GIMLi – Geophysical Inversion and Modelling Library – programmers tutorial

Thomas Günther* & Carsten Rücker†

December 13, 2011

GIMLi is a C++ class library for solving inverse and forward problems in geophysics. It was build in order to make inversion available to various forward modelling routines. Template programming algorithms are used to create a purely mathematical framework for solving physics problems. The inversion can apply various minimisation and regularization schemes, different transformation functions a region approach for sophisticated regularization and the chance to incorporate parametric and structural inversion.

In this tutorial we like to show the programmer how to work with GIMLi by means of different examples. A simple curve-fitting is used to show how different forward operators can be incorporated. Some small 1d examples demonstrate the use of different parameterisation and the partitioning of data and model. The region technique can be used for different target parameters or geological units.

Joining different inversion runs is one of the key issues of ongoing research. We show three different joint inversion (JI) types: one-parameter JI, petrophysical JI and structural JI. Finally the time-lapse inversion framework is presented and applied to ERT data using different algorithms.

*Leibniz Institute for Applied Geophysics, Hannover (Germany)
†Institute of Geology and Geophysics, University of Leipzig (Germany)

Contents

1. Introduction **3**
1.1. GIMLi – concept and overview . . . 3
1.2. Minimisation and regularization methods . . . 4
1.3. Transformation functions . . . 5
1.4. Parameterisation and the region technique . . . 6
1.5. Obtaining and building GIMLi . . . 6
1.6. Outline of the document . . . 8

2. A very simple example - polynomial curve fitting **9**
2.1. The first program in C++ . . . 9
2.2. A first Python program . . . 10
2.3. An own Jacobian . . . 12

3. General concepts using 1D DC resistivity inversion **13**
3.1. Smooth inversion . . . 13
3.2. Block inversion . . . 15
3.3. Resolution analysis . . . 16
3.4. Structural information . . . 17
3.5. Regions . . . 18

4. Enhanced techniques **20**
4.1. Combining different data types - MT 1d inversion . . . 20
4.2. Combining different parameter types - offsets in travel time . . . 21
4.3. What else? . . . 23

5. Joint inversion **24**
5.1. Classical joint inversion of DC and EM soundings . . . 24
5.2. Block joint inversion of DC/MRS data . . . 25
5.3. Structurally coupled cooperative inversion of DC and MRS soundings . . . 27
5.4. Petrophysical joint inversion . . . 29

6. Inversion templates **30**
6.1. Roll-along inversion . . . 30
6.2. Joint inversion . . . 30
6.3. Time lapse inversion . . . 31

A. Inversion properties **32**

B. Mesh types and mesh generation **33**

C. Region properties and region map file **35**

D. Transformation functions **37**

E. Vectors and Matrix types **38**
E.1. Block-Matrices . . . 38

1. Introduction

1.1. GIMLi – concept and overview

In geophysics, various physical processes and fields are used to gain information about the subsurface parameters. The fields and processes can be very well studied and understood by simulation assuming a parameter distribution. This so-called forward task can be done by using analytical formulae and numerical integration, or numerical schemes deploying finite difference or finite element techniques. In the recent years, very different studies have been presented that open up new methods to be used.
However, in almost all approaches the task is finally to derive subsurface parameters, i.e. the inverse problem has to be solved. Very often this is ill-posed, i.e. a variety of solutions is fitting the data within error bounds. Hence regularization methods have to be applied. There exist numerous inversion and regularization schemes, which do not have to be reinvented. Furthermore, often a resolution analysis is required in order to appraise the quality of the results. The idea of GIMLi is to present a very flexible framework for geophysical inversion and modelling such that it can be used from any forward operator. All problems such as optimization of regularization parameters, line search are solved generally. The GIMLi library is structured into four layers (Fig. 1) that are based on each other:

The basic layer holds fundamental algebraic methods and mesh containers for model parameterisation

The modelling®ion layer administrates the modelling classes that are based on a basis class and the connection to constraints and transform functions

The inversion layer is a template class for minimisation with different methods, inverse solvers, line search, λ optimisation and resolution analysis

In Inversion frameworks sophisticated techniques are formulated, e.g. time-lapse strategies, roll-along inversion or different kinds of joint inversion

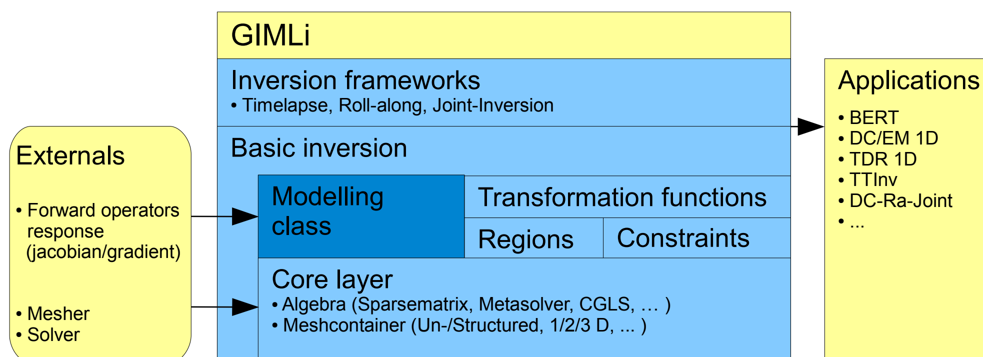

Figure 1: Scheme of the GIMLi library, applications and externals

External programs are, e.g., mesh generators and solvers for linear systems. For generating quality constrained irregular 2d and 3d meshes, we usually use `Triangle` (Shewchuk, 1996) and `TetGen` (Si, 2003). For solving linear systems we recommend the open-source collection

`SuiteSparse` (Davis, 2006), which contains multi-frontal direct&iterative solvers as well as reordering algorithms.
External forward operators can be easily linked against GIMLi. As soon they meet the requirements, the inversion can be setup and run with 2 lines.

1.2. Minimisation and regularization methods

Assume we have a discrete number of data d_i assembled in a data vector $\mathbf{d} = [d_1, \ldots, d_D]^T$. We want to find a parameter distribution represented by a discrete model vector $\mathbf{m} = [m_1, \ldots, m_M]^T$ such that the forwad response $\mathbf{f}(\mathbf{m})$ approximates $\mathbf{d}$. To each datum a variance δd_i is associated. By weighting the individual misfit $f_i - d_i$ with δd_i we obtain a unit-less misfit vector, such that different data are combined easily.
We try to minimise the misfit vector in a least squares sense using the objective function

$$\Phi_d = \sum_{i=1} D \left| \frac{d_i - f_i(\mathbf{m})}{\delta d_i} \right|^2 = \left\| \mathbf{D}(\mathbf{d} - \mathbf{f}(\mathbf{m})) \right\|_2^2 \to \min \quad \text{with} \quad \mathbf{D} = \mathrm{diag}(\delta d_i^{-1}) \quad . \tag{1}$$

More generally, the inverse data covariance matrix $\mathbf{W}_d^{-1}$ can be used instead of $\mathbf{D}$ if the variances are correlated. Since the problem is usually non-unique, regularization has to be applied. We concentrate on explicit and global regularization (CITE) and use a generalized matrix formulation (CITE) to obtain a model objective function

$$\Phi_m = \| \mathbf{W}^c \mathbf{C} \mathbf{W}^m (\mathbf{m} - \mathbf{m}^R) \|_2^2 \quad . \tag{2}$$

$\mathbf{m}^R$ represents a reference model. The matrix $\mathbf{C}$ is a derivative matrix or an identity matrix or a mixture of it. In the first neighboring relations are taken into account (smoothness constraints), whereas in the latter (zeroth order smoothness) they are not. The assumption of a smooth model is often the only choice to cope with ill-posedness and limited resolution. However the degree of smoothness can be controlled flexibly with the mostly diagonal weighting matrices $\mathbf{W}^b = \mathrm{diag}(w_i^b)$ and $\mathbf{W}^m = \mathrm{diag}(w_i^m)$. A derivative matrix consists of C rows or constraint equations constraining the M model cells corresponding to the rows. By choosing the w_i^m each model cell can be weighted individually such that a varying degree of smoothness or vicinity to the reference model can be achieved. Doing so, parameter constraints are applied using cell-dependent regularization. On the contrary, we are able to incorporate structural constraints by setting the the weights w_i^c for the individual model boundaries. For example, we can allow for arbitrary contrasts along a known boundary (e.g. from seismics or boreholes) in an otherwise smooth model (CITE).
Finally, a regularization parameter λ is used to weight Φ_d and Φ_m so that we minimize

$$\Phi = \Phi_d + \lambda \Phi_m = \left\| \mathbf{D}(\mathbf{d} - \mathbf{f}(\mathbf{m})) \right\|_2^2 + \lambda \left\| \mathbf{W}^c \mathbf{C} \mathbf{W}^m (\mathbf{m} - \mathbf{m}^R) \right\|_2^2 \to \min \quad . \tag{3}$$

Although the w_i are already regularization parameters it is easier to use relative w_i and optimized only the one external λ. In the knowledge of the variances, λ has to be chosen such that the data are fitted within their variances in a least squares sense. The inversion task can then be posed as

$$\min_{\mathbf{m}} \Phi_m \quad \text{subject to} \quad \chi^2 = \Phi_d / N = 1 \quad ,$$

which yields the same result as equation 3 for appropriate λ.

There are different methods for minimisation. The most popular one is a Gauss-Newton scheme since it has a fast convergence. However it requires the computation of the jacobian or sensitivity matrix with the elements $J_{ij} = \frac{\partial f_i(\mathbf{m})}{\partial m_j}$. Some physical problems allow for efficient sensitivity approximation. For small-scaled problems with fast forward operators it can be approximated by (brute force) variation of the m_j and a forward calculation.
In Gauss-Newton inversion, a big equation system consisting of the matrices above is solved for the model update. However, the left-hand side is not built up. Instead the equation is solved by conjugate-gradient based solvers that incorporate all matrices and vectors.
Alternative methods that do not need the storage of the are gradient-based. The gradient

$$\mathbf{g}(\mathbf{m}) = \left[\frac{\partial \Phi}{\partial m_1}, \frac{\partial \Phi}{\partial m_2}, \dots \frac{\partial \Phi}{\partial m_M}\right]^T$$

splits up in the model gradient and data gradient. The latter can be computed for some methods using adjoint field methods. Otherwise it can be computed by the perturbation method as well.
The easiest method is the steepest descent method where the model update is sought in the negative gradient. A more sophisticated and faster converging method is called nonlinear conjugate gradients (NLCG), where the preceding gradients are taken into account. A hybrid method between gradient and Gauss-Newton method is the quasi-Newton method. It successively stores the projections of the gradient directions and approximates the jacobian step-wise. Thus it starts with the linear convergence of gradient methods but ends up with quadratic Gauss-Newton convergence without storage of the jacobian matrix. This method is particularly interesting for higher-dimensional problems (Haber, 2005).
For all methods, an update $\Delta\mathbf{m}^{k+1}$ of the current model $\mathbf{m}^k$ is obtained. The model is updated using a step length τ^{k+1} such that equation 3 is minimized. The latter is called line search and can be done exact, by interpolation of the $f_i(\mathbf{m}^k + \Delta\mathbf{m}^{k+1})$ or by forward calculation of three step lengths and fitting a parabola.

