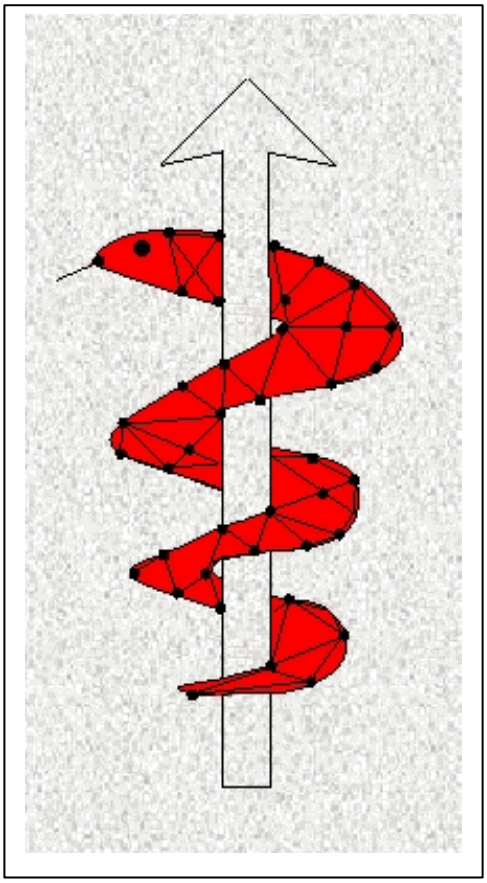
The IST Programme
Project No. 10378

SimBio

SimBio - A Generic Environment for Bio-numerical Simulation

http://www.simbio.de



D4.1b: Inverse Problem Methodology Release Notes Simbio_ipm_ver2001_04_30

Status:	Final
Version:	2.0
Security:	Public
Responsible:	A.N.T Software
Authoring Partners:	A.N.T Software MPI of Cognitive Neuroscience

Release History

Version	Date
1.0	06.04.2001
2.0	27.04.2001

The SimBio Consortium :

NEC Europe Ltd. – UK
A.N.T. Software – The Netherlands
CNRS-DR18 – France
Sheffield University – UK
MPI of Cognitive Neuroscience – Germany
Biomagnetisches Zentrum Jena – Germany
ESI Group – France
Smith & Nephew - UK

Contents:

SimBio: A Generic environment for bio-numerical simulation

D4.1b: Inverse Problem Methodology Design Report
Release Notes Release Notes: Simbio_ipm_ver2001_04_30

1. Introduction

The objective of WP 4 Subtask 4.1 is the generation of a generic inverse toolbox containing a large variety of inverse problem solvers and an error estimation package, which will include methods to estimate the sensitivity of the results of source reconstruction to inaccuracies in forward modeling.

The purpose of the current release of the generic toolbox is to show that the software design fits the requirement of providing a set of state of the art inverse algorithms and also providing simple interfaces to add additional methods. Thus, this release includes a set of algorithms, which can be combined to complete inverse procedures but also possibilities to select between optional algorithms showing the flexibility and generality of the software design. Further, this release contains an environment, which allows the coupling of the inverse toolbox to software components of the other SimBio partners.

This release notes cover the implemented algorithms. If an algorithm can be described by a short formula, it is provided. For other algorithms a reference to the literature is given. The HTML-Documentation of this release includes an access to all class interfaces of the inverse toolbox. A comprehensive overview about the algorithms can be found in the "D4.1a: Inverse Problem Methodology Design Report" (http://www.ccrl-nece.technopark.gmd.de/simbio/deliverables.html).

2. Software Design: General

The inverse toolbox uses a class structure implemented in ANSI C++, additionally using STL classes. Abstract classes are used to minimize the number of interfaces, to get a simple access to methods and to keep implementation details out of interfaces. Software of the NeuroFEM package, which is implemented using FORTRAN 90, is interfaced to the C++ classes.

The inverse toolbox is tested on a windows platform using the Microsoft Visual C++ software development environment and on Unix systems (Linux, SGI-Irix) using the Gnu C++-Compiler. This release contains libraries and a binary command line tool for Linux and SGI-Irix.

3. Inverse Toolbox

The inverse toolbox is implemented as a class structure. To be able to integrate algorithms in a simple and flexible way abstract classes are used. **The hierarchy of abstract classes provides an interface for entire categories of algorithms**. There is, e.g., an interface to address all analysis methods, one for all inverse analysis methods, one for all goal (or cost) functions, etc. The implementation of methods is done by classes derived from these abstract classes.
This principle has several major advantages:

(1) It allows for an elegant way to **combine different ingredients of an algorithm**. For example, the linear estimation procedure can be executed using any source space, any weighting method, any regularization, and any forward solution.
(2) The user can **derive new classes from the abstract interfaces** and thus implementing new forward solutions, weighting methods, etc. and use them together with the existing methods

without having to change any code within the toolbox or even knowing it (could be just object code or a library).

(3) The user can **address entire categories of algorithms**, e.g. all inverse procedures, by just using the appropriate abstract interface.

First a short overview about the abstract classes is given, followed by the description of the implementation classes. According to the general division of inverse methods into two branches. The description of the inverse toolbox is also separated into two parts: First, methods using a discrete parameter space, for parameters that have to be determined (e.g. source positions and orientations), second methods using a continuous parameter space for their parameters.

3.1 Abstract Class Interfaces: Discrete Parameter Space

This group contains inverse algorithms that allow only a finite number of discrete points as possible source locations. Such methods need a *grid*, provided by a *grid generator* that specifies these points and their neighbourhood relations. Discrete parameter spaces are used by linear and non-linear distributed source methods, by discrete non-linear dipole fit algorithms, and by scan methods, such as MUSIC. Figure 3.1 shows an overview about the class structure for methods using a discrete parameter space. Class names printed in grey are not yet realized in the current release of the inverse toolbox. The following paragraphs give a short description of the abstract class interfaces.

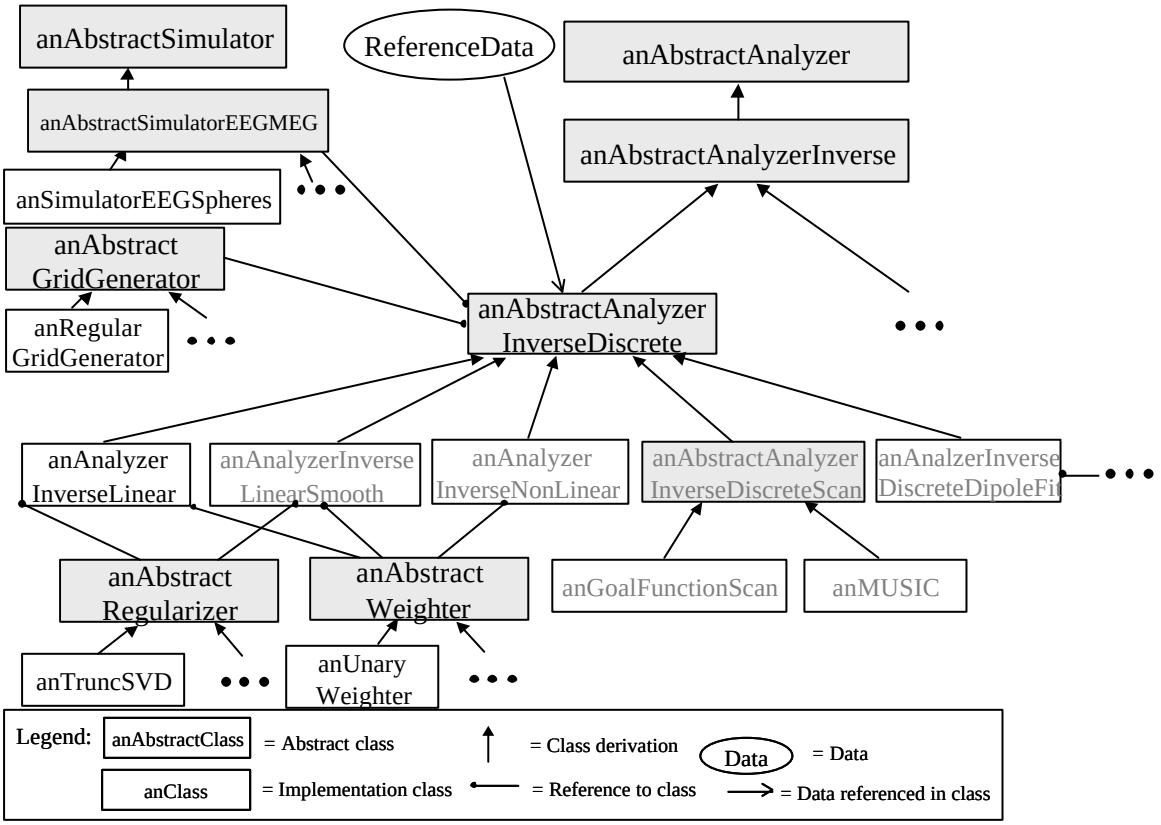


Fig. 3.1 Class structure for the implementation of methods using a discrete search space for their parameters. Class names printed in grey indicate classes not implemented in this release.

Abstract Analyzer

Class Name: anAbstractAnalyzer_c

Serves as a bases class for all types of analyzers. At the same time, it provides a generalized interface for any sort of data analysis that might be implemented into the tool box. A main component of this class is a pure virtual run() method, which is implemented in all classes derived from anAbstractAnalyzer_c.

Example:

```
void MyFunction(vector<anAbstractAnalyser_c &> list_of_operations)
{
        for (int i = 0; i < list_of_operations.size(); i++)
                list_of_operations[i].run();
}
```

In the above example *MyFunction* does not need to know, which particular analysis steps are represented by the members of the vector, it just carries them out.

Important public interface functions

- execute the method
  void run()

Abstract Inverse Analyzer

Class Name: anAbstractAnalyzerInverse_c

Serves as a base class and general interface for all types of inverse analyzers. It is derived from anAbstractAnalyzer_c. It contains a getResult() method, which returns a set of source parameters.

Important public interface functions

- execute the method
  void run()
- extract resulting source parameters
  bool getResult(utMatrix_t<double>& outParameters)

Abstract Inverse Discrete Analyzer

Class Name: anAbstractAnalyzerInverseDiscrete_c

Serves as base class for inverse methods having a discrete parameter space. It has a reference to an instance of anAbstractSimulatorEEGMEG_c and anAbstractGridGenerator_c. This enables the user to run any class that is derived from *anAbstractAnalyzerInverseDiscrete_c* with any forward solution and any definition of the source space that is implemented in the toolbox or created by the user him/herself.

Important public interface functions

- execute the method
  void run()

- extract resulting source parameters
  bool getResult(utMatrix_t<double>& outParameters)
- retrieve and set the grid generator for the source space generation
  anAbstractGridGenerator_c& getGridGenerator()
  bool setGridGenerator(anAbstractGridGenerator_c& inGridGenerator)
- retrieve and set simulator for forward calculation
  anAbstractSimulatorEEGMEG_c& getSimulator()
  bool setSimulator(anAbstractSimulatorEEGMEG_c& inSimulator);
- retrieve and set the reference data (measurement data) as a matrix with one row per channel and one column per time step
  void getReferenceData(utMatrix_t<double>& outReferenceData)
  void setReferenceData(utMatrix_t<double>& inReferenceData)

Abstract Grid Generator

Class Name: anAbstractGridGenerator_c

Base class for all kinds of grid generators. Grids are used to describe the search space (positions and directions of sources) and the head model used for forward calculations.

Important public interface functions

- generate and return grid (see header file for specification of *Grid_c*)
  Grid_c& generateGrid()

Abstract Regularizer

Class Name: anAbstractRegularizer_c

Base class for all kinds of regularizers. Used for matrix inversions of ill conditioned **lead field matrices**. Generates a regularized matrix from an input matrix.

Important public interface functions

- regularize matrix
  run(utMatrix_t<double>& inMatrix,utMatrix_t<double>& outMatrix)

Abstract Weighter

Class Name: anAbstractWeighter_c

Base class for all kinds of weighters. They are used for a spatial weighting of the points of the discrete search space. Can either be a diagonal matrix to weight each point independently or a matrix with non-diagonal values unequal zero to include neighbourhood relations. In case of a diagonal matrix only a vector containing diagonal elements is created.

Important public interface functions

- compute weights, either per grid point (vector) or as matrix
  computeWeights(Grid_c& inGrid, utVector_t<double>& WeigthingVector,utMatrix_t<double>& WeightingMatrix)

3.2 Implementation Classes: Discrete Parameter Space

Linear Inverse Analyzer

Class Name: anAnalyzerInverseLinear_c

Determines a solution of an inverse problem with the smallest quadratic norm (L2-norm) of the sources using the following formulas:

For the underdetermined case:

$$J = G \cdot L^T \cdot \mathrm{Re}\, g(L \cdot G \cdot L^T)^{-1} \cdot M$$

For the overdetermined case:

$$J = \mathrm{Re}\, g(L \cdot G \cdot L^T)^{-1} G \cdot L^T \cdot M$$

J = Source Strengths, M =Measured data, L=Leadfield Matrix, G= Weigthing Matrix, ^{T} = transposed of a matrix, $^{-1}$.= Pseudoinverse of a Matrix, Reg() = regularization operator

The linear inverse analyzer therefore needs the following methods:

- a forward solution for the computation of the lead field matrix *L*, provided by *anAbstractSimulatorEEGMEG_c*
- a weighting method for the computation of the weighting matrix *M*, provided by *anAbstractWeighter_c*
- a regularization operator, provided by *anAbstractRegulariser_c*
- a grid generator defining the positions and directions of the sources the strengths of which have to be determined, provided by *anAbstractGridGenerator_c*

How to create an linear inverse analyzer?

```
anAnalyzerInverseLinear_c(  const utMatrix_t<double>&     inReferenceData,
                            anAbstractSimulatorEEGMEG_c&  inSimulator,
                            anAbstractGridGenerator_c&    inGridGenerator,
                            anAbstractWeighter_c&         inWeighter,
                            anAbstractRegularizer_c&      inRegularizer);
```

Important public interface functions

- perform linear estimation
  void run();
- extract resulting parameter matrix
  bool getResult(utMatrix_t<double>& outParameters);

Grid Generators

Single Dipoles

Class Name: anGridGeneratorSingleDipoles_c

A class that can be filled with position and direction parameters of single dipoles. Can be used as a grid generator which provides a grid for the procedure which determines the linear parameters of dipole fit methods.

How to create an single dipole grid generator?

anGridGeneratorSingleDipoles_c( );

Important public interface functions

bool addGridNode(utVector_t<double> inPosition, utVector_t<double> NodeSourceDirections);

Regular Spherical Surface Grid

Class Name: anGridGeneratorSurfaceSphere_c

Generates a regular **angular** grid on the surface of a sphere. Radius of the sphere and spacing of the grid points can be set. Additionally a normal constrained can be set for the direction associated to sources on grid points.

How to create an spherical surface grid generator?

```
anGridGeneratorSurfaceSphere_c(   double              inRadius,
                                  utVector_t<double>& inCenter,
                                  double              inDelta,
                                  bool                inbNormalConst = false);
```

The first two parameters describe the sphere (in meters). *inDelta* gives the spacing of the points in α and ϑ direction in radians. If the flag *inbNormalConst* is set, the orientation of each grid point is radial, otherwise the 3 cartesian directions (x, y, z) will be assigned to each point.

Important public interface functions

- generate and return grid (see header file of *anAbstractGridGenerator_c* for specification of *Grid_c*)
  Grid_c& generateGrid()

Regular Spherical Volume Grid

Class Name: anGridGeneratorVolumeSphere_c

Generates a regular **cubic** 3 D grid inside of a sphere. Radius of the sphere and spacing of the grid points can be set.

How to create a spherical volume grid generator?

```
anGridGeneratorVolumeSphere_c(   double              inRadius,
                                 utVector_t<double>& inCenter,
                                 double              inDelta);
```

The first two parameters describe the sphere (in meters). *inDelta* gives the spacing of the points in meters. Each point is assigned the three canonical Cartesian directions .

Important public interface functions

- generate and return grid (see header file of *anAbstractGridGenerator_c* for specification of *Grid_c*)
  Grid_c& generateGrid()

FEM Volume Grid

Class Name: anVolumeGridGeneratorNeuroFEM_c

This class can be used as grid generator which provides a volume grid for the NeuroFEM simulator. The volume grid structure is read from a file.

How to create an volume grid generator?

anVolumeGridGeneratorNeuroFEM_c( );

Important public interface functions

- generate and return grid
  Grid_c& generateGrid(anSensorConfiguration_c& inSesnsorConfiguration);
- get influence nodes and directions of the grid
  void getInfluenceNodesNeuroFEM (utMatrix_t<double>& outInfluenceNodePositions,
  utMatrix_t<double>& outInfluenceNodeDirections);

FEM Surface Grid

Class Name: anSurfaceGridGeneratorNeuroFEM_c

This class can be used as grid generator which provides a surface grid as search space for the inverse source localsiation. The surface grid structure is read from a file.

How to create an surface grid generator?

anSurfaceGridGeneratorNeuroFEM_c( );

Important public interface functions

- generate and return grid:
  Grid_c& generateGrid();
- get influence nodes and directions of the grid:
  void getInfluenceNodesNeuroFEM (utMatrix_t<double>& outInfluenceNodePositions,
  utMatrix_t<double>& outInfluenceNodeDirections);

Regularizers

Truncated SVD regularizer

Class Name: anRegularizerTruncSVD_c

The singular value decomposition decomposes a matrix into three **orthogonal** matrices:

$$inMatrix = U \cdot W \cdot V^{T}$$

The matrix W is a diagonal matrix containing the singular values of *inMatrix*, while *U* and *V* consist of the left and right eigenvectors, respectively.
Small singular values (compared to the maximum singular value) of the matrix to be inverted are responsible for large values in the result and lead to a high sensitivity of the result to noise. To prevent this, small singular values are set to zero, resulting in a matrix W'. Subsequent the full matrix is reconstructed by multiplying the matrices:

$$outMatrix = U \cdot W' \cdot V^{T}$$

How to create a truncated SVD regularizer?

anRegularizerTruncSVD_c( );

There are no parameters.

