Software-Project 2017

Detailed Design

Real-Time Mesh Utilities

Petros Simidyan	Blerta Hamzallari	Felix Griesau	Marco Klamke
Julius Lerm	Lars Debor	Simon Heinke	Sugandha Sachdeva

last change: July 3, 2017

figures/mne-cpp.png

Contents

1 General Architecture

The features of the Real-Time Mesh Utilities are distributed onto two classes: the Surface Constrained Distance Calculation (from now on abbreviated with SCDC) and the Sensor Projecting can be found inside class **GeometryInfo**, while the Interpolation is housed inside class **Interpolation**.
Both classes lie within the library layer of MNE-CPP:
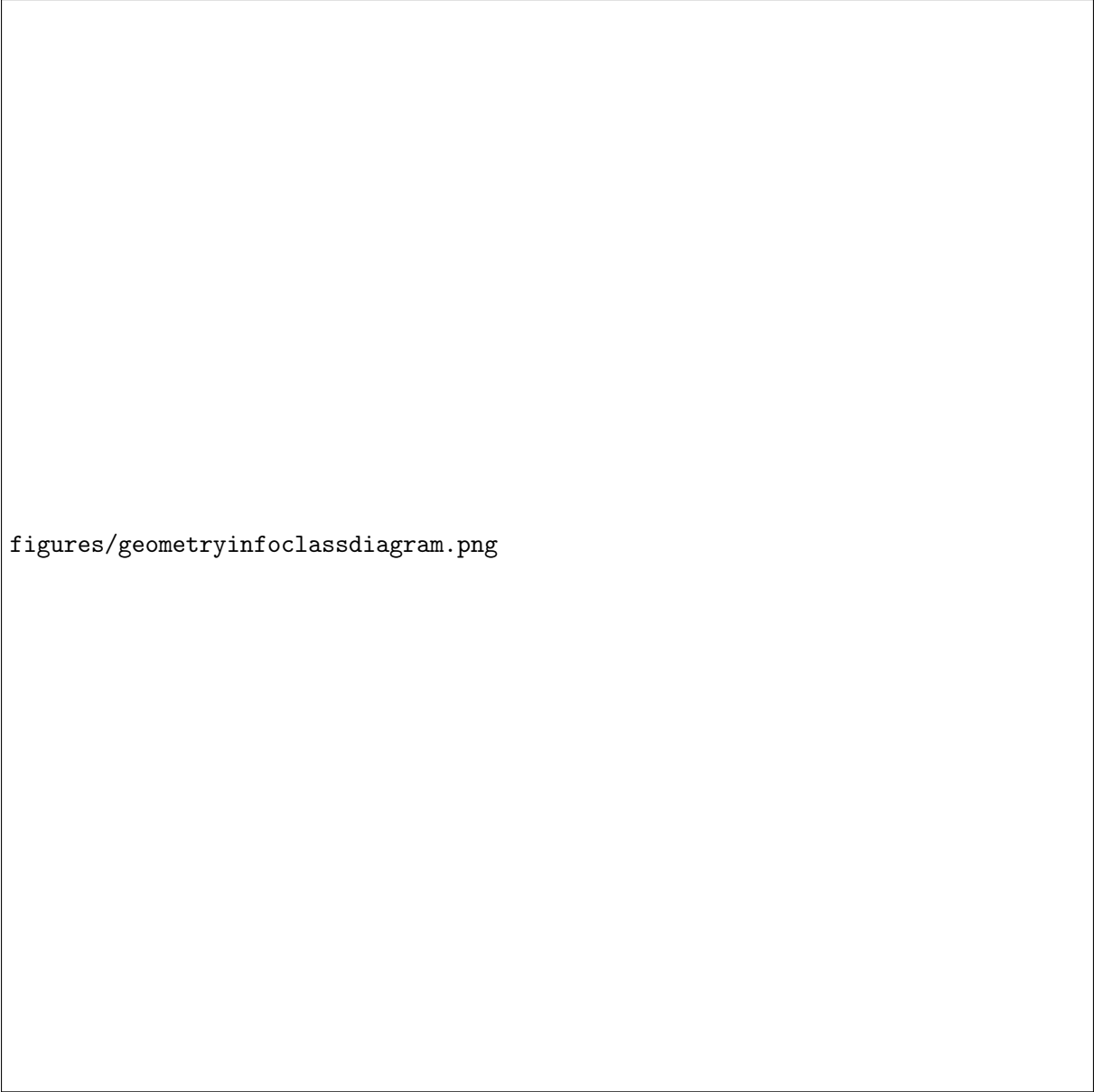
Figure 1: Overview of the MNE-CPP basic architectural layout

The classes are to be used by the application layer which includes all front end applications (i.e. MNE Scan, MNE Analyze and MNE Browse). The added functionality thus extends the preexisting framework.
For a seamless integration into mentioned front-end applications, the new features were added to the hierarchical GUI structure of MNE-CPP. This required for a so called DataTreeItem to be created. The latter allows easy modification of parameters of the two classes (see detailed class descriptions for more details).