Forward operator requirement

A forward operator is defined by a C++ class object derived from a base forward operator class `ModellingBase`. The only necessary function is called `response`, and it returns a forward response vector for a given model vector. Once an instance f of class has been defined, the forward response for the model vector model is called by f.response(model) or, more briefly, f(model).

1.3. Transformation functions

The forward problem is usually posed based on some intrinsic properties and measurements, e.g. the measured voltage is based on conductivity. However, we often want to use d and m values different from that, e.g. apparent resistivity and logarithm of resistivity. Motivation for that may be a better-posed system, additional constraints such as positivity or the incorporation of petrophysical constraints (Tarantola, 2001; Günther et al., 2008).
In any GIMLi application we can choose arbitrary transformation functions in any stage of the inversion. The inversion itself is using template classes such that the transformations are carried out on the fly. The choice affects model and data vector, but also data weighting and of course the gradient and jacobian. However the jacobian or gradient of the transformed inverse

problem are never explicit formed, instead they are incorporated into the inverse sub-problem by using derivatives.
A transformation $\hat{m}(m)$ needs three things: the transformation from m to $\hat{m}$ and back, and the derivative of $\hat{m}$ with respect to m. A number of useful transformation is already available. Others can be easily created or derived by combination of existing ones. See appendix D for details.

1.4. Parameterisation and the region technique

The discrete model parameters m_i can be freely defined (without spatial association - 0d) or be coefficients for a given model parameterisation (1d, 2d, 3d or 4d) or a mixture of it. A spatial parameterisaton is represented by a mesh containing the neighboring relations. Figure 2 shows different basic parameterisations. This can be structured (e.g. a finite difference discretisaton) or unstructured arranged by triangles or tetrahedra. There are various functions for mesh generation, export and import, see appendix B.

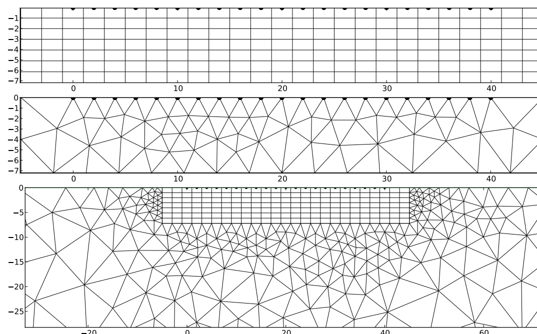

Figure 2: Different parameterisations: 0d (independent parameters), 1d mesh, 2d structured mesh (grid), 2d unstructured mesh, 2d mixed mesh, 3d mesh

A combination of different data is easily achieved by the described error weighting and combining two vectors, e.g. amplitude and phase data in MT, in one vector. Different parameters, can similarly be treated by different parts of the model vector, so-called regions. Regions can be, as the name states, parts of one mesh representing different geological units. Regions can also be different parameters on the same mesh (e.g. porosity and saturation) or on different parametrisations, e.g. a 2d/3d resistivity distribution and (0d) values for static shift in MT inversion.
In inversion the regions are by default independent from each other, but can as well be constrained to another. For each region the starting model value, but also constraint type, model and constraint control and the used transformation function can be freely defined. Special regions are a background region (not part of inversion) and a single-parameter region (all cells a compounded to one parameter). See appendix C for how to control the behaviour of regions.

1.5. Obtaining and building GIMLi

Since summer 2011, GIMLi is hosted on SourceForge under the project name libgimli[1]. See `http://sourceforge.net/projects/libgimli/` for additional information such as binary

[1]Note that since this change the BERT components were excluded from the project to a dedicated repository.

downloads, documentation, bug tracker and feature request. The code itself can be retrieved using subversion (SVN) using the address `https://libgimli.svn.sourceforge.net/svnroot/libgimli`. As usual the code contains a current development (trunk), experimental changes or side-projects (branches) and stable versions (tags).
The main code is located under `src` and applications in different sub-folders under `apps`. Under `doc` you will find Doxygen documentation and this tutorial including its code examples. To build the binaries, the GNU build system (Autotools) is used: first, the script `autogen.sh` runs the GNU autotools including `configure` and `make` runs the compilation. The configuration tries to detect the necessary libraries, i.e.

- LAPACK (Linear algebra package) and BLAS (basic linear algebra subprograms), see `www.netlib.org`
- Boost C++ libraries (`boost.org`), we use multithreading and python bindings
- SuiteSparse for solving sparse matrix systems (`http://www.cise.ufl.edu/research/sparse/SuiteSparse/`), we use Cholmod for symmetric matrices

Whereas LAPACK/BLAS and Boost can be made system-wide available on Linux systems using a package manager, SuiteSparse must be downloaded and built by hand. In the folder `external` there is a Makefile that tries to download and build LAPACK, BLAS and SuiteSparse. All libraries should be located in `external/lib`, On Windows systems, you can use ready-compiled dll files.
For reasons of platform-compatibility, our codes are adapted to the GNU compilers but should be working with any compiler suite. Whereas the GNU compiler collection is installed on any Linux or Mac system, on Windows MinGW (Minimalistic GNU for Windows, `www.mingw.org`) should be installed first. For Windows we also recommend to use CodeBlocks as compiler IDE for which we prepared project files (`**.cbp`) in the folder `mingw`.

PyGIMLi

Python (`www.python.org`) is a very powerful and easy-to-use programming language that combines the performance of numerical computation with high-end graphical output and user-interfaces. We recommend PythonXY (`www.pythonxy.com`), a distribution coming along with a lot of modules for scientific computing and development environments such as Spyder. The Python bindings for GIMLi, PyGIMLi, are located in the subfolder `python` make it very comfortable to write GIMLi applications. In the folder `python` you find a build script `build.sh` to build the binaries[2] as well as numerous other functions that can be used from python.

Installation

Independent whether you use pre-compiled or self-build binaries the path to the executables, libraries and python modules must be known using the variables PATH, LD_LIBRARY_PATH[3] and PYTHONPATH.

[2] The packages gccxml, pygccxml and pyplusplus are needen, but can also be retrieved using `buildToolChain.sh` in the folder `buildScripts.sh`
[3] Under windows there is no differentiation and PATH is for both.

1.6. Outline of the document

In the following chapters we like to document how to work with the library using different examples. A very simple example will illustrate how to setup a forward class and make the first inversion - polynomial curve fitting. Then, some basic geophysical examples explain how parameterisations are used and options are set. Three different joint inversion techniques are explained in the section 5. The last section deals with the subject time-lapse inversion. In the directory code there are the code examples used in the following.

2. A very simple example - polynomial curve fitting

2.1. The first program in C++

Example file polyfit0.cpp in the directory doc/tutorial/code/polyfit.
Let us start with the very simple inverse problem of fitting a polynomial curve of degree P

$$f(x) = p_0 + p_1 x + \ldots + p_P x^P = \sum_{i=0}^{P} p_i x^i$$

to some existing data y. The unknown model is the coefficient vector $\mathbf{m} = [p_0, \ldots, p_P]$. The vectorized function for a vector $\mathbf{x} = [\mathbf{x_1}, \ldots, \mathbf{x_N}]^\mathbf{T}$ can be written as matrix-vector product

$$\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} \quad \text{with} \quad \mathbf{A} = \begin{bmatrix} 1 & x_1 & \ldots & x_1^P \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & \ldots & x_N^P \end{bmatrix} = [\mathbf{1} \quad \mathbf{x} \quad \mathbf{x}^2 \ldots \mathbf{x}^P] \quad . \tag{4}$$

We set up the modelling operator, i.e. to return $\mathbf{f}(\mathbf{x})$ for given p_i, as a class derived from the modelling base class. The latter holds the main mimic of generating jacobian, gradients by brute force. The only function to overwrite is `response()`.

```
class FunctionModelling : public ModellingBase {
public:
    /*! constructor, nc: number of coefficients, xvec: abscissae */
    FunctionModelling( int nc, const RVector & xvec, bool verbose=false )
        : ModellingBase( verbose ), x_( xvec ), nc_( nc ){
        regionManager_->setParameterCount( nc ); //! instead of a mesh
    }
    /*! the main thing - the forward operator: returns f(x) */
    RVector response( const RVector & par ){
        RVector y( x_.size(), par[ 0 ] );    //!constant vector of p0
        for ( size_t i = 1; i < nc_; i++ ) //! p1 to pP
            y += pow( x_, i ) * par[ i ];   //! add pi*x^i
        return y;                          //! return sum
    }
    /*! define the startmodel */
    RVector startModel( ){ return RVector( nc_, 0.5 ); }
protected:
    RVector x_; //! abscissa vector x
    int nc_; //! number of coefficients
};
```

In the constructor the $\mathbf{x}$ vector and the number of coefficients are saved as protected variables[4]. The function `setParameterCount` setups the parameterisation as a 0d mesh of `nc` unknowns. The main type used is `RVector`, a vector of real (`double`) values.
We now want to apply the function to the real inversion of data and write a main program.

```
int main( int argc, char *argv [] ){
    int np = 1; /*! maximum polynomial degree fixed to 1 */
```

[4]Usually all variables are denoted with an underscore and declared as protected. Instead of accessing the values directly, we use set and get functions that control the validity of the arguments.

```
    RMatrix xy; //! two-column matrix from file holding x and y
    loadMatrixCol( xy, "datafile.dat" );
    /*! initialise modelling operator */
    FunctionModelling f( np + 1, xy[ 0 ] ); //! first data column
    /*! initialise inversion with data and forward operator */
    RInversion inv( xy[ 1 ], f );
     /*! the problem is well-posed and does not need regularization */
    inv.setLambda( 0 );
    /*! actual inversion run yielding coefficient model */
    RVector coeff( inv.run() );
    /*! save coefficient vector to file */
    save( coeff, "out.vec" );
    /*! exit programm legally */
    return EXIT_SUCCESS;
}
```

The data in the two-column data file is read into a real matrix (**RMatrix** xy;) whose columns can be assessed by xy[i]. We initialise the forward class as defined and the inversion by specifying data and forward operator. Then any options of the inversion can be set, such as the regularization parameter being zero.
Instead of using a fixed polynomial degree and a pre-defined file name we might specify this by the command line in order to have a user-friendly tool. For this, an option map is applied reading the last argument and the optionally defined -n switch.

```
    int np = 1;
    std::string datafile;
    OptionMap oMap;
    oMap.setDescription("Polyfit - fits two-column data with polynomials");
    oMap.addLastArg( datafile, "Datafile" );
    oMap.add( np, "n:", "np", "Number of polynomials" );
    oMap.parse( argc, argv );
    RMatrix xy;
    loadMatrixCol( xy, datafile );
```

The provided datafile `y_2.1x+1.1.dat` holds synthetic noisified data for a linear function. A call `curvefit -n1 y_2.1x+1.1.dat` yields values close to the synthetic ones.

