Ausführliches Beispiel
für die Klassen:
geometryInfo.cpp
interpolation.cpp

Dieses Dokument soll die Benutzung der Klassen geometryInfo und interpolation anhand eines ausführlichen Beispiels verdeutlichen. Ziel beider Klassen ist die Aufbereitung von MEG oder EEG Mess-Daten für eine graphische Echtzeit Ausgabe.
Es ist zu beachten, dass sich beide Klassen weder um Messung, noch um das tatsächliche Anzeigen der Daten kümmern.

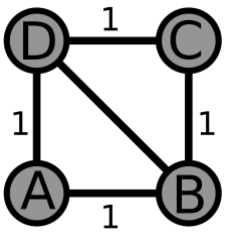
Voraussetzungen/Eingabe:

1. Mesh des Kopfes in folgender Form:
bemSurface:

```
QFile t_bemSurfaceVV("beispiel/example.fif");
MNEBem bemSurface(t_bemSurfaceVV);
```

Positionen der Knoten:

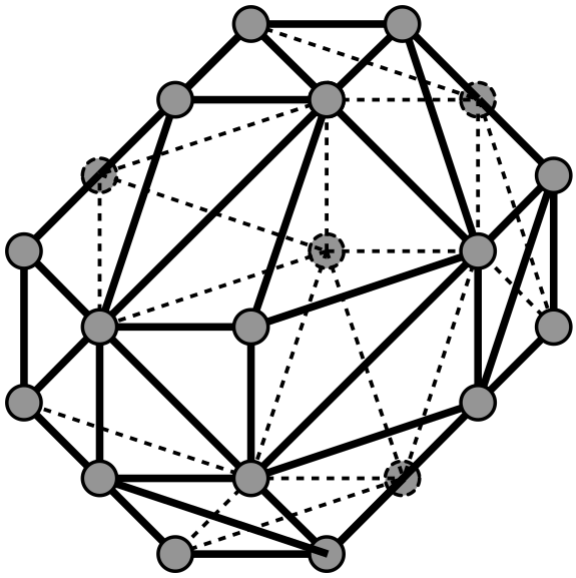
	x	y	z
A	0	0	0
B	1	0	0
C	1	1	0
D	0	1	0



Beziehungsweise auch ein größeres 3D Beispiel (laden erfolgt natürlich analog zum bemSurface):
bigSurface:

Da der Körper aus 24Knoten besteht werden wir an dieser Stelle auf die Auflistung sämtlicher Koordinaten verzichten.

Sämtliche Distanzen zwischen zwei Punkten in den Achsenebenen betragen 1



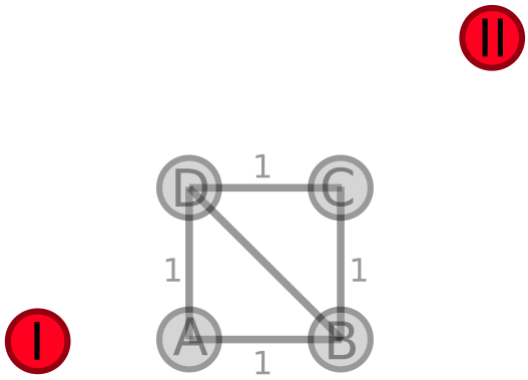
2. Positionen der Sensoren:

vecSensorPos:

```
// positions of EEG and MEG sensors
QVector<Vector3f> eegSensors;
QVector<Vector3f> megSensors;
//fill both QVectors with the right sensor positions
for( const FiffChInfo &info : evoked.info.chs){
        //EEG
        if(info.kind == FIFFV_EEG_CH && info.unit == FIFF_UNIT_V){
                eegSensors.push_back(info.chpos.r0);
        }
        //MEG
        if(info.kind == FIFFV_MEG_CH && info.unit == FIFF_UNIT_T){
                megSensors.push_back(info.chpos.r0);
        }
}
```

Positionen der Knoten:

	x	y	z
I	-1	0	0
II	2	2	0



3. Sensormessdaten in der Form:

vecMeasurmentData:

I	100
II	50

Ausgabe:

Farbwerte für jeden Knoten des Kopf-Mesh

Grundlegende Idee:
Aufteilung der Probleme in 2 Gruppen. Einen ersten Teil der bereits im Vorfeld offline berechnet werden kann und sich für unterschiedliche Mess-Daten nicht ändert (Für andere Kopf-Meshes oder neue Sensorpositionen schon) und einen zweiten Teil welcher die Messdaten live in Farbwerte für die Mesh-Knoten umwandelt.
Der erste Teil wirkt sich dabei auf die Ladezeit des Programms aus, der zweite Teil auf die Bildrate, mit welcher die Daten angezeigt werden können.