Important public interface functions

- Pass the settings vector
  bool setSettings(const utVector_t<double>& inSettings);

  The parameter vector can have 3 members.
  The first gives the cutoff criterion: if it is 1, all singular values below a certain **absolute** value are discarded; in case it is 2, all the cutoff boundary is given as fraction of the maximum value (**relativ**).
- Perform singular value decomposition (SVD)
  int SVD(utMatrix_t<double>& a, utVector_t<double>& w, utMatrix_t<double>& v)
  The input matrix *a* is overwritten by the left eigenvectors.
- Perform regularization
  void run(utMatrix_t<double>& inMatrix,utMatrix_t<double>& outMatrix);

Tikhonov regularizer

Class Name: anRegularizerTikhonov_c

The condition of a matrix is changed by adding a constant to the diagonal elements of the matrix.

outMatrix = inMatrix + l *I*

l is determined by the quotient of the variance of the noise and the variance of the sources

$l = \frac{s_n^2}{s_j^2}$.

Both values can only be estimated. In this implementation of the Tikhonov regularizer this happens in the following way.

Noise variance : s_n^2

Computed from the signal variance and the signal-to-noise ratio (passed).

Source variance: s_j^2

$$s_j^2 = \frac{\sum_{i,j=0}^{ns-1,ns-1} inMatrix[i][j] \cdot \sum_{ii=0}^{nt-1} M[i][ii]M[i][ii] + s_n^2 \sum_{i,j=0}^{ns-1,ns-1} inMatrix[i][j]}{\sum_{i,j=0}^{ns-1,ns-1} inMatrix[i][j]^2},$$

$M = MeasuredData, ns = NumSensors, \; nt = NumTimeSteps$

How to create a Tikhonov regularizer?

anRegularizerTikhonov_c(const utMatrix_t<double>& inReferenceData, double inSNR);

Both parameters describe the measurement data (note, that this regularizer is specially designed for lead field matrices). *InSNR* describes the noise variance as fraction of the maximum signal amplitude.

Important public interface functions

- Perform regularization
  void run(utMatrix_t<double>& inMatrix,utMatrix_t<double>& outMatrix);

Weighters

Unary weighter

Class Name: anUnaryWeighter_c

$$w_i = 1, \qquad i = 0,..., NumberSources,$$

Generates a vector with all elements equal to one. This weighter does not have any influence on the results of an inverse solution. It can be used if a linear estimation procedure is used without spatial weighting of source nodes.

How to create an unary weighter?

anUnaryWeighter_c(anAbstractSimulatorEEGMEG_c& inSimulator);

The simulator is inherited from the abstract interface and not needed. It can be any dummy.

Important public interface functions

- Compute weighting vector

```
bool computeWeights(   Grid_c&                 inGrid,
                       utVector_t<double>&     WeigthingVector,
                       utMatrix_t<double>&     WeightingMatrix,
                       utMatrix_t<double>&     LeadFieldMatrix);
```

  Again, all arguments except the reference to the weighting vector and the grid are not used . The grid provides the number of nodes, thus the length of the weighting vector.

Lead field matrix normalization weighter

Class Name: anWeighterLeadfieldNormalization_c

Generally inverse source reconstructions methods using the L2-norm constrained prefer sources close to the sensors. This can be prevented using a normalization of columns of lead field matrix. This weighter creates a vector using following formulas for its elements performing a weighting of the columns of the lead field matrix.

$$w_i = \frac{NumSensors \cdot NumSources}{Sum \cdot \sum_{j=0}^{ns-1} \left| Leadfieldmatrix[\,j][i] \right|}, \qquad i = 0,..., NumSources,\ ns = NumSensors$$

$$Sum = \sum_{i=0}^{NumSources} \frac{NumSensors}{\sum_{j=1}^{ns} \left| Leadfieldmatrix[\,j][i] \right|}$$

How to create an lead field normalization weighter?

anLeadfieldNormalizationWeighter_c(anAbstractSimulatorEEGMEG_c& inSimulator);

The simulator is inherited from the abstract interface and not needed. It can be any dummy.

Important public interface functions

- Compute weighting vector

```
bool computeWeights(    Grid_c&                 inGrid,
                        utVector_t<double>&     WeigthingVector,
                        utMatrix_t<double>&     WeightingMatrix,
                        utMatrix_t<double>&     LeadFieldMatrix);
```

3.2 Abstract Class Interfaces: Continuous Parameter Space

The second group of inverse procedures uses a continuous parameter space as search space for their parameters. For the determination of the parameters non-linear optimization procedures are used, which in turn need a goal function. These goal functions are specific for different source model options. For a comparison between estimated and measured data goal functions need a simulator and criteria, which provide a metric how to compare estimated and measured data. An additional search volume restricts the search space of parameters. Fig. 3.2 gives an overview about the class structure for inverse methods using a continuous parameter space.

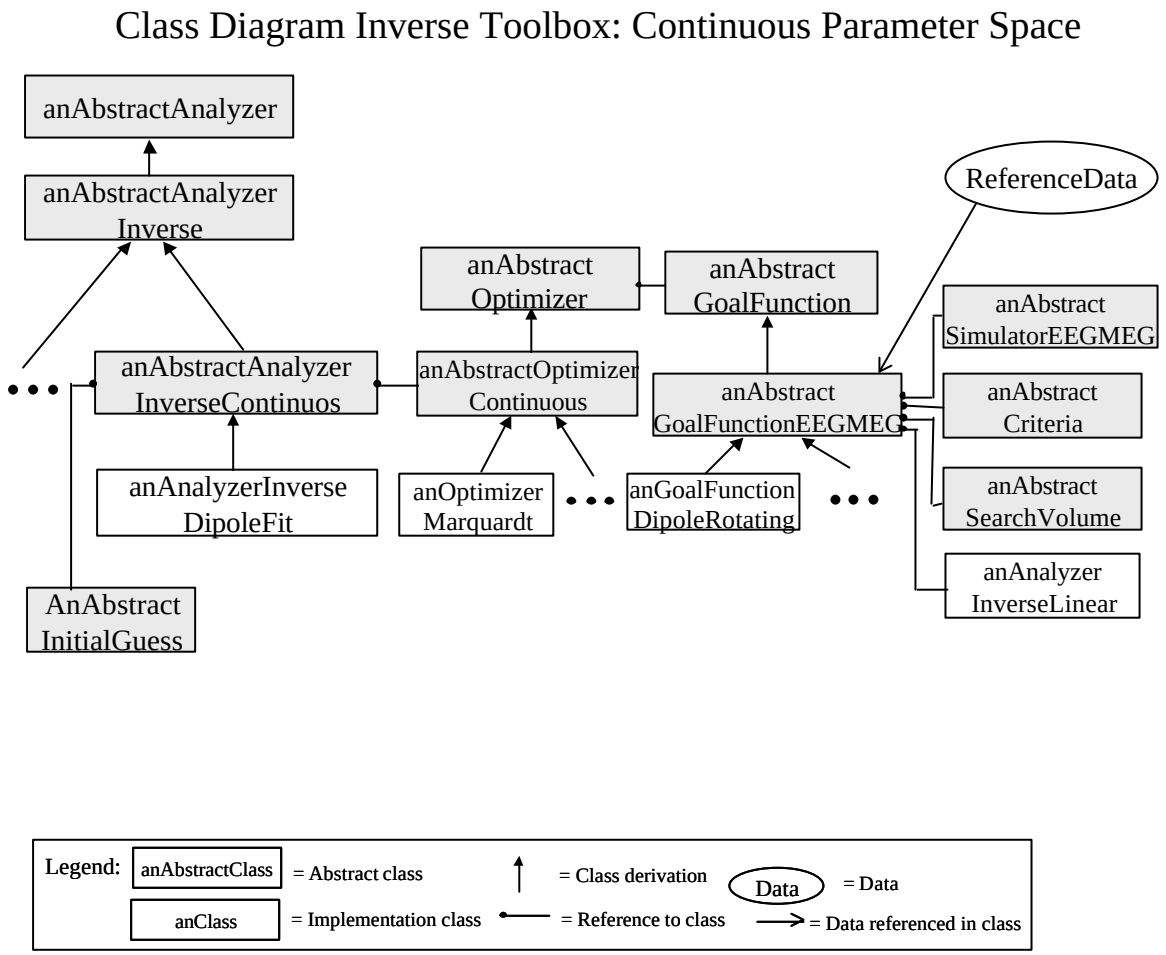


Fig. 3.2 Class structure for the implementation of methods using a discrete search space for their parameters.

Abstract Inverse Continuous Analyzer

Class Name: anAbstractAnalyzerInverseContinuous_c

Serves as base class for inverse methods having a continuous parameter space. It has a reference to an instance of anAbstractOptimizer_c and anAbstractInItialGuess_c.

Important public interface functions

- execute the method
  void run()

Abstract Optimizer

Class Name: anAbstractOptimizer_c

Serves as a base class for all kinds of optimization procedures. It has a reference to an instance of anAbstractGoalFunction_c. All the optimizer does is minimizing this goal function with respect to its parameters.

Important public interface functions

- find optimum of goal function
  virtual void run();

- get and set the settings vector. This vector is specific for the particular optimization procedure and therefore its members have a different meaning in each derived class.
  virtual bool setSettings(utVector_t<double> inSettings);
  virtual utVector_t<double>& getSettings();

- get and set the stopping criteria vector. The stopping criteria determine, when the optimization procedure considers the found parameter vectors as optimal. This vector is specific for the particular optimization procedure and therefore its members have a different meaning in each derived class.
  virtual bool setStopCriteria(utVector_t<double> inStopCriteria);
  virtual utVector_t<double>& getStopCriteria();

- retrieve the optimization quality vector. These values give information on quality criteria of the optimization, e.g. the goodness of fit. This vector is specific for the particular optimization procedure and therefore its members have a different meaning in each derived class.
  bool getOptimizeQuality(utVector_t<double>& outOptimizeQualityMeasure);

Abstract Optimizer Continuous

Class Name: anAbstractOptimizerContinuous_c

Serves a base class for all kind of optimization procedures having a continuous parameter space. This is the most important descendent of *anAbstractOptimizer_c.*

Important public interface functions

Apart from the functions already defined in its base class, the following additions are present:

- another run function to carry out the optimization, this time the initial parameters (initial guess) can be passed and the optimized parameters retrieved.
  virtual void run(utMatrix_t<double>& inParameter, utMatrix_t<double>& outParameter);

- carry out linear optimization. Many non-linear goal functions have implicit linear parameters (e.g. dipole strength), which need to be calculated after the non-linear iteration has finished.
  virtual void linearEstimation(utMatrix_t<double>& OptimizedParameters,
  utMatrix_t<double>& linearParameters);

Abstract Initial Guess

Class Name: anAbstractInitialGuess_c

Serves as base class for all kinds of initial guesses. Initial guesses are needed as start parameters for optimization procedures.

Important public interface functions

The class contains interface functions for automatically generating initial parameters as well as for setting them explicitely from outside.

- generate an initial guess
  virtual utMatrix_t<double> getInitialGuessParameters(int InitialGuessParamaters_height,
  int InitialGuessParamaters_length);
  The parameters give the dimension of the initial guess. The actual organization of the output matrix (which member stands for which parameter, e.g. dipole 1, position, x-coordinate) is determined in the derived classes.
- set initial guess from outside
  virtual bool setInitialGuessPositions(utMatrix_t<double>& inParameters);

Abstract Goal Function

Class Name: anAbstractGoalFunction_c

Serves as base class for all kind of goal functions. Goal functions are needed by optimization procedures. They form a very important concept, describing any kind of relationship that represents the degree of agreement between a hypothesis on the sources and the actual measurement.

Important public interface functions

- compute goal function value (goodness of fit) for passed set of parameters.
  virtual double computeGoalFunction(utMatrix_t<double>& inParameter);
- compute linear parameters of the goal function (e.g. dipole strengths), belonging to a certain set of non-linear parameters (e.g. positions).
  virtual void linearEstimation(utMatrix_t<double>& inParameter,
  utMatrix_t<double>& linearParameters);
- compute the Jacobi matrix, containing the derivatives of certain values (e.g. the differences of forward calculations at each sensor position) with respect to each parameter.
  virtual bool computeJacobiMatrix(utMatrix_t<double>& inParameters,
  utMatrix_t<double>& SqJacobi,
  utVector_t<double>& g);

- obtain additional goal function values, e.g. goodness of fit for each time step, etc.
  bool getAdditionalGoalFunctionValues(utVector_t<double>& outAdditionalValues);

Abstract Goal Function EEG/MEG

Class Name: anAbstractGoalFunctionEEGMEG_c

Serves as a base class for all kinds of goal functions used for inverse procedures for electrical and magnetic signals. It has references to instances of anAbstractSimulatorEEGMEG_c, which computes a forward solution. The forward solution is compared with the measured reference data. The anAbstractGoalFunctionEEGMEG_c has a further reference to an instance of anAbstractCriteria_c to define how simulated data and measured data are compared.

Important public interface functions

- the constructor is "protected", i.e. not directly accessible (only overloaded in derived classes). It is shown here anyway, because it makes clear, what ingredients are needed for all goal functions used for source localization from EEG and MEG.
  anAbstractGoalFunctionEEGMEG_c(const utMatrix_t<double>& inReferenceData,
  anAbstractSimulatorEEGMEG_c& inSimulator,
  anAbstractCriteria_c& inCriteria,
  anAbstractSearchVolume_c& inSearchVolume,
  anAnalyzerInverseLinear_c& inLinearEstimator);
  The goal function class uses a **simulator** to compute the forward solution for a certain source parameter set, it employs a **criterion** to compare it to the **reference data**. The **search volume** is used to compute extra penalties in case the positions of the sources are outside. Finally a **linear estimator** computes the linear parameters of the goal function.

- compute goal function value (goodness of fit) for passed set of parameters. This function passes the parameters to an *anAbstractSimulator*_c, then passes the obtained forward solution together with the measurements into an *anAbstractCriteria_c* object, which computes the goal function value.
  virtual double computeGoalFunction(utMatrix_t<double>& inParameter);

- compute projection matrix ($I - LL^{+}$). The norm of the product of projection matrix and reference data is normally used as goal function value (except for penalties)
  virtual void ProjectionMatrix(utMatrix_t<double>& inParameter,
  utMatrix_t<double>& outProjectionMatrix);

- compute the Jacobi matrix, containing the derivatives of certain values (e.g. the differences of forward calculations at each sensor position) with respect to each parameter.
  virtual bool computeJacobiMatrix(utMatrix_t<double>& inParameters,
  utMatrix_t<double>& SqJacobi,
  utVector_t<double>& g);

- compute a penalty for source leaving the search volume.
  double computePenaltySearchVolume(double& DistancetoSurface);

Abstract Criteria

Class Name: anAbstractCriteria_c

Serves as a base class for all kind of criteria. Criteria define a metric how data are compared.

Important public interface functions

- compute comparison criterion between two data sets
  virtual double computeCriteria(utMatrix_t<double>& Data1, cutMatrix_t<double>& Data2);

Abstract Search Volume

Class Name: anAbstractSearchVolume_c

Serves as a base class for all kinds of search volumes. Search volumes restrict the search space of parameters of a continuous parameter space. Methods allow to check whether parameters are inside the search space and to determine the distance of position parameters to the boundary of the search space. Points inside the search volume have negative distance values.

Important public interface functions

- determine if a parameter set is within the bounding volume (usually a position)
  virtual bool isInsideSearchVolume(const utVector_t<double>& inParameters);
- determine the distance of a parameter set (normally a position) to the boundary (where outside counts positive and inside negative)
  virtual double DistanceToSurface(const utVector_t<double>& inParameters);
- pass some settings to the class. The meaning of the members of the settings vector is specific for the particular descendent of this abstract class.
  virtual bool setSettings(const utVector_t<double>& inSettings);

3.4 Implemention Classes: Continuous Parameter Space

Analyzer for Dipole Fit:

Class Name: anAnalyzerInverseDipoleFit_c

Inverse method derived from anAbstractAnalyzerInverseContinuous_c to carry out non-linear dipole fit. The type of dipole fit (fixed, rotating moving) is determined by the goal function of the optimizer. The class has references to two other classes: (1) *anAbstractInitialGuess_c* determines the start values of the target parameters (e.g. the dipole positions), and (2) *anAbstractOptimizerContinuous_c* yields the method to optimize these parameters. The latter has in turn references to a goal function, the forward solution, etc.

How to create a dipole fit analyzer?

```
anAnalyzerInverseDipoleFit_c( int&                          NumDip,
                              anAbstractInitialGuess_c&     inInitialGuess,
                              anAbstractOptimizerContinuous_c& inOptimizer);
```

Important public interface functions

- perform dipole fit.
  void run();
- extract resulting parameter matrix
  bool getResult(utMatrix_t<double>& outParameters);
- set settings
  void setSettings(const utVector_t<double>& inSettings);
  The parameter vector contains parameters which can be transferred to the referenced *anAbstractOptimizerContinuous_c*. Thus it contains a concatenation of parameters to set the stopping criteria and other parameters of the optimizer. These vectors are specific for the

particular optimization procedure and therefore its members have a different meaning for different optimizers.

Initial Guess

Standard

Class Name: anInitialGuessStandard_c

Provides initial guess parameters for an optimization procedure. For dipole fit methods needing position parameters it provides dipole positions in the following way: For even numbers of dipoles, initial positions are placed symmetric in both hemispheres. For odd numbers an additional dipole is set on the z-axes. Dipole fit methods needing a additional start values for the direction are additionally provided with two start values describing the direction in polar coordinates.

Important public interface functions

- generate an initial guess
  utMatrix_t<double> getInitialGuessParameters(int InitialGuessParamaters_height , intInitialGuessParamaters_length);
  If *InitialGuessParamaters_height* is 3, the matrix which is returned contains three rows with x, y, z position parameters. If *InitialGuessParamaters_height* is 5, the matrix which is returned contains three rows with x, y, z position parameters and two additional rows containing direction parameters (polar coordinates). For other values of *InitialGuessParamaters_height* a matrix is returned with all parameters equal to 0.001.
- set initial guess from outside
  bool setInitialGuessPositions(utMatrix_t<double>& inPositions);
  Requires a matrix with 3 rows to set initial guess positions.