2.2. A first Python program

Example file polyfit.py in the directory doc/tutorial/code/polyfit.
Python is a very flexible language for programming and scripting and has many packages for numerical computing and graphical visualization. For this reason, we built Python bindings and compiled the library pygimli. As a main advantage, all classes can be used and derived. This makes the use of GIMLi very easy for non-programmers. All existing modelling classes can be used, but it is also easy to create new modelling classes.
We exemplify this by the preceding example. First, the library must be imported. To avoid name clashes with other libraries we suggest to import it to an easy name, e.g. by using `import pygimli as g`. As a result, all gimli objects (classes and functions) can be referred to with a preceding g., e.g. g.**RVector** is the real vector **RVector**. Next, the modelling class is derived from ModellingBase, a constructor is defined and the response function is defined.

```python
import pygimli as g
class FunctionModelling( g.ModellingBase ):
    # constructor
    def __init__( self, nc, xvec, verbose = False ):
        g.ModellingBase.__init__( self, verbose )
        self.x_ = xvec
        self.nc_ = nc
        self.regionManager().setParameterCount( nc )
    # response function
    def response( self, par ):
        y = g.RVector( self.x_.size(), par[ 0 ] )
        for i in range( 1, self.nc_ + 1 ):
            y += g.pow( self.x_, i ) * par[ i ];
        return y;
    # start model
    def startModel( self ):
        return g.RVector( self.nc_, 0.5 )
```

The pygimli library must once be imported (in this case under the name g) and all classes (e.g. modelling operators) can be used by g.classname, e.g. g.RVector is the already known vector of real values.

The main program is very easy then and the code is very similar to C++. Data are loaded, both forward operator and inversion are created. Inversion options are set and it the result of run is save to a file. That's it.

```python
    xy = g.RMatrix()
    g.loadMatrixCol( xy, datafile );
    # two coefficients and x-vector (first data column)
    f = FunctionModelling( options.np + 1, xy[ 0 ] )
    # initialize inversion with data and forward operator and set options
    inv = g.RInversion( xy[ 1 ], f );
    # constant absolute error of 0.01 (not necessary, only for chi^2)
    inv.setAbsoluteError( 0.01 );
    # the problem is well-posed and does not need regularization
    inv.setLambda( 0 );
    # actual inversion run yielding coefficient model
    coeff = inv.run();
    g.save( coeff, "out.vec" );
```

As a main advantage of Python, the actual computations can be easily combined with post-processing or visualization, even building graphical user-interfaces. In this code example we use matplotlib, a plotting library inside of pylab, which is comparable to MatLab.

```python
import pylab as P
P.plot( xy[0], xy[1], 'rx', xy[0], inv.response(), 'b-' )
P.show()
```

Similar to C++, command line options can be parsed using the class OptionParser, see the code file. The output is illustrated for two a synthetic function $y = 2.1x + 1.1$ noisified with Gaussian noise for two different orders in Figure 3.

In the following we continue the description with C++ but all are provided as well in Python without significant code changes.

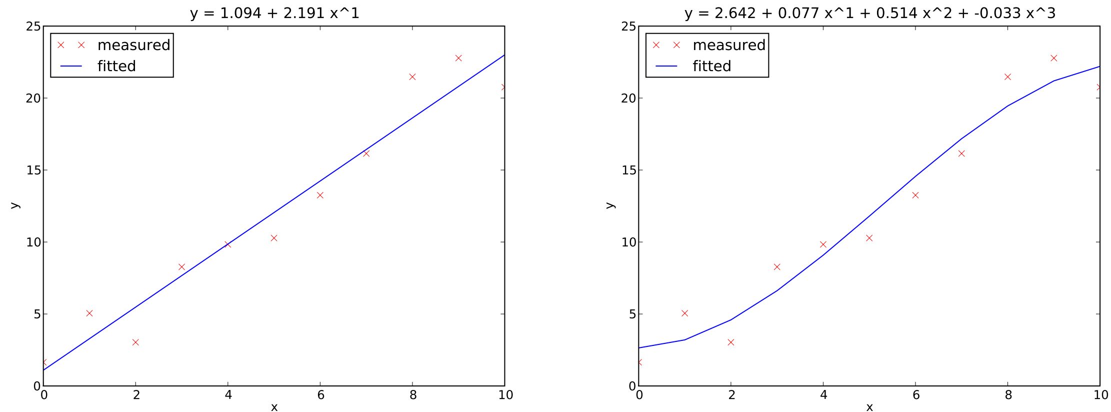

Figure 3: Polynomial fit for noisified synthetic data using first order (left) and third order (right) polynomials.

2.3. An own Jacobian

Example file polyfit1.cpp in the directory doc/tutorial/code/polyfit.

For the latter example, the underlying Gauss-Newton scheme creates a Jacobian matrix by brute force (perturbation). If we want to apply an own algorithm we overwrite the function createJacobian() in the modelling class by using the matrix **A** from (4):

```
void createJacobian( RMatrix & jacobian , const RVector & model ) {
    jacobian.resize( x_.size(), nc_ );
    for ( size_t i = 0 ; i < nc_ ; i++ )
        for ( size_t j = 0 ; j < x_.size() ; j++ )
            jacobian[ j ][ i ] = pow( x_[ j ], i );
}
```

The result of the inversion is of course the same as before. Note that the resize function checks for the right size and allocates space if necessary.

Alternatively we might to use other minimisation methods even though it is not necessary for this example. █(Steepest descent, NLCG, Quasi-Newton)

Note that RInversion is an instance of the template class Inversion < ValueType, MatrixType > with the value type **double** and the matrix type **RMatrix**[5]. One can, of course, use other types, e.g. the complex vector/matrix CVector/CMatrix. For many problems the jacobian matrix has only few entries and can be approximated by a sparse matrix. Therefore the matrix type **RSparseMapMatrix** exists, which is itself an instance of a template type with **long int** index and double values. The corresponding inversion is called RInversionSparse.

[5]RMatrix is a full matrix of real (double) values.

3. General concepts using 1D DC resistivity inversion

See cpp/py files in the directory `doc/tutorial/code/dc1d`.

3.1. Smooth inversion

Example file `dc1dsmooth.cpp`.
Part of the GIMLi library are several electromagnetic 1d forward operators. For direct current resistivity there is a semi-analytical solution using infinite sums that are approximated by Ghosh filters. The resulting function calculates the apparent resistivity of any array for a given resistivity and thickness vector. There are two main parameterisation types:

- a fixed parameterisation where only the parameters are varied
- a variable parameterisation where parameters and geometry is varied

Although for 1d problems the latter is the typical one (resistivity and thickness), we start with the first since it is more general for 2d/3d problems and actually easier. Accordingly, in the file dc1dmodelling.h/cpp two classes called `DC1dRhoModelling` and `DC1dBlockModelling` are defined. For the first we first define a thickness vector and create a mesh using the function `createMesh1d`. The the forward operator is initialized with the data.

```
RMatrix abmnr; loadMatrixCol( abmnr, dataFile ); //! read data
RVector ab2 = abmnr[0], mn2 = abmnr[1], rhoa = abmnr[2]; //! 3 columns
RVector thk( nlay-1, max(ab2) / 2 / ( nlay - 1 ) ); //! const. thickn.
DC1dRhoModelling f( thk, ab2, mn2 ); //! initialise forward operator
```

Note that the mesh generation can also be done automatically using another constructor. However most applications will read or create a mesh from in application and pass it.
By default, the transformations for data and model are identity transformations. We initialise two logarithmic transformations for the resistivity and the apparent resistivity by

```
RTransLog transRho;
RTransLog transRhoa;
```

Alternatively, we could set lower/upper bounds for the resistivity using

```
RTransLogLU transRho( lowerbound, upperbound);
```

Appendix D gives an overview on available transformation functions.
Next, the inversion is initialized and a few options are set

```
RInversion inv( data.rhoa(), f, verbose );
inv.setTransData( transRhoa );              //! data transform
inv.setTransModel( transRho );              //! model transform
inv.setRelativeError( errPerc / 100.0 ); //! constant relative error
```

A starting model of constant values (median apparent resistivity) is defined

```
RVector model( nlay, median( data.rhoa() ) ); //! constant vector
inv.setModel( model );                         //! starting model
```

Finally, the inversion is called and the model is retrieved using `model = inv.run();`
A very important parameter is the regularisation parameter λ that controls the strength of the smoothness constraints (which are the default constraint for any 1d/2d/3d mesh). Whereas w^c and w^m are dimensionless and 1 by default, λ has, after eq. (3), the reciprocal and squared unit of m and can thus have completely different values for different problems[6]. However, since often the logarithmic transform is used, the default value of $\lambda = 20$ is often a first guess. Other values are set by

```
inv.setLambda( lambda ); //! set regularisation parameter
```

In order to optimise λ, the L-curve (Günther et al., 2006; Günther, 2004) can be applied to find a trade-off between data fit and model roughness by setting `inv.setOptimizeLambda(true);`. For synthetic data or field data with well-known errors we can also call `model = inv.runChi1();`, which varies λ from the starting value such that the data are fitted within noise ($\chi^2 = 1$). We created a synthetic model with resistivities of 100(soil)-500(unsaturated)-20(saturated)-1000(bedrock) Ωm and thicknesses of 0.5, 3.5 and 6 meters. A Schlumberger sounding with AB/2 spacings from 1.0 to 100 m was simulated and 3% noise were added. Data format of the file `sond1-100.dat` is the unified data format[7].

