A Modular System for EM Inversion: Beta Version

1 Objectives and development history

The rationales for developing the modular inversion system documented here include:

- Having a "testbed" which can be used for developing inversion algorithms, with all of the "dirty" problem specific implementation details hidden from the inversion algorithm developer. In particular, one specific intended use of the system is to streamline development of 3D inversion methods, with initial debugging and testing of inversion codes conducted on 2D problems, before attempting application to more computationally challenging 3D problems. Note that one of our goals has been to develop more efficient algorithms which take better advantage of the structure of multi-transmitter EM inverse problems. This specific project goal has influenced the structure of the modular system.

- Having an inversion system that can be used with minimal modification on a range of different problems, with the capability for more rapid adaptation of the codes to treat problems of interest to other groups, e.g., Marine EM.

- Allowing for extensibility of practical inversion codes for specific problems. Two specific areas where extensibility might be useful, even for a simple well developed EM inverse problem like 2D MT: adding new measurement functionals (e.g., to invert for intersite transfer functions), and modifying the model parameterization.

These specific objectives have largely driven the devolopment, and influenced the structure of the code. So far we have focused mostly on understanding the basic issues, and developing a general framework for a modular system. The codes discussed here have been developed in f95, writing and testing code for a single case: 2D MT inversion. The higher level parts of the system are in fact designed to be general enough to allow use in other types of problems, including (especially) 3D MT, for which most of the components have been developed (at least in a preliminary form). However, actual testing of the modular system for inversion, has so far been restricted to 2D MT, and most of the modules documented here represent specific instances relevant to only this inverse problem. The hope is that this 2D system will provide a useful template for development of the specific modules required for 3D MT or for other (e.g., active source) problems. Also, as already noted, and as we shall discuss further below, the top level data space and inversion modules are intended to be general enough to ultimately be useful for a broad range of EM inverse problems. The next development step is to update already written 3D modules for compatability with the 2D system, and complete development and testing of a 3D MT inversion.

Our general approach has been to abstract the basic data objects that appear in an inversion scheme (model parameter, EM solution, data vectors), define these as essentially abstract data types, and develop basic methods for each class of object, as well as the necessary mappings between classes. At the highest levels of the code, only the abstract data objects are manipulated, with no reference to implementation specific attributes of

these objects. Our approach is thus "sort of" object oriented. I say "sort of" because the code is written in Fortran 95, and some aspects of an object oriented approach are difficult to implement in this programing language. Furthermore, some compromises in the object oriented approach seem warranted by the specific computationally intensive problems we are working on. For example, strict adherence to data hiding appears to result in too much overhead. Of course the system is very much under development, and it is anticipated that significant improvments could be realized. We do make extensive use of advanced features of Fortran 95 (such as they are): all code is written using modules, with explicit interfaces for all routines, data hiding through use of the "private" attribute (where practical), operator overloading, and extensive use of derived data types.

2 Overview

To provide an overview of the modular system being developed, it is useful to refer to the module dependency structure for our initial test application, the 2D MT inverse problem. The use dependencies of all modules for this application are illustrated in Figure 1 which is discussed in detail in the following sections.

2.1 Generic Inversion Modules

The top level in the diagram represents modules which have been developed for use with wide range of EM problems. Initial testing has been on 2D MT (TE+TM) but, care has been taken to make modules at this level independent of specific details of the 2D MT implementation. As will become apparent, this generallity has required making the code somewhat more complicated than would be required if only the specific 2D problem were to be addressed. While our intent is to keep this level generic with respect to the specific data type, model parameterization, problem dimensionality, source geometry, and many other factors, the implementation is fairly specific with respect to data space objects, which are public (and have public attributes) within the top level routines. This fits in with our objective to develop inversion routines that take advantage of the special multi-transmitter structure of EM datasets; hence we have built this structure into our implementation of the basic dataspace derived data types, and we make use of specific attributes of dataspace objects throughout the top level modules. In contrast, other objects, including model parameter and EM solution vectors are treated as abstract data types. These objects are manipulated and passed around by the top level modules, but internal attributes are never referenced. To use the top level routines for different EM inverse problems these abstract derived data types (i.e., model parameters, EM solution objects), must be instantiated for the specific EM inverse problems using the same generic data type names. This is necessary because there is no way to "rename" data types in f95. Methods referred to at the top level also have generic names, although renaming of procedures can be used here in f90. However, interfaces (type names, order of arguments) must be the same as in the 2D examples in our initial implementation.

2.2 Application Specific Modules

Just below the top level generic inversion modules sits a layer of modules that implement the basic operations needed for a spcific EM inverse problem, e.g., 3D MT; 2D MT; active source 2D; global observatory (3D spherical coordinate). These modules contain code that implements the forward mapping, as well as derivatives or data sensitivities. The data sensitivity calculation is broken down into a series of steps, using the forward code, data functionals, and the linearized mapping from model parameters to coefficients of the linear system of equations for the forward problem. The general mathematical formalism is described in the Appendix of the pre-print Egbert and Siripunvaraporn (2006). Note that the actual forward modeling code is not considered part of this level. Rather, this level contains "driver" routines which are called by the top level module routines to manage initialization, and solver calls required for both forward modeling and derivative calculations.

One key module at this second level (SolnRHS; actually in the third line, on the far leftm, in Figure 1) defines the generic EM solution space data type and methods. This is the specific implementation (for the particular EM inverse problem) of the abstract data type (ADT) used by Levl 1 modules to represent the absract EM solution. Note that for the 2D MT problem the basic EM solution is treated as a single scalar field (E_x or H_x for TE or TM modes, respectively), together with tags to keep track of the mode, and pointers to transmitter and model parameters. For 3D the basic EM solution would have to include 3 component vector fields for 2 source polarizations (corresponding to different boundary conditions) merged into a single EMsoln object. Note, for example, that evaluation of linearized (and non-linear) data functionals in 3D requires solutions for both polarizations. To allow an abstract treatment of EM solutions in the level I modules all of these details (one scalar 2D field, vs. a pair of 3D vector fields) must be hidden. The purpose of the SolnRHS module is to provide the interface between the top level routines and lower level routines where the details of the EM solution implementation must be referenced explicitly. A separate data type for forcing of the EM equations is also defined in this module. It is also necessary to define objects to represent data functionals (denoted by $\mathbf{L}$) with sufficient abstraction to accomodate multi-source vector fields, single component scalar fields, and other possibilities. This, together with generic methods for the high level EM solution space data types (basic, linear algebra, and dot products) are included in SolnRHS.

Adding new measurement operators would require additions at this level (to module DataFunc), with few (if any) changes elsewhere. the model parameter/solver coefficient sensitivity mappings (SensPDECoeff, formally denoted by the matrix $\mathbf{P}$) are also included at this level.

2.3 EM solver

The core of the inversion system is the EM solver. In the 2D case the solver was originally written by W. Siripunvaraporn (the source code is taken from REBOCC), and was "wrapped" into f95 modules to interface with the modular inversion system. The original code is encapsulated in module WSfwd2D. FwdTEmod and FwdTMmod contain f95 driver routines for initialization (forming and factoring matrices; setting boundary conditions), solving the numerical EM equations (back substitution), and cleanup. Details are given in Section 3.

2.4 Basics: Grids and EM Solutions

The lowest level routines are in modules Grid2D, Soln2D, and InterpEB2D. These routines are very closely tied to the specific finite difference modeling approach in two dimensions. Grid2D defines a datatype to hold 2D structured rectangular grid geometry information, soln2D provides basic data structures to hold scalar fields that can be defined on this grid, along with methods to create, zero, copy, form linear combinations, and dot products of these basic solution space objects. In addition to the usual array type data structures, sparse vector objects are supported (to represent efficiently observation functionals), along with basic/linear algebra/dot product methods for these objects. Some basic routines for mixed full/sparse methods are also included.

InterpEB2D forms the basis for the obeservation functionals. In particular, this module contains routines that construct the sparse vectors that represent interpolation functionals, used to compute the interpolated electric or magnetic field values at an arbitrary point in the domain from the values defined at the grid nodes in basic solution vectors (i.e., the data types defined in Soln2D). The structure here illustrates a key point: InterpEB2D is highly dependent on the grid; interpolation functions for a non-structured grid would look very different. However, once E and B are interpolated to the data site, formation of an impedance (tensor or scalar) is actually independent of the grid. This second step (as well as things like linearization of the impedance tensor) is handled at a higher level, in DataFunc. In principal a finite element implementation (which would require different basic solution space objects, different interpolation routines (and of course different solvers!) could be implemented with little or no change to the the Level II routines, including DataFunc. Conversely, a different EM problem (e.g., controlled source marine EM) would require modifications to routines such as DataFunc, but if the same grid and modelling approach were used, the basic E and B field interpolation routines would not need to be different.

