte toner favoriseres: Grundtone (1), terts (3), kvint (5).

Dur kvantiseringen foregår som på illustrationen af en oktav herunder:



Påskrevet er durskalaens trin (1-7) for en c-dur skala samt hvilken vej det ønskes at tonerne uden for skala skal kvantiseres til (røde pile).

Mol kvantiseringen er illustreret herunder:



Funktionen er opbygget af først en registrering af hvilken tone i oktaven, data er, og dernæst en kvantisering vha. switch case sætninger.

##### mapVelocity

Programflowet i mapVelocity er som følger:

1. Tjek om data er valid (0-127).
2. Hvis signal før var NOTEOFF og data overstiger den af brugeren satte lowerThreshold, sættes den kommandoen til NOTEON og noteOn-flaget sættes til 1 for den indstillede kanal.
3. Hvis signal før var NOTEON og data overstiger lowerThreshold, sættes kommandoen til AFTERTOUCH.
4. Hvis signal før var NOTEON og data er lavere end lowerThreshold, sættes kommandoen til NOTEOFF og noteOn flaget sættes til 0 for den indstillede kanal.

##### mapCCAbs

Programflowet i mapCCAbs er som følger:

1. Validér data (0-127).
2. Skalér data jf. af brugeren indstillede minVal\_ og maxVal\_
3. Sæt signal

##### mapCCRel

Herunder ses et flowchart for mapCCRel



#### Test

Test af MappingScheme er for de fleste funktioner testede for et begrænset, men nøje udvalgt sæt input data. De er som udført af designer med udgangspunkt i en white box approach, hvor alle signalveje testes, om end ikke alle mulige udfald afprøves. Som eksempel ville dette for funktionen mapKey være *12 gruntoner x 3 skalaer x 2 retninger x 127 mulige datainput = 9144 muligheder* for testens udfald. Netop denne funktion er således testet for et begrænset antal inputs for en specifik grundtone-skala kombination. Der henvises til endelig integrationstest for auditiv validering af signalvej for forskellige grundtoner og skalaer.

##### MidiSignal:

Til testen oprettes en vector af MidiSignaler:

int vectorsize = 16;

vector<MidiSignal> MidiSigs(vectorsize);

for (int i = 0; i < vectorsize; ++i)

{

MidiSigs[i].channel\_=0;

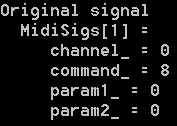
MidiSigs[i].command\_=NOTEOFF;

MidiSigs[i].param1\_=0;

MidiSigs[i].param2\_=0;

}

Ved print fås som forventet vores default MidiSignal:



##### Test af mapVelocity

Der oprettes et MappingScheme til test af velocity. Kun det pågældende MappingSchemes velocity parameter (her = 20) bestemmer velocity funktionen. Resten sættes til nul:

MappingScheme \*velocityPtr = new MappingScheme("one", //id

1, //kanal

"velocity", //param

"0", "0", "0", //key params

20, //velocity params

0, //Common CC param

0, 0, //CCAbs params

0); //CCRel params

MappingSchemets map-funktion testes med

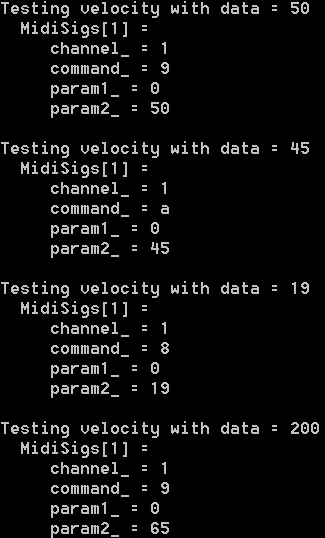
* Valid data med værdi over det satte ”threshold” (den satte velocity param): test af NOTEON
* Valid data med værdi over det satte ”threshold” (den satte velocity param): test af AFTERTOUCH
* Valid data med værdi under det satte ”threshold” (den satte velocity param): test af NOTEOFF
* Invalid data: test af validering

**Testkode, execution og observationer:**

velocityPtr->map(50, MidiSigs[1]);

cout << "\nTesting velocity with data = 50\n MidiSigs[1] = " << endl;

MidiSigs[1].print();

Vi observerer:   
command\_ = NOTEON (9), param2\_ = det satte datapunkt.

velocityPtr->map(45, MidiSigs[1]);

cout << "\nTesting velocity with data = 45\n MidiSigs[1] = " << endl;

MidiSigs[1].print();

Vi observerer:  
command\_ = AFTERTOUCH (a), param2\_ = det satte datapunkt.

velocityPtr->map(19, MidiSigs[1]);

cout << "\nTesting velocity with data = 19\n MidiSigs[1] = " << endl;

MidiSigs[1].print();

Vi observerer:  
command\_ = NOTEOFF (8), param2\_ = det satte datapunkt.

velocityPtr->map(200, MidiSigs[1]);

cout << "\nTesting velocity with data = 200\n MidiSigs[1] = " << endl;

MidiSigs[1].print();  
  
Vi observerer:  
command\_ = NOTEON (8), param2\_ = default data (65)

##### Test af mapKey

Der oprettes et MappingScheme til test af key. Kun det pågældende MappingSchemes key parameter (her grundtone = cis, skala = major, retning = falling) bestemmer key funktionen. Resten sættes til nul:

MappingScheme \*keyPtr = new MappingScheme( "two", //id

2, //kanal

"key", //param

"cis", "major", "falling", //key params

0, //velocity params

0, 0, 0, //Common CC param

0); //CCRel param

Denne konfiguration af key tester både ”vendingen” af data-retning, quantizeDiatonic med dur-skala og shift af toner jf. cis som grundtone.

**Forventet resultat**

Ved data = 120, shiftes tonen ned til 117.

Når tonen vendes bliver data = 119-117 = 2

data = 2 svarer til tonen D, der for skalaen cis du forventes blive rykket ned til grundtonen Cis, der her svarer til data = 1.

Data = 120 sendes to gange. Første gang forventes værdien sat, men command = NOTEOFF (8). Anden gang forventes command = NOTEON (9).

**Testkode**

cout << "\nTesting key with data = 120\n MidiSigs[1] = " << endl;

keyPtr->map(120, MidiSigs[1]);

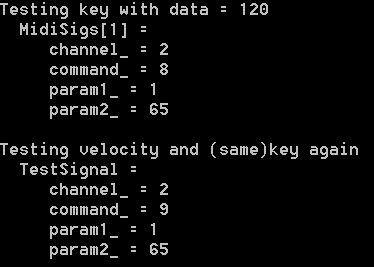
MidiSigs[1].print();

cout << "\nTesting velocity and (same)key again\n TestSignal = " << endl;

keyPtr->map(120, MidiSigs[1]);

MidiSigs[1].print();

**Testobservationer**

*Første map:*

Vi ser at param1\_ som forventet sættes til 1 (svarer til tonen Cis)  
Vi ser at command\_ = 8 (NOTEOFF) som forventet.

*Anden map:*

Vi ser at param1\_ som forventet stadig er 1  
Vi ser at command\_ = 9 (NOTEON) som forventet.

##### Test af mapCCAbs

Der oprettes et MappingScheme til test af absolut control change. Kun det pågældende MappingSchemes key parameter (her cc nummer = 10, min værdi = 40, max værdi = 120) bestemmer CCAbs funktionen. Resten sættes til nul:

MappingScheme \*CCAbsPtr = new MappingScheme("three", //id

3, //kanal

"CCAbs", //param

"0", "0", "0", //key params

0, //velocity params

10, 40, 120, //Common CC param: ccNum, minVal, maxVal

0); //CCRel params

**Testkode:**

CCAbsPtr->map(64, MidiSigs[1]);

cout << "\nTesting CCAbs with data = 64\n TestSignal = " << endl;

MidiSigs[1].print();

**Forventet resultat**:

Data = 64 er midtpunkt, hvis der havde været fuld range.

En range fra 40-120 har et midtpunkt på 80.

Der forventes derfor at param2 = 80.

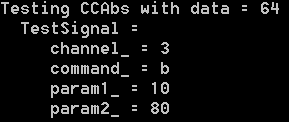
MappingSchemet er sat til CC nummer 10. Det forventes derfor at param1 = 10

Channel\_ forventes at være sat til 3, som sat i MappingSchemet.

Command\_ forventes at være CONTROLCHANGE

**Testobservationer**:

MidiSignalet sættes som forventet:



##### Test af mapCCRel

Der oprettes et MappingScheme til test af absolut control change. Kun det pågældende MappingSchemes key parameter (her cc nummer = 10, følsomhed = MEDIUM) bestemmer CCRel funktionen. Resten sættes til nul:

MappingScheme \*CCRelPtr = new MappingScheme("four", //id

4, //kanal

"CCRel", //param

"0", "0", "0", //key params

0, //velocity params

10, 40, 83, //Common CC param: ccNum, minVal, maxVal

MEDIUM); //CCRel params

**Testkode:**

Det satte MidiSignal fra test af mapCCAbs benyttes igen.

Der sendes to gange data der giver inkrementering af CC parameter-værdien og en gang, der dekrementerer.

CCRelPtr->map(127, MidiSigs[1]);

cout << "\nTesting CCRel with data = 127\n TestSignal = " << endl;

MidiSigs[1].print();

CCRelPtr->map(127, MidiSigs[1]);

cout << "\nTesting CCRelwith data = 127\n TestSignal = " << endl;

MidiSigs[1].print();

CCRelPtr->map(127, MidiSigs[1]);

cout << "\nTesting CCRelwith data = 13\n TestSignal = " << endl;

MidiSigs[1].print();

**Forventet resultat**:

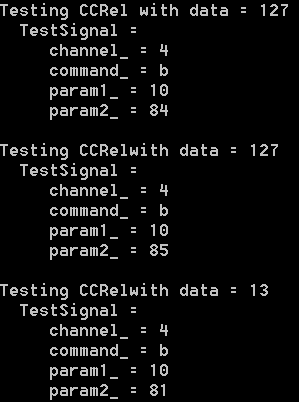
Data = 127 svarer til en inkrementering på 2.

Følsomheden er sat MEDIUM, hvilket svarer til en fordobling af den inkrementerede værdi.

Første kald sætter værdien til 84 (inkrementering på 4).

Ved andet kald sættes værdien til 85, da maxVal\_ nås.

Tredje kald dekrementerer signalet med 4, og vi forventer at få 81.

**Testobservationer**:

Resultatet ses til højre.

MidiSignalet sættes som forventet:

1. Som specificeret i afsnittet ”Krav til Mapping Scheme” [↑](#footnote-ref-1)