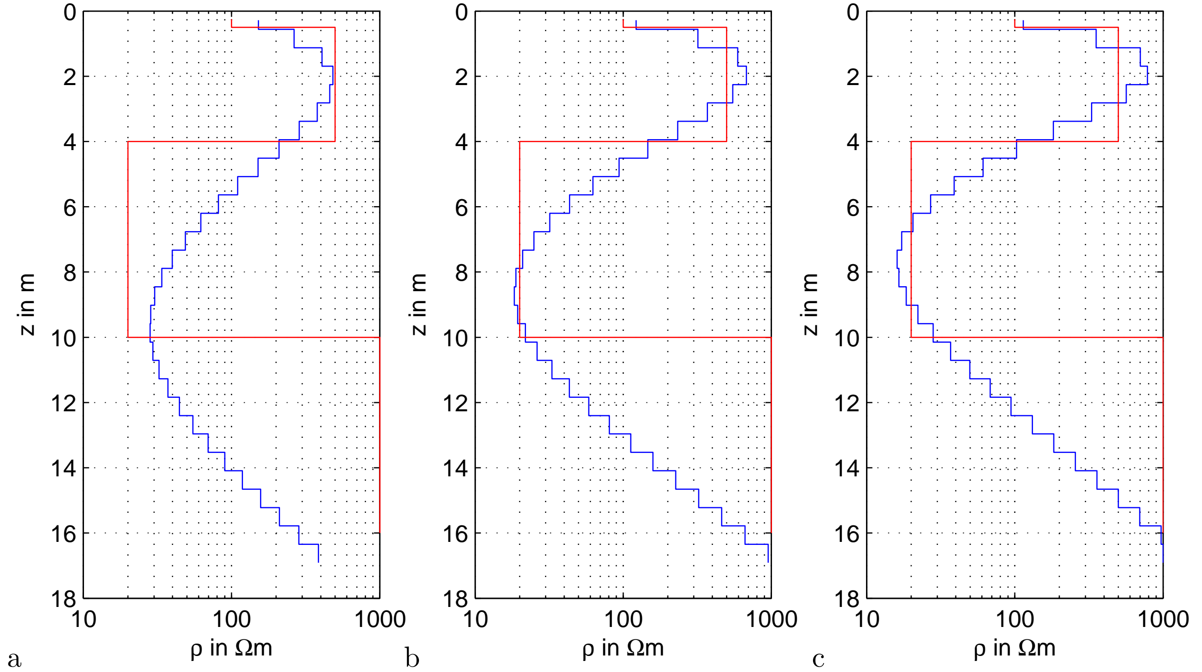

Figure 4: Smooth 1d resistivity inversion results for a) $\lambda = 200 \Rightarrow \chi^2 = 11.1$/rrms=10.1%, b) $\lambda = 20 \Rightarrow \chi^2 = 1.2$/rrms=3.3%, and c) $\lambda = 2 \Rightarrow \chi^2 = 0.6$/rrms=2.4%, red-synthetic model, blue-estimated model

Figure 4 shows the inversion result for the three different regularisation parameters 300, 30 and 3. Whereas the first is over-smoothed, the other are much closer at the reality. The rightmost figure over-fits the data ($\chi^2 = 0.3 < 1$) but is still acceptable. The L-curve method yields a

[6] The regularisation parameter has therefore to be treated logarithmically.
[7] See `www.resistivity.net?unidata` for a description.

value of $\lambda = 2.7$, which is too low. However, if we apply the χ^2-optimization we obtain a value of $\lambda = 15.2$ and with it the data are neither over- nor under-fitted.

3.2. Block inversion

Example file `dc1dblock.cpp` *in the directory* `doc/tutorial/code/dc1d`.
Alternatively, we might invert for a block model with unknown layer thickness and resistivity. We change the mesh generation accordingly and use the forward operator `DC1dModelling`:

```
DC1dModelling f( nlay, ab2, mn2 );
```

`createMesh1DBlock` creates a block model with two regions[8]. Region 0 contains the thickness vector and region 1 contains the resistivity vector. There is a region manager as a part of the forward modelling class that administrates the regions. We first define different transformation functions for thickness and resistivity and associate it to the individual regions.

```
RTransLog transThk;
RTransLogLU transRho( lbound, ubound );
RTransLog transRhoa;
f.region( 0 )->setTransModel( transThk );
f.region( 1 )->setTransModel( transRho );
```

For block discretisations, the starting model can have a great influence on the results. We choose the median of the apparent resistivities and a constant thickness derived from the current spread as starting values.

```
double paraDepth = max( ab2 ) / 3;
f.region( 0 )->setStartValue( paraDepth / nlay / 2.0 );
f.region( 1 )->setStartValue( median( rhoa ) );
```

For block inversion a scheme after Marquardt (1963), i.e. a local damping of the changing without interaction of the model parameters and a decreasing regularisation strength is favourable.

```
inv.setMarquardtScheme( 0.9 ); //! local damping with decreasing lambda
```

The latter could also be achieved by

1. setting the constraint type to zero (damping) by `inv.setConstraintType(0)`
2. switching to local regularization by `inv.setLocalRecularization(true)`
3. defining the lambda decreasing factor by `inv.setLambdaDecrease(0.9)`

With the default regularization strength $\lambda = 20$ we obtain a data fit slightly below the error estimate. The model (Fig. 5a) clearly shows the four layers (blue) close to the synthetic model (red).

[8] With a second argument `createMesh1DBlock` can create a block model with thickness and several parameters for a multi-parameter block inversion.

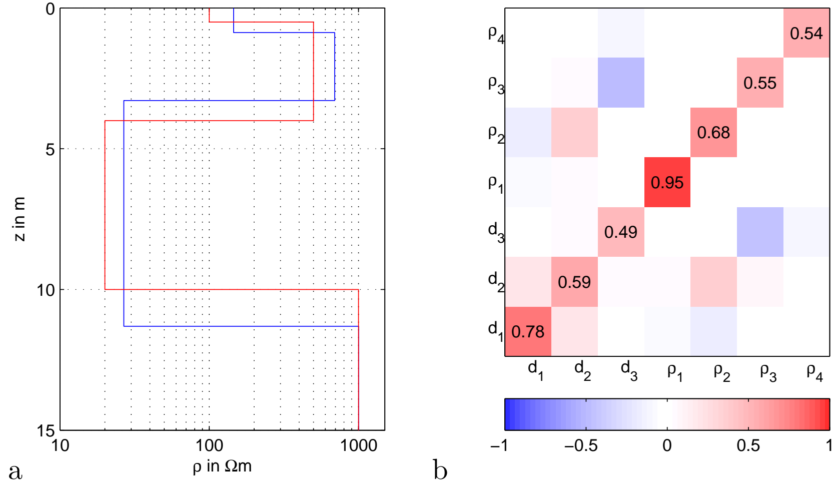

Figure 5: a) Block 1d resistivity inversion result (red-synthetic model, blue-estimated model)) and b) resolution matrix

3.3. Resolution analysis

One may now be interested in the resolution properties of the individual model parameters. The resolution matrix $\mathbf{R}^M$ defines the projection of the real model onto the estimated model:

$$\mathbf{m}^{est} = \mathbf{R}^M \mathbf{m}^{true} + (\mathbf{I} - \mathbf{R}^M)\mathbf{m}^R + \mathbf{S}^\dagger \mathbf{D}\mathbf{n} \quad , \tag{5}$$

(Günther, 2004) where $\mathbf{S}^\dagger \mathbf{D}\mathbf{n}$ represents the generalised inverse applied to the noise. Note that $\mathbf{m}^R$ changes to $\mathbf{m}^k$ for local regularisation schemes (Friedel, 2003).
Günther (2004) also showed that the model cell resolution (discrete point spread function) can be computed by solving an inverse sub-problem with the corresponding sensitivity distribution instead of the data misfit. This is implemented in the inversion class by the function modelCellResolution(iModel) where iModel is the number of the model cell. This approach is feasible for bigger higher-dimensional problems and avoids the computation of the whole resolution matrix. A computation for representative model cells can thus give insight of the resolution properties of different parts of the model.
For the block model we successively compute the whole resolution matrix.

```
RVector resolution( nModel ); //! create single resolution vector
RMatrix resM;                 //! create empty matrix
for ( size_t iModel = 0; iModel < nModel; iModel++ ) {
    resolution = inv.modelCellResolution( iModel );
    resM.push_back( resolution ); //! push back the single vector
}
save( resM, "resM" ); //! save resolution matrix
```

In Figure 5 the model resolution matrix is shown and the diagonal elements are denoted. The diagonal elements show that the resolution decreases with depth. The first resistivity is resolved nearly perfect, whereas the other parameters show deviations from 1. ρ_2 is positively connected with d_2, i.e. an increase of resistivity can be compensated by an increased resistivity. For ρ_3 and d_3 the correlation is negative. These are the well known H- and T-equivalences of

thin resistors or conductors, respectively, and show the equivalence of possible models that are able to fit the data within noise.

3.4. Structural information

Example file `dc1dsmooth-struct.cpp` *in the directory* `doc/tutorial/code/dc1d`.
Assume we know the ground water table at 4 m from a well. Although we know nothing about the parameters, this structural information should be incorporated into the model. We create a thickness vector of constant 0.5 m. The first 8 model cells are located above the water table, so the 8th boundary contains the known information. Therefore we set a marker different from zero (default) after creating the mesh

```
//! variant 1: set mesh (region) marker
f.mesh()->boundary( 8 ).setMarker( 1 );
```

This causes the boundary between layer 8 and 9 being disregarded, the corresponding w^c_8 is zero and allows for arbitrary jumps in the otherwise smoothed model. Figure 6 shows the result, at 4 m the resistivity jumps from a few hundreds down to almost 10.

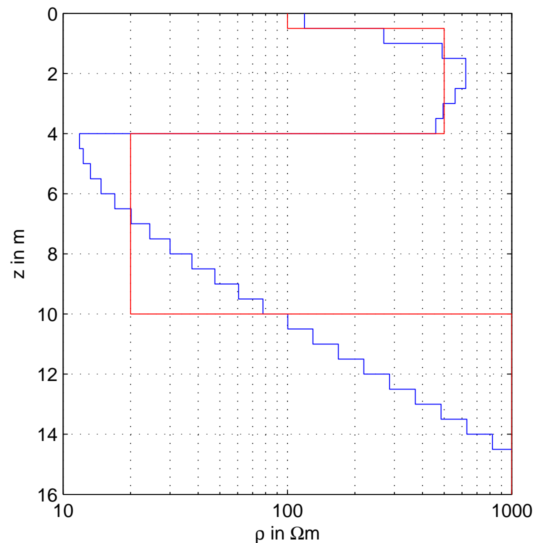

Figure 6: Inversion result with the ground water table at 4 m as structural constraint.

Note that we can set the weight to zero also directly, either as a property of the inversion

```
//! variant 2: application of a constraint weight vector to inversion
RVector bc( inv.constraintsCount(), 1.0 );
bc[ 6 ] = 0.0;
inv.setCWeight( bc );
```

or the (only existing) region.

```
//! variant 3: application of a boundary control vector to region
RVector bc( f.regionManager().constraintCount(), 1.0 );
bc[ 7 ] = 0.0;
f.region( 0 )->setConstraintsWeight( bc );
```

Of course, in 2d/3d inverse problems we do not set the weight by hand. Instead, we put an additional polygon (2d) or surface (3d) with a marker $\neq 0$ into the PLC before the mesh generation. By doing so, arbitrary boundaries can be incorporated as known boundaries.