Similar discussions apply to primative EM solution data types and associated methods defined in Soln2D, and the higher level encapsulation defined in SolnRHS. Methods and data objects in the higher level SolnRHS module are expressed in terms of methods and objects defined in the primative Soln2D module. Modification of the low level grid structure would require recoding lower level routines, but minimal (if any) modifications to SolnRHS (names of data types and procedures, if different would require modification).

2.5 Model Parameterization

On the far right near the bottom of Figure 1 is the model parameter module. The basic data object in this module is the model parameter. At present in our 2D MT implementation this is very simple, and classic: conductvity or log conductivity is defined independently on each of the cells in the numerical grid. A key point here is that we have made all attributes of the model parameter data type private. Thus the rest of the code does not know about, or make explicit use of, any details in the way the model parameter has been defined. To make all model parameter attributes private, it is necessary to include all functions and methods that need knowledge of model parameter details within the modelParameter module. This includes functions which map from the model parameter to the numerical grid (to define coefficients of the Maxwells equation operator, as discretized), and the model parameter

covariance, along with basic model parameter methods, algebra and dot products. Making modelParameter attributes private should allow the model parameterization to be modified without touching any other modules.

3 Module dependencies

Figure 1 gives the use dependencies of most of the main modules in the Modular2D system, as developed for the initial 2D MT application. Following are conventions used in this figure:

- Red lettering is used to denote modules which define data types that are used by other modules. These modules also generally contain basic methods for the defined data types.
- Black lettering is used to define modules that generaly define only methods; if there are new data types defined in these modules they are not intended to be public. These data types should typically be made "private", but I have not made sure this is done yet.
- The arrows define the usage dependency; arrowheads point at modules which use the module that the tail is touching, or inside of. I.e., the module being pointed at (e.g., DataFunc) uses all modules that point at it (SolnRHS, InterpEB2D, and DataGridInfo). In Fortran95 modules are inherited: if Module C uses module B, and module B uses module A, then by default everything in A that B has access to, C also has access to. Thus pretty much everything has access to module grid2d (except for DataSpace2D, which does not use any of the other modules in the figure). An important distinction between the arrows in the figure is whether or not the tail goes into the interior of the box. If the use arrow extends into the interior this means that the using module needs to make use of the details of the data objects; i.e., the attributes of data objects defined or available inside these modules need to be public to the using module. In the other case (tails not extending into the interior) data types or methods made available to the using module are more abstract; the using module does not need access to internal attributes of the data objects. In fact, the only module where an data type is defined which has no public attributes is the ModelParameter module. Note that there gnerally is only one "layer" of method modules around the basic data object modules that need to know details of the internal structure of the data objects. Higher level routines (for which the use arrows start outside of the boxes) treat the basic data objects abstractly, and are not supposed (by convention) to reference interal data object attributes. However, except for ModelParameters attributes are public, so the compiler cannot enforce this convention. Note that some unessary arrows appear, given the inheritance rules.
- There are 3 modules that are not shown: math_constants ioMod, and utilities. At present utlities.f90 just has one routine, errStop. math_constants and utilities are used by almost all other modules. ioMod is used by driver programs only.
- Modules near the bottom are generally more basic, and are used by modules higher in the stack. As outlined above the modules are loosely organized in layers, with all

modules in a layer having a similar (though not always identical) place in the modular hierarchy.

4 Module Details

All of the principal derived data types have a relatively standard set of "basic" methods, to create, deallocate and copy objects of this type. Also most (but not all) of these objects represent elements of a vector space, so linear algebra methods (multiplication by a scalar, linear combinations, adding) are provided, along with dot products. We have tried to use standard naming conventions for these basic/linear algebra/dot product methods:

create_$\langle dataType \rangle$ creates (including allocation of arrays) a ¡dataTYpe¿ object

deall_$\langle dataType \rangle$ deallocates arrays in a $\langle dataType \rangle$ object

copy_$\langle dataType \rangle$ copies a $\langle dataType \rangle$ object to another This is always overloads the "=" operator

zero_$\langle dataType \rangle$ zeros a $\langle dataTYpe \rangle$ object (this routine exits only if this is sensible/useful)

linComb_$\langle dataType \rangle$ forms linear combination of two objects of type $\langle dataType \rangle$.

dotProd_$\langle dataType \rangle$ forms dot product of two objects of type $\langle dataType \rangle$.

We do not provide a full description of all of the specific routines in the following, unless there is some special feature to take note of. The functionality of these general (and usually simple) routines should be fairly obvious from context, and can easily be discerned from the source code. All data type attributes, and routines are public, unless otherwise noted.

4.1 Base support modules

Module math_constants

This module sets a number of parameters used by other program units. In particular it (a) sets the precision used; in principal it should be possible to switch from double precision to single precision by changing parameter selectedPrec, but this has not been tested (and there might still be some places where type declarations do not make use of the parameter selectedPrec); (b) sets math and physics parameters (pi, mu, sigma_air, one, zero, etc.); (c) sets the sign convention for time dependence (exp isign omega t) again there might be some place where isign = -1 is still hard-coded; (d) defines some character strings that signify various options.

Module Utilities

Some general purpose utility functions; mostly these are a collection of functions used by W. Siripunvaraporn in the REBOCC code. Some may not actually be used in any of the 2D modules. See source code; nothing critical or profound here!

Module IoMod

I/O routines for most of the critical data objects, used to read or write to/from files. In general, these are simple fortran binary files, with enough header information to completely specify the data object, and not much more. The data types that are read and written are described in greater detail in subsequent sections. There are matlab routines for reading and writing all of the files that are used here. See the source code to understand the (simple) file formats.

`subroutine read_Vec2Dc(fid,cfile,vec)`

Opens *cfile* on unit *fid*, read in *vec* of type vec2Dc. Output object vec must be created before calling (i.e., need to know grid size!)

`subroutine write_Vec2Dc(fid,cfile,vec)`

Opens *cfile* on unit *fid*, writes out *vec* of type vec2Dc.

`subroutine openR_Vec2Dc(fid,cfile,header,nVec)`

Opens *cfile* on unit *fid* for reading *nVec* vec2Dc objects. This just reads the *header* and the number *nVec* of vec2Dc objects in the file; it does not read data for any actual vec2Dc objects. Note that the file format for multiple vec2Dc objects is not the same as the single vec2Dc object file read/written by `read_Vec2Dc`/`write_vec2Dc`.

`subroutine Vec2DcRead(fid,vec)`

Reads one object *vec* of type vec2Dc from unit *fid*, already opened with call to `openR_Vec2Dc`. At present, this routine just reads the next object. Could modify to allow skipping to read an arbitrary record number.

`subroutine openW_Vec2Dc(fid,cfile,header,nVec)`

Open *cfile* on unit *fid* for writing multiple vec2Dc objects. Writes *header*, number of vectors *nVec* to be put in file.

`subroutine Vec2DcWrite(fid,vec)`

Writes one object *vec* of type vec2Dc to unit *fid*, already opened/initialized by call to `openW_Vec2Dc`.

`subroutine write_EMsolnMTX(fid,cfile,eAll)`

Open *cfile* on unit *fid*, writes out object *eAll* of type EMsolnMTX (i.e., a collection of vec2Dc objects; almost the same as calling `openW_Vec2Dc`, then calling `Vec2DcWrite` nVec times).

`subroutine read_Grid2D(fid,cfile,grid)`

Opens *cfile* on unit *fid*, reads in fundamental rectangular grid paramters, and then creates gridDef2D object *grid*. Note that this routine first has to get the grid size (Ny, Nz) then create the grid (allocate arrays for Dy and Dz), then read in the grid spacing arrays.

```
subroutine write_Z(fid,cfile,nTx,periods,modes,nSites,sites,allData)
```

Opens file *cfile* on unit *fid* and writes a data file (impedances), outputing a dvecMTX object *allData*. The output file contains the number of "transmitters" *nTx*, followed by a dvec struture for each transmitter. For each transmitter the *periods* and *modes* are output, along with locations of the *sites*, the impedance data, and error standard deviations (in *allData*. This is set up for 2D complex impedances, and would need to be modified to output a more general data space object. Note also that a lot of the information passed to this routine (*periods, modes, nSites, sites, sites*) is normally accessed by routines which manipulate data vectors through data type, transmitter, and receiver dictionaries. These higher level data structures are not presently accesible to the IO routines (but could be!) Note also that this write routine assumes that the number of sites is the same for every transmitter; other routines that manipulate data space objects are more general.

```
subroutine read_Z(fid,cfile,nTx,periods,modes,nSites,sites,allData)
```

Reads files of the format output by `write_Z`. Creates/allocates for the dvecMTX output *allData*, after reading the number of transmitters that are included in the file, then reads everything in.

4.2 Grid related modules, including interpolation

Module Grid2D

Defines the data structure used to define the grid geometry, which is a rectangular finite difference grid in our application. A much more complicated data structure would be required to specify a non-structured grid, but a similar approach (making a single data structure which encapsulates a complete description of the grid) would be used.

Data Types

gridDef2D :: data structure to define the grid

Subroutines and Functions

```
subroutine create_Grid2D(Ny,Nz,Nza,grid)
```

Creates and a gridDef2D data structure *grid*, for a Ny x Nz finite difference grid with Nza air layers. After creation, fields grid%Dy, grid%Dz must be filled with node spacing values

```
subroutine gridCalcs(grid)
```

Pre-computes and stores derived grid geometry information for gridDef2D object *grid*. (e.g., positions of cell centers) that are useful for further calculations. This must be called after fields grid%Dy and grid%Dz are set.

`subroutine copy_Grid2D(gridOut,gridIn)`

Copy gridDef2D object *gridIn* to *gridOut*, allocating if necessary. If *gridOut* is already allocated checks to see if array sizes are compatible. This routine overloads = (so *gridOut=gridIn* results in this routine being called).

`subroutine deall_Grid2D(grid)`

Deallocates arrays (and nullifies pointers to) in gridDef2D object *grid*.

Module Soln2D

This module defines the basic objects used to represent solutions to the 2D EM equations, The basic object is a complex scalar field discretized on the rectangular finite difference grid. The module also contains a data type to represent sparse vectors on the same grids; these are used for efficient representation of data functionals (but could also be used for point source). The module includes basic create/destroy routines, vector space arithmetic, and dot products, for full storage and sparse vectors (plus some methods for one sparse and one full storage solution vector). Only those routines needed for modeling and initial work on 2D inversion have been developed here; some further cases might need to be developed for different applications/inversion algorithms.

Data Types

vec2Dc :: data structure to define complex EM solutions/derived fields on grid nodes, cell centers, faces
sparseVec2Dc :: data structure to define complex sparse EM solution space objects. Basic idea is to specify the number of non-zero vector entries (*nCoeff*), the indices J and K of the non-zero entries, and the complex values of the entries (C).

Note that for 2D we only deal with scalar fields, with discrete field values defined on nodes, cell centers, or cell faces. These are distinguished through structure field gridType, which can be either *NODE* or *CELL*/ The gridType should be the same for linear combinations, dot products etc.

Subroutines and Functions

Full storage vector routines:

Basic operations: *vec* is of type vec2Dc; *grid* is of type gridDef2D; *gridType* is a character*80 string; functionality should of these routines is self explanatory.

`subroutine create_Vec2Dc(grid,gridType,vec)`

`subroutine deall_Vec2Dc(vec)`

`subroutine zero_Vec2Dc(vec)`

`subroutine copy_Vec2Dc(vecOut,vecIn)`

This routine overloads =

`function dotProd_Vec2Dc(vec1,vec2,conj_Case) result(c)`

Note: *conj_case* is a logical variable; if false then this is not the usual Hermitian inner product; i.e., the dot product is formed without conjugating the first vector e1. If true, this is the usual Hermitian inner product. (This way of treating the need for the non-conjuaged IP is probably not a good idea!)

Sparse storage vector routines

`subroutine create_SparseVec2Dc(grid,gridType,nCoeff,newLC)`

Here one provides (a pointer to) the *grid*, the *gridType*, and the number of non-zero elements *nCoeff* to allow for. *newLC* is the sparseVec2Dc object created.

`subroutine deall_SparseVec2Dc(oldLC)`

`subroutine linComb_SparseVec2Dc(Lic1,c1,Lic2,c2,Loc3)`

This routine forms the sparse vector which is a linear combination of sparse vectors *c1*Lic1 + c2*Lic2 = Loc3*. The output sparse vector is created inside the routine (and the proper size is determined automatically).

`subroutine copy_SparseVec2Dc(SV2,SV1)`

Combined full/sparse storage vector routines

`subroutine add_SparseFull2Dc(cs,SV,FV)`

Computes the sum $cs*SV + V$ where SV is of type sparseVec2Dc (i.e., a complex sparse vector), and FV is of type vec2Dc (a complex 2D scalar field defined on the grid), and cs is a complex scalar. The result overwrites FV. This routine is used in the code to construct a complex full storage vector from a sparse vector by setting *cs = (1.0, 0.0)* and $FV = 0$. Note that additional functionality for linear combinations of sparse and full vectors might be needed at some point.

`function dotProd_SparseFull2Dc(SV,FV,Conj_Case) result(c)`

The comment on *Conj_Case* given above for `dotProd_Vec2Dc` applies here also.

Module InterpEB2D

This module provides interpolation functionals for both electric and magnetic fields at an arbitrary point within the model domain. Separate routines are provided for the TE electric field, and for the TM magnetic field solutions. In both cases "interpolation" of the other field component (i.e., magnetic for TE mode, electric for TM) are supported. These routines just create sparse vector representations of the basic interpolation functionals. Routines in this module are then called by higer level routines to actually apply data functionals, construct "combs" for inversion, etc. In addition to soln2D and grid, this module needs access to the

ModelParameter module. This is because calculation of electric fields for TM mode requires knowledge of the resistivity in the vicinity of the interpolation point. This module uses only the abstract data type ModelParam (passed as an argument to some routines in this module), and a function rhoC which computes resistivity associated with cell faces. This function is in the modelParameter module, and a function with identical interface and functionality would have to be coded for a modified model parameter model (then no changes would be required here). Also, the form of the function rhoC is very specific to the grid/modeling approach, and would need to be modified (along with all of the other routines in InterpEB2D) if the grid/modeling approach were modified.
Subroutines and Functions

`function minNode2D(x, xNode, nx) result(ix)`

This function finds the element of *xNode(nx)* which just bounds (from below) x. Output is an integer *ix*, giving the position in the input *xNode* array which provides the greatest lower bound on x (among all elements in *xNode*). Calling this function for each coordinate in an interpolation point allows the approprite nodes, cell centers, or faces which bound the point in the model domain (y,z). The resulting indicies are used to construct the interpolation operators for this point.

`subroutine NodeInterpSetup2D(grid,x,mode,LC)`

Sets up coefficients in sparse vector LC for evaluation/interpolation at the two dimensional postion vector x for a field defined on a TE or TM mode *grid*. Input parameter *mode* (character*2) is used to define which case to do the interpolation for; the only difference is that there is no air layer in the TM mode case, so vertical node numbering is slightly different. The interpolation method is a bi-linear spline. This routine allocates arrays in the output sparse vector LC, deallocating first if necessary. Note that this sets up for interpolation of E for TE mode solutions, and for B for TM mode solutions.

`subroutine BinterpSetUp_TE(grid,x,LC)`

This routine sets up coefficients in sparse vector LC for evaluation/interpolation of a horizontal magnetic field component at the location given by the two dimensional position vector x, using a magnetic field vector defined on faces of a stagered 2D *grid* (for TE electric fields are defined at nodes). For direct application of the sparse vector to interpolation of the magnetic field, this field would have to be computed first on faces. In actual usage, this sparse vector is used by `BfromEseup_TE` to construct an interpolator which can be applied directly to the electric field solution object defined (for 2D) on nodes. Again, the interpolation method is bilinear splines. Interpolation of vertical magnetic. Because this routine is only called by other routines in the same module, `BinterpSetUp_TE` has been made private (no reason why this could not be changed, if needed for some other purpose).

`subroutine BfromESetUp_TE(grid,x,omega,LC)`

This routine sets up coefficients in sparse vector LC for evaluation/interpolation of a horizontal magnetic field component at the location given by the two dimensional position vector x, using an electric field vector defined on nodes of the standard 2D TE electric fields *grid*. This calls `BinterpSetUP_TE`. Note that the computation of magnetic fields from electric requires the angular frequency *omega*.

```
subroutine EinterpSetUp_TM(grid,x,sigma,LC,Q)
```

This is the analogue of `BinterpSetUp_TE` for interpolation of electric fields from faces of the 2D *grid*. This interpolation makes use of the model parameter *sigma* to obtain the resistivity on the cells near x, which are required for acurate interpolation. Argument Q (an array of 4 sparseVec2Dc objects) is optional, and is used for computation of the derivative of data functional coefficients with respect to variations of the model parameter. This routine is only called by `EinterpSetUp_TM`, and has been declared private to this module.

```
subroutine EfromBSetUp_TM(inGrid,x,omega,sigma,LC,b0,Q)
```

This is the TM mode analogue of `BfromESetUp_TE`, computing the sparse vector LC needed for evaluation of the electric field at the location given by 2 dimensional vector x, directly from the TM magnetic field solution vector. The background magnetic field solution ($b0$) is an optional input, and the sparseVec2D object Q is an output. These two optional arguments are required to compute the derivative of of the data functional coefficients with respect to variations in the model parameters. Both optional arguments should be present, or neither.

Detailed discussion of the meaning of Q is beyond the scope of this documentation! Briefly: we assume that the model parameter determines conductivity of each cell (by some mapping; in the model parameter module implemented to date this mapping is trivial). Q gives the component, due to to dependence of the data functional on conductivity, of the derivative (with respect to the block conductivities) of the electric field at location x. (Hard to describe without detailed equations!)

4.3 The ModelParameter module

There is only one module in this group, because we have merged all routines which need access to model parameter attributes into a single module. This allows making model parameter attributes private, and prevents any other parts of the code from depending on specifics of our model parameterization. The rationale for this strategy is that we anticipate that modifications in model parameterization might be particularly useful.

Data Types

modelParam :: This is the derived data type used to store parameters that determine conductivity/resistivity needed to define the partial differential equation coefficients. This specific implementation is based on blocks allows for linear/log conductivity/resistivity As noted repeatedly already, all attributes are PRIVATE to this module. (but the data type

itself is public). One detail to note: two parameter types are supported in this implementation: linear and (natural) log conductivity. These are distinguished by the value of modelParam%paramType., a character string.
mCov :: This is the derived data type used to store parameters that define the model covariance, as well as arrays useful for implementation. This data type is used only within this module, and the full data type is private.

Subroutines and Functions

There are essentially three subgroups of routines in this module: the usual create_, deall_, etc., objects; the model covariance (smoother); and routines which implement the mapping from model parameters to the grid (including a linearized mapping, and its adjoint). The first group are for the most part self-explanatory, following the usual naming and argument conventions.

```
subroutine create_modelParam(grid,paramtype,m)
```

Inputs are *grid* (type gridDef2D) and *paramType* (character string; LOGE or COND). This routine in principal has to be used to create new modelParam objects. In a different implementation, one might need different input information (of different types than here ... *grid* of type gridDef2D, a character string denoting *paramType*. To avoid having to adhere

```
subroutine deall_modelParam(m)
```

```
subroutine copy_modelParam(mOut,mIn)
```

As noted above, a call to this can be used to create a new modelParam object (*mOut*) with the same properties as *mIn*. Using this approach in higher level routines for creation of a new model parameter circumvents needing to refer to things such as *grid* or *paramtype* which might be irrelevant to a different model parameter implementation.

```
subroutine zero_modelParam(m)
```

```
function dotProd_modelParam(m1,m2) result(r)
```

Note that modelParam is always assumed to be real.

```
subroutine linComb_modelParam(a1,m1,a2,m2,m)
```

```
subroutine write_Cond2D(fid,cfile,m)
```

This is a write routine, exactly as in ioMod, but for writing out modelParam objects, using unit *fid* for file *cfile*. This has to be in this module because the write routine obviously needs access to the details of the model parameter m in order to write out the components that define the parameter.

```
subroutine read_Cond2D(fid,cfile,m)
```

This is the comparable read routine.

```
subroutine writeAll_Cond2D(fid,cfile,header,nSigma,sigma)
```

This writes out an array *sigma* of *nSigma* modelParam objects in one call, first opening *cfile* on unit *fid* (and adding user defined descriptive character*80 string *header* at the top of the file).

The second group of routines in modelParam implements mappings from the conductivity parameter to the discrete grid. To the extent that the mapping is non-linear, linearization of the mapping also needs to be provided, as does the adjoint of this linear mappping. Here there are two distinct modeling problems, for TE and TM modes. Separate mappings are required for the two cases. If the model parameterization representation were changed, all of these routines would also need to be modified.

```
subroutine CondParamToArray(m,Ny,Nz,Cond)
```

Maps from modelParam object m to a real array defined on grid nodes, as required for the TE forward operator. The output *Cond* is a real array of dimension (*Ny,Nz*). A similar mapping routine would be required for any modified model parameterization. This routine is only used by forward modeling setup routines, which were not originally written using derived data types (hence the output *Cond* is just a simple array). This mapping function is used for both TE and TM mode modeling.

```
function rhoC(m,j,k) result(r)
```

Computes resistivity for cell *j,k* using input modelParam m. This function defines how the abstract conductivity parameter is mapped to cell resistivites needed for TM electric field interpolation functions. The derivative of this function is required for evaluation of linearized data functionals, and for construction of the direct parameter space part of the "comb".

The remaining routines in this group are only called by routines in module SensPDECoeff. Thus, changing interfaces on these routines (which might be appropriate for some different model parameterizations) would only require modifications of one additional module.

TE mode routines:

```
subroutine CellToNode(m,NodeCond,m0)
```

Linearized mapping from conductivity or log conductivity (m), defined as a modelParam object (i.e., in this implementation conductivity defined on Earth CELLS), to interior Earth NODE conductivities, averaging over adjacent cells. Overwrites inputs on these cells, leaving other nodes (in air, on grid boundaries) unaltered *m0* is background conductivity for linearization – this is an optional argument, required when *CellCond%paramType == LOGE*. The output (*NodeCond*) is of type vec2Dc, created with *gridType = NODE*.

```
subroutine NodeToCell(NodeCond,m,m0)
```

This is the adjoint (transpose) of `CellToNode`. Input (*NodeCond*) is of type Vec2Dc,created with *gridType = NODE*; output is a modelParam (m); *m0* is an optional argument, required for linearization if *m%paramType = LOGE*.

TM mode routines:

```
subroutine CellToFace(m, m0, FaceRes_y,FaceRes_z)
```

Linearized mapping from modelParam (perturbation) m to perturbation (about *m0*) in resistivities on y and z faces of cell. Background is model parameter *m0*, outputs are vec2Dc (*FACE*) objects *FaceRes_y*, *FaceRes_z*. Background is always needed for this implementation, since the relationship between m (conductivity or log conductivity) and resistivity is always non-linear.

```
subroutine FaceToCell(FaceRes_y,FaceRes_z,m0,m)
```

Adjoint of `CellToFace`, linearized about *m0*.

```
subroutine QtoModelParam(Qj,m0,dm_real,dm_imag)
```

This completes implementation of the transpose of operator "Q", the component of the data sensitivity due to dependence of the data functionals on the model parameter (i.e., the interpolation coefficients for E and B). This implements $\partial(\mathbf{Q}_j^T \rho_{cell})/\partial\mathbf{m}$ where $\mathbf{m}$ is the model parameter, ρ_{cell} are corresponding cell resistivities, and $\mathbf{Q}_j$ is the sparse vector giving the derivative of the data wrt cell resistivities. *Qj* is an input sparseVec2Dc object and is complex, *m0* the background conductivity (used for computing the derivative of ρ_{cell} wrt $\mathbf{m}$), and *dm_real*, *dm_imag* are output modelParam objects that give this component of the derivative, for the real and imaginary parts of the data, respectively. Note that *dm_imag* is optional.

The final group of routines in the modelParameter module implement the model covariance. These routines were written by W. Siripunvaraporn, and have just been added to this module. Details are not completely understood by me, and these routines have not actually been tested.

Public Routines:

```
subroutine setup1DCM(grid)
```

Using gridDef2D data structure *grid*, initialize covariance smoother (parameters that control smoothing length scales are hard coded at present!) This call initializes a data structure that is saved, and is used in subsequent calls to the model covariance operator.

```
subroutine solveDiff(m)
```

The model covariance operator. Calling this applies the smoother to modelParam m, overwriting the input.

These two public covariance routines call some private routines (`setupYCM,setupZCM,` verb—yDiff,zDiff—) which are used to implement the covariance smoother. A pair of similar public routines will always be needed: for initialization, and to apply the covariance operator.

4.4 Forward Solver Modules

There are 4 modules in this group: SolnRHS, which defines data types used for input and output to the forward solver, and the forward solver core modules. These core modules are described in section V.

Module SolnRHS

This module essentially provides a higher level wrapper for vec2Dc objects. It is used to define objects that represent EM solutions in a more abstract way, and hides some details of the lower level implementation from the top level inversion modules. For the 2D MT modular implementation this extra layer may appear rather pointless; however it will be essential for 3D MT, and it there are already good reasons in terms of module dependency structure to have this extra layer. More explicitly: for 3D the EMsoln objects defined here will contain complex vector field solutions for 2 source polarizations. This is necessary, since elements of the impedance tensor require both source polarizations for evaluation of the corresponding data functinals. So for 3D there is a clear distinction between objects that represent a vector field (a component of an EM solution) and the full two component EM soluton itself. With regard to dependencies: it turns out to be useful to have the following information attached to an EM solution object: a pointer to the grid; a pointer to the modelParam; transmitter information (what sources were used to compute this solution?) At the same time it is useful to have basic objects which can define scalar (2D) or vector (3D) fields on the grid defined at a low level in the module hierarchy; these basic objects (i.e., vec2Dc in this implementation) are useful for many purposes, and furthermore the modelParam module must know about these basic objects (the conductivity mappings in particular require access to these objects to define mapping from parameter space to grid). Including a pointer to a modelParam object in a vec2Dc object creates issues of circularity, if the modelParm module depends on the soln2D module. So we have used soln2D to represent the basic fields, and then define a higher level object which can wrap one or more of these objects, together with information about how the solution was computed into a single data structure.

This is the rationale of this module. In the specific application most of the routines mearly call routines of similar name that are defined in soln2D (and manage some other pointers and descriptive information).

Data Types

EMsoln :: The basic EM solution data type, wrapping an object of type vec2Dc with information about mode, transmittier (frequency/period), model parameter, and grid.
EMsparse : A comparable wrapper for a sparseVec2Dc. This really looks useless in this application! But note that data functionals for 3D MT act on a pair of fields, and will require a pair of sparse vectors for correct representation.
RHS : A data structure to store the right hand side (forcing and boundary conditions (BC)) for the EM system to be solved. This wraps a vec2Dc object, and carries boundary conditions separately. If there is no forcing inside the domain (a boundary value problem, as for MT) the vec2Dc object is not used. This structure also carries flags "mode" and "adj" which determine how the solver is called.

Subroutines and Functions

The following routines follow the usual naming conventions, implementing basic (minimal needed) operations for the three data types defined in this module. Functionality is identical to comparably named routines in soln2d.

```
subroutine create_EMsoln(grid,gridType,e)
subroutine deall_EMsoln(e)
subroutine copy_EMsoln(eOut,eIn)
subroutine zero_EMsoln(e)
function dotProd_EMsoln(e1,e2,conj_Case) result(c)
subroutine create_EMsparse(grid,gridType,nCoeff,LC)
subroutine deall_EMsparse(LC)
subroutine linComb_EMsparse(Lin1,c1,Lin2,c2,Lout)
subroutine create_RHS(grid,gridType,b)
subroutine deall_RHS(b)
subroutine zero_RHS(e)
```

The next two are mixed sparse/full storage routines, again as in soln2d:

```
subroutine add_EMsparseSoln(cs,SV,FV)
function dotProd_EMSparseSoln(SV,FV,Conj_Case) result(c)
```

The last routine adds a sparse vector to the source part of a RHS object.

```
subroutine add_EMsparseRHS(cs,SV,FV)
```

4.5 Application Specific Modules

This group of modules contains routines that are called directly by routines in the top level inversion modules. To keep these generic, it is necessary to follow strict rules on names, interfaces, and functionality of these routines which interact with the top level routines. At the same time, these routines implement very application specific details: what sorts of data are available, and how are predictions computed from the numerical solution; how do variations in the model parameters change coefficients of the EM differential equation (and hence the solution), etc. Routines at this level provide an interface between the very specific details of how solutions are represented on a discrete grid, how the conductivity is parameterized, and how the equations are solved, with the more abstract manipulations used at the inversion level. There are four modules: one is essentially a data sharing module, the other three implement the abstract operators $\mathbf{L}$, $\mathbf{S^{-1}}$, and $\mathbf{P}$ which can be used to give a formal expression for the data sensitivities or Jacobian. (There is an additional term, $\mathbf{Q}$ needed when data functionals directly depend on the model parameters. This is only needed in some cases, and is (mostly) implemented along with $\mathbf{L}$ in module DataFunc.)

Module SensPDEcoeff

This module computes "forcings" for the sensitivity calculation (and adjoints). This module is specific to the numerical implementation of the solver, in this case for 2D TE and

TM MT finite difference modeling. The module works with the "natural" representations of conductivity: defined on nodes for TE, and (as resistivity) on faces for TM; for TM data functionals resistivity defined on cells is also required. Mappings from the potentially more flexible earth conductivity parameter to these fixed, grid-specific representations is implemented in module modelParameter, which this module uses. This module has minimal dependence on the specific way that conductivity is parameterized. These details are handled through the basic mappings (including linearizations and adjoints) which are provided in the modelParameter module. This module does not define any new data types.

There are only two public routines, which as presently coded are wrappers for private routines. Only the public routines can be called by routines outside of this module. These are the only routines for which the explicit interface used for the 2D implementation must be maintained for other applications.

```
subroutine Pmult(e0,dm,e)
```

This computes $\mathbf{e} = \mathbf{P}\delta\mathbf{m}$, using EMsoln *e0* for the background for linearization. *e0* carries information about mode (TE or TM) transmitter (frequency), as well as pointers to background model parameters and the grid. *dm* is a modelParam data object (perturbation to model parameter), and *e* is a RHS data object (right hand side (forcing+BC) for the forward problem. E.g., for TE mode the output corresponds to $i\omega\mu E_x\delta\sigma$.

```
subroutine PmultT(e0,e,dm_Real,dm_Imag)
```

This is the adjoint of `Pmult`, and again works for TE and TM modes, automatically figuring out which is appropriate. Again *e0* is the background solution, and now *e* is the input. Note that *e* is now an object of type EMsoln (it is the solution of the adjoint problem. Purists might cringe ... shouldn't this input be of type RHS, since the output of `Pmult` is? Well, it's not!) The outputs *dm_Real, dm_Imag* are now ModelParam objects, one each for real and imaginary parts (this is needed to allow sensitivities of real and imaginary parts of a complex data to be computed with a single complex adjoint solve.) The argument for the imaginary part output is optional.

The remaining routines, which do all of the actual work, are private. These are called by `Pmult` and `PmultT`, and by other routines in the module.

```
subroutine P_TE(e0,dm,e)
subroutine P_TE_T(e0,e,dm_Real,dm_Imag)
subroutine P_TM(e0,dm,e)
subroutine P_TM_T(e0,e,dm_Real,dm_Imag)
```

These are just like `Pmult` and `PmultT`, but only for TE/TM mode. It would be simple to merge `P_TE` and `P_TM` into Pmult (and similarly for the transpose) to make a single routine. The TE mode routines call `NodeToCell`, `CellToNode`, and various basic solution routines to create and deallocate temporary objects. The TM routines are more complicated, involving the following curl operators, which are at present only used internally within this module (and are private):

`subroutine curlB(b,Jy,Jz)`

Computes curl b, mapping from Node -¿ Face. Inputs are the magnetic field defined on nodes b, an object of type vec2Dc). Outputs *Jy, Jz* are vec2Dc objects (used to store what are essentially currents defined on faces), which should be created with *gridType = FACE_EARTH*. (This is coded to map onto all faces, including boundaries)

`subroutine curlE(Ey,Ez,b)`

Computes curl E mapping from Face -¿ Node. Now *Ey, Ez* are inputs (vec2Dc of gridType *FACE_EARTH*; representing electric fields on faces), and outputs are b of type vec2Dc of *gridType = NODE_EARTH*.

`subroutine curlE_T(b,Ey,Ez)`

Transpose of curlE.

Module dataGridInfo

This module depends only on the grid2d module. It is primarily used for "data sharing" among higher level routines. The data that is stored in this module include: a copy of the grid definition (type gridDef2D), and two "dictionaries" used to provide detailed information about data types, and transmitters (sources). The grid is stored as a gridDef2D object named *SolnRHS_grid*. This copy of the grid is available to routines DataFunc, and EMsolver, after it has been set by a call to the initialization routine `set_SolnRHS_grid`. Having this grid available makes it simpler to keep the top level routines more general, in particular avoiding the need to refer to objects of type gridDef2D (which are specific to the 2D finite difference modeling application) in top level routines.

The idea behind the dictionaries is that the details of what might be needed to describe a "data type" or a "transmitter" will be highly problem dependent. For example, for MT the frequency (and poassibly the mode–TE or TM) completely defines the transmitter. For an active source problem, location and orientation of the source would also be required (along with a different set of possible transmitter types!) To maintain flexibility and reusability for top level routines, data space objects (defined below) carry integer indicies to a look up table (of indeterminate form) called the transmitter and dataType dictionaries. Thus no matter how complicated the data structure required to describe the data type and transmitter, at the top level these are always reduced to a pair of integers. At places in the following we may refer to "data type iDT" or transmitter "iTX", where iDT and iTX are integer arguments. More correctly these are just indicies which point to the data type or transmitter dictionaries. Routines which would need access to the details of a transmitter (to set up the forcing for the forward problem, for example), get this information from the dictionary. This module contains these dictionaries, and routines in level two have full access to these dictionaries, and to the details of their implementation.

With a minor exception (discussed subsequently) the top level routines do not refer to specific components of these dictionaries, but rather just pass around the look up indicies. Thus, if these dictionaries are modified for a diffrent application, no changes to top level

routines should be required. For further discussion of the dictionaries (including the "reciever dictionary") see the DataSpace module documentation below.

Data Types

MTtx :: Derived data type used to represent an entry in the transmitter dictionary. The transmitter dictionary (*txDict*) is an array of MTtx objects. The form is very specficic to 2D MT. Note that no reference to *txDict* is made in the top level routines; only routines in modules DataFunc and EMsolver refer to *txDict*. For problems with mixed data types it would be possible to have multiple transmitter dictionaries with different names (DataFunc and EMsolver would of course have to deal with these different details).

dataType :: Derived data type used to represent the EM "data type". Here we have two data types: TE and TM mode impedances. One could imagine a broad range of other data types (apparent resistivity, tipper, intersite transfer function, impedance phase) even within the context of the 2D MT problem. More generally this could be a starting point for joint inversion, as well as for distinguishing between different ways of presenting the same sort of data. As for *txDict* an array of objects of type dataType forms the data type dictionary *typeDict*.

The boundary between transmitter and dataType is a bit fuzzy, and arguably for the MT impedance case considered here, dataType is redundent. Every data vector has a pointer to both a dataType and a transmitter, so there is some flexibility in where some information is kept. For example, for the 2D MT problem we could have put the mode in the transmitter dictionary, and then, it might seem that there would be no need for a type dictionary. However, the way the top level routines are written there still needs to be a type dictionary, and are some specific things that dataType objects have to contain: the two logical attributes *isComplex* and *calcQ*. These define, respectively whether the data type is complex, and whether the data functional has an explicit dependence on the model parameter. Impedance data has *isComplex = .true.*, and amplitude data by itself would have *isComplex = .false.* (Note that amplitude and phase, as a pair can be viewed as the logarithm of the complex impedance, and the pair apparent resistivity and phase can thus be coupled into a data type that is complex–data functionals would have to be defined correctly for this.) For our implementation of interpolation functionals, TE mode has *calcQ = .false.* and TM mode has *calcQ = .true.* These two attributes HAVE to be defined for any dataType dictionary element. They are referrenced by top level inversion routines (to allow efficiencies, that can be obtained by working with real and complex parts of data together). No other field in the typeDict dictionary is accessed by top level routines, so these may be modified. Also, note that the top level routines refer to typeDict specifically; to use top level routines without modification this dictionary must have this name.

Note that if one were to implement multiple transmitter dictionaries, a flag would have to be set in the dataType entry to allow use of the correct transmitter dictionary for derefenecing the transmitter index.

Subroutines and Functions

```
subroutine set_SolnRHS_grid(grid)
```

This just copies *grid* into *SolnRHS_grid*, where it is saved and made available through this module to level II routines in modules DataFunc and EMsolver.

```
subroutine TXdictSetUp(nTx,Periods)
subroutine TypeDictSetup()
```

These are simple routines for setting up the relatively simple transmiter and data type dictionaries, given a list of periods and assuming that there are only TE and TM impedance data types. More sophistication in managing definition of new data types would probably be useful in the long run.

Module DataFunc

This module defines and implements the data functionals defined in the type dictionary. The last comment on the previous module ("More sophistication ...") applies here as well. At present this implements 2D MT data functionals for TE and TM impedances only. The module contains (1) a receiver dictionary (*rxDict*), which is comparable to the transmitter dictionary, but for receivers; (2) routines for evaluation of impedances (and ultimately other interpretation parameters); (3) routines to compute data functionals for linearized impedances, and ultimately other interpretation paramters. The basic idea is that data are stored in structures (defined in module dataSpace; see below) which contain indices into data type, transmitter, and receiver dictionaries in addition to actual data values. The actual data space object types are not used by routines in this module, which only compute data values or set up linearized data functionals required for sensitivity calclations for a single observation. However the indices stored with the data vectors are passed to these routines, which have access to the associated dictionaries, and use these to control computations.

Data Types

MTrx :: This data type is trivial, containing just a single real 2-vector that defines location (y,z) of the MT observation. In practice, z always corresponds to the air/earth interface, but we allow for observations to be computed at any location in the domain. An array of MTrx objects is the receiver dictionary, which is private to this module and called *rxDict*. Indicies into *rxDict* are used to look up the location of the measurement. Again, the rationale for the receiver dictionary (which does seems overkill for the present 2D MT application) is to allow for measurement types that might require additional parameters to define the specific observation. (E.g., for network MT with very long dipoles one would need to define both ends of a dipole, reuiring 4 real numbers). With the dictionary approach at the level of the data space objects (the top level in the modular system!) all receiver descriptors look identical: an integer index into a dictionary that is not directly used (or accessible) to the top level routines. As for the transmitter dictionary, it would be possible to declare more than one receiver dictionary within module dataFunc; then the data type dictionary would have to include information about which receiver dictionary to use for elements of a particular dvec object. (E.g., if one wanted to mix MT/network MT in a single inversion, one approach would be to have separate dictionaries for the MT site locations, and the long dipoles). The definitions of the receiver dictionaries would be in module DataFunc, which at any rate is where details of all evaluation functionals would have to be coded.

Subroutines and Functions

The subroutines in this module are called by top level inversion modules. We have given all of these routines "generic" names. The names are not specific to the 2D MT problem, although the functionality is. The key point is that these are the methods that must be provided for any specfic application. If in a new application (e.g., 3D MT) the same functionality is provided by different routines with the same name (and the same interface, in terms of abstract data types), then no changes to the higher level routines should be required. Note that in f90/95 one can rename and overload procedures, so it would be possible to use more informative (i.e., specific to the application) names for procedures. A stickier point for Fortran is data types, as discussed elsewhere.

`subroutine RXdictSetUp(nSites,siteLocations)`

This sets up the receiver dictionary, given a list *nSites siteLocations*. This is a simple minded implementation for our simple case.

`subroutine nonLinDataFunc(ef,iDT,iRX,Z)`

This implements application of the non-linear data functional: Given an input EMsoln *ef* (which has a transmitter index attached already), for an observation of data type index *iDT*, and receiver index *iRX*, compute the observed data. To be generic, and to support complex data, as well as data with multiple components (for example, think impedance tensors) the output *Z* is ALWAYS declared as a complex (one dimensional) array (a single real scalar can be returned in this, if appropriate... and so can a 2x2 complex impedance tensor). For the specific implementation here, the frequency (and mode) can be obtained from the input EM solution *ef*, the mode and site location can be found by using *iDT* and *iRX* with the aid of *typeDict* and *rxDict*. The routine then calls appropriate interpolation routines in InterpEB2D to compute interpolation functionals for E and B field components; E and B interpolated to the data site are computed by forming the dot product of the interpolation functionals with the EM solution; and the ratio $Z = \mathrm{E/B}$ is formed and returned.

`subroutine linDataFunc(e0,iDT,iRX,Lz,Qz)`

This implements formation of the linearized data functional corresponding to the non-linear data functional implemented by routine nonLinDataFunc. Now *e0* is the background solution; inputs *iDT* and *iRX* serve the same function as for the non-linear functional. This routine returns, as an array of complex sparse vectors *Lz(:)*, the data functionals for the linearization of the impedance (around the background *e0*). *Qz(:)* is an optional array of sparse vectors, that are used to compute the component of the derivatives due to direct dependence of the non-linear data functional on the model parameter. As for the non-linear functional, *Lz* and *Qz* MUST be arrays to support efficient handling of multiple component data (e.g., impedance tensor elements).

`subroutine EMSparseQtoModelParam(Q,sigma0,dsigmaReal,dSigmaImag)`

This is another routine involved with implementation of **Q**, the part of the Jacobian due to dependence of interpolation functionals on model parameters (needed only for TM). This is just a wrapper routine, needed for technical reasons [to break a circular module use dependency, while keeping modelParam fully encapsulated, with no public attributes. This routine just calls routine `QtoModelParam` (in module ModelParam), after extracting the soln2D sparse vector from it's EMsparse data type wrapper (a type which ModelParam does not have access to ... or dependencies get circular)].

Module EMsolver

This is a high level interface/control module used by top level routines for initializing and using the solver core, described for the 2D finite difference implementation in the next section. The key public routines in this module have generic (abstract data type) arguments and can thus be called from the top-level inversion routines. A similar interface will generally be required for specific use of the top-level inversion modules, to hide implementation specific solver details. As is the case for DataFunc, to use the top level modules without changes, all public routines in the solver interface module will have to use the generic names, parameter lists and abstract functionality as in this example for 2D MT.

Data Types

EMsolnMTX :: derived data type for storing solutions from multiple transmitters. In many situations with multiple transmitters it might be useful to save background solutions for all transmitters for further subsequent calculations. This is just a convenient data structure for such storage.

Subroutines and Functions

There are two private data objects with attribute save in this module, which recall is specfic to 2D MT (even while the interfaces and names of the public routines are intended to be general): *b0* of type RHS, and a character string to keep track of which mode (TE or TM) was solved for on the last call to any routine within the module. The idea (not clear it is useful or implemented correctly at this point) is to minimize unnecessary reinitialization of the solver.

```
subroutine create_EMsolnMTX(d,eAll
```

Standard creation routine, for type EMsolnMTX.

```
subroutine initSolver(iDT,m,e0,e,comb)
```

This is the initialization routine, setting up the solver for data type *iDT*, model parameter *m*. As part of the setup a single EMsoln object *e0* is created for storage of the solution, and if necessary the scratch RHS data structure b0 is created (this is a module variable, so it is not returned to the calling routine). Outputs *e* and *comb* are optional arguments, which should be used when intiallizing for a sensitivity calculation. *e* is of type EMsoln and *comb* is of type RHS; these will be used for the solution and RHS for sensitivity calculations (note that *e0* is still needed to store a background solution in this case).

`subroutine exitSolver(e0,e,comb)`

Cleans up (deallocates), undoing the allocation of initSolver.

`subroutine fwdSolve(iTX,iDT,e0)`

Driver for 2D forward solver; finishes setting up equations, plus boundary conditions for transmitter *iTX* data type *iDT* , and returns the solution in *e0*. NOTE that the BC are set up internally, not provided as an input. The RHS object *b0*, which is used to store the BC input to the actual solver, is local to this module. Note that this routine automatically calls UpdateFreq to complete initialization of the solver (and trigger matrix factorization) for a particular frequency. Thus the matrix is refactored every time this routine is called. This probably should be modified to increase efficiency for direct sensitivity calculations!

`subroutine sensSolve(iTx,iDT,comb,e)`

Driver for the solver, for use with sensitivity calculations. This differs from fwdSolve only in that the forcing (*comb* of type RHS) is provided by the calling routine, not generated internally as for a standard forward calculation. This variant is required for sensitivity type calculations, where the forcing has to be computed based on the data locations/values or model parameter perturbation.

4.6 Data Space and Inversion Modules

These are the top-level inversion modules, which call only "generic interface" routines from the next level down, plus model parameter routines (which also have a generic interface). On the left in the diagram is the DataSpace module, which defines data space objects. These are public to the inversion modules–I.e. we do NOT try to hide data space object attributes from these top-level routines, since we want to take advantage of the data structure of multi-transmitter inverse problems in our inversion algorithms.

Module DataSpace

This module defines the basic data space objects, and the standard basic/linear algebra/dot product methods. In contrast to other basic data types everything about these data objects is intended to be public to the top levels. This is because the epxlicit structure of the data space objects is intrinsic to the data space inversion strategies that this system is (primarily) built to implement. In contrast, none of the lower level routines depend on the data space objects, except through the notion of dictionaries for data type, transmitter, and receiver. The structure of the data space objects will accomodate most imaginable frequency domain EM data types, but the specific structure is in some sense rigid.

Data Types

dvec: This is the basic building block for data space objects, containing data of a specific type for a single transmitter, but (possibly) for multiple components, and for multiple receivers. The number of components (*nComp*) is determined by the data type, and is fixed for all *nSites* receivers in a dvec object. All data objects are taken as real; complex data are represented as

two components, one for the real part, the other for the imaginary part. Thus, for example, 2D TE mode complex impedances for a series of N_s sites would have $nComp = 2$ and $nSite = N_s$. Note that for data with the type dictionary attribute *isComplex = .false.* *nComp* must be even. Two real arrays of size (*nComp, nSite*) are used to store data and data error standard deviations. It is possible to not actually store errors; in this case the logical *errorBar* attribute is set to false. Each dvec also carries indicies into the transmitter and data type dictionaries, and an array of indicies (one for each of the *nSite* sites) into the receiver dictionary.

dvecMTX: This is the basic object for a full data vector. This is essentially just an array of *nTX* dvec objects. Note that there is no restriction on the sizes (or types) of the component dvec objects. These may mix objects of different types (TE, TM, DC resistivity; some of which may be real, some complex) with different numbers of components or sites. I envision that all components would have/not have error bars, however. (the optional error bar feature is to allow data space objects to be more compactly stored for use in inversion algorithms; actual data always does have to have error bars!) There is also another attribute for this object: "normalized", which is used to keep track of whether or not the data vector has been normailized by the data errors. This is used for bookkeeping in the inversion algorithms.

Subroutines and Functions

The bulk of the routines in this module are standard basic data object routines, comparable in function and interface to those previously defined for other basic data types such as EM solutions and model parameters.

```
subroutine create_Dvec(nComp, nSite, d, errBar, iTx, iDt )
```

The last three arguments (*errBar, iTX and iDT*) are optional; these can be set before calling the create routine.

```
subroutine deall_Dvec(d)
subroutine copy_Dvec(d2,d1)
subroutine linComb_Dvec(a,d1,b,d2,dOut)
function dotProd_Dvec(d1,d2) result(r)
subroutine scMultAdd_Dvec(a,d1,d2)
```

The last of these returns *a*d1 + d2 in d2.* There are essentially the same routines provided for the full MTX (multi-transmitter) data vectors:

```
subroutine deall_DvecMTX(D)
subroutine copy_dvecMTX(D2,D1)
subroutine linComb_DvecMTX(a,D1,b,D2,Dout)
subroutine scMultAdd_DvecMTX(a,D1,D2)
function dotProd_DvecMTX(D1,D2) result(r)
```

And there are a few additional routines:

```
subroutine normalize_dvecMTX(d,dNorm)
```

This normalizes a dvecMTX object using error bars stored in the data structure. If the *normalized* attribute is *.true.* already nothing is done. *dNorm* is an optional output argument; if this is not present *d* is overwritten. Output dNorm must be allocated before calling.

`function count_dvecMTX(d) result(Ndata)`

This just counts the total number of data in a dvecMTX object–i.e., each component dvec object has ncomp*nSite elements; this routine sums over all transmitters to compute the total length *Ndata* of the data vector.

`subroutine SSdiff_DvecMTX(allData,allCalc,misfit)`

Computes sum of squared differences between two dvecMTX objects (*allData, allCalc*), normalizing by the data error standard deviation in allData (does not currently check to see if data errors are provided, or it data are already normalized.

Module MeasComb

This module manages measurement functionals, at the top level. Routines here deal with the isComplex and calcQ attributes, but otherwise do not reference specific attributes of data (or model) types. Note that here "comb" is not an abreviation of "combination" as in "linear combination"; rather this refers to the comb (as in hair comb) that defines the forcing of the adjoint for gradient calculations. (The comb is a sum of sparse vectors, one at each data site; if plotted an array of isolated spikes, somewhat like a comb).

`subroutine linDataMeas(e0,ef,d,dSigma)`

Multiply $\delta\mathbf{e}$ by $\mathbf{L}$ (and $\delta\mathbf{m}$ by $\mathbf{Q}$ if appropriate) for a single transmitter. Given perturbed (ef) and background (e0) electric field solutions evaluate *linearized* data functionals (applied to ef) for all sites represented in a dvec object. *dSigma* is an optional input, the model parameter perturbation that was used to solve for the solution perturbation *ef*. This is required if the data functionals depend on the model parameter, when this routine is used to multiply $\mathbf{J}$ (sensitivity matrix) times a model parameter vector.

`subroutine linDataComb(e0,d,comb,Qcomb)`

Create $\mathbf{L}^T$ and bfQ^T for a single transmitter. I.e., given the background electric field solution *e0* and a dvec object *d* for one transmitter, compute the adjoint of the measurement operator: i.e., *comb* of type RHS constructed from the scaled superposition of data kernals, scaled by the conjugate of data values. E.g., if *d* contains residuals, this can be used to set up for gradient calculation. *Qcomb* is an optional output, of type modelParam. *Qcomb* corresponds to the part of the sensitivity due to derivatives of data functionals with respect to model parameters, and needs to be included as an argument only if $calcQ = .true.$

`subroutine dataMeas(ef,d)`

Given the solution for a single TX (*ef*), compute predicted data at all sites, returning result in *d*. Data type, transmitter, site location, etc. is obtained from *d*, with reference to the dictionaries. Calls `nonLinDataFunc` to do the actual impedance calculation.

Module SensMatrix

This module implements, in an abstract way applicable to a broad range of EM problems, forward and sensitivity calculations. The routines in this module act on the highest level, e.g., mapping from modelParam objects to dvecMTX objects. A single call forward operator, as well as computation of the full Jacobian, and multiplication of arbitrary model space (or data space) vectors by the Jacobian (or its transpose). These represent the key operators needed to define a number of variants on gradient based optimization algorithms.

`subroutine calcSensMatrix(d,sigma0,dsigma)`

Calculate sensitivity matrix for data in *d*, linearized about model parameter *sigma0*. Result is returned in the array of model parameters *dsigma*, each of which corresponds to one row of the Jacobian (sensitivity matrix; **J**). Various standard initializations are required before calling this routine (loading grid, setting the modelParameter, calling setWSparams and to initialize (as described in other modules) and setting up data type, transmitter, and receiver dictionaries.

`subroutine Jmult(delSig,sigma0,d,eAll)`

Calculate product of Jacobian (sensitivity matrix, **J**) and a model parameter perturbation *delSig* for all transmitters in dvecMTX object *d*. If optional input parameter *eAll* is present, it must contain solutions for all transmitters (for conductivity parameter *sigma0*) in the same order that the dvec objects are stored in *d*. This last optional input argument can be used to avoid recomputing background solutions, e.g., in a CG solution scheme.

`subroutine JmultT(sigma0,d,dsigma,eAll)`

Outputs the transpose of of the Jacobian **J** times data vector *d* (type dvecMTXX); background conductivity (for linearization of Jacobian) is *simga0*. Output is a single conductivity parameter *dsigma*. If optional input parameter *eAll* is present, it must contain solutions for all transmitters (for conductivity *sigma0*), in the proper order.

`subroutine JmultT_MTX(sigma0,d,dsigma,eAll)`

Outputs results of multiplying the (dvec object) components of *d* by the transpose of the corresponding single transmitter data Jacobian (i.e., $\mathbf{J}_l^T \mathbf{d}_l, l = 1, ...N_f$). Input is the dvecMTX object *d*, and the background conductivity *sigma0*. Output is an array of conductivity parameters, stored in allocatble array *dsigma*, one for each transmitter. The sum of the array *dsigma* over transmitters is the same as the conductivity parameter output of JmultT. If optional input parameter *eAll* is present, it must contain solutions for all transmitters (for conductivity parameter *sigma0*) in the same order that the dvec objects are stored in *d*. This last optional input argument can be used to avoid recomputing background solutions, e.g., in a CG solution scheme.

`subroutine fwdPred(sigma,d,eAll)`

Calculate the predicted data for data types, transmitters, and receivers defined by dvecMTX object *d*. for conductivity parameter *sigma*. Optionally returns array of EM solutions *eAll*, one for each transmitter (if optinal argument *eAll* of type EMsolnMTX is present).

Module DCG

5 Forward Solver Core

This section describes the "modular" version of the 2D MT forward modeling routines constructed using forward modeling code from the REBOCC (Reduced Basis Occam) code of Siripunvaraporn and Egbert (2000). This provides (a) a simple example of "quick and dirty" modularization of some existing code; (b) a set of 2D modeling modules which can be used to support open boundary condition generation routines for the modular 3-D code; and (c) an initial modeling core for use with the initial 2D MT implementation of the modular inversion system.

Note that within the framework of the modular system there are higher level driver routines, in module EMsolver, which actually interact with and manage the initialization, update, and solver routines described here. These are described in the detailed module descriptions of the previous section.

There are three modules: **FwdTMmod, FwdTEmod, and WSfwd2D**.

The basic idea has been to take forward modeling code from the REBOCC 2D inversion of W. Siripunvaraporn essentially as is, and write some interface routines in f90 which provide a cleaner, more clearly defined, and more limited interface. These wrapper/interface module routines, can be called to set up and solve the 2D TE and TM equations without reference to the original subroutines taken from REBOCC; but all of the hard computations, are still done with the original codes. The interface modules, for TM and TE mode forward calculations, are contained in FwdTMmod and FwdTEmod. Usage is described below. These modules also contain, as private data, all coefficient and temporary work arrays needed for solution of the equations. There are routines for setting up and modifying all arrays efficiently, so that these may be used repeatedly for multiple equation solutions if desired. In particular, a simple LU factorization (with pivoting) is used for equation solution, After forming and factoring the system of equations solutions for a number of right hand sides can be computed.

The interface routines call subroutines that were originally part of the REBOCC 2D MT inversion code. These are now all in a single module, WSfwd2D. This module contains private module integer variables which are available to all routines in the module, and are used to define sizes of automatic arrays in the routines of WSfwd2D. In the original code integer parameters to set array sizes were set in an include file, and array sizes were fixed at compile time. In the modular code all arrays are dynamically allocated at run time. To minimize changes to the code, the same array size parameter names were used, but these are now module variables, which are set by an initialization routine, which must be called before the first use of any routines in WSfwd2D.

A second aspect of the wrapping is that the arrays that were originaly declared in driver programs for REBOCC are private variables in the wrapper modules, with the save attribute.

arrays are allocated with the appropriate size in setup routines which are in the interface modules FwdTMmod and FwdTEmod, and are called from higher level routines in the overall modular system (or other driver programs). These arrays (still with the names used in REBOCC) are then passed to the old REBOCC subroutines where the actual computation is done.

Modules FwdTEmod/FwdTMmod

Subroutines and Functions:

TE versions from FwdTEmod are listed; corresponding TM versions are in FwdTMmod. These follow the same naming convention, and have identical functionality (except they initialize for and solve the TM problem).

`subroutine FWD2DSetupTE(grid,m,IER)`

This routine does initial allocation and setup of coefficient arrays using input *grid* (of type gridDef2D) and model parameter *m* (of type modelParam). *IER* is returned as an error flag (needs work!). Note that this initial setup does not depend on frequency, and does not need to be redone as long as the grid and conductivity model remain unchanged.

`subroutine UpdateFreqTE(per)`

This routine updates the frequency dependent part of the operator, and factors the matrix in preparation for solver calls. This routine must be called after `FWD2DSetupTE`, before the first solver call, and then each time solutions for a different frequency are required.

`subroutine UpdateCondTE(Sigma)`

This routine updates the grid conductivity. Before calling the solver again, `UpdateFreqTE` must be called again to set the frequency, and to factor the matrix. This routine calls `CondParamToArray` to set conductivity in the array used by the REBOCC 2D modeling routines. If the model parameterization is changed, this mapping function (in module modelParam) will of course need to be changed; no changes inside the modeing code should be needed, however.

`subroutine Fwd2DsolveTE(b,Esol,IER)`

This routine computes the actual solution of the TE system with sources and bondary conditions specified in RHS data object *b*. The solution is returned in a complex array *Esol* of dimension Ny+1 x Nz+1 (Hmmm ... why haven't we gotten around to using vec2Dc, or even EMsoln object?)

`subroutine SetBoundTE(per,EXB)`

This sets boundary conditions for the forward problem (by solving a 1D problem, using conductivity as presently set in the TE modeling module).

`subroutine Fwd2DdeallTE()`

This just cleans up and deallocate arrays created when `FWD2DSetupTE` was initially called.

There are also some private routines, which are called only by other routines in this module: `EarrayToIntVec, IntVecToEarray, addEarrayToIntVec, multEarrayByArea`. These should not be needed for any outside use of the 2D modeling module, and are not described here (see source code for details).

Module WSfwd2D

Subroutines and Functions

There is really only one routine in this module that users need to be aware of. All others are called by routines in the driver/wrapper modules (FwdTEmod/FwdTMmod), but not by any other part of the modular system.

`subroutine SetWSparams(Ny,Nz,Nza)`

This is called to initialize array declaration size variables, which were initially integer parameters in an include file. *Ny* and *Nz* are the full grid size (including air layers) and *Nza* is the number of air layers. This has to be called before anything involving any kind of 2D modeling is invoked.