Best of Grid

Class Name: anInitialGuessBestofGrid_c

For a set of regular placed dipoles a goal function is computed. Dipole parameters of the dipole with the highest goal function value are then used as initial guess parameters. This procedure leads only to results if the number of dipoles is one. If the number of dipoles is greater then one, the same initial guess parameters are created as by using the anInitialGuessStandard_c.

How to create an initial guess: best of grid?

anInitialGuessBestofGrid_c(anAbstractGoalFunctionEEGMEG_c& inGoalFunction);

Paremeters:

inGoalFunction	Goal Function for EEG/MEG returns value, which has to be optimized by changing dipole positions. Penalties included in the goal function ensure that restrictions of the parameter space are fulfilled.

Important public interface functions

- generate an initial guess
  utMatrix_t<double> getInitialGuessParameters(int InitialGuessParamaters_height , intInitialGuessParamaters_length);
  If *InitialGuessParamaters_height* is 3, the matrix which is returned contains three rows with x,y,z

position parameters. Position parameters are determined as described above. If *InitialGuessParamaters_height* is 5, the matrix which is returned contains three rows with x,y,z position parameters and two additional rows containing direction parameters (polar coordinates). For other values of *InitialGuessParamaters_height* a matrix is returned with all parameters equal to 0.001.

- set initial guess from outside
  bool setInitialGuessPositions(utMatrix_t<double>& inPositions);
  Requires a matrix with 3 rows to set initial guess positions.

Optimizer

Marquardt

Class Name: anOptimizerMarquardt_c

For the optimization of nonlinear parameters the Levenberg Marquardt optimizer is realized [2],[3]. Stopping criteria of the algorithm, which can be set are: minimum descent of the goal function, minimum shift of the position between to iterations, minimum rotation between to iterations and maximum number of iterations. The absolute value of the goal function is not used as a criterion to stop the optimization procedure. Parameters, which influence the behaviour of the optimization algorithm are lambda, ny and a factor to increase lambda, after one unsuccessful iteration step. Both stopping criteria and parameters influencing the optimization process can be set, but values, which are proven to be satisfactory to determine parameters of dipoles, are used by default.

How to create an marquardt optimizer?

anOptimizerMarquardt_c(anAbstractGoalFunction_c& inGoalFunction);

Parameters:

inGoalFunction	Parameters have to be optimized w.r.t to a goal function. A goal function may also include penalties to limit the search space for parameters.

Important public interface functions

- perform parameter optimization
  run(utMatrix_t<double>& inParameter, utMatrix_t<double>& outParameter);
  Paramters: *inParameter* Start parameters for optimization procedure
  outParameter Resulting optimized parameters
- set optimization procedure settings
  bool *setSettings(utVector_t<double> inSettings);*
  The settings vector contains following items: initial lambda, factor to increase lambda, ny
- set default procedure settings (approved for source localization)
  bool setDefaultSettings();
- set stop criteria of optimization procedure
  bool setStopCriteria(utVector_t<double> inStopCriteria);
  The stop criteria vector contains following items: Goodness of fit (%) (not used as a stopping criterion !!), minimal descent of the goal function value between to optimization steps, maximum number of iterations, minimum position shift between two iterations, minimum rotation of the direction between two iterations

- set default stopping criteria (approved for source localization)
  bool setDefaultStopCriteria();
- compute additional parameters which can be determined by linear estimation
  void linearEstimation(utMatrix_t<double>& OptimizedParameters,
  utMatrix_t<double>& linearParameters);
  Parameters: *OptimizedParameters* Parameters determined by Marquardt optimizer
  linearParameters Resulting parameters of linear estimation procedure. The linear estimation procedure is defined inside the respective goal function.

Goalfunctions

Rotating Dipole Fit

Class Name: anGoalFunctionDipoleRotating_c

Serves as goal function for rotating dipole fit procedures. For all time steps one position is determined for each dipole by the non-linear optimization procedure. The class contains further a method to determine directions and magnitudes for each time step for each dipole by linear estimation.

The goal function of the rotating dipole fit can also be used for a moving dipole fit by using the anAnalyzerInverseDipoleFit_c with a rotating goal function feeding the method for each time step separately with measured data.

How to create a goal function for a rotating dipole fit ?

anGoalFunctionDipoleRotating_c(const utMatrix_t<double>& inReferenceData,
anAbstractSimulatorEEGMEG_c& inSimulator,
anAbstractCriteria_c& inCriteria,
anAbstractSearchVolume_c& inSearchVolume,
anAnalyzerInverseLinear_c& nLinearEstimator);

For a description of parameters have a look at an *anAbstractGoalFunctionEEGMEG_c.*

Important public interface functions

- perform computation of goal function value
  double computeGoalFunction(utMatrix_t<double>& inParameter);
  Paramters: *inParameter* Parameters, for which goal function value is computed.
- get number of parameters, which have to be determined by non-linear optimization procedure for a dipole (returns 3)
  GetNumberOfParametersPerDip()
- compute projection matrix *(specific for goal function type)*
  ProjectionMatrix(const utMatrix_t<double>& inParameter,
  utMatrix_t<double>& outProjectionMatrix);
- compute dipole parameters which can be determined by linear estimation (direction and magnitude)
  linearEstimation(utMatrix_t<double>& inParameter ,
  utMatrix_t<double>& linearParameters);
  Parameters: *inParameter* Parameters determined by non linear optimizer
  linearParameters Resulting parameters of linear estimation procedure.

Fixed Dipole Fit

Class Name: anGoalFunctionDipoleFixed_c

Serves as goal function for fixed dipole fit procedures. For all time steps one position and one direction is determined for each dipole by the non-linear optimization procedure. The class contains further a method to determine the magnitudes for each time step for each dipole by linear estimation.

How to create a goal function for a fixed dipole fit?

```
anGoalFunctionDipoleFixed_c( (   const utMatrix_t<double>&   inReferenceData,
                                 anAbstractSimulatorEEGMEG_c& inSimulator,
                                 anAbstractCriteria_c&        inCriteria,
                                 anAbstractSearchVolume_c&    inSearchVolume,
                                 anAnalyzerInverseLinear_c&   inLinearEstimator);
```

For a description of parameters have a look at an *anAbstractGoalFunctionEEGMEG_c.*

Important public interface functions

- perform computation of goal function value
  double computeGoalFunction(utMatrix_t<double>& inParameter);
  Paramters: *inParameter* Parameters, for which goal function value is computed.
- get number of parameters, which have to be determined by non-linear optimization procedure for a dipole (returns 5)
  GetNumberOfParametersPerDip()
- compute projection matrix (specific for goal function type)
  ProjectionMatrix(const utMatrix_t<double>& inParameter, utMatrix_t<double>& outProjectionMatrix);
- compute dipole parameters which can be determined by linear estimation (direction and magnitude)
  linearEstimation(utMatrix_t<double>& inParameter , utMatrix_t<double>& linearParameters);
  Parameters: *inParameter* Parameters determined by non linear optimizer
  linearParameters Resulting parameters of linear estimation procedure.

Criteria

Minimum Square Error

Class Name: anCriteriumMinimumSquareError_c

Determines the Euclidean norm between two arrays as return value for the criterion:

$$Value = \sum_{i,j=0}^{n-1,m-1} \left(data1[i][j] - data2[i][j] \right)^2$$

How to create a criterion: minimum square error?

anCriteriaMinimumSquareError_c();

Important public interface functions

- compute criteria :
  double computeCriteria(utMatrix_t<double>& inData1,
  const utMatrix_t<double>& inData2);

  Parameters:

inData1	Values of first data set
inData2	Values of second data set

Search Volumes

Infinite

Class Name: anSearchVolumeInfinite_c

Parameters are always inside of the search volume.

How to create a infinite search volume?

anSearchVolumeInfinite_c();

Important public interface functions

- check whether parameters are inside search volume (always returns true)
  bool isInsideSearchVolume(const utVector_t<double>& inParameters);
- get distance to surface (returns always -10E35)
  double DistanceToSurface(const utVector_t<double>& inParameters);

Sphere

Class Name: anSearchVolumeSphere_c

For points inside a sphere, which is determined by its center and its radius, the isInsideSearchVolume() method returns true. The DistanceToSurface() method returns the distance between a point and the surface of the sphere. The distance is negative for points inside of the sphere.

How to create a spherical search volume?

anSearchVolumeSphere_c(utVector_t<double> inCenter, double inRadius);

Parameters:

inCenter	Center of spherical search volume (x,y,z)
inRadius	Radius of spherical search volume

Important public interface functions

- check whether parameters are inside spherical search volume
  bool isInsideSearchVolume(const utVector_t<double>& inParameters);

- get distance to surface
  double DistanceToSurface(const utVector_t<double>& inParameters);
- set center of sphere
  bool setCenter(utVector_t<double>& inCenter);
- set radius of spheres
  bool setRadius(double& inRadius);
- set settings
  bool setSettings(const utVector_t<double>& inSettings);
  The settings vector contains the following items: radius, center x, center y, center z

Sphere with Boundaries

Class Name: anSearchVolumeWithBoundariesSpheres_c

A set of spheres around one point can be defined. It can be defined for each spacing between the spheres whether it belongs to the search volume or not. The isInsideSearchVolume() method returns true if a point lies between the surfaces of two spheres, for which the spacing is set as a valid search volume. The DistanceToSurface() method returns the smallest distance to the nearest surface of a bounding sphere.

How to create a search volume with several spherical boundaries?

anSearchVolumeWithBoundariesSpheres_c (utVector_t<double> in_Center,
utVector_t<double> in_Radii,
utVector_t<int> inValidSearchVolumes);

Parameters:

inCenter	Center of spherical search volume (x,y,z)
inRadii	Radii of spheres
inValidSearchVolumes	every spacing between to spheres can be set as a valid or invalid search volume by setting values to one or zero.

Important public interface functions

- check whether parameters are inside spherical search volume
  bool isInsideSearchVolume(const utVector_t<double>& inParameters);
- get distance to surface
  double DistanceToSurface(const utVector_t<double>& inParameters);
- set center of sphere
  bool setCenter(utVector_t<double>& inCenter);
- set radii of spheres
  bool setRadii(utVector_t<double>& inRadii);
- set spaces between spheres as valid or invalid search volumes
  bool setValidSearchBoundaries(utVector_t<int>& ininsideValidSearchBoundaries);

3.5 Simulator Classes

Abstract Class Interface

Class Name: anAbstractSimulator_c

The purpose of a simulator is to make a forward computation for a given set of parameters. For inverse methods this forward solution is used for a comparison to measured data. A computegainmatrix() method is provided to perform forward computations.

Important public interface functions

- compute gain matrix
  void computeGainMatrix(const utVector_t<double>& inParameters,
  utMatrix_t<double>& outSimData));
  inParameters contains parameters as input for the forward calculation. OutSimData contains the simulation results

Abstract Class Interface EEG/MEG

Class Name: anAbstractSimulatorEEGMEG_c

anAbstractSimulatorEEGMEG_c serves as a base class for all kind of simulators for EEG and MEG forward computations. It contains an overloaded computegainmatrix() method with positions and directions of sources as input parameters.

Important public interface functions

- compute gain matrix
  computeGainMatrix(const utVector_t<double>& inParameters,
  utMatrix_t<double>& outSimData))
  Paramter Description*:*
  inParameters contains parameters as input for the forward calculation.
  OutSimData contains the simulation results

- compute gain (or lead field) matrix
  computeGainMatrix(int NumberPoints,
  const utMatrix_t<double>& inPos,
  const utMatrix_t<double>& inDir,
  utMatrix_t<double>& outSimData)

  Parameter Description
  NumberPoints number of reconstruction points
  inPosition matrix with positions (number of columns = number of positions)
  inDir direction of dipoles (number of columns = number of positions)
  (see box below for interpretation)

Each column stands for one reconstruction point. If the matrix has only one column, this column applies to all reconstruction points.
The number of rows determines the direction information provided:

0 rows (matrix is 0 x 0 altogether)	at each reconstruction point, **3 dipoles** are created with standard orientations into x, y, and z directions
2 rows	at each reconstruction point, **1 dipole** is created, the directions are given in spherical coordinates (1^{st} row – α, 2^{nd} row – ϑ)
3 rows	at each reconstruction point, **1 dipole** is created, the directions are given in cartesian coordinates (1^{st} row – x, 2^{nd} row – y, 3^{rd} row – z)
4 rows	at each reconstruction point, **2 dipoles** are created, the directions are given in spherical coordinates (α_1, ϑ_1, α_2, ϑ_2)
6 rows	at each reconstruction point, **2 dipoles** are created, the directions are given in cartesian coordinates (x_1, y_1, z_1, x_2, y_2, z_2)
8 rows	at each reconstruction point, **3 dipoles** are created, the directions are given in spherical coordinates (α_1, ϑ_1, α_2, ϑ_2, α_3, ϑ_3)
9 rows	at each reconstruction point, **3 dipoles** are created, the directions are given in cartesian coordinates (x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3)

If the matrix has any other number of rows (1, 5, 7, or more than 9), the 0 row case is assumed. Some possibilities in the above table are grayed, these are not available at the moment (0 rows will be assumed), but will be in the future.

Spherical Head Model

Class Name: anSimulatorEEGSpheres_c

Implementation class of a simulator using a head model consisting of up to 4 concentric spheres defined by their center, radii and conductivities. These spheres can be used to represent the following tissues: scalp, skull, cerebro-spinal fluid (CSF), brain . An analytical solution based on Legrendre polynomials is used for calculation of EEG potentials [4].

How to create a simulator for a spherical head model?

```
anSimulatorEEGSpheres_c(  anSensorConfiguration_c&  inSensorconfiguration ,
                          utVector_t<double>&       inRadii,
                          utVector_t<double>&       inCenter,
                          utVector_t<double>&       inConductivities);
```

Parameters Description:

inSensorconfiguration	Desription of the Sensorconfiguration (EEG Electrodes)
inRadii	Radii of the spheres (starting from the outermost sphere. Up to 4 Radii can be set.
inCenter	Center of all spheres (x., y, z)
inConductivities	Conductivity values for the spheres

Important public interface functions

- perform a computation of the simulation results
  void computeGainMatrix(int NumberDip,
  const utMatrix_t<double>& inPos,
  const utMatrix_t<double>& inDir,
  utMatrix_t<double>& outSimData);

  For a parameter description see anAbstractSimulatorEEGMEG_c

- set and get radii of spheres
  bool setRadii(utVector_t<double>& inRadii);
  bool getRadii(utVector_t<double>& outRadii);
- set and get center of spheres
  bool setCenter(utVector_t<double>& inCenter);
  bool getCenter(utVector_t<double>& outCenter);
- set and get conductivity values
  bool setConductivities(utVector_t<double>& inConductivities);
  bool getConductivities(utVector_t<double>& outConductivities);
- get number of spheres
  int getNumberofSpheres();

Boundary Element Head Model

Class Name: anSimulatorEEGBEM_c

The simulator using a boundary element model (BEM) of the head takes the individual, non spherical shape of the main tissue boundaries within in the head into account: scalp-surface, inside and outside boundary of the skull, surface of the brain. Each of the boundaries is discretized into triangular elements. The determination of electric potentials of a segment of a boundary is described in [5].

How to create a simulator for a boundary element head model?

anSimulatorEEGBEM_c(anSensorConfiguration_c& inSensorconfiguration,
anAbstractGridGenerator_c& inBEM_GridGenerator);

Parameters Description:

inSensorconfiguration	Desription of the Sensorconfiguration (EEG Electrodes)
inBEM_GridGenerator	Grid containing boundary element description of the head model

Important: Constructor for BEM simulator will be changed for the final release of the inverse toolbox!!

Important public interface functions

- perform a computation of the simulation results
  void computeGainMatrix(int NumberDip,
  const utMatrix_t<double>& inPos,
  const utMatrix_t<double>& inDir,
  utMatrix_t<double>& outSimData);

  For a parameter description see anAbstractSimulatorEEGMEG_c

Finite Element Head Model

Class Name: anSimulatorEEGNeuroFEM_c

Computes the eeg-potential distribution on a finite element grid.

How to create a simulator for a finite element head model?

```
anSimulatorEEGNeuroFEM_c(anSensorConfiguration_c&          inSensorConfiguration,
                         anVolumeGridGeneratorNeuroFEM_c&  inVolumeGridGenerator);
```

Parameters Description:

inSensorconfiguration	Desription of the Sensorconfiguration (EEG Electrodes)
inVolumeGridGenerator	Grid generator of a finite element volume grid

Important public interface functions:

- perform a computation of the simulation results

```
void computeGainMatrix( int                          NumberDip,
                        const utMatrix_t<double>&    inPos,
                        const utMatrix_t<double>&    inDir,
                        utMatrix_t<double>&          outSimData);
```

For a parameter description see anAbstractSimulatorEEGMEG_c

3.6 Additional Classes

Sensor configuration

Class Name: anSensorConfiguration_c

Class serves for the description of sensors. Electrodes are described by a label and the position. Gradiometers for MEG measurements are provided with additional parameters describing the coil system and an integration model for the approximation of the magnetic flux.

How to create a sensor configuration?

```
anSensorConfiguration_c();
anSensorConfiguration_c(anAbstractDataHandler_c& inDataHandler);
```

Parameters Description:

InDataHandler	Link between measured data and input data for numerical methods. It provides the sensor configuration with sensor positions

Important public interface functions

- set sensor type (EEG, MEG, EEG+MEG)
  bool setSensorType(sensortype_e inSensorType);
- get sensor type
  sensortype_e getSensorType();

- get number of EEG electrodes
  int getNumberEEGElectrodes();
- add electrode
  bool addEEGElectrode(std::string inLabel,utVector_t<double> inPosition);
- remove electrode
  bool removeEEGElectrode(int Index);
- get electrode label
  std::string getElectrodeLabel(int Index);
- get electrode position
  utVector_t<double> getEEGElectrodePosition(int Index);
- change electrode position
  bool changeEEGElectrodePosition(int Index, utVector_t<double> inPosition);

Members variables for the description of MEG sensors are already available in the sensor configuration class, but methods to access these parameters are not implemented yet.