3.5. Regions

Example file `dc1dsmooth-region.cpp` *in the directory* `doc/tutorial/code/dc1d`.
In the latter sections we already used regions. A default mesh contains a region with number 0. A block mesh contains a region 0 for the thickness values and regions counting up from 1 for the individual parameters. Higher dimensional meshes can be created automatically by using region markers, e.g. for specifying different geological units.
In our case we can divide part above and a part below water level. In 2d or 3d we would, similar to the constraints above, just put a region marker $\neq 0$ into the PLC and the mesh generator will automatically associate this attribute to all cells in the region.
Here we set the markers of the cells ≥ 8 to the region marker 1.

```
Mesh * mesh = f.mesh();
mesh->boundary( 8 ).setMarker( 1 );
for ( size_t i = 8; i < mesh->cellCount(); i++ )
    mesh->cell( i ).setMarker( 1 );
```

Now we have two regions that are decoupled automatically. The inversion result is identical to the one in Figure 6. However we can now define the properties of each region individually. For instance, we might know the resistivities to lie between 80 and 800 Ωm above and between 10 and 100 Ωm below. Consequently we define two transformations and apply it to the regions.

```
RTransLogLU transRho0( 80, 800 );
RTransLogLU transRho1( 10, 1000 );
f.region( 0 )->setTransModel( transRho0 );
f.region( 1 )->setTransModel( transRho1 );
```

Additionally we might try to improve the very smooth transition between groundwater and bedrock. We decrease the model control (strength of smoothness) in the lower region by a factor of 10.

```
f.region( 1 )->setModelControl( 0.1 );
```

The result is shown in Figure 7. The resistivity values are much better due to adding information about the valid ranges. Furthermore the transition zone in the lower region is clear.

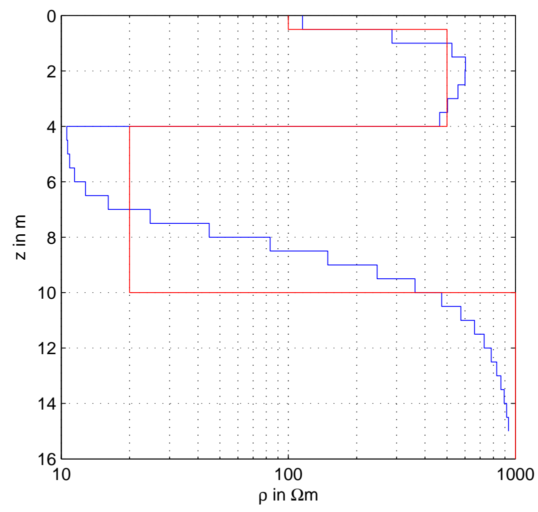

Figure 7: Inversion result using two regions of individual range constraint transformations and regularization strength (model control).

4. Enhanced techniques

4.1. Combining different data types - MT 1d inversion

File `doc/tutorial/code/enhanced/mt1dinv0.cpp`
In magnetotelluric (MT) inversion for every period an amplitude ρ^a and a phase ϕ is computed from the electric and magnetic registrations. The file `1000_100_1000_n5_1.dat` contains a synthetic three layer case of 1000Ωm-100Ωm-1000Ωm with 5 % Gaussian noise on the ρ^a and 1 degree on the phases. The three columns T, ρ^a and ϕ are read extracted as vectors by

```
RMatrix TRP; //! real matrix with period(T), resistivity(R) & phase(P)
loadMatrixCol( TRP, dataFileName ); //! read column-based file
size_t nP = TRP.rows(); //! number of data
RVector T( TRP[0] ), rhoa( TRP[1] ), phi( TRP[2] ); //! columns
```

It is based on the forward operator MT1dModelling in `src/em1dmodelling.h/cpp`, giving back a vector that consists of the amplitudes and phases for each period. Of course both have completely different valid ranges. The phases a linearly related between 0 and $\pi/2$, whereas the amplitudes are usually treated logarithmically. On the (1d block) model side, thickness and apparent resistivity use log or logLU transforms as done for DC resistivity.

```
/*! Model transformations: log for resistivity and thickness */
RTransLog transThk;
RTransLogLU transRho( lbound, ubound );
/*! Data transformations: log apparent resistivity, linear phases */
RTransLog transRhoa;
RTrans transPhi;
```

Since amplitudes and phases are combined in one vector, we create a cumulative transformation of the two by specifying their lengths. Similarly, the assumed relative error of the ρ^a and ϕ are combined using a cat command.

```
CumulativeTrans< RVector > transData; //! combination of two trans
transData.push_back( transRhoa, nP ); //! append rhoa trans (length nP)
transData.push_back( transPhi, nP ); //! append phi trans
RVector error( cat( RVector(nP,errRhoa/100), RVector(errPhase/phi) ) );
```

Similar to DC resistivity inversion, we create a 1d block mesh, initialise the forward operator and set up options for the two regions (0-thicknesses,1-resistivities). Starting values for the ρ_i and d_i are computed by the mean apparent resistivity and an associated skin depth.

```
MT1dModelling f( T, nlay, false );
double medrhoa = median( rhoa ); //! median apparent resistivity
double medskindepth = sqrt( median( T ) * medrhoa ) * 503.0; //!skin d.
f.region( 0 )->setTransModel( transThk ); //! transform
f.region( 1 )->setTransModel( transRho );
f.region( 0 )->setConstraintType( 0 ); //! min. length
f.region( 1 )->setConstraintType( 0 );
f.region( 0 )->setStartValue( medskindepth / nlay );
f.region( 1 )->setStartValue( medrhoa );
/*! Real valued inversion with combined rhoa/phi and forward op. */
RInversion inv( cat( rhoa, phi ), f, verbose, dosave );
```

The rest is done as for DC resistivity block inversion. In Figure 8 the inversion result and its resolution matrix is shown. The model is very close to the synthetic one and represents an equivalent solution. This is also represented by the resolution matrix, where ρ_1, ρ_3 and d_1 are resolved almost perfectly, whereas ρ_2 and d_2 show slight deviations.

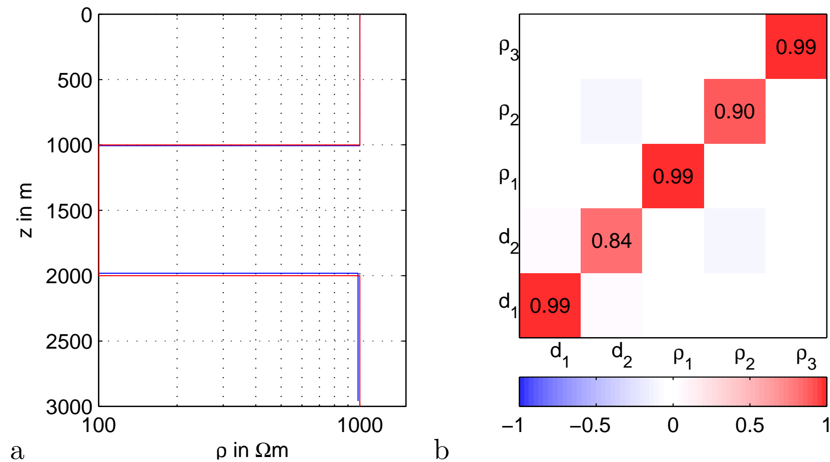

Figure 8: a) Block 1d resistivity inversion result (red-synthetic model, blue-estimated model)) and b) resolution matrix

One can easily test the inversion only based on ρ^a or ϕ by increasing the errors of the others by a large factor. According to (1) the corresponding weight goes to zero. By skipping ϕ the model deviates slightly and the cell resolutions for ρ_2 and d_2 decrease to 0.9 and 0.86, respectively. If the ρ^a are neglected the solution becomes obviously non-unique. Only d_1 is determined pretty well, the other parameters obtain cell resolutions of 0.6-0.8. Note also that for local regularisation the resolution does not contain the resolution of the preceding models (Friedel, 2003).

4.2. Combining different parameter types - offsets in travel time

Next, we consider a 2d travel-time tomographic problem. In the library there is a forward operator called `TTDijkstraModelling`, which is using a Dijkstra (Dijkstra, 1959) algorithm that restricts the ray paths to mesh edges. Although this is only an approximation, it is sufficiently accurate for high-quality meshes. Assume the zero point (shot) of the traces is not exactly known. This might be due to long trigger cables, problems in the device or placing besides the profile. Aim is to include an unknown delay for each shot position into the inversion[9].
First, we derive a new forward modelling class `TTOffsetModelling` from the existing `TTDijkstraModelling` (abbreviated by `TTMod`) since we want to use their functionality. Additionally to the existing class we need the number of shot positions and a simple 0d/1d mesh holding the offset values for them added to the original mesh.

```
class TTOffsetModelling : public TTMod {
public:
```

[9]A similar problem is the issue of static shift in MT inversion caused by local conductivity inhomogeneity, which shifts the apparent resistivity curves.

```
    TTOffsetModelling( Mesh & mesh, DataContainer & data )
      : TTMod( mesh, data ) {
        //! find occuring shots, and map them to indices starting from zero
        shots_ = unique( sort( dataContainer.get("s") ) );
        std::cout << "found " << shots_.size() << " shots." << std::endl;
        for ( size_t i = 0 ; i < shots_.size() ; i++ )
            shotMap_.insert( std::pair< int, int >( shots_[ i ], i ) );
        //! create new region containing offsets with special marker
        offsetMesh_ = createMesh1D( shots_.size() );
        for ( size_t i = 0 ; i < offsetMesh_.cellCount() ; i++ )
            offsetMesh_.cell( i ).setMarker( NEWREGION );
        regionManager().createRegion( NEWREGION, offsetMesh_ );
    }
...
```

The two functions response and createJacobian need to be overwritten. However we want to expand the original functions by the changes needed. This is straight forward for the response vector. First part of the model is the slowness vector whose response is calculated calling the original function.

```
    RVector TTOffsetModelling::response( const RVector & model ){
        //! extract slowness from model and call old function
        RVector slowness( model, 0, model.size() - shots_.size() );
        RVector offsets( model, model.size() - shots_.size(), model.size());
        RVector resp = TTMod::response( slowness ); //! normal response
        RVector shotpos = dataContainer_->get( "s" );
        for ( size_t i = 0; i < resp.size() ; i++ ){
            resp[ i ] += offsets[ shotMap_[ shotpos[ i ] ] ];
        }
        return resp;
    }
```

For the Jacobian the case is a bit more complicated. Instead of increasing the size of the matrix a-posteriori, we use a block matrix type H2SparseMapMatrix consisting of two horizontally concatenated matrices called by H1() and H2() [10]. The first is the normal way matrix holding the path lengths. The second is a matrix with a value of 1 in the position of the shot number and 0 elsewhere.

```
    RVector TTOffsetModelling::createJacobian( H2Matrix & jacobian,
                                               const RVector & model ){
        //! extract slowness from model and call old function
        RVector slowness( model, 0, model.size() - shots_.size() );
        RVector offsets( model, model.size() - shots_.size(), model.size());
        TTMod::createJacobian( jacobian.H1(), slowness );
        jacobian.H2().setRows( dataContainer_->size() );
        jacobian.H2().setCols( offsets.size() );
        //! set 1 entries for the used shot
        RVector shotpos = dataContainer_->get( "s" );
        for ( size_t i = 0; i < dataContainer_->size(); i++ ) {
            jacobian.H2().setVal( i, shotMap_[ shotpos[ i ] ], 1.0 );
        }
```

[10]See appendix on matrices E for existing matrix types.

```
}
```

As a result the model vector holds both slowness values and the offsets, which can be sliced out of the vector for individual post-processing.