2   GeometryInfo

The class **GeometryInfo** holds all needed functionality for the SCDC and the Sensor Projecting (see Functional Specification for further details). Since all included methods are static, the class itself does not have to be instantiated, thus it also is declared static and the default constructor is forbidden. The class **GeometryInfo** does not have any class members.

figures/geometryinfoclassdiagram.png

Figure 2: Interface of GeometryInfo

2.1 Member Functions

2.1.1 scdc

The method **scdc** originally was thought to receive only one input argument, i.e. the MNEBemSurface (object that holds the needed adjacency information). Based on this, it was supposed to calculate the full distance table of all contained vertices. Since the latter would have needed around 160 gigabytes of memory for some meshes, it was decided that the method also receives a subset of vertices. It then only calculates all distances for each vertex of said subset. The subset argument is defaulted with an empty vector, in case a subset could not be provided or is not wanted.

After first tests revealed that an unrestricted distance calculation would take far too long, a third argument was introduced: a double-value that acts as a distance threshold and thus restricts the distance calculation to vertices which lie within the said radius. This seems a bit imprecise or careless at first glance, but it actually is reasonable for this specific application: bilateral stimulation of cells inside the brain only occurs up to a certain distance. The third argument is defaulted with the maximum double-value, in case a distance threshold could not be provided or is not wanted.

After these restrictions and adaptions were agreed upon, the question which algorithm to choose arose. While Pettie and Ramachandrans algorithm had the best asymptotic run-time, further research revealed that an implementation would go far beyond the scope of this project, since the algorithm uses several other subroutines (such as Prims method for minimal spanning trees). Another possible candidate for solving the problem was Thorups algorithm. Testing of a publicly available implementation showed that it actually did worse than another algorithm, when the above mentioned restrictions to the problem were applied. The only algorithm that resulted in a lower computation time was an iterative version of Dijkstra's algorithm (see below for more details).

The **scdc** method allocates the necessary memory and initializes the distance table. Since two instances of **iterativeDijkstra** would not interfere with each other (because they work on different columns of the distance table), it was decided for **scdc** to start several threads. According to the number of available threads (or cores, depending on hyperthreading etc.), the passed subset of vertices gets split into equally sized parts, whose limiting indices are then passed to an instance of **iterativeDijkstra**. The **scdc** method then was supposed to wait for all threads to finish by using a periodic check, so it could return the filled distance table to the caller.

To further improve performance, this procedure was slightly changed: one instance of **iterativeDijkstra** gets executed on the main thread, i.e. the thread that called **scdc**. This avoids unnecessary thread changes for the periodic check and in that way decreases computation time.

Figure 3: Sequence diagram of method **scdc**. Arguments to calls are not included to ensure readability

2.1.2   iterativeDijkstra

The method **iterativeDijkstra** receives six input arguments: Firstly a pointer to the distance table which the results are to be written into. Secondly a reference to an MNEBemSurface holds the necessary adjacency information. As described above, the **iterativeDijkstra** method receives a vector of indices which point to vertices inside the MNEBemSurface. Added to that, it gets passed two indices which limit the section of said vector that the respective instance of **iterativeDijkstra** is responsible for.
Since the external requirements (i.e. low computation time and overall high efficiency) induce some restrictions (see description of **scdc** for more details), a distance threshold gets passed as the sixth

argument. Because Dijkstra's algorithm is so well known, its general implementation is not described in this document. Apart from a standard version of Dijkstra's algorithm, **iterativeDijkstra** ignores vertices that have a higher distance than said threshold, that means it does not check their neighbors for possibly better paths.

This method is private since it only gets called by **scdc** as a subroutine and is not part of the external interface of class **GeometryInfo**.
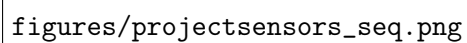
2.1.3   projectSensors

The method projectSensors receives two input arguments: An MNEBemSurface for the adjacency information and a vector of sensor positions in 3D space. It was intended for **projectSensors** to use a k-D tree for fast lookups during the mapping of sensors. After the k-D tree was implemented, it was striking that average times for a lookup (which should be logarithmic in theory) were far too slow to sustain an overall efficient run-time of the program. Added to that, the method for building the tree could only be executed on one thread, since the structural dependencies within a k-D tree are far to complicated to be divided onto two ore more threads.

A simple linear search that finds the minimal distance between any vertex of the mesh and the respective sensor resulted in a better run-time. Although this was originally thought of as a verification tool due to its simplicity, it was then extended and optimized.

Since two instances of linear search would not interfere with each other, it was decided for **projectSensors** to start several threads. According to the number of available threads (or cores, depending on hyperthreading etc.), the passed vector of sensor positions gets split into equally sized parts, whose limiting iterators are then passed to an instance of **nearestNeighbor**. Each instance returns a vector of IDs which refer to vertices of the mesh. The **projectSensors** method was supposed to wait for all threads to finish by using a periodic check, so it could append the resulting vectors in the right order and return the outcome to the caller.

To further improve performance, this procedure was slightly changed: one instance of **nearestNeighbor** gets executed on the main thread, i.e. the thread that called **projectSensors**. This avoids unnecessary thread changes for the periodic check and thus decreases computation time.

Figure 4: Sequence diagram of method **projectSensors**. Arguments to calls are not included to ensure readability

2.1.4   nearestNeighbor

The method **nearestNeighbor** receives three input arguments: An MNEBemSurface, and two iterators. The first iterator points to the start and the second to the end of a section of a vector. It then iterates over every sensor position between the two iterators and searches for the vertex with the closest distance to the respective sensor. In order to achieve this, it compares the distances of all vertices stored inside the MNEBemSurface. After a shortest distance is found for a specific sensor, the ID of the corresponding vertex is pushed back into a result vector. After the limiting iterator has been reached, this result vector is then returned to the caller.
This method is private, since it only gets called by **projectSensors** as a subroutine and is not part of the external interface of class **GeometryInfo**.

2.1.5   matrixDump

This method is implemented for testing and verifying purposes only. It receives a matrix and a string. It then writes the content of the matrix into a file that is named with said string.

2.1.6   squared

This method is implemented for better readability only, thus it is declared private. It receives a double-value and returns it squared. In order to avoid efficiency losses through stack frames, it is declared inline.

3   Interpolation

The class **Interpolation** holds all needed functionality to create a weight matrix based on a distance table and to interpolate sensor signals. Since all included methods are static, the class itself does not have to be instantiated, therefore it also is declared static and the default constructor is forbidden.
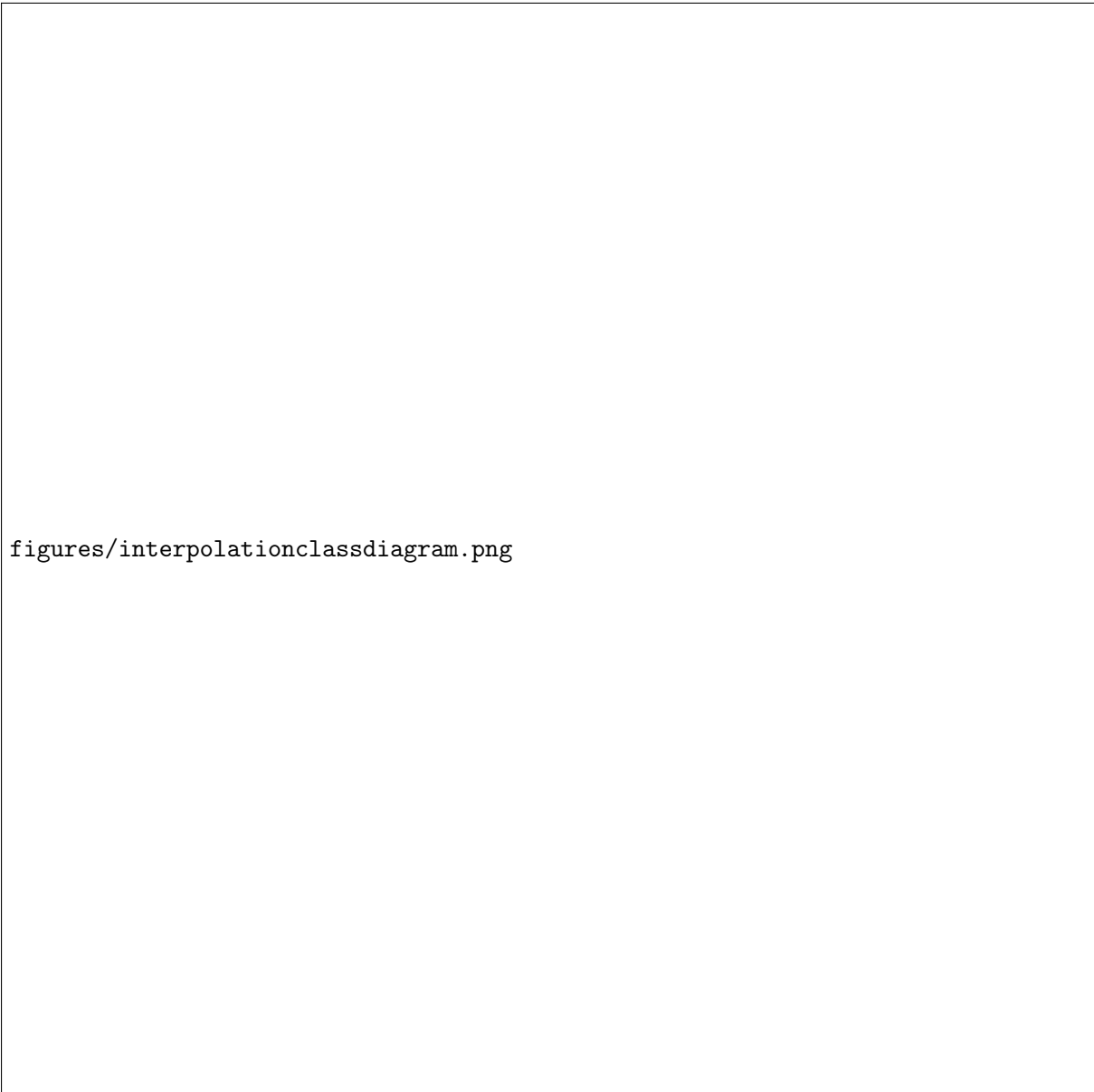
figures/interpolationclassdiagram.png

Figure 5: Interface of Interpolation

3.1 Member Objects

3.1.1 Weight Matrix

The class **Interpolation** holds one static class member: **m_interpolationMatrix** which is a shared pointer to a sparse matrix (i.e. the weight matrix). Since a user of the **Interpolation** class does not care about internal representation of the weight matrix and is only interested in fast interpolation of signals, it was decided that the weight matrix should be stored as a member of the class. In case access to the matrix itself should be needed, a designated **getter**-function was included (see method **getResult** for more details).
Originally, this member was supposed to be of type **Eigen::MatrixXd** which is a standard matrix of double-values with variable dimensions. After first tests revealed that depending on the distance threshold chosen during the calculation of distances on the mesh (see section 2.1.1) the weight matrix contained around 97% of null-entries, the weight matrix was changed to type **Eigen::SparseMatrix<double>** which is optimized for sparse matrices. This implied some changes to the calculation of the weight matrix within the method **createInterpolationMat** because **Eigen::SparseMatrix<double>** do not provide the same interface as **Eigen::MatrixXd**.

3.2 Member Functions

3.2.1 createInterpolationMat

The method **createInterpolationMatrix** receives four input arguments: Firstly a vector of IDs which represent the vertices which have been assigned to a sensor during the Sensor Projecting (see section 2.1.3 for more details). Secondly a shared pointer to a distance table, i.e. a matrix of double-values. Since the weight matrix can be calculated using a variety of mathematical interpolation functions (e.g. a linear function, a quadratic function, etc.), the method **createInterpolationMat** was thought to receive some kind of value to represent the chosen function. That could for example be achieved by defining global constants and encoding the functions with them. For a higher level of versatility it was decided that **createInterpolationMat** should receive a pointer to a function, which in turn receives a double-value and returns a double-value and thus fulfills the internal requirements of the weight matrix calculation. A minimal preset of usable functions was provided as public members of the class (see section 3.2.5 for more details).
Since bilateral stimulation of cells in the brain only occurs up to certain distances, a double-value is received as a fourth argument. It acts as a distance threshold, that means that the interpolation weight of vertices with a higher distance than said threshold is set to zero.

3.2.2 interpolateSignal

The method **interpolateSignal** receives a vector of double values which represents a specific moment of brain activity, i.e. the values that were recorded by sensors. The weight matrix gets multiplied with the input vector and thus produces a vector that contains the brain activity for all vertices of the distance table that was used to create the weight matrix.

3.2.3 clearInterpolationMatrix

The method **clearInterpolationMatrix** resets the weight matrix, i.e. frees all memory allocated for the respective member object.

3.2.4 getResult

This method is implemented in case the calculated weight matrix is needed as an object itself. It returns a smart pointer to the weight matrix.

3.2.5 linear, gaussian, square

These methods are implemented to provide a minimal preset of functions to use during the calculation of the interpolation weight matrix (see method **createInterpolationMatrix** for more details).
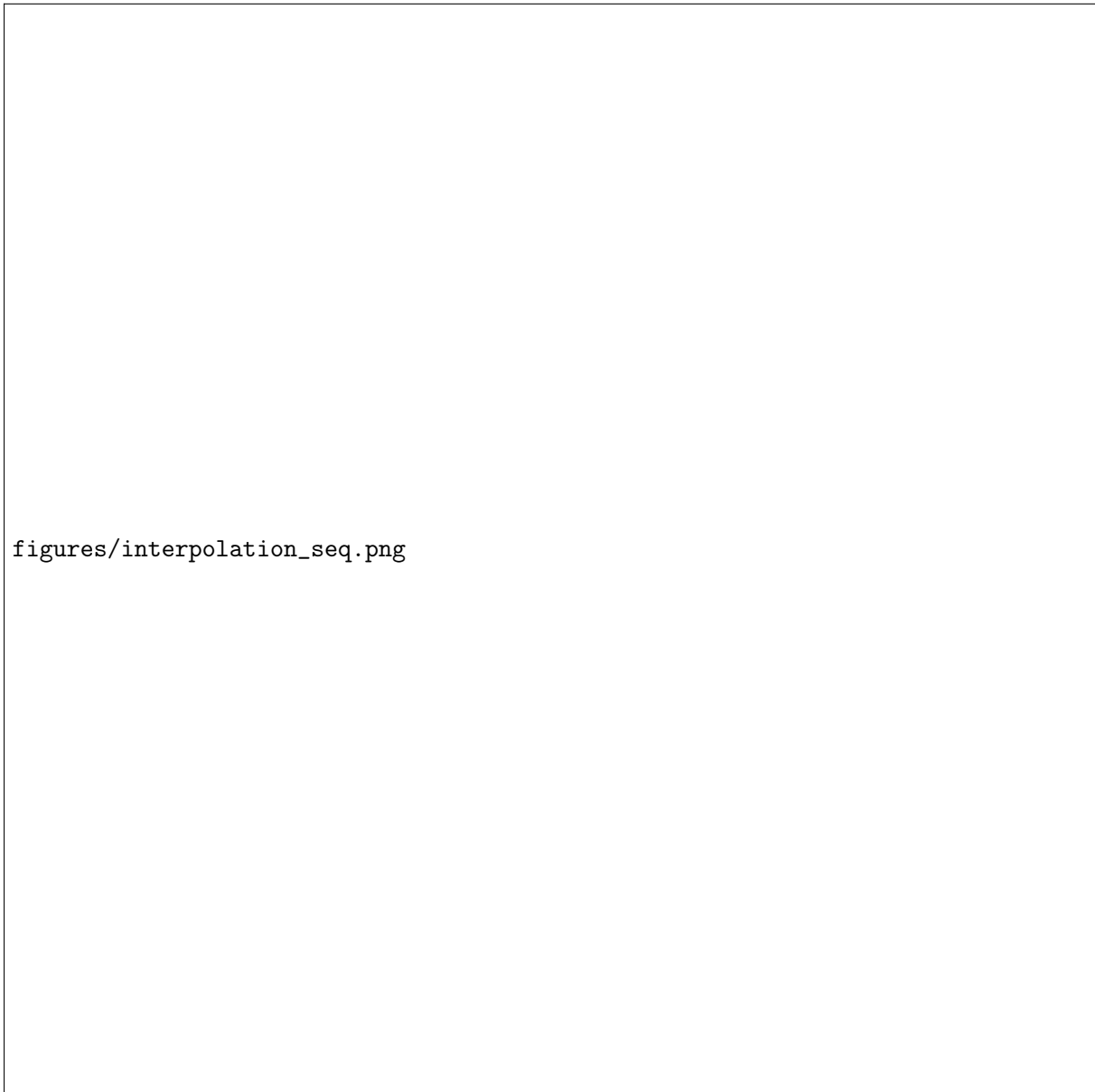
figures/interpolation_seq.png

Figure 6: Sequence diagram for class **Interpolation**. Arguments to calls are not included to ensure readability

4   SensorDataTreeItem

The class SensorDataTreeItem is used for the communication between our previously described classes and the Disp3D environment within the MNE-CPP framework. The most important member is a pointer to an RtSensorDataWorker, which is described in chapter five. Most of the functions in this class are used to control said RtSensorDataWorker. Only the relevant member functions are mentioned below, since the full description of this class and how it interacts with the Disp3D environment would go beyond the scope of this project.
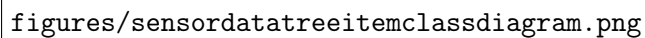
figures/sensordatatreeitemclassdiagram.png

Figure 7: Interface of the SensorDataTreeItem

4.1   Member Objects

4.1.1   m_bIsDataInit

This member of type bool is used as a flag. It should verify that the Item is initialized.

4.1.2   m_iUsedSensors

This member of type QVector<int> stores all indices of the used sensors out of given FiffEvoked object. For example, for a measurement with EEG signals the vector stores all indices of EEG sensors.

4.1.3   m_lSensorsBad

This member of type QVector<int> stores all indices of bad channels. Bad channels can be marked during a measurement and the member will be updated accordingly.

4.1.4   m_pSensorRtDataWorker

This member is a pointer to the items worker. The worker itself is described in section 5.

4.2   Member Functions

4.2.1   Constructor

The constructor of the SensorDataTreeItem initializes its member m_bIsDataInit with the value *false* and then calls the private method *initItem*.

4.2.2   initItem

This method is private since it only gets called by the constructor.
It sets the items textual description that is visible to the user and initializes all MetaTreeItems. The latter are needed to give the user control options when controlling the program through the Disp3D menu:

StreamStatus The start/stop option for the interpolation and visualization process.

ColorMapType The active color map that is used to visualize the data is chosen in a drop-down menu.

VisualizationTresholds These thresholds are used to control maximum and minimum values in the visualization.

StreamingTimeInterval Allows to set the minimal time that one frame should be displayed.

LoopedStreaming If this option is set, the worker will stream the same block of measurement repeatedly. Otherwise it will wait for new real-time and display it once.

DistanceThreshold This meta item allows on-the-fly change of the distance threshold used in the surface constrained distance calculation and the creation of the interpolation matrix. Changing this value triggers a recalculation of the latter by calling *calculateSurfaceData*.

InterpolationFunction This meta item allows on-the-fly change of the function used in the creation of the interpolation matrix. It provides a drop-down menu for the user.

4.2.3 init

This method must be called before any data can be streamed.
It instantiates the worker member (described in section 5), extracts sensor coordinates, registers bad channels and instructs the worker to calculate all data that is needed for the later live interpolation (i.e. the surface constrained distance calculation and the weight matrix).
This method sets the *m_bIsDataInit* flag to *true*.

4.2.4 addData

This method adds a new block of real-time data to the list of sensor data that is to be visualized. The data is filtered so that only the values of the used sensors (see member m_iUsedSensors) are passed to the worker.

4.2.5 updateBadChannels

This method allows on-the-fly marking of bad channels. It receives a FiffInfo objects and extracts all bad channels. It then passes their indices to the worker.
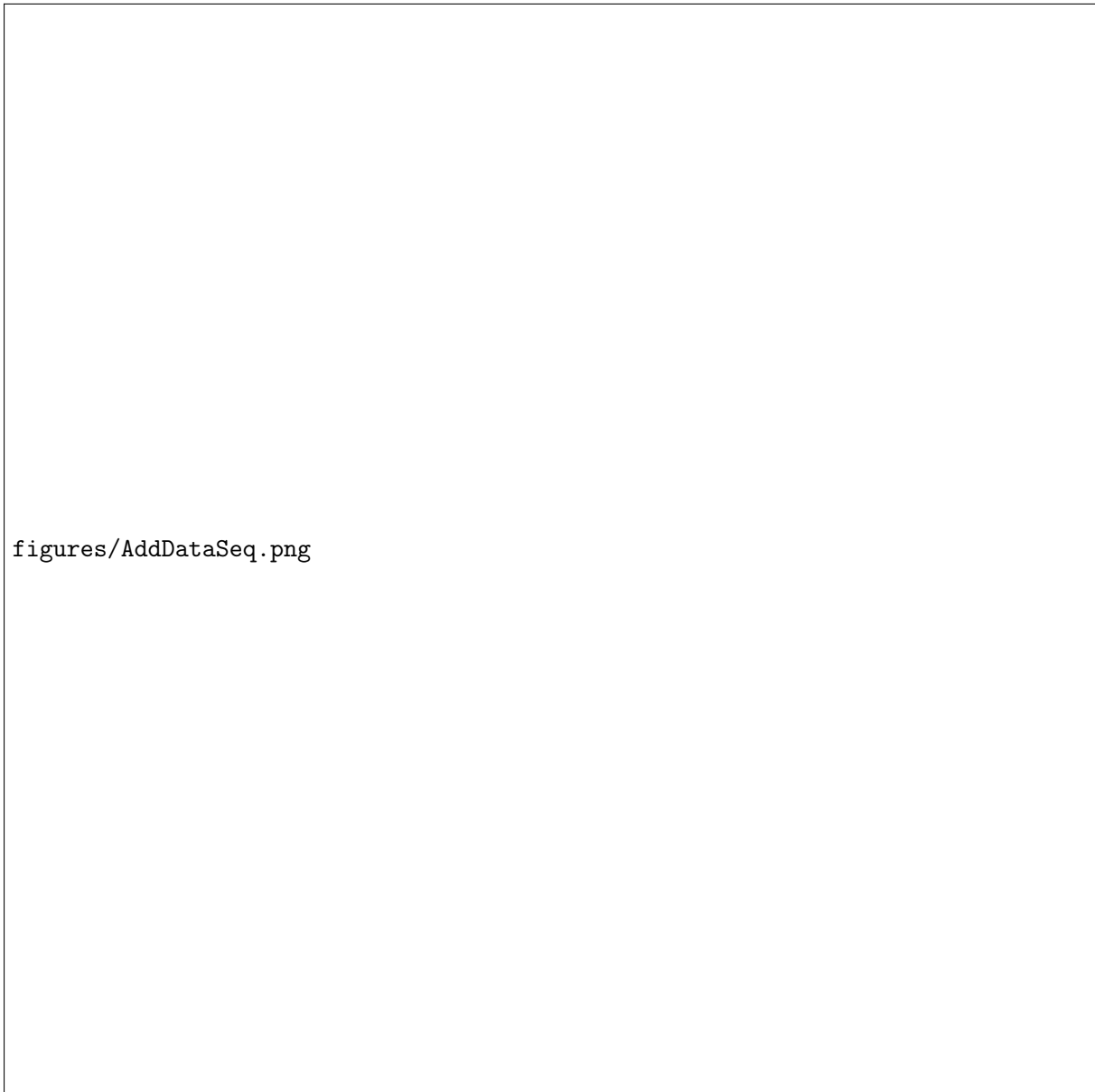
figures/AddDataSeq.png

Figure 8: Sensor data is added from hierarchically higher items

5 RtSensorDataWorker

This class manages real-time interpolation of sensor signals and streaming of visual output. It derives from the class QThread and therefore provides possibilities to manage threading and periodic output of visual frames.
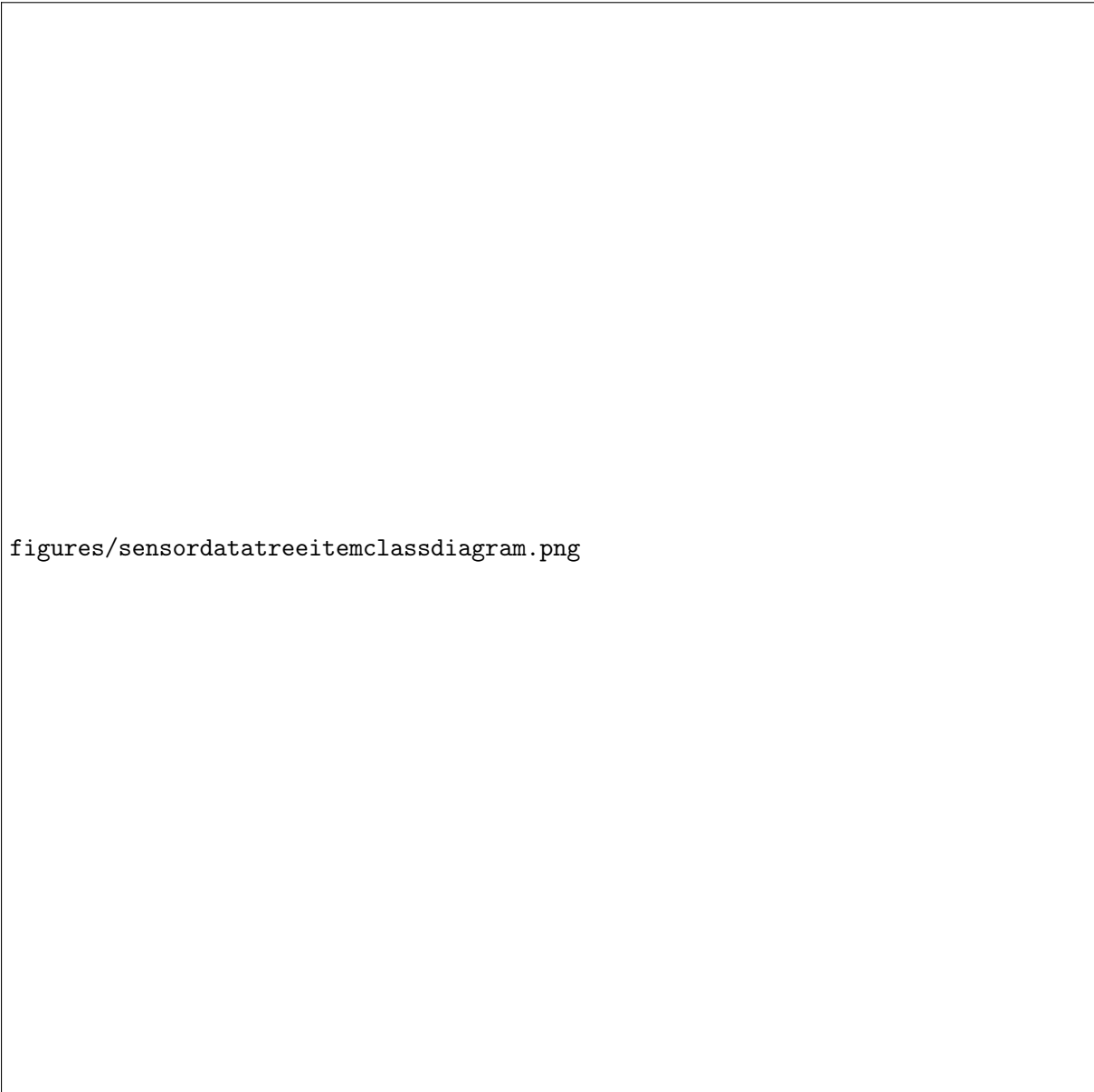
figures/sensordatatreeitemclassdiagram.png

Figure 9: Interface of the SensorDataTreeItem

5.1 Member Objects

5.1.1 m_qMutex

This member manages semaphore functionality: Since the user often interacts with the item, respectively the worker, members and methods are often accessed by several threads. To avoid data races or even crashes, all critical functions use this mutex.

5.1.2 m_lData

This list holds all currently available sensor activity as vectors of double values.

5.1.3 m_bIsRunning

This flag allows to stop the main loop of the worker. It is set to *true* when the worker is started and is set to *false* when the worker is stopped.

5.1.4 m_bSurfaceDataIsInit

This flag displays if the worker has received the necessary information for the surface constrained distance calculation and the computation of the interpolation matrix. It is used for rejecting calls to methods that need this information and would produce critical behaviour if the data is not calculated yet.

5.1.5 m_lInterpolationData

This struct holds all necessary information for the live interpolation: the used sensor type, the distance threshold, the mesh, the sensor information and all results of the necessary computations: the mapped sensor subset, the distance table and the weight matrix.
It acts as a container for said data objects.

5.2 Member Functions

5.2.1 Constructor

The constructor initializes all members with default values. The *m_lInterpolationData* struct is instantiated and filled with default values as well.

5.2.2 addData

This method receives a block of sensor activity represented as a matrix of double values. It splits up the matrix into several columns and appends each column to the list of sensor signals (member *m_lInterpolationData*).

5.2.3 calculateSurfaceData

This method is overloaded: there is a public method *calculateSurfaceData* that receives an MNEBemSurface, a vector of sensor positions, a FiffInfo objects and a sensor type. It stores all passed input arguments inside the member *m_lInterpolationData* and then calls the private method *calculateSurfaceData*. The latter calculates the surface constrained distances and the weight matrix based on the data that is stored inside the *m_lInterpolationData* member.

5.2.4   updateBadChannels

This method receives a FiffInfo object that holds the bad channel information. It stores the passed reference inside the *m_lInterpolationData* member and then recalculates surface data.

5.2.5   run

This is the workers main loop.
According to the users settings, it averages the next sensor signals by removing them from the member *m_lInterpolationData*. It then generates vertex colors based on this averaged signal by calling *generateColorsFromSensorValues* and publishes the result to all relevant items (i.e. the items that manage visualization). After all computations are done, the *run* method calls *QThread::msleep* to sleep for a certain (depending on the time needed for the previous computations).

5.2.6   generateColorsFromSensorValues

The method *generateColorFromSensorValues* performs several checks on the workers members to ensure that the following critical computations will succeed. It then interpolates vertex activity by passing the signal it received from the workers main loop to *Interpolation::interpolateSignal*. The interpolated result then gets passed to *normalizeAndTransformToColor*.

5.2.7   normalizeAndTransformToColor

The method *normalizeAndTransformToColor* normalizes the passed signal according to the passed thresholds. It then used the passed function pointer to generate RGB values for each vertex.

figures/PublishColorsSeq.png

Figure 10: Interpolated visual data is passed through the Disp3D environment

6 Glossary

Mesh describes the representation of a planar graph in 3D space.

Eigen is a C++ template library for linear algebra that includes matrices, vectors, numerical solvers, and related algorithms

MatrixXd is a specialized class from the Eigen framework. It represents a matrix of double-values (Matrix**X**d) that has variable dimensions (Matrix**X**d)

SparseMatrix<T> is an optimized version of standard Eigen matrices that is designated for sparse matrices.

MNEBemSurface is a class from the MNE-CPP framework that holds information about the represented surface and its geometry.

SCDC (surface constrained distance calculation) determines the shortest path between two points on a tessellated surface.

MNE-CPP is a cross-platform C++ framework, which provides MEG/EEG tools and applications for fast non-invasive brain monitoring.

Pettie and Ramachandran Seth Pettie and Vijaya Ramachandran published their paper "A Shortest Path Algorithm for Real-Weighted Undirected Graphs" in 2005.

Prim Robert C. Prim had republished a previously known algorithm for finding minimal spanning trees in 1957.

Thorup Mikkel Thorup is a computer scientist who published his algorithm for solving the single source shortest path problem in 1997.

Dijkstra Edsger W. Dijkstra was a computer scientist who published his algorithm for solving the single source shortest path problem in 1959.

Linear search or sequential search is a method for finding a target value within a list.