Der erste Teil ist in 3 Zwischenschritte unterteilt und bildet die Initialisierungsphase, da hier bereits Kopf-Mesh, Sensorpositionen (und Messart (EEG, MEG) ?) festgelegt werden.

1. Zuerst müssen den Sensoren genaue Orte(Knoten) auf der Kopfoberfläche zugewiesen werden (Annahme: Wert wurde an dieser Stelle ermittelt) → projectSensors

2. Danach müssen die Distanzen zwischen den Sensoren(Sensorknoten) und allen Knoten ermittelt werden (später benötigt um die Aktivitäten aller Knoten zu „erraten“) → scdc
    2.1 Im Anschluss müssen noch die Sensoren betrachtet werden, welche keine, bzw. falsche Daten liefern (bad channels) (Annahme: diese Sensoren liegen in unendlicher Ferne, und haben daher keinen Einfluss) → filterBadChannels

3. Als letztes muss eine Gewichtsmatrix erstellt werden, über welche dann immer wieder aus den Sensorwerten alle anderen Werte abgeleitet werden können → createInterpolationMat

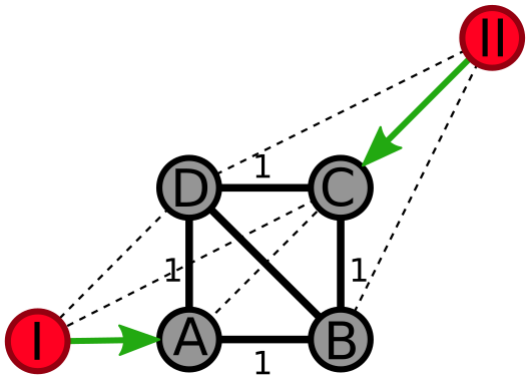
Im zweiten Teil sollen dann möglichst schnell aus den Sensorwerten die Werte für alle Knoten „erraten“ werden (dazu wird zwischen den gemessenen Werten interpoliert)

Ausführliche Verwendung und Erklärung:

projectSensors:
Eingabe: Surface, SensorPos
Ausgabe: MappedSubset

Die Funktion projectSensors, weißt jedem Sensor genau einen Knoten aus unserem Mesh(Surface) zu. Dabei muss einem Sensor der Punkt auf dem Mesh zugewiesen werden, welcher möglichst exakt der Position entspricht an welcher der Sensor die Daten misst.
Da bereits im Vorfeld, per Hand, Mesh und Sensoren möglichst passgenau übereinandergelegt werden, eignet sich eine einfache lineare Suche nach dem Knoten im Mesh, welcher den geringsten euklidischen Abstand vom Sensor aufweist.
Beispiel:

Wie leicht zu sehen ist, stimmen die Positionen der Sensoren noch nicht genau mit den Mesh-Knoten überein. In der späteren Darstellung wird aber nur das eingefärbte Mesh angezeigt. Daher müssen die Senoren jetzt auf unser Mesh „projeziert“ werden.



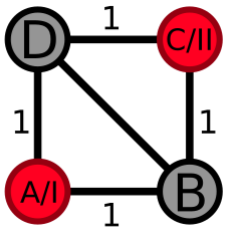
Schauen wir uns die Abstände zwischen Senoren und Mesh-Knoten genauer an:

	A	B	C	D
I	1	2	$\sqrt{5}$	$\sqrt{2}$
II	$\sqrt{8}$	$\sqrt{5}$	$\sqrt{2}$	$\sqrt{5}$

Wir nehmen also an, dass Sensor I seinen Wert im Knoten A gemessen hat und Sensor II im Knoten C.
Das Ergebnis von projectSensors würde also wie folgt aussehen:

```
MNELIB::MNEBemSurface Surface = bemSurface;
QVector<Vector3f> SensorPos = vecSensorPos;
QSharedPointer<QVector<qint32>> MappedSubset = GeometryInfo::projectSensors(Surface, SensorPos);
```