4.3. What else?

- Full waveform TDR inversion?
- Gravity 2d or 3d inversion?
- what is enhanced?

5. Joint inversion

The term joint inversion denotes the simultaneous inversion of a number of different data types. We can classify different types according to the relation of the associated parameters:

1. Identical parameters is historical the classical joint inversion, e.g. both DC and EM aim at ρ.
2. Parameters are indirectly connected by petrophysical relations, e.g. ERT and GPR both aiming at water content.
3. Independent parameters. In this case only the structures can be linked to each other. For simple models this can involve the inversion of the geometry. For fixed models structural information is exchanged[11].

5.1. Classical joint inversion of DC and EM soundings

File `doc/tutorial/code/joint/dcem1dinv.cpp`
First, let us consider to jointly invert different electromagnetic methods, e.g. direct current (DC) and Frequency Domain Electromagnetic (FDEM). For the latter we assume a two-coil system in horizontal coplanar model with 10 frequencies between 110 Hz and 56 kHz. Whereas DC resistivity yields apparent resistivities, the FDEM data are expressed as ratio between secondary and primary field in per cent.
The two ready methods DC1dModelling and FDEM1dModelling are very easily combined since they use the same block model. In the response function the two vectors are combined. We create a new modelling class that derives from the base modelling class[12] and has two members of the individual classes, which must be initialized in the constructor. Alternatively we could derive from one of the two classes and use only a member of the other.

```
class DCEM1dModelling : public ModellingBase {
public:
    DCEM1dModelling( size_t nlay, RVector & ab2, RVector & mn2,
                     RVector & freq, double coilspacing, bool verbose ) :
        ModellingBase( verbose ), // base constructor
        fDC_( nlay, ab2, mn2, verbose ), // FDEM constructor
        fEM_( nlay, freq, coilspacing, verbose ) { // DC constructor
          setMesh( createMesh1DBlock( nlay ) ); // create mesh
    }
    RVector response( const RVector & model ){ // paste together responses
        return cat( fDC_.response( model ), fEM_.response( model ) );
    }
protected:
    DC1dModelling fDC_;
    FDEM1dModelling fEM_;
};
```

In the response function both response functions are called and combined using the cat command. We set the usual transformation (log for apparent resistivity and logLU for the resistivity) and inversion (Marquardt scheme) options as above. In case of identical responses

[11] Note that this type is formally a structurally coupled cooperative inversion.
[12] In order to use the classes, dc1dmodelling.h and em1dmodelling.h have to be included.

(e.g. apparent resistivities) this would be the whole thing. Here we have to care about the different data types (cf. section 4.1), i.e. always positive, log-distributed ρ_a from DC and possibly negative, linearly distributed, relative magnetic fields. The transformations are again combined using CumulativeTrans

```
RTransLog transRhoa;
RTrans transEM;
CumulativeTrans< RVector > transData;
transData.push_back( transRhoa, ab2.size() );
transData.push_back( transEM, freq.size() * 2 );
```

In the code we create a synthetic model synthModel, calculate the forward response and noisify it by given noise levels.

```
/*! compute synthetic model (created before) by calling f */
RVector synthData( f( synthModel ) );
/*! error models: relative percentage for DC, absolute for EM */
RVector errorDC = synthData( 0, ab2.size() ) * errDC / 100.0;
RVector errorEM( freq.size() * 2, errEM );
RVector errorAbs( cat( errorDC, errorEM ) );
/*! noisify synthetic data using the determined error model */
RVector rand( synthData.size() );
randn( rand );
synthData = synthData + rand * errorAbs;
```

The inversion is converging to a χ^2 value of about 1, i.e. we fit the data within error bounds. Finally a resolution analysis is done to determine how well the individual parameters (2 thicknesses and 3 resistivities) are determined. We can compare it with single inversions by drastically increasing the error level for one of the methods by a factor of 10. Table 1 shows the resulting diagonal values of the resolution matrix for a three-layer model. The first layer is well resolved in all variants except the first layer resistivity for EM. Considering the values for the other resistivities we can clearly see that EM is detecting the good conductor and DC describes the resistor as expected from the theory.

Method	d_1=20 m	d_2=20 m	ρ_1=200 Ωm	ρ_2=10 Ωm	ρ_3=50 Ωm
Joint inversion:	0.98	0.46	0.98	0.67	0.57
EM dominated:	0.97	0.36	0.71	0.66	0.20
DC dominated:	0.96	0.21	0.97	0.32	0.62

Table 1: Resolution measures for Joint inversion and quasi-single inversions using an error model increased by a factor of 10.

5.2. Block joint inversion of DC/MRS data

If the underlying parameters of the jointed inversion are independent, a combination can only be achieved via the geometry. For the case of a block 1d discretization both methods are affected by the layer thicknesses.
Similar to the above example, we create a combined modelling class that is derived from one method, in this case MRS1DBlockModelling. This one is a block model (water content and

thickness) variant of the magnetic resonance sounding (MRS) modelling MRSModelling. The class has a DC resistivity forward modelling and holds the number of layers nlay.
The model is created in the constructor using createMesh1DBlock(nlay, 2) that is able to hold, additionally to the thickness (region 0), multiple properties, here water content (region 1) and resistivity (region 2). From the model vector the thickness, water content (or their combination) and resistivity has to be extracted and the result of the two forward calls are combined using the cat command. The Jacobian is by default brute force, which is quite cheap for block models.

```
class DC_MRS_BlockModelling : public MRS1dBlockModelling{
public:
    DC_MRS_BlockModelling( size_t nlay, DataContainer & data, RMatrix & KR,
                           RMatrix & KI, RVector & zvec, bool verbose ) :
        MRS1dBlockModelling( nlay, KR, KI, zvec, verbose ), nl_( nlay ) {
            setMesh( createMesh1DBlock( nlay, 2 ) ); //two-properties
            Mesh mymesh = createMesh1DBlock( nlay ); //single block mesh
            fDC_ = new DC1dModelling( mymesh, data, nlay, verbose );
        }
    virtual ~DC_MRS_BlockModelling( ){ delete fDC_; }

    RVector response( const RVector & model ){
        //! extract resistivity , watercontent & thickness from model vec
        RVector thk( model, 0 , nl_ - 1 );
        RVector wcthk( model, 0 , nl_ * 2 - 1 );
        RVector res( model, nl_ * 2 - 1 , nl_ * 3 - 1 );
        return cat( MRS1dBlockModelling::response( wcthk ),
                    fDC_->rhoa( res, thk ) );
    }
protected:
    DC1dModelling *fDC_;
    int nl_;
};
```

In order to use the class, we have to build a cumulative data transform as in subsection 4.1. Model transformations are logarithmic to ensure positive values, additionally an upper bound of 0.4 is defined for the water content.

```
    RTrans transVolt;     // linear voltages
    RTransLog transRhoa; // logarithmic apparent resistivities
    CumulativeTrans< RVector > transData;
    transData.push_back( transVolt, errorMRS.size() );
    transData.push_back( transRhoa, dataDC.size() );
    RTransLog transRes;
    RTransLogLU transWC(0.0, 0.4);
    RTransLog transThk;
```

In order to achieve a Marquardt inversion scheme, the constraint type is set to zero for all regions:

```
    f.regionManager().setConstraintType( 0 );
```

Appropriately, the transformations and starting values are set.

```
    f.region( 0 )->setTransModel( transThk );
    f.region( 0 )->setStartValue( 5.1 );
    f.region( 1 )->setTransModel( transWC );
    f.region( 1 )->setStartValue( 0.1 );
    f.region( 2 )->setTransModel( transRes );
    f.region( 2 )->setStartValue( median( dataDC.rhoa() ) );
```

We use a synthetic model of three layers representing a vadoze zone ($\rho = 500\Omega$m, 0.1% water, 4m thick), a saturated zone ($\rho = 100\Omega$m, 40% water content, 10m thick) and a bedrock ($\rho = 2000\Omega$m, no water). 3 % and 20 nV Gaussian noise are added to the DC and MRS data, respectively. Figure 9 shows the result of the combined inversion, which is very close to the synthetic model due to the improved information content from two models.

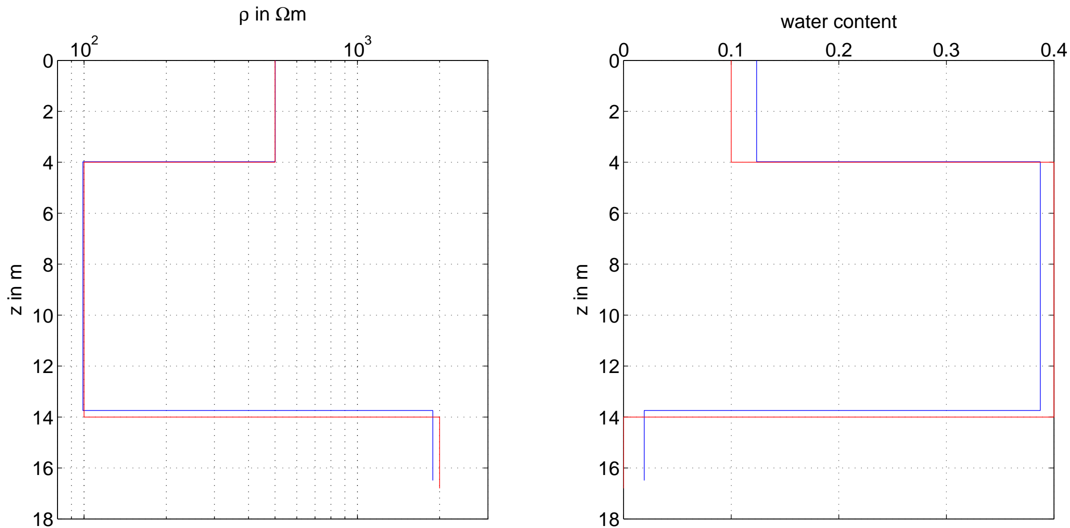

Figure 9: Joint inversion result of block-coupled DC resistivity (left) and MRS (right) sounding.