DataHandler

Class Name: anAbstractDataHandler_c

anAbstractDataHandler_c serves as a base class for all kind of data handlers. It serves as link between measured data with properties like physical units or sampling rate and input data for numerical methods. The implementation of a data handlers depends on properties of the operating system, file formats, etc.. Thus no concrete data handler is implemented in the inverse toolbox.

How to create a data handler?

anAbstractDataHandler_c();

Important public interface functions

- get an array containing data for a start point and a given interval length
  bool GetData(long lSample, short scSamples, utMatrix_t<double>& inData);
- get position and direction information of signal sources
  void GetSignalSources(utMatrix_t<float>& pos, utMatrix_t<float>& ori,utVector_t<bool>&);
- get number of samples in data set
  long GetSampleCount() cons;t
- get sampling frequency
  float GetSamplingFrequency() const;
- get a block of data, which contains in each layer data for one event, which is described in an eventlist
  bool GetDataForEventList(const std::vector<anAnalyzerAveragerEventParameters_c>& eventlist,
  utVector_t<int>& ZeroPos,
  utVector_t<int>& NumSamples,
  utVector_t<bool>& bEpochComplete,
  utBlock_t<double>& data):

Signal Processing

Measured data for inverse procedures like EEG and MEG have time dependent properties. To describe these properties and to perform data processing prior to inverse procedures the toolbox provides signal processing algorithms.

Data Windowing

Class Name: anAbstractDataWindow_c

Data windowing algorithms are used for algorithms working with a limited amount of data, to provide a smooth onset and fading of the input data for the subsequent signal processing. To be able to exchange windowing methods in a simple way, a abstract class interface is introduced.

Important public interface functions

- window data
  windowData1D(const utVector_t<double>& inputSamples,
  utVector_t<double>& outputSamples);
- perform inverse windowing
  windowDataInverse1D(const utVector_t<double>& inputSamples,
  utVector_t<double>& outputSamples);

Square Window

Class Name: anDataWindowSquare_c

Implementation class derived from anAbstractDataWindow_c. Windowing using a square window does nothing to the data. Windowed data are identical to the input data (see formula).

$w_j = 1,\ j = 0,...,N-1,\ N$ =Data length

How to create data window?

anDataWindowSquare_c();

Important public interface functions

- window data
  windowData1D(const utVector_t<double>& inputSamples,
  utVector_t<double>& outputSamples);
- perform inverse windowing
  windowDataInverse1D(const utVector_t<double>& inputSamples,
  utVector_t<double>& outputSamples);

Bartlett Window

Class Name: anDataWindowBartlett_c

Implementation class derived from anAbstractDataWindow_c. Using a Bartlett window data are multiplied with a triangle using the following formula.

$$w_j = 1 - \left| \frac{j - \frac{1}{2}N}{\frac{1}{2}N} \right|,\ j = 0,...,N-1,\ N = \text{Data length.}$$

How to create data window?

anDataWindowBartlett_c();

Important public interface functions

- window data
  windowData1D(const utVector_t<double>& inputSamples,
  utVector_t<double>& outputSamples);
- perform inverse windowing
  windowDataInverse1D(const utVector_t<double>& inputSamples,
  utVector_t<double>& outputSamples);

Welch Window

Class Name: anDataWindowWelch_c

Implementation class derived from anAbstractDataWindow_c. Using a welch window data are multiplied with a parabel using the following formula

$$w_j = 1 - \left(\frac{j - \frac{1}{2}N}{\frac{1}{2}N} \right)^2,\ j = 0,...,N-1,\ N = \text{Data length.}$$

How to create data window?

anDataWindowWelch_c();

Important public interface functions

- window data
  windowData1D(const utVector_t<double>& inputSamples,
  utVector_t<double>& outputSamples);
- perform inverse windowing
  windowDataInverse1D(const utVector_t<double>& inputSamples,
  utVector_t<double>& outputSamples);

Transformations

Transformations transfer data from one domain to a second domain.

Abstract Analyzer Transformation:

Class Name: anAbstractAnalyzerTransform_c

anAbstractAnalyzerTransform_c is derived from an anAbstractAnalyzer_c and serves as a base class for all kind of transformations. To provide the class with measured data it has a reference to an instance of an anAbstractDataHandler_c.

The constructor of the abstract analyzer for transformations cannot be accessed:

anAbstractAnalyzerTransform_c(anAbstractDataHandler_c& inDataHandler,
anAbstractDataWindow_c& inDataWindow);

Parameters Description:

inDataHandler	serves as an interface to measured data
inDataWindow	performs data windowing prior to transformation

Important public interface functions

- perform transformation (pure virtual method). Implementation resides in derived classes
  virtual void run() = 0;
- get result data
  bool getResult(utMatrix_t<double>& outParameters);
- acvtivate data windowing (by default windowing is activated)
  bool ActivateWindowing();
- deactivate windowing
  bool DeActivateWindowing();

Abstract Analyzer for invertibleTransformations:

Class Name: anAbstractAnalyzerTransformInvertible_c

All transformations, which are invertible, are derived from anAbstractAnalyzerTransformInvertible_c. It contains additional methods to run transformations and to retrieve the results of inverse transformations.

The constructor of the abstract analyzer for invertable transformations cannot be accessed:

anAbstractAnalyzerTransformInvertible_c(anAbstractDataHandler_c& inDataHandler,
anAbstractDataWindow_c& inDataWindow);

Parameters Description:

inDataHandler	serves as an interface to measured data
inDataWindow	performs data windowing prior to transformation

Important public interface functions

- perform transformation (pure virtual method). Implementation resides in derived classes.
  virtual void run() = 0;
- get result data
  bool getResult(utMatrix_t<double>& outParameters);
- perform inverse transformation (pure virtual method). Implementation resides in derived classes.
  virtual void runInverse(utMatrix_t<double>& inData) = 0;
- get result data of inverse transformation
  bool getInverseResult(utMatrix_t<double>& outParameters);
- acvtivate data windowing (by default windowing is activated)
  bool ActivateWindowing();
- deactivate windowing
  bool DeActivateWindowing();

Fourier Transformation

Class Name: anAnalyzerTransformFFT_c

Is derived from anAbstractAnalyzerTransformInvertible_c. The detemination of the results of the fast fourier transformation and the inverse fast fourier transformation is adapted from [3].

How to create fourier transformator?

anAnalyzerTransformFFT_c(anAbstractDataHandler_c& inDataHandler,
anAbstractDataWindow_c& inDataWindow);

Parameters Description:

inDataHandler	serves as an interface to measured data
inDataWindow	performs data windowing prior to transformation

Important public interface functions

- perform fourier transformation
  void run();
- get result data
  bool getResult(utMatrix_t<double>& outParameters);
- perform inverse fourier transformation
  void runInverse(utMatrix_t<double>& inData) ;
- get result data of inverse transformation
  bool getInverseResult(utMatrix_t<double>& outParameters);
- acvtivate data windowing (by default windowing is activated)
  bool ActivateWindowing();
- deactivate windowing
  bool DeActivateWindowing();
- set settings
  void setSettings(const anAnalyzerTransformFFTParameters_s& Settings);

How to create fourier transformation settings

```
anAnalyzerTransformFFTParameters_s( long                          nStartSample,
                                    long                          nNumberofSamples,
                                    const anFrequencyBands_c&     srcFrequencyBands,
                                    double                        dFrequencyResolution = 0.,
                                    anTransformationResultType_e  eResultType =
                                    RESULT_POWERSPECTRUM);
```

Following result types are available:

1. RESULT_UNKNOWN,
2. RESULT_POWERSPECTRUM,
3. RESULT_LOGSPECTRUM,
4. RESULT_DBSPECTRUM,
5. RESULT_AMPLITUDESPECTRUM,
6. RESULT_COMPLEXSPECTRUM.

Averager

Class Name: anAnalyzerAverager_c

Is derived from anAbstractAnalyzer_c. The method averages epochs of signal with respect to certain trigger points. The getResult() method provides a matrix containing the averaged signal (first set of columns), the number of epochs taken into account for each data point (on row) and the variance for each data point (third set of columns). To have access to the data the anAnalyzerAverager_c class has a reference to an instance of an anAbstractDataHandler_c and to a vector of the class anAnalyzerAveragerEventParameters_c, which is used to describe the starting points and intervals for the averaging procedure.

How to create an averager ?

```
anAnalyzerAverager_c(std::vector<anAnalyzerAveragerEventParameters_c>& EventList,
                     anAbstractDataHandler_c&                          inDataHandler,
                     bool                                              UseOnlyCompleteEpochs=true);
```

Parameters Description:

inDataHandler	serves as an interface to measured data
EventList	description of events (averager zero time, pre- and post stimulus intervals)
UseOnlyCompleteEpoch	flag, which can be set to false, if also incomplete intervals should be averaged

Important public interface functions

- perform averaging
  void run();
- get result data
  bool getResult(utMatrix_t<double>& outParameters);

Statistics

Abstract Class Interface

Class Name: anAbstractAnalyzerStatistic_c

anAbstractAnalyzerStatistic_c serves as a base class for all statistical procedures. It is derived from anAbstractAnalyzer_c. The constructor takes a matrix with data, which can be a set of random variables or stochastic processes.

The constructor of the abstract analyzer for statistical methods cannot be accessed:

anAbstractAnalyzerStatistic_c(const utMatrix_t<double>& inData);

Parameters Description:

inData	Each row of the matrix contains the values of one random variable or stochastic process.

Important public interface functions

- perform statistics (pure virtual method). Implementation resides in derived classes.
  virtual void run() = 0;
- get result data
  bool getResult(utMatrix_t<double>& outParameters);

Statistic Moments for one random variable

Class Name: anAnalyzerStatisticMomentsOneRandomVariable_c

The run() method of anAnalyzerStatisticMomentsOneRandomVariable_c determines the statistic moments of each row of the matrix set by the constructor. The statistic moments (mean, variance, standard deviation, skewness, kurtosis) are determined using following formulas:

Mean: $\bar{x} = \frac{1}{N}\sum_{j=1}^{N} x_j$

Variance: $Var(x_1 \ldots x_N) = \frac{1}{N-1}\sum_{j=1}^{N}(x_j - \bar{x})^2$

Standard deviation: $s\ (x_1 \ldots x_N) = \sqrt{Var(x_1 \ldots x_N)}$

Skewness: $Skew(x_1 \ldots x_N) = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{x_j - \bar{x}}{s}\right)^3$

Kurtosis: $Kurt(x_1 \ldots x_N) = \left(\frac{1}{N}\sum_{j=1}^{N}\left(\frac{x_j - \bar{x}}{s}\right)^4\right) - 3$

How to create an analyzer that determined the statistic moments of a random variable ?

anAnalyzerStatisticMomentsOneRandomVariable_c(const utMatrix_t<double>& inData,
anStatisticMomentsResultType_e eResultType =
MOMENTS_MEAN_VARIANCE_STANDARDDEV);

Parameters Description:

inData Each row of the matrix contains the values of one random variable or stochastic process.

eResultType Following options for the result type are available:
MOMENTS_UNKNOWN, MOMENTS_MEAN,
MOMENTS_MEAN_VARIANCE_STANDARDDEV,
MOMENTS_ALL_MOMENTS

Important public interface functions

- compute statistic moments
  void run() ;
- get result data
  bool getResult(utMatrix_t<double>& outParameters);

One row of the result matrix contains the statistic moments in the following order:

$$resultmatrix[i][0] = mean, resultmatrix[i][1] = \text{var}\,iance,$$
$$resultmatrix[i][2] = s\tan dard\_deviation, resultmatrix[i][3] = skewness,$$
$$resultmatrix[i][4] = kurtosis, i = 0,..., number\_of\_random\_\text{var}\,iables$$

Utilities

Vectors, matrices and 3 D data blocks are frequently used by algorithms of the inverse toolbox. Thus, the inverse toolbox includes facilities to use vectors, matrices and blocks in a secure and efficient way. Template classes are created to use the same functionality for different data types (int, long, float, double, etc.)

Vector

Class Name: utVector_t

Template class for vector operations. Includes secure allocation and access, as well as basic operations.

How to create a vector ?

utVector_t();
utVector_t(int,T=0); create empty vector of certain length

Important public interface functions

- allocation
  void allocate(int);
- length
  unsigned length() const;

- reference vector element
  T& operator[](int);
- assign vector to another vector
- *utVector_t<T>& operator=(const utVector_t<T>&);*
- fill with one value
  utVector_t<T>& operator=(const T&);
- add another vector (same size)
  utVector_t<T>& operator+=(const utVector_t<T>&);
  utVector_t<T> operator+(const utVector_t<T>&);
- subtract another vector (same size*)*
  utVector_t<T>& operator-=(const utVector_t<T>&);
  utVector_t<T> operator-(const utVector_t<T>&);
- scalar multiplication with another vector (same size)
  T operator(const utVector_t<T>&) const;*
- multiply with scalar
  utVector_t<T>& operator=(const T&);*
  utVector_t<T> operator(const T&) const;*
- equality
  bool operator==(const utVector_t<T>&) const;
- inequality
  bool operator!=(const utVector_t<T>&) const;
- euclidian norm
  double norm() const;
- sum of elements
  double sum() const;
- normalise to euclidian norm
  void normalise();
- concatenate two vectors
  utVector_t<T> operator|(const utVector_t<T>&) const;

Matrix

Class Name: utMatrix_t

Template class for matrix operations. Includes secure allocation and access, as well as basic operations.

How to create a matrix ?

utMatrix_t();
utMatrix_t(int,int,T=0); create empty matrix of given dimensions

Important public interface functions

- allocation
  void allocate(int,int);
- horizontal dimension
  unsigned length() const;
- vertical dimension
  unsigned height() const;
- get a copy of column
  utVector_t<T> column(int) const;
- copy vector into ith row of matrix
  void setrow(const utVector_t<T>&,int);

- copy vector into ith column of matrix
  void setcolumn(const utVector_t<T>&,int);
- remove the ith row of matrix
  void removerow(int);
- remove the ith column of matrix
  void removecolumn(int);
- assign matrix to another matrix
  utMatrix_t<T>& operator=(const utMatrix_t<T>&);
- fill with one value
  utMatrix_t<T>& operator=(const T&);
- add another matrix (same size)
  utMatrix_t<T>& operator+=(const utMatrix_t<T>&);
  utMatrix_t<T> operator+(const utMatrix_t<T>&);
- subtract another matrix (same size)
  utMatrix_t<T>& operator-=(const utMatrix_t<T>&);
  utMatrix_t<T> operator-(const utMatrix_t<T>&);
- multiplication of two matrices (special dimension requirements)
  utMatrix_t<T> operator(const utMatrix_t<T>&) const;*
- multiplication of a matrix with a diagonal matrix (values are stored in a vector) from the right side
  void multiplyrightdiagonal(const utVector_t<T>& in,utMatrix_t<T>& out);
- multiplication of a matrix with a diagonal matrix (values are stored in a vector) from the left side
  void multiplyleftdiagonal(const utVector_t<T>& in, utMatrix_t<T>& out);
- multiply with scalar
  utMatrix_t<T>& operator=(const T&);*
  utMatrix_t<T> operator(const T&) const;*
- equality
  bool operator==(const utMatrix_t<T>&) const;
- inequality
  bool operator!=(const utMatrix_t<T>&) const;
- frobenius norm
  double norm() const;
- determine transpose of a matrix
  void transpose();
- concatenate horizontally
  utMatrix_t<T> operator|(const utMatrix_t<T>&) const;
- concatenate vertically
  utMatrix_t<T> operator&(const utMatrix_t<T>&) const;

Block

Class Name: utBlock_t

Template class for matrix operations. Includes secure allocation and access, as well as basic operations.

How to create a block ?

utBlock_t();
utBlock_t(int,int,int,T=0); create empty block with given dimensions

Important public interface functions

- allocation
  void allocate(int,int,int);
- horizontal dimension
  unsigned length() const;
- vertical dimension
  unsigned height() const;
- third dimension
  unsigned depth() const;
- assign block to another block
  utBlock_t<T>& operator=(const utBlock_t<T>&);
- fill with one value
  utBlock_t<T>& operator=(const T&);
- add another block (same size)
  utBlock_t<T>& operator+=(const utBlock_t<T>&);
  utBlock_t<T> operator+(const utBlock_t<T>&);
- subtract another block (same size)
  utBlock_t<T>& operator-=(const utBlock_t<T>&);
  utBlock_t<T> operator-(const utBlock_t<T>&);
- multiply with scalar
  utBlock_t<T>& operator=(const T&);*
  utBlock_t<T> operator(const T&) const;*
- equality
  bool operator==(const utBlock_t<T>&) const;
- inequality
  bool operator!=(const utBlock_t<T>&) const;
- concatenate horizontally
  utBlock_t<T> operator|(const utBlock_t<T>&);
- concatenate vertically
  utBlock_t<T> operator&(const utBlock_t<T>&);
- concatenate depth
  utBlock_t<T> operator^(const utBlock_t<T>&);

4. Access to methods of the inverse toolbox

To give access to the classes of the inverse toolbox a multi layer interface is implemented. It provides a set of user scenarios on three different levels. A user scenario consists of a reasonable combinations of methods and contains a complete source modeling procedure.

In the center is the generic inverse toolbox providing inverse methods as a set of classes.

The first shell (UIF I) is realized as a class and provides a set of methods each of them representing one scenario. Data exchange is done by using pointers to data.

The next level (UIF II) gives access to the user scenarios with additional file input/output facilities.

The outer level (UIF III) will allow to construct a user scenario on a command line level.

The three user interfaces are isomorph. A method or command of the two outer levels consist besides of functions specific to its level of just one call of a method of the next deeper layer. The three user interfaces can be easily extended, as it is described in chapter 4.4

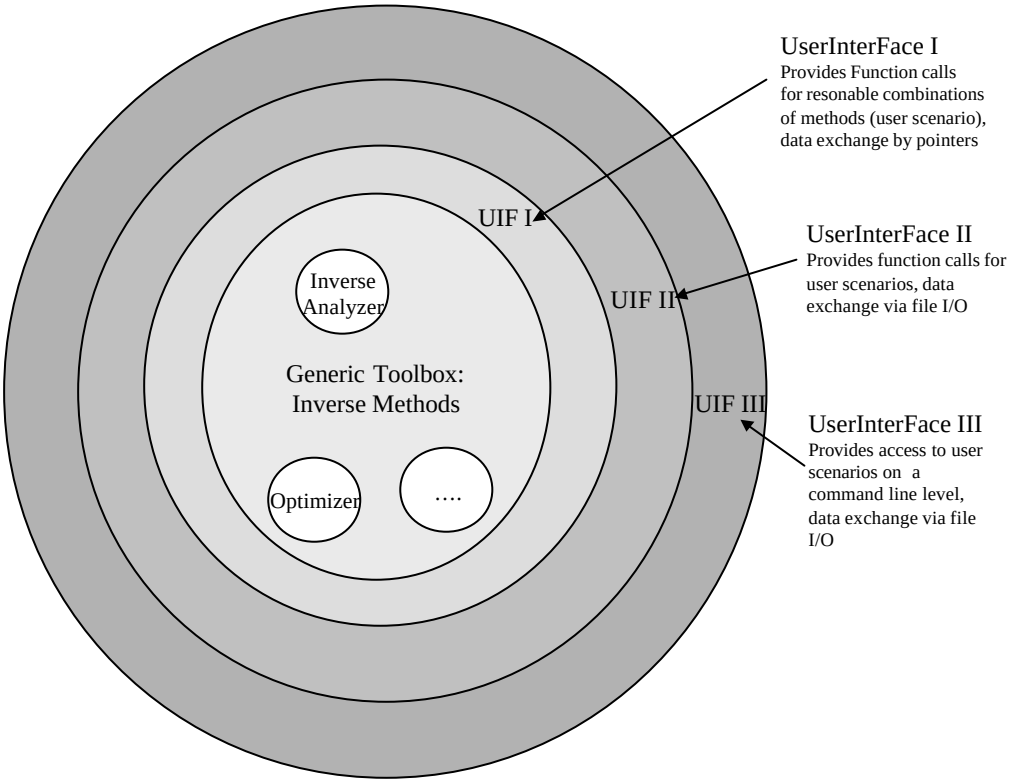


Fig.41 Three shell user interface

4.1 User interface I

User interface I shall give the opportunity to use the methods of the inverse toolbox in a complete application. User interface I is implemented as a class, which contains in the current release the following methods to provide user scenarios.

uif1_linear_estimation_oncortex(parameter_1, parameter_2, …, parameter_n)
uif1_linear_estimation_onbrainsurface(parameter_1, parameter_2, …, parameter_n)
uif1_linear_estimation_inbrainvolume(parameter_1, parameter_2, …, parameter_n)

uif1_movingdipolfit(parameter_1, parameter_2, …, parameter_n)
uif1_rotatingdipolfit(parameter_1, parameter_2, …, parameter_n)
uif1_fixeddipolfit(parameter_1, parameter_2, …, parameter_n)

Parameters that have to be set invoking a method can be divided into five groups:

1. Parameters, which supply the methods with grids, measured signals and measurement conditions like electrode positions.

2. The result of the source modeling.

3. Parameters, which are used as switches, to select between combinations of algorithms.

4. A class providing additional parameters to define detailed settings of algorithms.

5. For dipole fit methods, the number of dipoles can be set as an additional parameter.

Table 4.1 gives an overview about possible combinations of algorithms, which can be composed on the level of user interface I. An exact parameter definition for the methods is given in the appendix part A.

	Forward – type	Linear-inverse type	Non-linear inv. type	Inverter-type (Regu-larizer)	Sensor-type	Optimizer	Criteria	Search-volume	Intial Guess
Linear estima-tion	FEM	L2	-	TSVD Tikhonow	EEG	-	-	Cortex-grid Brain-surface-grid Brain-volume-grid	-
Dipole-fit	Spheres FEM	-	-	TSVD Tikhonow	EEG	Simplex Marquardt	Minumum Square Error	Entire Brain	Stand-ard

Table 4.1 Overview about combinations of algorithms, which are provided in the current release of the user interfaces

4.2 User interface II

User interface II provides the same methods as user interface I. User interface II provides methods which can read files generated providing the methods with data.

Each Method of user interface II corresponds to one method of user interface I. File input and output operations are wrapped around the call of the corresponding method of user interface I.

Files that can be read by the methods of UIF II contain:

reference data	(Vista file format, ASCII file format, ASA file format)
sensor configuration	(Vista file format, ASA file format)
grids	(NeuroFEM)
parameter file	(format description: see appendix B)

The objective of the parameter file is to provide additional information to set detailed properties of methods of the inverse toolbox.

The output of results of the inverse toolbox are written into a Vista file.

Names of methods of user interface II start with “uif2”. An exact description of the parameters of the methods of UIF II is available in the appendix part A.

4.3 User interface III

The third user interface level provides access to the methods of the inverse toolbox on a command line level. Commands are invoked with a set of arguments, to specify the execution of the command. User interface III provides the same set of inverse methods as the above described user interfaces. Commands of user interface III have identical properties as methods of the inner levels by calling a method of UIF II which in turn calls a method of UIF I.

Commands of user interface III look like:

uif3 argument1 argument2 argument3

Arguments for commands define the type of source modeling, filenames of input and output files and switches, to select between combinations of algorithms. Filenames must be specified with a prefix for the filetype. Following specifiers are used in UIF3 to determine a file type:

refdata=
searchspacegridfile=
headgridfile=
leadfieldfile=
sensorfile=
resultfile=
parameterfile=

Switches defining details of an inverse procedure and file names can be set in an arbitrary order. It is insured that only reasonable combinations of algorithms can be selected and if possible default parameters are set, when parameters are omitted.

For dipole fit methods the number of dipoles can be set by the following argument:

ndip=NumberofDipoles

As for the preceding user interfaces exact definitions of the commands are given in part A of the appendix.

4.4 Adding user scenarios

Adding new methods to the inverse toolbox or the wish to create new combinations of already implemented methods evoke the need of creating new user scenarios. This can be done by combining methods of the inverse toolbox to a new user scenarios. This chapter shall give a brief guideline to implement a new scenario for the three interface levels.

User interface I is defined as a class. Thus in the corresponding include file a new method has to be added. To provide an example, a part of the class definition can be found in the appendix part C. The

method must have arguments specifying input and output parameters as for example reference data, sensor configurations and result data. Additional switches are possible to allow a greater variability of combinations of methods. Inside the implementation of the methods, a constructor for all classes which are necessary for the determination of the source parameters has to be invoked. Then the run() method for the parameter determination can be called, followed by a call of getResult(), which delivers a matrix with the estimated source parameters as it is shown in the following example.

```
// prepare method

// initialize simulator
anSimulatorEEGSpheres_c        sim(sensorconfig,inRadii,Center,inConductivities);

// Initialize searcg volume
anSearchVolumeSphere_c         SearchVolume(Center, inRadii[2]);

// initialize weighter
anUnaryWeighter_c              weighter(sim);

// initialize regulatizer
anRegularizerTruncSVD_c        regularizer;

// initialize criterium
anCriteriaMinimumSquareError_c squareerror;

// initialize initial guess
anInitialGuessStandard_c       initialguess;

// initialize grid generator
anGridGeneratorSingleDipoles_c dipolegrid;

// initialize linear estimator
anAnalyzerInverseLinear_c linearestimator(refdata,sim,dipolegrid,weighter,regularizer);

// initialize goal function
anGoalFunctionDipoleRotating_c goalfunction(refdata,sim,squareerror,searchsolume,
                                            linearestimator);

// initialize optimizer
anOptimizerMarquardt_c         optimizer(goalfunction);

// inverse methos
anAnalyzerInverseDipoleFit_c   method(numdip,initialguess,optimizer);

// run dipole fit and get resultmatrix
utMatrix_t<double> resultmatrix;
method.run();
method.getResult(resultmatrix);
```

User interface II is also realized as a class. Methods of user interface II create data matrices and classes from files before calling the method of user interface I. The results can be written in a file subsequent to the call of the method of UIF I. An example is given inside appendix C.

User Interface III uses the argument argc (argument count) and the array of arguments argv (argument value) of the main function of a program. Thus the values of argv can be used to identify the name of the scenario and to define names of input and output files as well as switches. To ensure that the array argv contains reasonable values, their contents should be checked before calling a method of user interface II.

4.5 Interaction with NeuroFEM

As a basis for the development of the NeuroFEM-software, the software package CAUCHY'97, described under
http://www.rwth-aachen.de/neurologie/Ww/Neurologie/cauchy/CauchyFunctionality.htm and in [7, 11, 13] was taken and strongly redesigned..

A C++ class-structured software replaces old FORTRAN77 CAUCHY'97 kernel routines and enables the development of the SIMBIO-software on parallel platforms. The NeuroFEM-simulator has been derived from an abstract simulator class. A hierarchic class structure is used to reduce the number of interfaces and to keep them clean of implementation details. The toolbox provides abstract class interfaces for grid generators, forward simulators using a discrete and continous search space. NeuroFEM tools are interfaced to classes that are derived from these abstract classes.

The integrated class structure allows future comparisons with boundary-element-based forward simulations and analytical series expension formulas for spherical shell geometries, which both are also derived from the abstract simulator-class. The goal is an influence-study of tissue anisotropy on the various inverse algorithms of the ST4.1 toolbox. A dynamical memory management has been introduced throughout NeuroFEM, which replaces the former static allocations and enables a user-friendly application on distributed memory platforms. A first test was carried out where the NeuroFEM-simulator was used as forward simulator within an inverse dipole fit method.

In EEG/MEG-source localisation, the source is usually modelled as a mathematical equivalent current dipole, i.e. a current source and a sink which are infinitively close together in the human cortical layer. This point-like equivalent current dipole has been shown to be an adequate model for the synchronous polarisation of a cortical surface of about $30mm^2$. The point-like source directly leads to a singularity in the related potential which has to be treated numerically. One possibility is the "blurred dipole" where current monopoles are placed at neighboring FE-mesh nodes around the dipole location such that the resultant moment matches that of the mathematical dipole [7]. Another possibility is the subtraction method where the "singularity-potential" for a mathematical dipole in an unbounded homogeneous conductor is calculated analytically and the correction is carried out numerically on the realistic geometry ([12 ,8]). The correction is calculated with the FE method so that the sum of "singularity-" and "correction-", the "total-potential", obeys the charge continuity law within the head and the Neumann boundary conditions at the surface. The subtraction method and the node-shift (ns) mesh generation approach (see ST1.2. report) have been validated in a 4-layer sphere model where a spherical harmonics series expansion of the dipole potential can be derived for anisotropically conducting layers [9, 10]. To validate the subtraction method, we assumed the following isotropic conductivities in the 4 layer model: 0.33, 0.0049, 1.0 and 0.34 S/m. Together with nodeshifted FE-mesh, the magnification error (optimum 1) was 1.053 and the relative difference measure (optimum 0) 0.023 for 6 electrodes at all extreme sphere surface positions.
For tight coupling between ST4.1 and WP3 in the case of a continuous parameter space for the inverse reconstruction described here, the NeuroFEM tool was extended to provide a source simulation also for dipoles which are not on the discrete grid, but in the continuous space. The simulator is able to simulate several dipoles in the continuous space in the same time. In the case of a discrete parameter space, the coupling between WP3 and ST4.1 is on a simple file level or via close coupling, the leadfield matrix can also be directly handed to the inverse tools if the application runs on the same computer. The actual NeuroFEM tool is integrated in the ST4.1 software release.

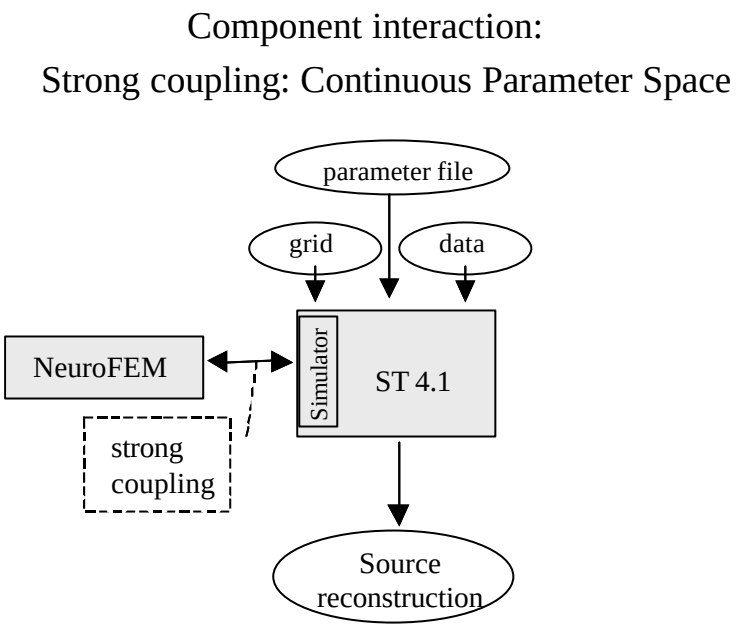


Fig. 4.2 Coupling between NeuroFEM and other parts of the inverse toolbox ST4.1. Methods of ST4.1 directly call NeuroFEM algorithms (strong cupling).

The inverse toolbox provides accurate modeling of the various tissues and anisotropy through individualized high resolution finite element modeling. This results in large, sparse linear equation system with many different right-hand-sides (sources). Therefore, the use of appropriate fast solvers is necessary with regard to the solution time and applicability for inverse source localization.

Preconditioned Krylov-subspace-methods are among the most attractive iterative methods. The FEM package NeuroFEM, which is developed within this framework provides these up to date modelling and solution techniques for de FE-simulation. Within this developpment compare incomplete threshold-factorization- with multigrid- preconditioners, the latter is known to be an optimal method with respect to the operation count and memory. Since the geometric construction of a grid-hierarchy is difficult (we only have tensor measurements for the finest level), we use a pure algebraic multigrid (AMG) preconditioner. [6].

Solver Comparison

The performance tests with ILDLT- and AMG-preconditioning, both taking anisotropy into account, showed no remarkable difference in the solver times. The calculation times of the Jakobi-preconditioned Krylov-solvers were up to a factor 1.25 longer for the anisotropic models, probably because of less-clustered eigenvalue spectra. Up to a relative L_2-residual of 10^{-3}, the differences between the ILDLT- and the AMG-preconditioned Krylov methods are minimal. From 10^{-5} up to 10^{-10}, the AMG-preconditioned CG method is the fastest in all tested cases and a factor 2 to 3 times faster than the best-tuned ILDLT-preconditioned Krylov method. Because of the loss of about 3 digits for the potential from the source to the electrodes, the interesting residuals begin at about 10^{-5}.

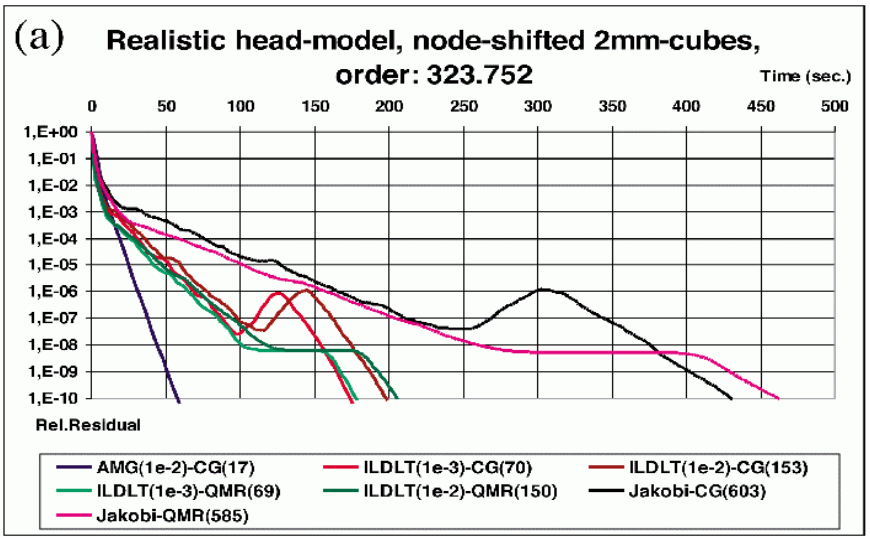


Fig 4.3: Comparison of different preconditioners for CG- and QMR-method : Realistic head model, rel. residual L2-norm-accuracy up to 10^{-10}

With regard to the inverse problem, acceptable calculation times can only be reached through a parallelization of the presented solvers.

4.6 Specification of graphical user interface

Commands defined on the level of user interface III can be invoked from a graphical user interface to provide an easy access to the methods of the inverse toolbox. The following paragraphs give a guideline for the development of a user interface for the methods realized in the current release.

The purpose of the graphical user interface is to specify the inverse method. The selection of files (reference data, search space grid, head grid, leadfield matrix, sensor configuration, additional parameters) and checking of their existence should be done by the integrated SimBio environment without the need of interaction. Thus, there is no need for file selections dialogues in the graphical user interface.

The interface should have two levels to specify the method. On the first level a pre selection of the method should be available to define main properties of the inverse method. The second level allows an refinement of the properties of the method. On this level default values are provided wherever possible.

Methods that can be selected on the first level are of 2 different categories. These categories and their subtypes are shown in table 4.2. together with the respective command line argument. On the first level one type of method has to be chosen exclusively.

Main Category	Subcategory	Command line argument
Linear Estimation	On Cortex	uif3_linear_estimation_oncortex
	On Brain Surface	uif3_linear_estimation_onbrainsurface
	In Brain Volume	uif3_linear_estimation_inbrainvolume
Dipole Fit	Moving Dipole	uif3_movingdipolfit
	Rotating Dipole	uif3_rotatingdipolfit
	Fixed Dipole	uif3_fixeddipolfit

Table 4.2 Methods accessible via the graphic user interface: Main category, subcategory, command line argument.

On the second level arguments allow further selections to specify the method. They have to be set exclusively. For each main category of methods these arguments are identical.

Categories of arguments and argument values for linear estimation methods are given in table 4.3

Linear Estimation	Command line argument
Head Model	FEM
Inverse Type	**L2**
Inverter Type	**TruncatedSVD**
	Tikhonow

Table 4.3 Categories and values for the refined definition of linear estimation methods (default values are printed in a bold font).

Continuous dipole fit methods have the number of dipoles as the second command line argument. Following arguments further specify the method (table 4.4)

Dipole Fit	Command line argument
Head Model	Sphere
	FEM
Inverter Type	**TruncatedSVD**
	Tikhonow
Optimizer Type	**Marquardt**
Criteria Type	**MinimumSquareError**
Search Volume Type	**InEntireBrain**
Intial Guess Type	**Standard**

Table 4.4 Categories and possible values for the refined definition of dipole fit methods (default values are printed in a bold font).

Subsequent to the selection of a method and their possible refined definition a button should be available to start the inverse calculations by sending a command according to user interface III with the argument values specified by the user. Following example presents a possible command that can be started by the graphical user interface:

```
uif3 uif3_movingdipolefit refdatafile=EEG.v headgridfile=head.v
sensorfile=10_20.v parametefile=spheres.par resultfile=moving.v ndip=1
Sphere Marquardt MinimumSquareError TruncatedSVD InEntireBrain
Standard
```

4.7 Error reporting

The command line application of UIF3 returns a zero by a successful execution and –1 if the execution failed. To enable a more sophisticated error reporting error messages are appended to the parameter file in case of errors during the inverse procedure. If no parameter file is specified by the command line arguments, a file named “error.par” will be created for error reporting. Error messages consist as shown in the example of following items: error code, error type, origin of the error, error description.

```
[Errors]
code= 02201
type= fatal
origin= NeuroFemSimulatedAnealing
description= No convergence!
```

These error messages should be read from the parameter file and displayed to the user via the graphical user interface.

5. Documentation

The release contains an documentation using HTML. The documentation includes:

1. an overview about implemented methods containing references to the literature,
2. a graphical representation of the class structure with links to the class definitions,
3. a complete class list with links to the class definitions,
4. a cookbook, how to work with the class interfaces,
5. a description of the user interfaces including a complete parameter list.

6. Directory Structure

Figure 6.1 shows the directory organization of the current release of the inverse toolbox. All directories are subdirectories of simbio_ipm_ver2001_04_30.

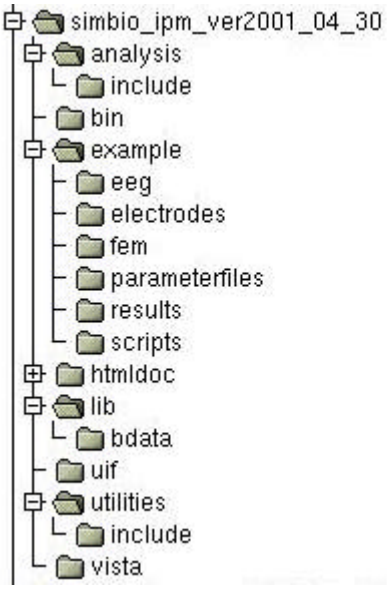


Fig. 6.1 Directory structure of st4.1 preliminary release “simbio_ipm_ver2001_04_30

An Overview over the contents of the directories is given in table 6.1.

Directory Name	Description
analysis/include	class definitions of inverse toolbox
Bin	binary directory: contains binary for **uif3**
example/eeg	contains an example EEG file
example/electrodes	contains example electrode files
example/fem	contains example FEM head model and scripts for dipole fit methods using a FEM head model

	(must be started in this directory !!)
example/results	contains example result files
example/scripts	contains example script files
htmldoc	contains html documentation
Lib	contains libraries of the inverse toolbox, NeuroFEM, Fortran libraries, and a vista library
Uif	contains source and include files of user interfaces. Additionally a Makefile is provided
Utilities/include	class definitions of template classes for vectors, matrices and blocks.
Vista	contains source and include files for vista file i/o

Tab. 6.1 Description of the directories delivered with the preliminary release of the inverse toolbox of ST 4.1.

7. Examples

a) Contiuous Parameter Space: Dipole Fit

Fig. 7.1 shows on the left side an averaged spike signal of a patient suffering from epilepsy. Spikes were aligned w.r.t to the maximum peak amplitude for the averaging procedure. Vertical blue lines indicate time steps for which a rotating dipole fit is performed using a Marquardt optimizer to determine non linear dipole parameters (position and direction) and linear estimation to compute the magnitudes of the dipoles. The example shows a propagation of the sources from lateral to central brain structures during the time course of the spike.

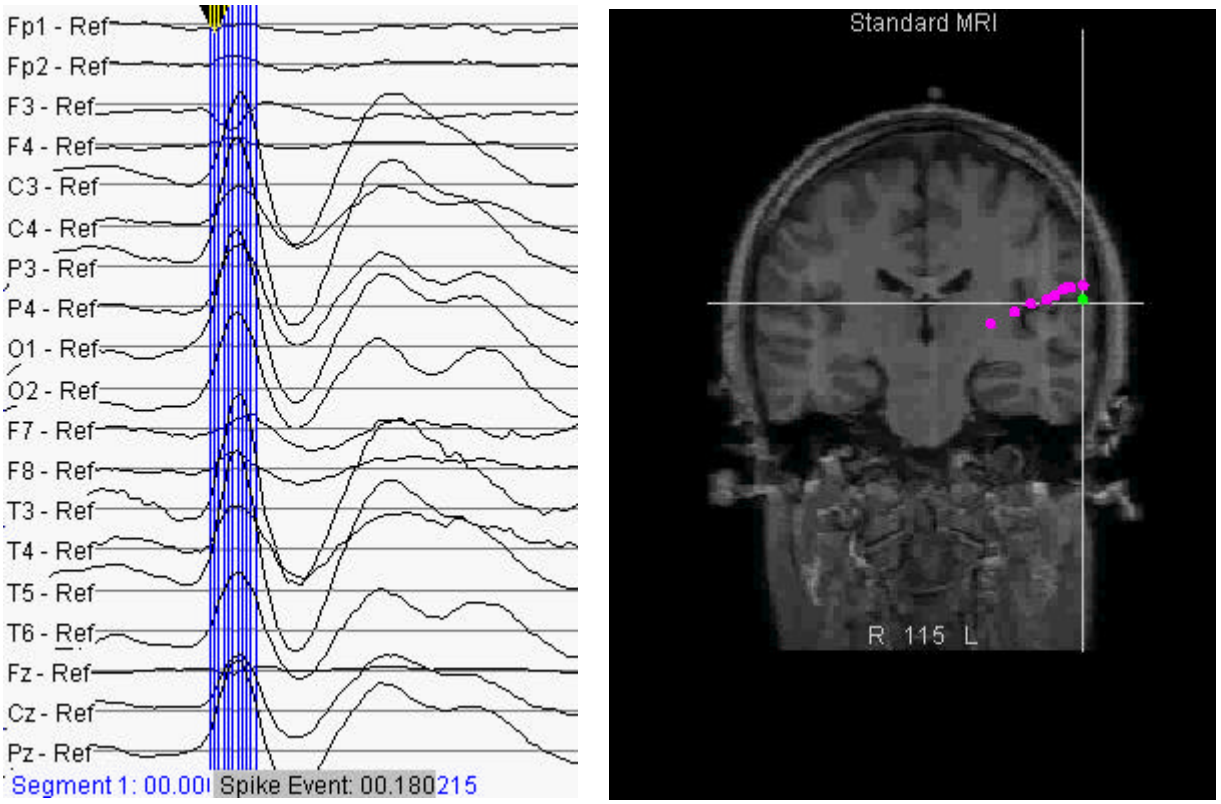


Fig 7.1 Averaged spike of a patient suffering from epilepsy and reconstructed dipole positions for single time steps indicated by blue vertical lines.

b) Discrete Parameter Space: Linear Estimation

Fig 7.2 shows simulation results for an artificial radial source. The search space is defined by source positions regularly distributed on a sphere, with a predefined direction normal to the surface of the sphere. Source magnitudes are determined by linear estimation. The results on the left side were determined without weighting, the results on the right side were computed using a leadfield columns normalization weighting procedure. The smoother and more widespread distribution of the source magnitudes calculated by using the weighting procedure is obvious. This result is inline with the objective to remove the preference of source positions near to the sensors.

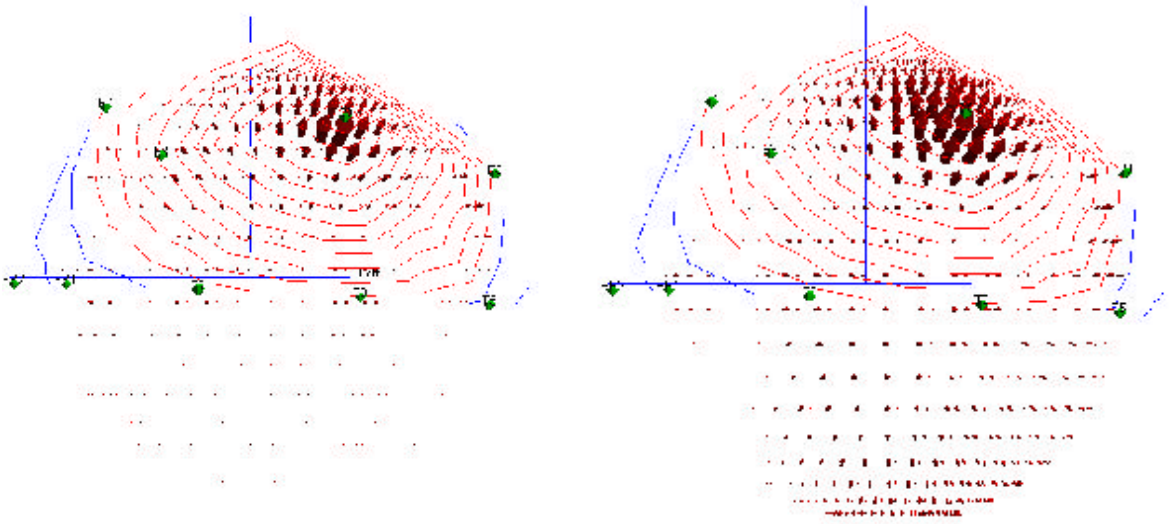


Fig. 7.2 Linear Estimation: Source reconstruction of a radial source by linear estimation. Grid points are regularly distributed on the surface of a sphere with a predefined direction normal to the surface of the sphere. On the left side no weighting is used, on the right side a procedure, which normalizes the columns of the leadfield matrix, is used, thus, leading to a more widespread distribution of sources.

c) NeuroFem Simulator

Two models:
The isotropic conductivity values of the 5-tissue realistic head model were chosen as follows (in S/m): skin 0.33, skull 0.0042, CSF 1.0, grey matter 0.33 and white matter 0.14. Isotropic (Rea-cube) and node-shifted (Rea-ns-cube) 2 mm cube meshes were generated. A 4-layer sphere model (cond. 0.33, 0.0042, 1.0 and 0.33) was constructed and meshed with 2 mm isotropic and 2 mm node-shifted cube elements. A radial:tangential anisotropy of 1:10 was assigned for the "skull" layer. A geometry-based mesh generator was used to generate a tetrahedral mesh for the isotropic and anisotropic 4-layer-sphere model. The different FEM equation solvers were tested on this model.

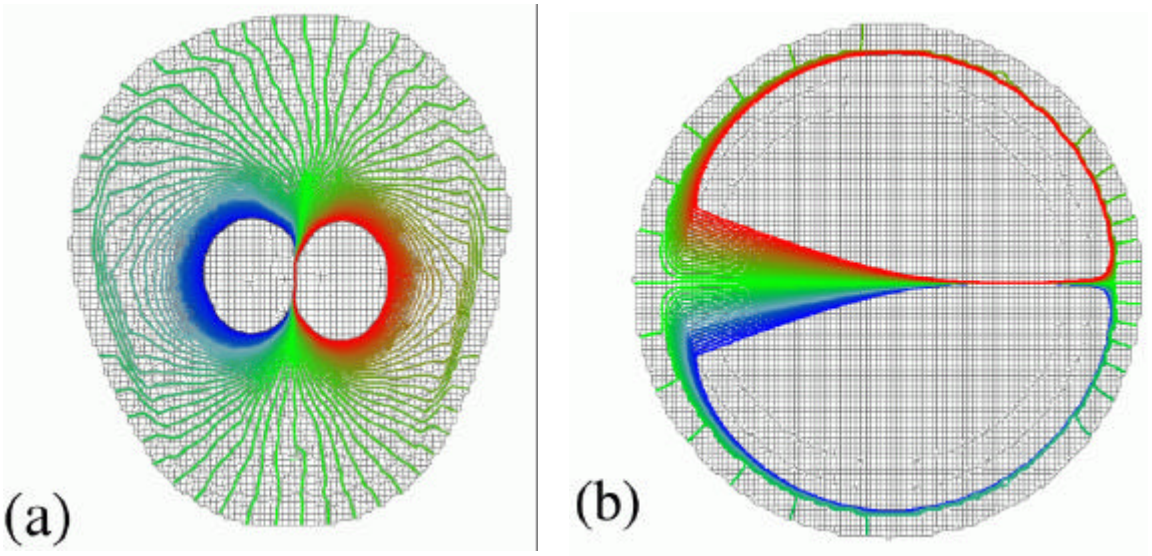


Figure 7.3 a) Realistic 5-tissue-model: Isopotential-lines from -2 V to 2 V on a coronal layer of the node-shifted FE-mesh. b) Anisotropic 4-layer sphere model: Isopotential-lines from -0.1 V to 0.1 V on a layer of the node-shifted FE-mesh.

References:

[1] D4.1a: Inverse Problem Methodology Design Report, (http://www.ccrl-nece.technopark.gmd.de/simbio/deliverables.html), 2000

[2] Marquardt D. An algorithm for least squares estimation of nonlinear parameters. SIAM J. Appl. Math., 1963, 11:431-441

[3] Press W.H., Flannery B., Teukolsky S.A., Vetterling, W.T. Numerical Recipes, The Art of Sceintific Computing. Cambridge University Press, 1993

[4] de Munck J. A mathematical and physical interpretation of the electromagnetic field of the brain. PhD thesis, University of Amsterdam, The Netherlands, 1989

[5] Zanow F. Realistically shaped models of the head and their application to EEG and MEG. Ph.D. thesis, University of Twente Enschede, The Netherlands, ISBN 9036509416, 1997

[6] Wolters C.H.,Reitzinger S.,Basermann A.,Burkhardt S., Hartmann U.,Kruggel F. & Anwander A. Improved tissue modelling and fast solver methods for high resolution FE-modelling in EEG/MEG-source localization**.** In J.Nenonen, R.J. Ilmoniemi & T. Katila (Eds.),Proceedings of the 12th International Conference on Biomagnetism (Biomag 2000),Helsinki University of Technology,Espoo,Finland (in press). (Web-address:http://biomag2000.hut.fi/papers/162a.pdf).

[7] Buchner, H., Knoll, G., Fuchs, M., Rienacker, A., Beckmann, R., Wagner, M., Silny, J., and Pesch, J. (1997) Inverse localization of electric dipole current sources in finite element models of the human head. Electroencephalography and Clinical Neurophysiology, 102:267-278

[8] Marin, G., Guerin, C., Baillet, S., Garnero, L. and Meunier, G. (1998) Influence of skull anisotropy for the forward and inverse problem in EEG: Simulation studies using FEM on realistic head models, Human Brain Mapping, 6, 250-269.

[9] de Munck, J.C. (1988) The potential distribution in a layered anisotropic spheroidal volume conductor, J.Appl.Phys., 64 (2), 464-470.

[10] de Munck, J.C. and Peters, M. (1993), A fast method to compute the potential in the multi sphere model, IEEE Trans. Biomed. Eng., 40(11), 1166-1174.

[11] Rienaecker, A., Buchner, H. and Knoll, G., Comparison of Regularized Inverse EEG Analyses in Finite Element Models of the Individual Anatomy, In: Witte, H., Zwiener, U., Schack, B. and Doering, A. (eds.), Quantitative and Topological EEG and MEG Analysis, Druckhaus Mayer Verlag GmbH Jena-Erlangen,1997, pp.459 463.

[12] von Rango, J., Schlitt, H.A., Halling, H. and Mueller-Gaertner, H.-W. (1996) Finite Integration Techniques for the MEG Forward Problem. , In: Witte, H., Zwiener, U., Schack, B. and Doering, A. (eds.), Quantitative and Topological EEG and MEG Analysis, Druckhaus Mayer Verlag GmbH Jena-Erlangen,1997, pp.336-338.

[13] Wolters, C.H., Beckmann, R.F., Rienaecker, A. and Buchner, (1999), Comparing regularised and non-regularised nonlinear dipole fit methods: A study in a simulated sulcus structure, Brain Topography, 1999, Vol.12 (1), 3-18.

Appendix A: Commands and parameters for user interfaces

List of methods and parameters of user interface I

1. Inverse methods:

Linear Estimation:
uif1_linear_estimation_oncortex(inReferenceData, inSearchSpaceGrid, inHeadGrid, inSensorConfiguration, inSensorType, inHeadModelSpheres, inParameters, outResult, inForwardType, inInverseType, inInverterType)
uif1_linear_estimation_onbrainsurface(inReferenceData, inSearchSpaceGrid, inHeadGrid, inSensorConfiguration, inSensorType, inParameters, outResult, inForwardType, inInverseType inInverterType)
uif1_linear_estimation_inbrainvolume(inReferenceData inSearchSpaceGrid, inHeadGrid, inSensorConfiguration, inSensorType, inLeadFieldMatrix, inParameters, outResult, inForwardType, inInverseType, inInverterType)
Dipole Fit:
uif1_movingdipolfit(inReferenceData, inHeadGrid, inSensorConfiguration, inSensorType, outResult, inParameters, nDipoles, inForwardType, inOptimizerType, inCriteriaType, inInverterType inSearchVolumeType, inIntialGuessType)
uif1_rotatingdipolfit(inReferenceData, inHeadGrid, inSensorConfiguration, inSensorType, outResult, inParameters, nDipoles, inForwardType, inOptimizerType, inCriteriaType, inInverterType inSearchVolumeType, inIntialGuessType)
uif1_fixeddipolfit(inReferenceData, inHeadGrid, inSensorConfiguration, inSensorType, outResult, inParameters, nDipoles, inForwardType, inOptimizerType, inCriteriaType, inInverterType inSearchVolumeType, inIntialGuessType)

All methods return a boolean value, indicating either a successful execution or a failure.