I	A
II	C

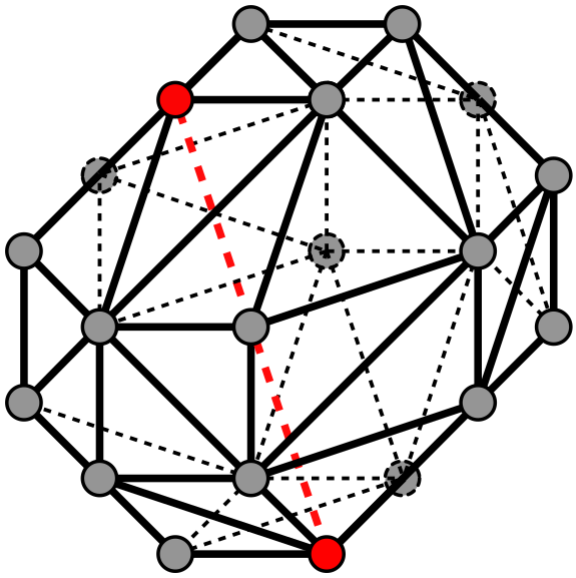


Es werden nur die zu den Sensoren gehörige Knotennummern in der Reihenfolge der Sensoren gespeichert.

scdc:
Eingabe: Surface, CancelDistance, MappedSubset
Ausgabe: scdcMap

Die surface constraint distance calculation (scdc) berechnet die Oberflächenentfernung aller Punkte zueinander in einem Mesh(Surface).
Um die Entfernung zwischen zwei Punkten zu berechnen gibt es verschiedene Möglichkeiten. Die einfachste, ist aber auch gleichzeitig die Ungenaueste. Denn statt die Entfernung auf der Oberfläche zu berechnen wird einfach die euklidische Distanz berechnet, siehe Abb.:

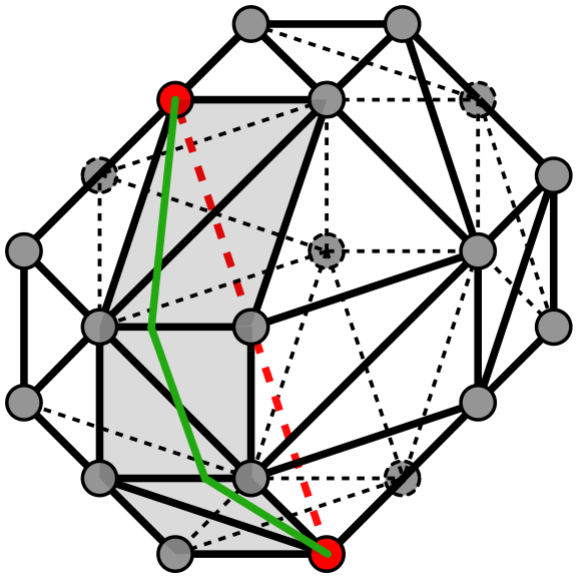
Bei unserer Festlegung, dass alle x-, y- oder z-Achsen parallele Entfernungen zwischen den Knoten die Länge 1 haben, hätten wir eine Euklidische Distanz von $\sqrt{10}$ LE also etwa 3,162 LE



Die eigentliche, kürzeste Entfernung auf der Oberfläche würde über die in der nächsten Abbildung grau markierten Flächen und entlang der grün markierten Linie verlaufen:

Die exkate Entfernung beider Punkte auf der Oberfläche wäre also:
$\sqrt{(2*\sqrt{2}+1)^2+1^2}$ LE also in etwa 3,957 LE

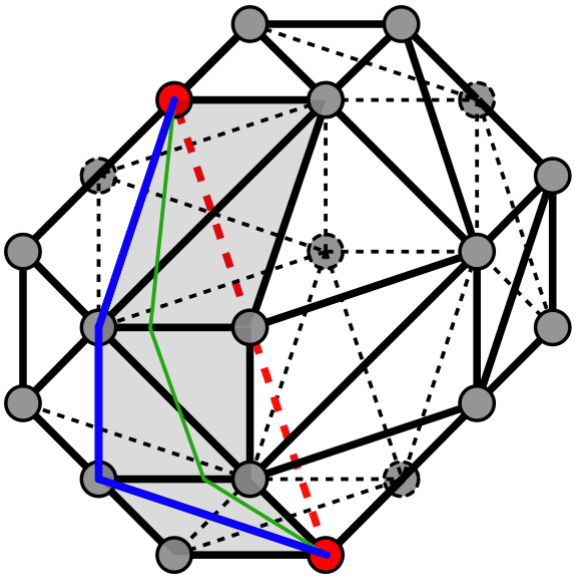
Da dies jedoch sehr aufwändig zu berechnen ist, da die Entfernung mehrere Dreiecksflächen in den unterschiedlichsten Winkeln schneidet und eine Menge Prozessorzeit in Anspruch nehmen würde, haben wir uns nur für eine näherungsweise Berechnung entschieden.



Wir werden zwar auch auf der Oberfläche rechnen, jedoch entlang der Dreiecksverbindungen zwischen den Knoten, daher entlang der Kanten des Mesh.
Denn dafür eignet sich auch ein Dijkstra Algorithmus:

Die blaue Kante in der Abbildung entspricht der berechneten Entfernung:

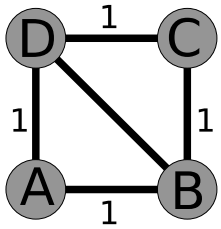
$\sqrt{(\sqrt{2})^2+1^2}+1+\sqrt{2}$ was in etwa 4,146 entspricht. Wir treffen damit zwar nicht die exakte Entfernung, können aber mit einem Annehmbaren Zeitaufwand ein deutlich genaueres Ergebnis liefern, ohne die Ladezeiten unseres Programms zu lang werden zu lassen.



Bei dem folgenden einfachen Mesh würde als Resultat der scdc alsofolgende Tabelle herauskommen:

```
MNELIB::MNEBemSurface Surface = bemSurface;
QSharedPointer<QVector<qint32>> MappedSubset = nullptr;
QsharedPointer<MatrixXd> scdcMap = GeometryInfo::scdc(Surface, MappedSubset, CancelDistance);
```

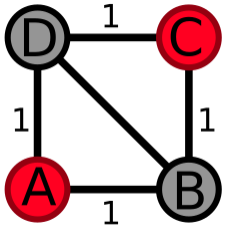
	A	B	C	D
A	0	1	2	1
B	1	0	1	$\sqrt{2}$
C	2	1	0	1
D	1	$\sqrt{2}$	1	0



Es würde daher Speicherplatz für n*n Entfernungen benötigt, wenn n der Anzahl der Knoten im Mesh entspricht. Da dies jedoch zu viel ist und auch in der Berechnung zu lange dauern würde (für n = 200.000 → 320GB), muss die Anzahl der zu berechnenden Entfernungen drastisch verkleinert werden. Dabei hilft uns das Ergebnis der vorherigen Berechnung (MappedSubset). Wir werden für die Interpolation nur noch die Distanzen zwischen einem Sensor und allen anderen Knoten benötigen, weswegen wir auch nur diese Berechnen werden:

```
MNELIB::MNEBemSurface Surface = bemSurface;
QSharedPointer<QVector<qint32>> MappedSubset = GeometryInfo::projectSensors(Surface, SensorPos);
QsharedPointer<MatrixXd> scdcMap = GeometryInfo::scdc(Surface, MappedSubset, CancelDistance);
```

	A	C
A	0	2
B	1	1
C	2	0
D	1	1



Wir reduzieren damit den benötigten Speicherplatz auf n*m, wobei n nach wie vor die Anzahl der Knoten im Mesh und m die Anzahl der Sensoren ist. Damit reduzieren wir den benötigten Speicherplatz mindestens um den Faktor 1000 und können ihn gut handhaben.
Da aber auch die Rechenzeit kostbar ist und die Ladezeit unseres Programms so gering wie möglich gehalten werden soll, können wir noch eine weitere Einschränkung vornehmen. Da anzunehmen ist, dass ein gemessener Sensorwert nur für diesen Teil des Gehirnes die Aktivität darstellt, wird er die Aktivitäten in anderen, weit entfernten Teilen des Gehirns nicht beeinflussen. Er sollte deswegen bei einer Interpolation auch nicht betrachtet werden. Die CancelDistance gibt an (in m) bis zu welcher Distanz Entfernungen um einen Sensor berechnet werden.
Bei einer CancelDistance von 1,5 sieht die Tabelle wie folgt aus:

```
MNELIB::MNEBemSurface Surface = bemSurface;
QSharedPointer<QVector<qint32>> MappedSubset = projectSensors(Surface, SensorPos);
double CancelDistance = 1.5;
QsharedPointer<MatrixXd> scdcMap = GeometryInfo::scdc(Surface, MappedSubset, CancelDistance);
```

	A	C
A	0	∞
B	1	1
C	∞	0
D	1	1

Da die Standartbelegung aller Werte unendlich ist, bleiben nun auch einige Distanzen unendlich. Sollten jedoch Entfernungen jeglicher Länge berechnet werden (also keine CancelDistance angenommen werden) so kann dies auf zwei Arten geschehen. Entweder wird eine CancelDistance gewählt, welche größer ist als alle Entfernungen zwischen Sensoren und beliebigen Knoten, oder sollte dies nicht klar, bzw. zu unsicher sein, kann die CancelDistance auf unendlich gesetzt werden:

```
#define DOUBLE_INFINITY std::numeric_limits<double>::infinity()
double CancelDistance = DOUBLE_INFINITY;
```

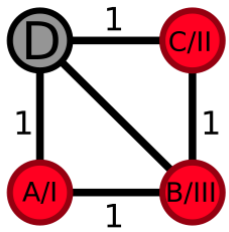
Für einen Kopf eignen sich CancelDistance Werte zwischen 0,05 und 0,2.

Bisher haben wir jeden Sensor berücksichtigt. Es ist jedoch so, dass einige Sensoren falsche Signale liefern. Die Gründe dafür sind vielseitig (defekter Sensor, zu große Entfernung zwischen Kopfhaut und Sensor, …). Bei der späteren Interpolation ist es dann jedoch so, dass fehlerhafte Messwerte, nicht nur den einen Punkt auf der Kopfoberfläche verfälschen, sondern sich aufgrund der Interpolation auch im Umkreis negativ auf das Ergebnis auswirken. Aufgrund dessen werden diese Sensoren auch als badChannel markiert und müssen entsprechend behandelt werden. Da später auch für diese Sensoren Messwerte ankommen, lohnt es sich diese Vorzeitig herauszufiltern. Am einfachsten geht es, in dem wir annehmen, dass die Distanz zu einem Knoten welcher einen Sensor repräsentiert der als badChannel markiert ist immer unendlich ist.

Nehmen wir in unserem Beispiel also an, dass nicht nur die Knoten A und C einen Sensor repräsentieren sondern auch der Knoten B einen Sensor repräsentiert:

Das Ergebnis der scdc dazu sieht dann also wie folgt aus:

	A	B	C
A	0	1	∞
B	1	0	1
C	∞	1	0
D	1	∞	1



Nehmen wir nun an das der Sensor III auf dem Knoten B als badChannel markiert ist, erhalten wir durch Anwendung der Funktion filterBadChannel folgendes Ergebnis:

```
QsharedPointer<MatrixXd> scdcMap = GeometryInfo::scdc(Surface, MappedSubset, CancelDistance);
FIFFLIB::FiffInfo fiffInfo = fiffInfo;
int SensorType = iSensorType;
QVector<qint32> GeometryInfo::filterBadChannels(scdcMap, fiffInfo, SensorType);
```

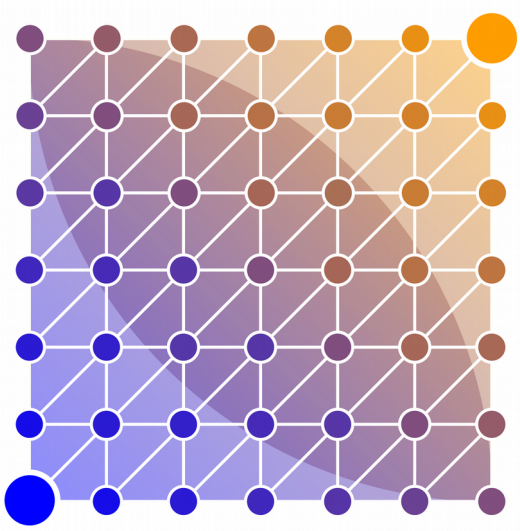
	A	B	C
A	0	∞	∞
B	1	∞	1
C	∞	∞	0
D	1	∞	1

Die Entfernung eines jeden Knotens zu diesem Knoten wurde nun als unendlich angenommen, weswegen später kein Knoten von diesem Sensor beeinflusst wird.
fiffInfo stellt dabei sämtliche Informationen über die Sensoren bereit und SensorType gibt den Typ des Sensors an (FIFFV_EEG_CH oder FIFFV_MEG_CH).

CreateInterpolationMat:
Eingabe: MappedSubset, scdcMap, interpolationFunction, CancelDistance, fiffInfo, SensorType
Ausgabe: weightMatrix

Es muss eine Gewichtsmatrix erstellt werden. Das heißt, dass eine Matrix benötigt wird, in welcher für jeden einzelnen Knoten steht, zu wie viel Prozent er von welchen Sensoren beeinflusst wird. Benötigt wird das ganze, damit der letzte Schritt, die Interpolation, bzw. das ausrechnen der Werte, einen möglichst geringen Zeitaufwand besitzt.

Benötigt wird die Interpolation um fehlende Inormationen zu „erraten“ bzw. zu berechnen. Unser Mesh besteht aus viel mehr Knoten als wir Sensorwerte bekommen. Wir wissen als nur an den Positionen der Sensoren die Gehirnaktivität. Jedoch gibt es auch im restlichen Teil Aktivitäten. Wir gehen davon aus das diese zwischen den Sensoren in etwa einer Mischungen aus beiden gemessenen Werten entspricht. In dem nebenstehenden Beispiel ist dies farblich verdeutlicht. Auf einem Mesh mit 49 Knoten gibt es nur zwei gemessene Werte (links unten und rechts oben). Die restlichen Knotenpunktwerte müssen berechnet werden. Je näher ein Punkt dabei an einem Sensor dran ist, desto ähnlicher sind sich beide Intensitäten. In Punkten die in gleicher weise von beiden Sensoren entfernt sind ist auch die Intensität in gleicher Weise von beiden Punkten beeinflusst.



Neben der oben dargestellten linearen Interpolation gibts es auch noch verschiedene andere Interpolations-Möglichkeiten. Um den Nutzer völlige Freiheit zu geben, wird die genutzte Funktion als Parameter mit übergeben. Aktuell sind 4 verschiedene Interpolations-Funktionen implementiert (linear, quadratisch, kubisch und die gauß-Funktion).

Das Ergebnis für das Beispiel wäre:

```
QSharedPointer<QVector<qint32>> MappedSubset = projectSensors(Surface, SensorPos);
QsharedPointer<MatrixXd> scdcMap = GeometryInfo::scdc(Surface, MappedSubset, CancelDistance);
double (*interpolationFunction) (double) = Interpolation::linear;
double CancelDistance = 1.5;
FIFFLIB::FiffInfo fiffInfo = fiffInfo;
int SensorType = iSensorType;
QSharedPointer<SparseMatrix<double> > weightMatrix =
Interpolation::createInterpolationMat(MappedSubset, scdcMap, interpolationFunction, CancelDistance,
fiffInfo, SensorType)
```

	A	C
A	1	0
B	0,5	0,5
C	0	1
D	0,5	0,5

Für die Interpolations-Funktion stehen dabei folgende Varianten zur Verfügung:

```
double (*interpolationFunction) (double) = Interpolation::linear;
double (*interpolationFunction) (double) = Interpolation::square;
double (*interpolationFunction) (double) = Interpolation::cubic;
double (*interpolationFunction) (double) = Interpolation::gaussian;
```

interpolateSignal:
Eingabe: weightMatrix, MeasurmentData
Ausgabe: fullInterpolatedVec

Diese Funktion dient der eigentlichen live-Interpolation. Aus dem Eingangssignal, welches nur Werte für die Sensoren enthält wird mithilfe einer einfachen Matrix-Vektor-Multiplikation ein Vektor erzeugt, welcher für jeden Knoten aus dem Mesh einen Intensitätswert besitzt.

```
QSharedPointer<SparseMatrix<double> > weightMatrix =
        Interpolation::createInterpolationMat(MappedSubset, scdcMap, interpolationFunction,
        CancelDistance, fiffInfo, SensorType);
VectorXd MeasurementData = vecMeasurmentData;
QSharedPointer<VectorXf> fullInterpolatedVec = Interpolation::interpolateSignal(weightMatrix,
MeasurmentData);
```

Beispiel:
Eingangssignal:

100
0

Berechnetes Ausgangssignal:

100
50
0
50

Im letzten Schritt erfolgt dann eine Umwandlung in die, den Intensitäten entsprechenden, Farbwerte.