5.3. Structurally coupled cooperative inversion of DC and MRS soundings

File `doc/tutorial/code/joint/dc_mrs_joint1d.cpp`
In many cases it is not clear whether the model boundaries observed by different methods are identical or how many of them are existing. Nevertheless we expect a similarity of the structure, i.e. the gradients. On smooth model discretizations of any dimension the exchange of geometrical information can be achieved using the constraint control function (Günther and Rücker, 2006). Main idea is to decrease the weight for the roughness operator of one method depending on the partial derivative of the other. A large roughness as a possible interface should enable the interface on the other side by a low weight. There are different possible functions for doing so. Originally, a iteratively reweighted least squares scheme was proposed that incorporates the whole distribution. Here we use a simple function

$$w_c(r) = \frac{a}{|r| + a} + a \tag{6}$$

where w_c is the weight, r is the roughness value and a is a small quantity. For $r \to 0$ the weight $w_c = 1 + a$ lies slightly above 1, for $r \to \infty$ it becomes a.

In this case we apply it to DC resistivity and MRS sounding for a smooth 1d model. The latter operator is linear and thus realized by a simple matrix vector multiplication of the kernel function and the water content vector. We initialise the two independent inversions and run one iteration step each. In the iteration loop we calculate the function of one roughness and set it as constraint weight for the other before running another inversion step.

```
invMRS.setMaxIter( 1 );
invDC.setMaxIter( 1 );
invMRS.run(); //! init and run 1 step
invDC.run(); //! init and run 1 step
double a = 0.1;
RVector cWeight( nlay - 1 );
for ( int iter = 1; iter < maxIter; iter++ ) {
    cWeight = a / ( abs( invDC.roughness() ) + a ) + a;
    invMRS.setCWeight( cWeight );
    cWeight = a / ( abs( invMRS.roughness() ) + a ) + a;
    invDC.setCWeight( cWeight );
    invDC.oneStep();
    invMRS.oneStep();
}
```

Figure 10 shows the inversion result for the above mentioned three-layer case. Without coupling (a) the transitions between the layers are quite smooth. Much more significant jumps in both parameters occur when structural coupling is applied (b) and make the interpretation of both layer thickness and representative parameters less ambiguous.

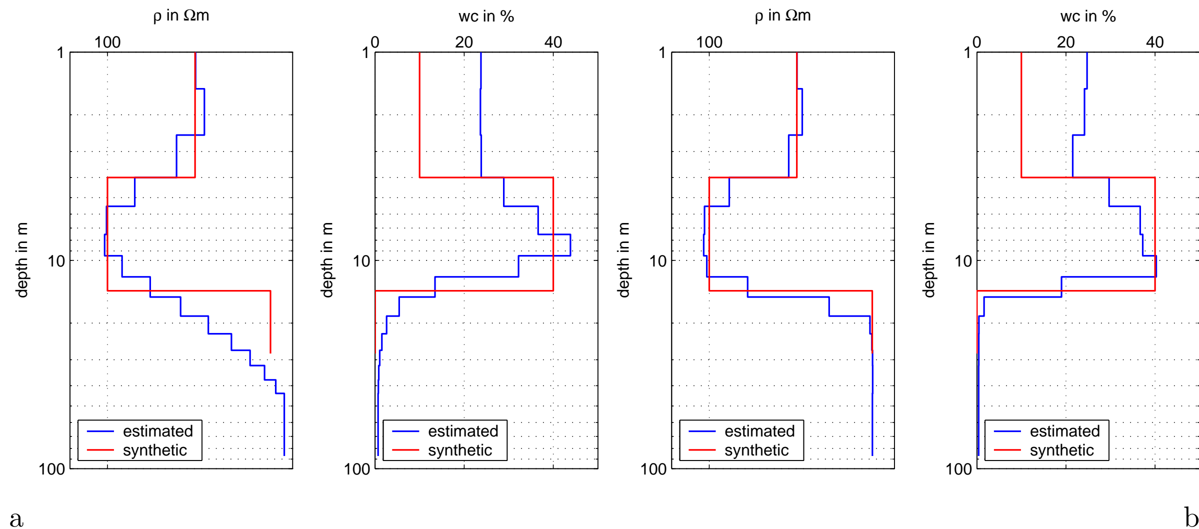

Figure 10: Synthetic model (red) and inversion results (blue) for DC (left) and MRS (right) 1D inversion without (a) and with (b) structural coupling

Of course the coupling does not have to affect the whole model. The constraint weight vector can as well be set for an individual region such as the aquifer. See inversion templates on how to do structural coupled inversion more easily and flexibly.

5.4. Petrophysical joint inversion

Target: water content in a soil column using GPR (CRIM equation) and DC (Archie equation) ▮TO BE IMPLEMENTED

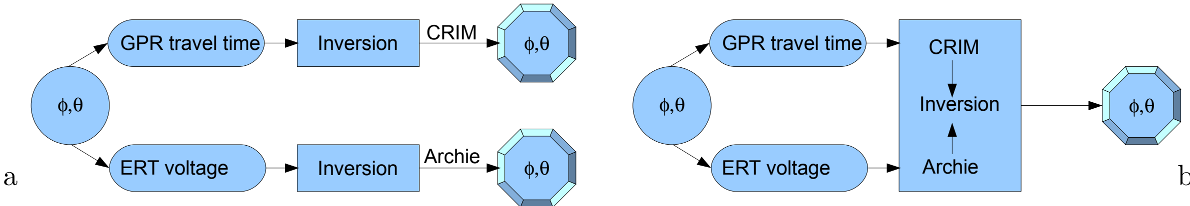

Figure 11: Scheme for separate inversion (a) and petrophysical joint inversion (b) of GPR and ERT data to obtain an image of porosity or water content

6. Inversion templates

There are certain tasks that go beyond a classical inversion scheme but still are method-independent. They can be formulated more generally in order to be applied to a wider range of applications. There are several cases

Roll-along inversion: Geophysical data are often acquired along profiles. Since numerical work usually goes by N^2, it can be much more efficient to do the work piece-wise. However, the continuity must be ensured.

Joint inversion: As described above, the different joint inversions do not have to be programmed individually. Furthermore, more than two methods can be coupled.

Time-lapse inversion: Although there are several approaches for efficient inversion along the time axis, they are method-independent and can be formulated generally.

6.1. Roll-along inversion

▮BLA

6.2. Joint inversion

Assume we have inversions invA, invB and invC, which (or parts of which) are to be coupled.

Block inversion

Block inversion: mesh1d containing thickness and several parameters. ▮TO BE IMPLEMENTED

Structural coupling

As described in section 5.3, the structural coupling consists mainly some preparation steps and a main iteration for the coupling that does individual runs and coupling.
Generally, there can be two possibilities for each inversion: (i) to couple the whole inversion model (as above) or (ii) to couple only one specified region. For example, structures in an aquifer are to be imaged by cross-hole ERT and GPR. Whereas for the latter it is sufficient to restrict to the aquifer, for ERT the unsaturated zone and a clay layer must be taken into account.
For more than 2 inversions the coupling must be generalized: One can imaging a chain (or ring) or a star scheme coupling. Whereas in the first the inversions are coupled pair-wise, in the latter the constraint weights are mixed.

```
SCCInversion SCC;
SCC.append( invA );
SCC.append( invB, regionnumber );
SCC.run();
```

Options: type of coupling, e.g. IRLS scheme or self-defined function. Ring or star connection. Terminating criterion.
▮TO BE IMPLEMENTED

6.3. Time lapse inversion

Strategies

Example

References

Davis, T. A. (2006). *Direct Methods for Sparse Linear Systems.* SIAM Book Series on the Fundamentals of Algorithms. SIAM, Philadelphia.

Dijkstra, E. W. (1959). *Numerische Mathematik*, chapter 1, pages 269–271.

Friedel, S. (2003). Resolution, stability and effiency of resistivity tomography estimated from a generalized inverse approach. *Geophys. J. Int.*, 153:305–316.

Günther, T. (2004). *Inversion Methods and Resolution Analysis for the 2D/3D Reconstruction of Resistivity Structures from DC Measurements.* PhD thesis, University of Mining and Technology Freiberg. available at `http://fridolin.tu-freiberg.de`.

Günther, T., Müller-Petke, M., Hertrich, M., and Rücker, C. (2008). The role of transformation functions in geophysical inversion. In *Ext. Abstract, EAGE Near Surface Geophysics Workshop. 15.-17.9.08, Krakow (Poland).*

Günther, T. and Rücker, C. (2006). A general approach for introducing information into inversion and examples from dc resistivity inversion. In *Ext. Abstract, EAGE Near Surface Geophysics Workshop. 4.-6.9.06, Helsinki(Finland).*

Günther, T., Rücker, C., and Spitzer, K. (2006). 3-d modeling and inversion of DC resistivity data incorporating topography - Part II: Inversion. *Geophys. J. Int.*, 166(2):506–517.

Haber, E. (2005). Quasi-Newton methods for large-scale electromagnetic inversion problems. *Inverse Problems*, 21:305–323.

Marquardt, D. W. (1963). An algorithm for least-squares estimation of non-linear parameters. *J. Soc. Ind. App. Math.*, 11:431–441.

Shewchuk, J. R. (1996). Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Lin, M. C. and Manocha, D., editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag. From the First ACM Workshop on Applied Computational Geometry.

Si, H. (2003). TetGen: A 3D Delaunay Tetrahedral Mesh Generator. `http://tetgen.berlios.de`.

Tarantola, A. (2001). Logarithmic parameters. `http://web.ccr.jussieu.fr/tarantola/Files/Professional/Papers_PDF/Music.pdf`.

A. Inversion properties

An inversion is a template class of the underlying data vector Vec and the matrix type Mat. It is initialised with one of the following constructors:

```
//! very simple and empty inversion
Inversion( bool verbose = false, bool dosave = false )
//! Usual inversion constructor with data and FOP
Inversion( Vec data, ModellingBase & forward, bool verbose, dosave)
//! Complete constructor including transformations
Inversion( Vec data, ModellingBase & forward,
    Trans & transData, Trans & transModel, bool verbose, bool dosave )
```

The properties are not visible itself, instead there are setter and getter functions. Setters:

```
setRelativeError( double/Vec error );  //! set relative data error
setAbsoluteError( double/Vec error );  //! set absolute data error
setForwardOperator( ModellingBase & forward ); //! set forward operator
setTrans( transData, transModel );    //! set transformations
setTransData( transData );            //! set data transformation
setTransModel( transModel );          //! set model transformation
setLineSearch( bool isLinesearch );   //! switch line search on/off
setRobustData( bool isRobust );       //! IRLS (robust) data weighting
setBlockyModel( bool isBlocky );      //! IRLS (blocky) model constraints
setLambda( double lambda );           //! regularisation strength
setOptimizeLambda( bool optLambda );  //! L-curve optimization
setMaxIter( int maxIter );            //! define maximum iteration number
setModel( Vec model );                //! set model vector
setModelRef( Vec referenceModel );    //! set reference model vector
setCWeight( Vec cWeight );            //! set constraint control vector
setMWeight( mWeight );                //! set model control vector
```

Getter

```
ModellingBase & forwardOperator();      //! forward operator
uint boundaryCount(), modelCount(), dataCount();//! # boundaries/cells/data
bool lineSearch(), blockyModel(), robustData(), optimizeLambda();//!options
double getLambda();                       //! regularisation strength
int maxIter();                            //! maximum iteration number
Vec model(), response(), roughness();     //! model/response/roughness vector
Vec cWeight(), mWeight();                 //! constraint/model control vector
getPhiD(), getPhiM(), getPhi(), getChi2(); //! objective function parts
```

Run inversion and other actions

```
Vec model = run();                    //! runs the whole inversion
Vec model = oneStep();                //! runs one inversion step
Vec model = runChi1();                //! runs changing lambda such that chi^2=1
robustWeighting();                    //! applies robust data weighting
constrainBlocky();                    //! apply blocky model contraints
echoStatus();                         //! echo chi2/phi/phiD/phiM/iteration
Vec modelCellResolution( iRes );      //! compute a column of resolution matrix
```

B. Mesh types and mesh generation

There are different mesh types and ways how to generate them. There is only one base mesh class holding the nodes/vertices/coordinates, cells (defined by the bounding vertices) and boundaries revealing the neighboured cells. Every node, cell and boundary has a marker that defines the behaviour in modelling (e.g. an electrode node or boundary conditions) or inversion (e.g. the region number or a known sharp boundary).