2. Parameters I

Parameter	Type	Description
InReferenceData	utMatrix_t<double>	Matrix containing the reference data
InSearchSpaceGrid	Class derived from: anAbstractGridGenerator_c	class containing search space grid
InHeadGrid	Class derived from: anAbstractGridGenerator_c	class containing head model (grid)
inSensorConfiguration	SensorConfiguration	class containing the description of the sensor configuration
InSensorType	sensortype_e {sensortype_EEG, sensortype_MEG}	type of sensors: EEG, MEG
InLeadFieldMatrix	utMatrix_t<double>	Matrix containing precomputed leadfield matrix

InParameters	class InverseParameters_c
	{
	public:
	InverseParameters_c();
	virtual ~InverseParameters_c();
	public:
	eegspheres_c m_EEGSpheresSettings;
	eegbem_c m_BEMSettings;
	optimizer_c m_OptimizerSettings;
	initialguess_c m_InitialGuessSettings;
	tsvd_c m_TSVDSettings;
	tikhonov_c m_TikhonovSettings
	NeuroFEMSim_c m_NeuroFEMSimSettings;
		Class containing settings for inverse methods

	NeuroFEMSolver_c m_NeuroFEMSolverSettings; NeuroFEMInverseGeneral_c m_NeuroFEMInvers GeneralSettings; NeuroFEMAnnealing_c m_NeuroFEMAnnealingSettings; NeuroFEMRegularization_c m_NeuroFEMRegularizationSettings; NeuroFEMTSVD_c m_NeuroFEMTSVDSettings; };	

	class eegspheres_c
	{
	public:
	eegspheres_c();
	virtual ~ eegspheres_c();
	public:
	int m_NumSpheres;
	utVector_t<double> m_Radii;
	utVector_t<double> m_Conductivities;
	utVector_t<double> m_Center;
	bool m_bUpdatedSettings;
	};
		class containing parameters of spherical head model

	class eegbem_c
	{
	public:
	eegbem_c();
	virtual ~ eegbem_c();
	public:
	std::string m_MatrixFileName;
	bool m_bUpdatedSettings;
	};
		Class containing BEM settings

	class optimizer_c
	{
	public:
	optimizer_c();
	virtual ~optimizer_c();
	public:
	double m_goodnessoffit;
	double m_minimaldescent;
	int m_maximumnumberofiterations;
	double m_minimumshift;
	double m_minimumrotation;
	bool m_bUpdatedSettings;
	};
		Class containing optimizer settings

	class initialguess_c
	{
	public:
	initialguess_c();
	virtual ~initialguess_c();
	public:
	int m_numinitialguesspos;
	utVector_t<double> posx;
	utVector_t<double> posy;
	utVector_t<double> posz;
	bool m_bUpdatedSettings;
	};
		Class containing initial guess settings

	class tsvd_c
	{
	public:
	tsvd_c();
	virtual ~tsvd_c();
	public:
	int m_cutoffcriterium;
	double m_abscutoff;
	double m_relcutoff;
	bool m_bUpdatedSettings;};
		Class containing truncated SVD settings

	class tikhonov_c
	{
	public:
	tikhonov_c();
	virtual ~ tikhonov_c();
	public:
	double m_SNR;
	bool m_bUpdatedSettings;
	};
		Class containing Tikhonov regularizer settings

	class NeuroFEMSim_c
	{
	public:
	NeuroFEMSim_c();
	virtual ~NeuroFEMSim_c();
	public:
	int m_analyticsolution;
	int m_degreeofintegration;
	double m_toleranceforwardsolution;
	int m_solvermethod;
	double m_simulationdipolethreshhold;
	double m_simulationepsilondirac;
	double m_dipolesmoothness;
	int m_dipolemodelorder;
	double m_dipolemodelscale;
	double m_dipolemodellambda;
	double m_dipolemodeldistance;
	bool m_bUpdatedSettings;
	};
		Class containing NeuroFEM simulator settings

	class NeuroFEMSolver_c
	{
	public:
	NeuroFEMSolver_c();
	virtual ~NeuroFEMSolver_c();
	public:
	std::string m_pebbelsparameterfilename;
	std::string m_pilutssparameterfilename;
	bool m_bUpdatedSettings;
	};	

	class NeuroFEMInverseGeneral_c
	{
	public:
	NeuroFEMInverseGeneral_c();
	virtual ~NeuroFEMInverseGeneral_c();
	public:
	double m_toleranceinversesolution;
	bool m_bUpdatedSettings;};
		Class containing NeuroFEM general inverse settings

	class NeuroFEMAnnealing_c
	{
	public:
	NeuroFEMAnnealing_c();
	virtual ~NeuroFEMAnnealing_c();
	public:
	int m_numberofdipoles;
	double m_starttemperature;
	double m_decreasingfactor;
	int m_maximumnumberofsteps;
	bool m_bUpdatedSettings;
	};
		Class containing NeuroFEM simulated annealing settings

	class NeuroFEMRegularization_c
	{
	public:
	NeuroFEMRegularization_c();
	virtual ~NeuroFEMRegularization_c();
	public:
	double m_lambdainverse;
	double m_lambdaend;
	double m_lambdaincrement;
	double m_desirecechisquare;
		Class containing NeuroFEM regularization settings

	bool m_bUpdatedSettings; };	

	class NeuroFEMTSVD_c
	{
	public:
	NeuroFEMTSVD_c();
	virtual ~NeuroFEMTSVD_c();
	public:
	int m_svdworkarray;
	double m_r_parameter;
	bool m_bUpdatedSettings;
	};
		Class containing NeuroFEM TSVD settings

OutResult	utMatrix_t<double>	Matrix containing the result
Ndipoles	Int	Number of dipoles for dipole fit

3. Parameters II (switches)

Parameter	**Possible Values**	**Default**
inForwardType	forwardtype_Sphere forwardtype_FEM	
inInverterType	invertertype_Tikhonow, invertertype_TruncatedSVD	invertertype_TruncatedSVD
inOptimizerType	optimizertype_MarquardtOptimizer	optimizertype_Marquardt Optimizer
InCriteriaType	criteriatype_MinimumSquareError	criteriatype_MinimumSquare Error
inSearchVolumeType	searchvolumetype_InEntireBrain	searchvolumetype_InEntireBrain
InIntialGuessType	intialguesstype_Standard	intialguesstype_Standard

List of methods and parameters of user interface II

1. Inverse Methods:

Linear Estimation:
uif2_linear_estimation_oncortex(inReferenceDataFileName, inCortexGridFileName, inHeadGridFileName, inSensorConfigurationFileName, inParameterFileName, outResultFilename, inForwardType, inInverseType, inInverterType)
uif2_linear_estimation_onbrainsurface(inReferenceDataFileName, inBrainSurfaceGridFileName, inHeadGridFileName, inSensorConfigurationFileName, inParameterFileName, outResultFilename, inForwardType, inInverseType, inInverterType)
uif2_linear_estimation_inbrainvolume(inReferenceDataFileName, inBrainVolumeGridFileName, inHeadGridFileName, inSensorConfigurationFileName, inLeadFieldFileName, inParameterFileName, outResultFilename, inForwardType, inInverseType, inInverterType)
Dipole Fit:
uif2_movingdipolfit(inReferenceDataFileName, inHeadGridFileName, inSensorConfigurationFileName, inParameterFileName, outResultFilename, nDipoles, inForwardType, inOptimizerType, inCriteriaType, inInverterType, inSearchVolumeType, inIntialGuessType)
uif2_rotatingdipolfit(inReferenceDataFileName, inHeadGridFileName, inSensorConfigurationFileName, inParameterFileName, outResultFilename, nDipoles, inForwardType, inOptimizerType, inCriteriaType, inInverterType, inSearchVolumeType, inIntialGuessType)
uif2_fixeddipolfit(inReferenceDataFileName, inHeadGridFileName, inSensorConfigurationFileName, inParameterFileName, outResultFilename, nDipoles, inForwardType, inOptimizerType, inCriteriaType, inInverterType, inSearchVolumeType, inIntialGuessType);

Additional methods for file I/O

bool uif2_read_ReferenceData(const string inReferenceDataFileName, utMatrix_t<double>& inReferenceData, ReferenceDataSettings_c& inReferenceDataSettings, anSensorConfiguration_c& inSensorConfiguration, SignalFileType_e FileType = VistaIn);
bool uif2_read_SensorConfiguration(const string inSensorConfigurationFileName, anSensorConfiguration_c& inSensorConfiguration, sensortype_e& inSensorType, ElectrodeFileType_e FileType = VistaElec);
bool uif2_write_ResultData(const string outResultFilename, utMatrix_t<double>& outResult, ModelType_e ModelType = Model_Rotating, int nDipoles = 1, ModelType_e ModelType = Model_Moving, ResultFileTtpe_e FileType = VistOut);
bool uif2_read_ParameterFile(const string inParameterFileName, ReferenceDataSettings_c& inRefDataSettings, InverseParameters_c& inInverseParameters);

Input-Output File Types

enum SignalFileType_e	ASCIIIn VistaIn ASAIn	VistaIn
enum ElectrodeFileType_e	VistaElec ASAElec	VistaElec
enum ResultFileType_e	VistaOut ASAOut	VistaOut

2. Parameters I

Parameter	Type	Description	Default File Format
InReferenceDataFileName	string	Filename of file containing the reference data	Vista
InCortexGridFileName	string	Filename of file containing the description of the cortex grid	Vista
InBrainSurfaceGridFileName	string	Filename of file containing the description of the brain surface grid	Vista
InBrainVolumeGridFileName	string	Filename of file containing the description of the brain volume grid	Vista
InHeadGridFileName	string	Filename of file containing the description of the head grid (needed for simulator)	Vista
inSensorConfigurationFileName	string	Filename of file containing the reference data	Vista
OutResultFilename	string	Filename of file containing the results	Vista
InParameterFileName	string	Filename of file containig settings of methods	See Appendix B

3. Parameters II (switches)

Parameter	**Possible Values**	**Default**
InForwardType	forwardtype_Sphere forwardtype_FEM	
InInverseType	linearinversetype_L2	linearinversetype_L2
InInverterType	invertertype_Tikhonow, invertertype_TruncatedSVD	invertertype_TruncatedSVD
inOptimizerType	optimizertype_MarquardtOptimizer	optimizertype_Marquardt Optimizer
InCriteriaType	criteriatype_MinimumSquareError	criteriatype_MinimumSquare Error
inSearchVolumeType	searchvolumetype_InEntireBrain	searchvolumetype_InEntireBrain
inIntialGuessType	intialguesstype_Standard	intialguesstype_Standard

List of command arguments of user interface III

1. Inverse Methods:

Linear Estimation:
uif3_linear_estimation_oncortex refdatafile=*inReferenceDataFileName* searchspacegridfile=*inCortexGridFileName* headgridfile=*inHeadGridFileName* sensorfile=*inSensorConfigurationFileName* parameterfile=*inParameterFileName* resultfile=*outResultFilename* inForwardType inInverseType inInverterType
uif3_linear_estimation_onbrainsurface refdatafile=*inReferenceDataFileName* searchspacegridfile=*inBrainSurfaceGridFileName* headgridfile=*inHeadGridFileName* sensorfile=*inSensorConfigurationFileName* parameterfile=*inParameterFileName* resultfile=*outResultFilename* inForwardType inInverseType inInverterType
uif3_linear_estimation_inbrainvolume refdatafile=*inReferenceDataFileName* searchspacegridfile=*inBrainVolumeGridFileName* headgridfile=*inHeadGridFileName* sensorfile=*inSensorConfigurationFileName* parameterfile=*inParameterFileName* resultfile=*outResultFilename* inForwardType inInverseType inInverterType
Dipole Fit:
uif3_movingdipolefit refdatafile=inReferenceDataFileName headgridfile=*inHeadGridFileName* sensorfile=*inSensorConfigurationFileName* resultfile: parameterfile=*inParameterFileName* *outResultFilename* nDip=*nDipoles* inForwardType inOptimizerType inCriteriaType inInverterType inSearchVolumeType inIntialGuessType
uif3_rotatingdipolefit refdatafile=inReferenceDataFileName headgridfile=*inHeadGridFileName* sensorfile=*inSensorConfigurationFileName* parameterfile=*inParameterFileName* resultfile=*outResultFilename* nDip=*nDipoles* inForwardType inOptimizerType inCriteriaType inInverterType inSearchVolumeType inIntialGuessType
uif3_fixeddipolefit refdatafile=inReferenceDataFileName headgridfile=*inHeadGridFileName* sensorfile=*inSensorConfigurationFileName* parameterfile=*inParameterFileName* resultfile=*outResultFilename* nDip=*nDipoles* inForwardType inOptimizerType inCriteriaType inInverterType inSearchVolumeType inIntialGuessType

2. Values for arguments used as switches

Parameter	**Possible Values**	**Default**
InForwardType	Sphere FEM	
InInverseType	L2	L2
InInverterType	Tikhonow TruncatedSVD	TruncatedSVD
InOptimizerType	Marquardt	Marquardt
InCriteriaType	MinimumSquareError	MinimumSquareError
InSearchVolumeType	InEntireBrain	InEntireBrain
InIntialGuessType	Standard	Standard

Appendix B Description of parameter file

The parameter file is divided into sections. Each section starts with a section keyword in brackets [sectionkeyword]. This keyword is followed by a set of lines consisting of a keyword followed by "=" and the value which shall be assigned to a variable, which is determined by the keyword.

Lines containing a comment begin with a "#".

The following list contains the sections of the parameter file and the keywords and data types to determine properties of methods indicated by the section keyword. Keyword and data must be separated by a " ". If more than one value belongs to keyword, values must be given in the following line separated by " ". It is not mandatory to set all possible values, which can be set by the parameter file. Parameters not set in the parameter file are set to default values.

New sections or new section members will be documented in supplements to this design report.

Parameter file:

```
[ReferenceData]
# Number of samples in ReferenceDataFile
numsamples= int
# Number of channels in ReferenceDataFile
numchan= int
# Startsample for inverse procedure
startsample= int
# Stopsample for inverse procedure
stopsample= int

[EEGSphereSimulator]
# Number of spheres
eegnumspheres= int
# Radii of spherical head model
radii=
floatvalue floatvalue ...
# Conductivities of speherical head model
conductivites=
floatvalue floatvalue ...
floatvalue floatvalue ...
# Center of spheres
centerx= floatvalue
centery= floatvalue
centerz= floatvalue

[EEGBEMSimulator]
# File name of system matrix (if present)
systemmatrixfilename= string

[NeuroFEMSimulator]
# different analytical solutions for the EEG problem
analyticalsolution= int
# degree of Gaussian integration (2 is recommended)
degreeofintegration= int
# residuum of the forward solution
toleranceforwardsolution= floatvalue
# SOLVER (1:Jakobi-CG, 2:IC(0)-CG, 3:AMG-CG, 4:ILDLT-CG)
solvermethod= int
```

```
# threshold (percentage of the greatest dipole strength) of all dipoles to appear in the result files
dipolethreshold= floatvalue
# blurring single loads to adjacent nodes by means of the Gaussian (normal)
#                                                    distribution
sourcesimulationepsilondirac= floatvalue
# dipole modeling; weighting of the source distribution with the power of the declared value
dipolemodelingsmoothness= int
# power of the dipole moments to be considered
dipolemodelingorder= int
# necessary internal scaling factor; should be larger than twice the element edge length
dipolemodelingscale= floatvalue
# Lagrangian multiplier for the (inverse) dipole modeling
dipolemodelinglambda= floatvalue
# source-sink separation of the analytical dipole model
dipolemodelingdistance= floatvalue

[NeuroFEMSolver]
# parameter file for Pebbles solver
pebbles= string
# parameter file for Piluts solver
piluts= string

[NeuroFEMInverseGeneral]
#  residuum of the inverse solution
toleranceinversesolution= floatvalue

[NeuroFEMAnnealing]
#  in a simulated annealing calculation it gives the number of n dipoles to be searched;
#  in a FOCUSS procedure the stopping criteria for the number of remaining dipoles
numberofdipoles= int
# determines the start level of the SA-procedure
starttemperature= floatvalue
# decreasing factor
decreasingfactoroftemp= floatvalue
# number of the overall iterations of a SA procedure
maximumannealingsteps= int
```

[NeuroFEMRegularization]
# Lagrangian multiplier l in the functional $A+lB;$
# start value for the convergence test
lambdainverse= floatvalue
# final value for the convergence test
lambdaend= floatvalue
# factor for the convergence test
lambdaincrement= floatvalue
# χ^2 value to reach while doing a lambda iteration
desiredchisquarevalue= floatvalue

[NeuroFEMTSVD]
# memory space size for a SVD calculation
svdworkarray= nt
# r-parameter
rparameter= floatvalue

[InitialGuess]
Number of initial guess positions
numinitialguesspos=
Initial guess positions
posx=
floatvalue floatvalue ….
posy=
floatvalue floatvalue ….
posz=
floatvalue floatvalue …..

[SearchVolume]
#radius of spherical search volume
searchvolumeradius= 0.07618
#center of spherical search volume
#center
searchvolumecenterx= 0.004474
searchvolumecentery= 0.004474
searchvolumecenterz= 0.04897

[Optimizer]
# Goodness of fit
goodnessoffit= floatvalue
# Minimal descent of goal function
minimaldescent= floatvalue
# Maximum number of iterations
maximumnumberofiterations= int
# Minimum shift of positions
minimumshift= floatvalue
# Minimum rotation of Moment
minimumrotation= floatvalue

[RegularizationTSVD]
# cutoff criterium 1=absolut, 2=relativ
cutoffcriterium= int
# absolut cutoff value for TSVD

```
abscutoff= float
# relativ cutoff value for TSVD
relcutoff= float

[RegularizationTikhonov]
# estimated signal to noise ratio
estimatedSNR= float


 [Error]
# Error code
code = string
# Error type
type = string
# Origin of error
origin = string
# Error description
description = string
```

Appendix C: Examples for the creation of new user scenarios

Implementation example of user interface I:

Class definition

```
//$1 06.02.2001 Matthias D. created

#ifndef __UIF1_c_H__
#define __UIF1_c_H__


// UIF1_c.h: interface for the UIF1_c class.
//
//////////////////////////////////////////////////////////////////////



#include <utilities/include/utvector_t.h>
#include <utilities/include/utmatrix_t.h>
#include <utilities/include/utblock_t.h>

// Include files of analysis toolbox

#include <analysis/include/anregularizertruncsvd_c.h>
#include <analysis/include/ancriteriaminimumsquareerror_c.h>
#include <analysis/include/ansearchvolumesphere_c.h>
#include <analysis/include/ansearchvolumeinfinite_c.h>
#include <analysis/include/anoptimizermarquardt_c.h>
#include <analysis/include/angoalfunctiondipolerotating_c.h>
#include <analysis/include/angoalfunctiondipolefixed_c.h>
#include <analysis/include/ansensorconfiguration_c.h>
#include <analysis/include/ansimulatoreegspheres_c.h>
#include <analysis/include/aninitialguessstandard_c.h>
#include <analysis/include/ananalyzerinversedipolefit_c.h>
#include <analysis/include/ananalyzerinverselinear_c.h>
#include <analysis/include/anunaryweighter_c.h>
#include <analysis/include/angridgeneratorsingledipoles_c.h>

///// enumerators are used for switches to determine details of the methods

enum forwardtype_e {forwardtype_Sphere, forwardtype_BEM,forwardtype_FEM};
enum linearinversetype_e {linearinversetype_L2, linearinversetype_Loreta};
enum invertertype_e {invertertype_Tikhonow, invertertype_TruncatedSVD};
enum optimizertype_e {optimizertype_SimplexOptimizer,
optimizertype_MarquardtOptimizer, optimizertype_SimulatedAnnealing};
enum criteriatype_e {criteriatype_MinimumSquareError, criteriatype_MaximumEntropie,
criteriatype_MaximumProbability};
enum searchvolumetype_e {searchvolumetype_InEntireBrain};
enum intialguesstype_e {intialguesstype_Standard};

….

class UIF1_c
{
public:
	UIF1_c();
	virtual ~UIF1_c();
…
```

```
//  Continuous Parameter Space: UserFunctions for dipole(s)


bool uif1_rotatingdipolfit(const utMatrix_t<double>& inReferenceData,
                           // Matrix containing reference (measured) data
                           anAbstractGridGenerator_c& inHeadGrid,
                           // Head Grid
                           anSensorConfiguration_c inSensorConfiguration,
                           // Configurstion of the sensors (electrodes, MEG)
                           sensortype_e inSensorType,
                           // Switch for the selecetion of the sensortype
                           utMatrix_t<double>& outResult,
                           // Resultmatrix
                           InverseParameters_c&  inParameters,
                           // Class Containing Settings of Inverse Methods
                           int nDipoles = 1,
                           // Number of Dipoles
                           forwardtype_e inForwardType = forwardtype_BEM,
                           // Switch for the selecetion of forward model
                           optimizertype_e inOptimizerType =
                           optimizertype_MarquardtOptimizer,
                           // Switch for the selection of the optimizer
                          criteriatype_e  inCriteriaType  =
                          criteriatype_MinimumSquareError,
                          // Switch for the selection of criteria
                          invertertype_e inInverterType  =
                          invertertype_TruncatedSVD,
                          // Type of regularization,    Default: Truncated Singular
                          // Value Decomposition
                          searchvolumetype_e inSearchVolumeType =
                          searchvolumetype_InEntireBrain,
                          // Switch for the selection of the search volume type
                          intialguesstype_e inIntialGuessType =
                          intialguesstype_Standard);
                          // Switch for the selection of initialguess
```

….

Class implementation

….

```
bool UIF1_c::uif1_rotatingdipolfit(const utMatrix_t<double>& inReferenceData,
                                   // Matrix containing reference (measured) data
                                   anAbstractGridGenerator_c& inHeadGrid,
                                                       // Head Grid
                                   anSensorConfiguration_c inSensorConfiguration,
                                   // Configurstion of the sensors (electrodes, MEG)
                                   sensortype_e inSensorType,
                                   // Switch for the selecetion of the sensortype
                                   utMatrix_t<double>& outResult,
                                   // Resultmatrix
                                   InverseParameters_c&  inParameters,
                                                // Class Containing Settings of
                                   Inverse Methods
                                   int nDipoles,
                                   // Number of Dipoles
                                   forwardtype_e inForwardType,
                                   // Switch for the selecetion of forward model
                                   optimizertype_e inOptimizerType,
                                               // Switch for the selection of the
                                   optimiser
                                                       criteriatype_e  inCriteriaType,
                                                                   // Switch for the
                                   selection of criteria
```

```
                                invertertype_e inInverterType,
                                       // Type of regularization,    Default:
                                Truncated
                                       // Singular Value Decomposition
                                searchvolumetype_e inSearchVolumeType,
                                       // Switch for the selection of the search
       volume
                                intialguesstype_e inIntialGuessType)
                                // Switch for the selection of initialguess

{

// Selection of type of forward model
anAbstractSimulatorEEGMEG_c                  *sim=NULL;

switch(inSensorType)
{
case sensortype_EEG:
        switch (inForwardType)
              {
               case forwardtype_Sphere:
                     sim= new anSimulatorEEGSpheres_c(inSensorConfiguration,
                     inParameters.m_EEGSpheresSettings.m_Radii,
                     inParameters.m_EEGSpheresSettings.m_Center,
                     inParameters.m_EEGSpheresSettings.m_Conductivities) ;
                     break;
              case forwardtype_BEM:
                     sim = new
                     anForwardSimulatorBEMEEG_c(inSensorConfiguration,head_grid);
                     break;

              case forwardtype_FEM:
                     sim = new anForwardSimulatorNeuroFEMEEG_c(inSensorConfiguration,
                     head_grid);
                     break;

              default:
                     return false;

              }

       break;

case sensortype_MEG:
        switch (inForwardType)
              {
              case forwardtype_Sphere:
                     sim = new anForwardSimulatorSpheresEEG_c(inSensorConfiguration)
                     break;
              case forwardtype_BEM:
                     sim = new      anForwardSimulatorBEMMEG_c(inSensorConfiguration,
                     head_grid);
                     break;
              case forwardtype_FEM:
                     sim = new anForwardSimulatorNeuroFEMMEG_c(inSensorConfiguration,
                     head_grid);
                     break;
              default:
                     return false;

              }

     break;


}


// Selection of type of criteria
anAbstractCriteria_c                   *crit=NULL;
```

```
switch (inCriteriaType)
{
	case criteriatype_MinimumSquareError:
		crit = new anCriteriaMinimumSquareError_c;
		break;
	case criteriatype_MaximumEntropie:
		crit = new anMaximumEntropie_c;
		break;
	 case criteriatype_MaximumProbability:
		crit = new anMaximumProbability_c;
		break;
	default:
		delete head_grid;
		delete sim;
		return false;
}


// Selection of type of search volume
anAbstractSearchVolume_c	*vol=NULL;

switch (inSearchVolumeType)
{
	case searchvolumetype_InEntireBrain:
		vol = new anSearchVolumeInfinite_c;
		break;
	default:
		delete head_grid;
		delete sim;
		delete crit;
		return false;
}



// Selection of Regularization
anAbstractRegularizer_c	*reg=NULL;


switch (inInverterType)
{
	case invertertype_Tikhonow:
		reg = new anRegularizerTikhonow_c;
		break;
	case invertertype_TruncatedSVD:
		reg = new anRegularizerTruncSVD_c;
		break;
	default:
		delete head_grid;
		delete sim;
		delete crit;
		delete vol;
		return false;
}


// initialize Grid Generator
anGridGeneratorSingleDipoles_c dipolegrid;

// intitialize weighter
anUnaryWeighter_c		weighter(*sim);			// define
weighter

// initialize linear estimator
anAnalyzerInverseLinear_c
linearestimator(inReferenceData,*sim,dipolegrid,weighter,*reg);

anGoalFunctionDipoleRotating_c
goalfunction(inReferenceData,*sim,*crit,*vol,linearestimator);
```

```
// Selection of type of forward otpimizer
anAbstractOptimizerContinuous_c     *opt=NULL;

switch (inOptimizerType)
{
              case optimizertype_SimplexOptimizer:
                     opt = new anSimplexOptimizer_c(goalfunction);
                     break;
              case optimizertype_MarquardtOptimizer:
                     opt = new anOptimizerMarquardt_c(goalfunction);
                     break;
              case optimizertype_SimulatedAnnealing:
                     opt = new SimulatedAnnealingOptimizer_c(goalfunction);
                     break;
              default:
                     delete head_grid;
                     delete sim;
                     delete crit;
                     delete vol;
                     delete reg;
                     return false;

}


// Initial Guess


anAbstractInitialGuess_c           *inguess=NULL;

switch (inIntialGuessType)
{
       case intialguesstype_Standard:
              inguess = new anInitialGuessStandard_c;
              break;
       default:
              delete head_grid;
              delete sim;
              delete crit;
              delete vol;
              delete reg;
              delete opt;
              return false;

}


// anAnalyzerInverseFit

anAnalyzerInverseDipoleFit_c      inversefit(nDipoles,*inguess,*opt);

// Moving Dipole Fit
inversefit.run();

inversefit.getResult(outResult);

delete head_grid;
delete sim;
delete crit;
delete vol;
delete inguess;
delete opt;
delete reg;
return true;

}
```

```
//
//////////////////////End of method uif1_rotatingdipolfit/////////////////
////////////////////////////////////////////////////////////////////////////
…
```

Implementation example of user interface II:

Class definiton

```
class UIF2_c
{
public:
       UIF2_c();
       virtual ~UIF2_c();
```

…

```
bool uif2_rotatingdipolfit(string inReferenceDataFileName,
                           // Inputfilename: File containing reference
                           // (measured) data
                           string inHeadGridFileName,
                           // Inputfilename: File with description of the head grid
                           string inSensorConfigurationFileName,
                           // InputFile:Configurstion of the sensors
                           // (electrodes, MEG-gradiometer)
                           string inParameterFileName,
                           // InputFilename: ParameterFile
                           string outResultFilename,
                           // OutputFilenam: File with   Resultmatrix
                           int nDipoles = 1,
                                       // Number of Dipoles
                           forwardtype_e inForwardType = forwardtype_BEM,
                           // Switch for the selecetion of forward model
                           optimizertype_e inOptimizerType =
                                  optimizertype_MarquardtOptimizer,
                                  // Switch for the selection of the optimizer
                           criteriatype_e  inCriteriaType  =
                                              criteriatype_MinimumSquareError,
                          // Switch for the selection of criteria
                           invertertype_e inInverterType =
                           invertertype_TruncatedSVD,
                           // Type of regularization,    Default: Truncated
                           // Singular Value Decomposition
                           searchvolumetype_e inSearchVolumeType =
                           searchvolumetype_InEntireBrain,
                           // Switch for the selection of the search volume type
                           intialguesstype_e inIntialGuessType =
                           intialguesstype_Standard);
                           // Switch for the selection of initialguess
```

…

Class implementation

```
bool UIF2_c::uif2_rotatingdipolfit(string inReferenceDataFileName,
                                   // Inputfilename: File containing reference
                                   // (measured) data
                                   string inHeadGridFileName,
                                   // Inputfilename: File with description of the
                                   // head grid
                                   string inSensorConfigurationFileName,
                                   // InputFile:Configurstion of the sensors
                                   // (electrodes, MEG-gradiometer)
                                   string inParameterFileName,
                                   // InputFilename: ParameterFile
                                   string outResultFilename,
                                   // OutputFilenam: File with   Resultmatrix
                                   int nDipoles,
                                   // Number of Dipoles
                                   forwardtype_e inForwardType,
                                   // Switch for the selecetion of forward model
```

```
                                optimizertype_e inOptimizerType,
                                // Switch for the selection of the optimizer
                                criteriatype_e  inCriteriaType,
                                // Switch for the selection of criteria
                                invertertype_e inInverterType,
                                // Type of regularization,    Default:
                                       // Truncated Singular Value Decomposition
                                searchvolumetype_e inSearchVolumeType,
                                // Switch for the selection of the
                                // search volume type
                                intialguesstype_e inIntialGuessType)
                                // Switch for the selection of initialguess
{

anAbstractGridGenerator_c       *inHeadGrid = NULL;

switch (inForwardType)
       {
       case forwardtype_Sphere:
              break;
         case forwardtype_BEM:
                           inHeadGrid =
newanGridGeneratorBEMFromFile_c(inHeadGridFileName);
              break;
       case forwardtype_FEM:
              inHeadGrid = newanGridGeneratorFEMFromFile_c(inHeadGridFileName);

              break;
       default:
              return false;

       }


// Intialization of grid in brain volume and for head model

//anHeadModelGridGeneratorfromFile_c          head_grid (inHeadGridFileName);
//headgrid_is_generated = true;

// Read SensorConfiguration from File
anSensorConfiguration_c inSensorConfiguration;
sensortype_e            inSensorType=sensortype_unknown;
uif2_read_SensorConfiguration(inSensorConfigurationFileName, inSensorConfiguration,
inSensorType);

// Decode Parameterfile
ReferenceDataSettings_c  ReferenceDataSettings;
InverseParameters_c        Parameters;

uif2_read_ParameterFile(inParameterFileName, ReferenceDataSettings, Parameters);

// Read Referencedata
utMatrix_t<double> inReferenceData;
uif2_read_ReferenceData(inReferenceDataFileName, inReferenceData,
ReferenceDataSettings);


utMatrix_t<double> outResult;
```

```
bool bResult = m_UIF1.uif1_rotatingdipolfit(inReferenceData,
                                    // Matrix containing reference (measured) data
                                    *inHeadGrid,
                                    // Head grid
                                    inSensorConfiguration,
                                    // Configurstion of the sensors (electrodes, MEG-//
                                    gradiometer)
                                    inSensorType,
                                    // Switch for the selecetion of the sensortype
                                    outResult,
                                    // Resultmatrix
                                    Parameters,
                                    // Parameters of methods
                                    nDipoles,
                                    // Number of Dipoles
                                    inForwardType,
                                    // Switch for the selecetion of forward model
                                    inOptimizerType,
                                    // Switch for the selection of the optimizer
                                    inCriteriaType,
                                    // Switch for the selection of criteria
                                    inInverterType,
                                    // Type of regularization,    Default: Truncated //
                                    Singular Value Decomposition
                                    inSearchVolumeType,
                                    // Switch for the selection of the search volume
                                    inIntialGuessType);
                                    // Switch for the selection of initialguess


if (!uif2_write_ResultData(outResultFilename, outResult, Model_Rotating,
nDipoles))return false;


return bResult;
}

//
//////////////////////End of method uif2_rotatingdipolfit ////////////////////////
```

Implementation example of user interface III:

```
#include "uif1_c.h"
#include "uif2_c.h"


// Method to interpete argv
bool decodeKeyword(char* keyword);

int main(int argc, char* argv[])

{
// return value of inverse procedure
bool bsuccess = false;

for(int i = 1; i < argc; i++)
       {
       char* keyword = argv[i];
       decodeKeyword(argv[i]);
       }

if (ValidInverseProcedure)
{

////////////////////////////////////////////////////////////////////////////////
//////////////////////////
```

```
//
// Dipole Fit methods
//

// Moving Dipole Fit

if (inInverseProcedureType == invereseproceduretype_movingdipolfit )
       {
        if(ReferenceDataFileNamePresent && HeadGridFileNamePresent &&
           SensorFileNamePresent && ResultFileNamePresent)
           {
           if(!forwardtypeset)    printf("\nNo forward type set. Instead
                                          deafult is used:   BEM\n");
           if(!optimizertypeset)  printf("\nNo optimizer type set. Instead default
                                          is used: Marquardt\n");
           if(!criteriatypeset)   printf("\nNo criteria type set. Instead default
                                          is used:MinimumSquareError\n");
               if(!invertertypeset)      printf("\nNo inverter type set. Instead
           default
                                          is used:  TruncatedSVD\n");
           if(!searchvolumetypset)  printf("\nNo search volume set. Instead
                                          deafult is used:  InEntireBrain\n");
           if(!intialguesstypeset)  printf("\nNo initial guess set. Instead
                                          deafult is used:  Standard\n");

// Start inverse procedure

           bsuccess=uif2.uif2_movingdipolfit(ReferenceDataFileName,
                                          HeadGridFileName,
                                          SensorFileName,
                                          ParameterFileName,
                                          ResultFileName,
                                          nDip,
                                          inForwardType,
                                          inOptimizerType,
                                          inCriteriaType,
                                          inInverterType,
                                          inSearchVolumeType,
                                          inIntialGuessType);
              }
       else
              {
               printf("\nNot all necessary file names set !\n");
              }
       }
```

……

```
bool decodeKeyword(char* keyword)
{
/// Decodes Command Line Arguemnts


////////////////////////////////////////////////////////////////////////////
/////////////////////////
// Inverse Procedure


if (stricmp(keyword,"uif3_linear_estimation_oncortex")== 0)
{
inInverseProcedureType = invereseproceduretype_linear_estimation_oncortex;

printf("\nInverse Prcodure: uif3_linear_estimation_oncortex\n");

ValidInverseProcedure= true;}

if (stricmp(keyword,"uif3_linear_estimation_onbrainsurface")== 0)
{
inInverseProcedureType = invereseproceduretype_linear_estimation_onbrainsurface;

printf("\nInverse Prcodure: uif3_linear_estimation_onbrainsurface\n");

ValidInverseProcedure= true;}

if (stricmp(keyword,"uif3_movingdipolefit")== 0)
{
inInverseProcedureType = invereseproceduretype_movingdipolfit;

printf("\nInverse Prcodure:      uif3_movingdipolrfit\n");

ValidInverseProcedure= true;}
……

}
```