0d mesh

Zero dimension means that there are several parameters without any neighbouring relation. Consequently 0th order constraints are used. There is no special mesh generator, instead a 1d mesh is created and the constraint type is set to zero. Alternatively, the forward operator is initialised without mesh and the parameter number is set by setParameterCount.

1d mesh

A real 1d mesh subdivides the subsurface in vertically or horizontally aligned elements and can be created by the following functions:

```
/*! Generate 1d mesh with nCells cells (size 1) and nCells+1 nodes */
Mesh createMesh1D( uint nCells, uint nClones = 1 );
/*! Generate simple one dimensional mesh with nodes in RVector pos */
Mesh createMesh1D( const RVector & x );
```

nClones can be used to create models for different parameters such as resistivity and phase. Result is one mesh with two sub-meshes that are individual regions. See section 3.1 for an example.

1d block model

A 1d block model consists of (nLayers-1) thicknesses and nLayers values for a number of properties. The thicknesses and properties are individual 1d meshes.

```
/*! Generate 1D block model of thicknesses and properties */
Mesh createMesh1DBlock( uint nLayers, uint nProperties = 1 );
```

See section 3.2 for a resistivity block inversion or 5.2 for joint block inversion using two properties.

2d regular mesh

A regular (FD like) 2d model consists of regularly spaced rectangles. They can be created by the number of elements in x- or y-direction or vectors of the enclosing nodes:

```
/*! Generate simple 2d mesh with xDim*yDim cells of size 1 */
Mesh createMesh2D( uint xDim, uint yDim );
/*! Generate simple 2d mesh from node vectors x and y */
Mesh createMesh2D( const RVector & x, const RVector & y );
```

2d general mesh

Generally a 2d mesh - a regular one is just a special case - can consist of triangles or quadrangles (deformed rectangles) or a mix of it. They are created by mesh generators such as triangle (Shewchuk, 1996). Input for the mesh is a piece-wise linear complex (PLC) comprising nodes edges and region markers. Meshing is done externally and loaded using Mesh.load(filename);.

3d regular mesh

A regular (FD like) 3d model consists of regularly spaced hexahedra. They can be created by the number of elements in x/y/z-direction or vectors of the enclosing nodes:

```
/*! Generate regular 3d mesh with xDim*yDim*zDim cells of size 1 */
Mesh createMesh3D( uint xDim, uint yDim, uint zDim );
/*! Generate regular 3d mesh from node vectors x, y and z */
Mesh createMesh3D( const RVector & x, & y, & z );
```

3d general mesh

At the moment, a 3d mesh can consist of tetrahedrons or hexahedrons, but prisms or pyramids could be easily implemented.

C. Region properties and region map file

Some properties can be set for each region individually using `f.region(i)`.

```
setMarker( int marker );
setBackground( bool background );
setSingle( bool single );
setStartVector( const RVector & start );
setStartValue( double start );
setModelControl( double mc );
setModelControl( const RVector & mc );
setBoundaryControl( double or RVector bc );
%setZPower( double zpower );
setZWeight( double zweight );
setTransModel( Trans & tM );
setConstraintType( uint type );
setLowerBound( double lb );
setUpperBound( double ub );
```

```
uint parameterCount(), boundaryCount(); //! parameters/boundaries
```

The region manager controls the individual regions. It is initialised from the forward operator and interprets the mesh with its markers

```
setMesh( const Mesh & mesh );
Region * createRegion( int marker, const Mesh & mesh ); //! create region
Region * region( int marker ); //! get an individual region by marker
uint parameterCount(), constraintCount(), boundaryCount(); //! counters
RVector createStartVector(); //! create starting model vector
RVector createModelControl(), createBoundaryControl(); //! create vectors
RVector createFlatWeight( double zPower, double minZWeight ); //! zpower
loadMap( const std::string & fname ); //! set region properties from file
```

The latter region map file simplifies the setting of region properties and is comfortable for testing different values. It is a column file with the description of the columns in the first row after a # sign:

```
#No start Ctype MC  zWeight Trans lBound uBound
0   100   1     1   0.2     log   50     1000
1   30    0     0.2 1       log   10     200
```

The example represents a two layer case, e.g. an unsaturated (0) and a saturated (1) zone with different starting resistivities. Smoothness constraints with enhanced layering is applied in the first and minimum length in the latter. Both use a log/logLU transformation with specific upper and lower bounds.
Instead of the number, an asterisk (*) can be used to set properties for all regions. In one region file, several blocks as above can be stated, e.g.

```
#No start Trans
*   100   log
#No lBound uBound
1   20     300
2   50     1000
```

defines an equal model transformation and starting value, but different lower and upper bounds.
There are two special types of regions: background and single regions. The background region is not part of the inversion, the values are prolongated (filled up) for the forward run. On the contrary, the cells of a single region are held constant and treated as one parameter in inversion. They are specified as above using the keywords `background` and `single`, e.g.

```
#No single
* 1
#No background
0 1
```

defines all regions as single parameter regions except number zero, which is background.
By default, regions are decoupled from each other, i.e. there are no smoothness contraints at the boundary. However, it might be useful to have those, e.g. by constraining a region of known parameters by borehole data to the neighboring cells or just to stabilize inversion. In this case, inter-region constraints can be defined in the region file. The text

```
#Inter-region
*  *   0.1
1   2   1
```

sets weak connection between all regions, except regions 1 and 2 are normally connected.

D. Transformation functions

For data and model parameter arbitrary transformations can be applied. A transformation is a C++ class derived from a base class (the identity transformation), which mainly consists of four functions, each returning a vector for a given vector:

trans the forward transformation function: $y(x)$

invTrans the inverse of the function: $x(y)$

deriv the derivative of the function: $y'(x)$

error the transformation of associated errors $\delta y(\delta x)$

The latter is defined in the base class. The inversion as mathematical operation is done in the y domain, whereas the physics is described in the x domain.
Besides the presented transformations, you can define your own transformations by deriving from the base class and overwriting the first three functions. If the inverse transformation is not known analytically, there is a class **TransNewton**, in which the inverse function is obtained by Newton iteration.
However, there are a lot of already existing transformation classes that can be used or combined:

Basic transforms

TransLinear (a,b): $y(x) = a * x + b$

TransPower (n): $y(x) = x^n$

TransExp (x_0,y_0): $y(x) = y_0 \cdot e^{-x/x_0}$

TransInv : $y(x) = 1/x$ (specialisation of TransPower)

TransLog : $y(x) = \log(x)$

Range transforms

The logarithm restricts x to be positive, i.e. sets a lower bound 0. Instead of 0, a lower bound x_l can be set. By combining two logarithmic functions a lower and an upper bound can be combined. Similar can be obtained by a cotangens function.

TransLog (x_l): $y(x) = \log(x - x_l)$

TransLogLU (x_l,x_u): $y(x) = \log(x - x_l) - \log(x_u - x)$

TransCotLU (x_l,x_u): $y(x) = -\cot((x - x_l)/(x_u - x) \cdot \pi)$

Combination

Different transformations can be combined by either
TransNest(y^1,y^2): $y(x) = y^1(y^2(x))$
TransAdd(y^1,y^2): $y(x) = y^1(x) + y^2(x)$
Since for the latter the inverse cannot be combined by the two inverses, it is derived from transNewton, a base class whose inverse is achieved by a Newton iteration. So any not-so-easily-invertible function can be derived from transNewton and does not require to define invTrans.
There is a cumulative transformation CumulativeTrans, which applies a vector of transformations for different part of the model. This meta-transformation is applied in the region technique but can also be defined for one region. More often it is used to combine different data or model types, see sections 4.1, 4.2 and 5.1 for examples.

Special transformations

Some geophysically relevant transformations have been already defined:

TransLogMult (y_0) is a **TransNest** of **TransLog** and **TransMult**: $y(x) = y_0 \log(x_0)$, e.g. for using the logarithm of the apparent resistivity ($\rho^a = GR$)

TransCRIM (ϕ,ϵ_m,ϵ_w): the complex refractive index model (CRIM) - derived from **TransLinear**

TransArchie (ρ_w) - Archie's equation, derived from **TransPower**

Note that there are predefined types based on real (double) vectors beginning with an R, e.g. RTransLogLU is actually TransLogLU< **RVector** >.

E. Vectors and Matrix types

▮
std :: vector
template< **class** ValueType > **class** Vector
RVector,FVector, BVector,IVector
load/save
complex values using Complex = std :: complex<**double**> as a vector CVector
template < **class** ValueType > **class** Matrix
RMatrix, FMatrix
load/saveMatrixCol
template< **class** ValueType, **class** IndexType > **class** SparseMapMatrix
RSparseMapMatrix for double and unsigned int
modelling column-compressed sparse matrix **SparseMatrix**

E.1. Block-Matrices

Large matrices for Jacobian (J) and constraint matrix (C)
holding individual matrices by stacking matrices saves space and makes allocation easier
Horizontal stacking (H types): individual models (or model parts)
Vertical stacking (V types): individual data (J) or constraints (C)

2 matrices (2) of arbitrary type, N matrices (N) of identical type, repetition (R) of identical matrices

H2Matrix< Matrix1, Matrix2 > example: 2 data sets for 1 model

V2Matrix< Matrix1, Matrix2 > example: 1 model, 2 data sets

D2Matrix< Matrix1, Matrix2 > example: 2 models, 2 data sets

HNMatrix< Matrix > example: LCI

VNMatrix< Matrix > (N vertical identical matrices)

DNMatrix< Matrix > (diagonal matrices, Block-Jacobi) example: LCI (identical models with individual Jacobians)

DRMatrix< Matrix > (example: