Modular System for Electromagnetic Inversion

Gary D. Egbert[1]*, Anna Kelbert[1] and Naser Meqbel[1]
[1] College of Oceanic and Atmospheric Sciences, Oregon State University,
104 COAS Admin Bldg., Corvallis, OR *97331-5503*, USA

Abstract

Building on the mathematical framework developed in the companion paper (*Egbert and Kelbert*, 2012) we describe a general modular system of computer codes for inversion of electromagnetic (EM) geophysical data. The modular approach can simplify maintenance of the inversion code, as well as development of new capabilities – e.g., allowing for new data types such as the inter-site transfer functions in magnetotellurics (MT), or modifying model regularization. It also allows for rapid adaptation of inversion algorithms developed for one purpose (e.g., three-dimensional MT) to other EM problems (e.g., controlled source EM). Basic data objects (model parameters, solution vectors, data vectors) are treated as abstract data types, with a standard set of methods developed for each class, including creation and destruction, and, as appropriate, linear algebra or other vector space methods. Operators required for gradient computations are developed as mappings between these basic object classes. Only these abstract data objects and mappings are manipulated by higher level Jacobian and inversion routines, with no reference to the problem specific details required for an EM method, or for the numerical implementation of the forward solver. Problem-specific components are instantiated at the lowest levels of the system, with details hidden from generic top-level routines by an interface layer. Our implementation, called ModEM, is written in Fortran 95, but also provides a more general template for EM inverse code development. ModEM is readily adaptable to joint inversion, both for multiple EM data types, and for EM in combination with other sorts of geophysical data. Parallelization using the Message Passing Interface (MPI) standard has been implemented at the top level, and is thus applicable to any problem embedded in ModEM. To illustrate the flexibility of the system, we consider applications to two- and three-dimensional MT inversion, as well as a simple controlled source EM problem.

*E-mail: egbert@coas.oregonstate.edu

1 Introduction

This paper is a companion to *Egbert and Kelbert* (2012, hereinafter "Paper I"), where we have presented a general mathematical framework for EM inversion methods based on the Jacobian of the non-linear forward mapping from model parameters to observations. Here we build on this basic mathematical framework to develop a modular system of computer codes (ModEM) for solving a broad class of frequency domain EM inverse problems.

There are several motivations for the developments discussed here. First, ModEM can function as a "test-bed" for comparing inversion algorithms, and for prototyping new approaches, such as the hybrid and multi-frequency schemes described in *Egbert* (2010). With the modular implementation, translating an abstract mathematical description of an inversion search algorithm into actual computer code is greatly simplified, and code reuse is maximized. Furthermore, ModEM allows the same inversion code to be used for a range of EM problems, and allows initial debugging and testing of algorithms on computationally fast two-dimensional (2D) problems before application to the more challenging three-dimensional (3D) case. More generally, ModEM allows for more rapid adaptation of inversion codes developed and tested for one purpose (e.g., 3D magnetotellurics (MT)) to other EM problems (e.g., controlled source EM (CSEM)). Finally, a modular approach can simplify maintenance of an inversion code, as well as development of new capabilities–e.g., allowing easier modification of the model parametrization (including different regularizations, and inclusion of prior information), or treatment of new data types (e.g., inter-site transfer functions in MT).

The first step in this enterprise, described in Paper I, is to abstract the basic data objects (e.g., model parameters, EM solution, data vectors) and operators (e.g., the Jacobian, model covariance) and to characterize their dependence and interactions. As detailed in Paper I, the Jacobian, which is central to all gradient based inversion schemes, can be decomposed (through application of the chain rule) into a series of simpler operators depending on the forward solver, the model parametrization, and the measurement process, and interactions among these components. The basic objects are treated as abstract data types, with a standard set of methods developed for each class, including creation and destruction, and, as appropriate, linear algebra or other vector space methods. Then, the component operators required for gradient computations are developed as mappings between the basic object classes. At the highest levels of the inversion system only these abstract data objects, and the abstracted mappings, are manipulated, with no reference to implementation specific details. Our approach could be termed object oriented, although in fact we frequently deviate from a strict adherence to this programming model, due to concerns about efficiency in the computationally intensive 3D EM inversion problem. For example, strict adherence to data hiding appears to result in too much overhead. Furthermore, the code has been

written in Fortran 95, and some aspects of an object oriented approach are difficult to implement in this programming language. We do make extensive use of advanced features of Fortran 95: all code is written using modules, with explicit interfaces for all routines, we hide data through use of the "private" attribute where practical, use operator overloading, and make extensive use of derived data types.

The paper is organized as follows. In Section 2 we briefly summarize the key results and notation of Paper I. Then in Section 3, we provide an overview of the modular system, and describe the core modules which define data and model space objects, and the methods needed to manipulate these. The next three sections describe the rest of the modular system, which is organized into three levels (Figure 1). The lowest level (Section 4) includes modules that define and implement the discrete numerical forward problem. The highest level (Section 6) includes the generic (problem independent) modules that define abstract operations with the Jacobian, inversion search algorithms, and parallelization. The intermediate level (Section 5) provides application specific details and an interface that hides low-level details from the generic top-level routines.

Throughout this paper we consider three specific examples. The first two, 2D and 3D MT, were used extensively also in Paper I. For the third case, we consider modifications required of lower level modules developed for 3D MT to invert controlled source EM (CSEM) data. In this example we take both sources and receivers to be vertical magnetic dipoles. For maximum clarity, and to avoid ambiguity, all modules discussed have been given specific names. The modules, which refer to a conceptual piece of code (and may in practice denote a group of modules), are written in **bold**. Any specific data types or subroutines mentioned in this text are written `verbatim`. *Italics* is reserved for the abstract terms, and for emphasis.

2 Preamble: Summary of Paper I

In our discussion we refer frequently to theory developed in Paper I, using notation which is summarized in Table 1. For completeness we summarize key points here. ModEM provides a general framework for solving regularized EM inverse problems, i.e., minimization of a penalty functional of the form

$$\mathcal{P}(\mathbf{m},\mathbf{d}) = (\mathbf{d}-\mathbf{f}(\mathbf{m}))^T\mathbf{C}_\mathbf{d}^{-1}(\mathbf{d}-\mathbf{f}(\mathbf{m})) + \nu(\mathbf{m}-\mathbf{m}_0)^T\mathbf{C}_\mathbf{m}^{-1}(\mathbf{m}-\mathbf{m}_0) \tag{1}$$

to recover an Earth conductivity model parameter vector $\mathbf{m}$, which provides an adequate fit to a data vector $\mathbf{d}$. In (1), $\mathbf{C_d}$ is the covariance of data errors, $\mathbf{f}(\mathbf{m})$ defines the forward mapping, $\mathbf{m}_0$ is a prior or first guess model parameter, ν is a trade-off parameter, and $\mathbf{C_m}$ (or more properly $\nu^{-1}\mathbf{C_m}$) defines the model covariance or regularization term.

The forward mapping requires solution of the frequency domain EM partial differential equation (PDE), which in discrete form is written generically as

$$\mathbf{S_m e} = \mathbf{b}. \tag{2}$$

The subscript $\mathbf{m}$ here denotes the dependence of the PDE operator on the unknown model parameter, and in the following is often omitted; $\mathbf{e}$ represents the discrete EM field solution; and $\mathbf{b}$ is the forcing (boundary conditions and/or source terms). Typically $\mathbf{e}$ will represent only the *primary* field (e.g., electric) that is actually solved for; the other *dual* field (e.g., magnetic) is then computed via a simple transformation operator $\mathbf{h} = \mathbf{Te}$. Simulated observations are computed from the solution $\mathbf{e}$ (and possibly $\mathbf{m}$) via

$$d_j = f_j(\mathbf{m}) = \psi_j(\mathbf{e}(\mathbf{m}), \mathbf{m}). \tag{3}$$

Using the chain rule a general expression for the Jacobian (or sensitivity matrix) $\mathbf{J}$ $(= \partial\mathbf{f}/\partial\mathbf{m})$ can be given:

$$\mathbf{J} = \mathbf{L}\mathbf{S}_{\mathbf{m}_0}^{-1}\mathbf{P} + \mathbf{Q}. \tag{4}$$

In (4) the matrix

$$\mathbf{P} = -\left.\frac{\partial}{\partial \mathbf{m}}\left(\mathbf{S_m e}_0\right)\right|_{\mathbf{m}=\mathbf{m}_0} \tag{5}$$

gives the sensitivity of the operator-solution product to perturbations in the model parameter; this depends on details of the numerical model implementation and the conductivity parametrization. Assuming the forward operator can be written in the form

$$\mathbf{S_m e} \equiv \mathbf{S}_0\mathbf{e} + \mathbf{U}\ (\pi(\mathbf{m}) \circ \mathbf{Ve}), \tag{6}$$

where $\mathbf{S}_0$, $\mathbf{U}$ and $\mathbf{V}$ are some linear operators that do not depend on the model parameter vector $\mathbf{m}$, $\pi(\mathbf{m})$ is a (possibly non-linear) mapping from the model parameter space to the numerical grid, and $(\circ)$ denotes the component-wise multiplication of vectors, an explicit expression for the operator $\mathbf{P}$ can be given,

$$\mathbf{P} = -\mathbf{U}\ diag(\mathbf{Ve}_0)\Pi_{\mathbf{m}_0}. \tag{7}$$

In (7), $\Pi_{\mathbf{m}_0}$ is the Jacobian of the model parameter mapping $\pi(\mathbf{m})$ evaluated at the background model parameter $\mathbf{m}_0$. All of the usual forms for 2D and 3D EM induction operators can be expressed as in (6); e.g., for the second-order 3D staggered-grid finite difference equation for the electric field

$$\nabla \times \nabla \times \mathbf{E} + i\omega\mu\sigma\mathbf{E} = 0 \tag{8}$$

$\mathbf{S}_0$ corresponds to the discrete curl-curl operator, $\mathbf{U} \equiv i\omega\mu\mathbf{I}$, $\mathbf{V} \equiv \mathbf{I}$, and $\pi(\mathbf{m}) \equiv \sigma(\mathbf{m})$ is a mapping from the model parameter space to the cell edges, where electric field components are defined.

The matrix $\mathbf{L}$ in (4) represents the linearized data functionals. This can be decomposed into two sparse matrices as $\mathbf{L} = \mathbf{A}^T \Lambda^T$, where columns of Λ are sparse vectors which represent evaluation functionals for point observations of the electric and magnetic fields. The operator $\mathbf{T}$ is generally involved in evaluation of dual field components. The matrix $\mathbf{A}$ depends on details of the (generally non-linear) observation functionals (e.g., impedance, apparent resistivity), which may combine magnetic and electric measurements from one or more locations. When either the evaluation functionals, or the field transformation operator $\mathbf{T}$ have an explicit dependence on the model parameter there is an additional term in the sensitivity matrix, which we have denoted $\mathbf{Q}$ in (4).

The Jacobian represents a linear mapping, giving the perturbation to the data resulting from a model parameter perturbation ($\delta\mathbf{d} = \mathbf{J}\delta\mathbf{m}$). A wide range of gradient-based inversion algorithms make use of this operator, along with the transpose or adjoint ($\delta\mathbf{m} = \mathbf{J}^T\delta\mathbf{d}$). ModEM does not generally form the Jacobian, or the component matrices in (4), but rather implements the solver for the discrete system $\mathbf{S}_\mathbf{m}^{-1}$, the operators $\mathbf{P}$, $\mathbf{L}$, $\mathbf{Q}$, and the compound Jacobian operator $\mathbf{J}$, together with adjoints. These operators, together with model and data covariances, are then used to implement a range of gradient-based inversion algorithms. As discussed in Paper I, and in detail below, in most EM inverse problems there is significant structure to the data vector, implied by the multiplicity of transmitters and receivers. This structure is also reflected in the Jacobian and the component matrices, and in the organization of ModEM.

3 Data Space, Model Space, and System Overview

The inputs and outputs of an inversion algorithm (namely, data vectors $\mathbf{d}$ and model parameters $\mathbf{m}$) are defined in modules **DataSpace** and **ModelSpace**, shown at the top and bottom, respectively, of Figure 1 where the structure of the modular system is summarized. These modules have no intrinsic dependence on other parts of the system, which is organized into three layers. At the top (Level I) modules are generic, intended to be used without change for a wide range of problems. The actual inversion search algorithms are implemented at this level, as are driver routines for calculations with the Jacobian. The Level I routines use *data vector* ($\mathbf{d}$) and *model parameter* ($\mathbf{m}$) objects, along with the *solution vector* objects that define the solution ($\mathbf{e}$) and forcing ($\mathbf{b}$) for the forward problem, and a suite of methods and operators which must be instantiated, for each specific application, through appropriate Level II and III modules.

The basic numerical modeling methods – grids, EM field objects, forward solver, EM field interpolation functionals – are implemented in Level III. No specific data type or procedure names defined at this level are referenced by the generic Level I routines, so there is a great deal of flexibility in actual implementation. In fact, as for one of the examples discussed below, it may

Symbol		Represents
$\mathbf{m}, \mathbf{m}_0$	$\in \mathcal{M}$	model parameter vectors
$\pi(\mathbf{m})$	$: \mathcal{M} \mapsto \mathcal{S}_{P,D}$	mapping from model parameter to primary or dual grid
$\Pi_{\mathbf{m}_0}$	$: \mathcal{M} \mapsto \mathcal{S}_{P,D}$	$\partial\pi/\partial\mathbf{m}$, evaluated at $\mathbf{m}_0$
$\mathbf{C}_\mathbf{m}$	$: \mathcal{M} \mapsto \mathcal{M}$	model covariance
$\mathbf{e}, \mathbf{e}_0$	$\in \mathcal{S}_P$	solution vectors on the primary grid
$\mathbf{d}$	$\in \mathcal{D}$	data vector
$\psi_j(\mathbf{e}(\mathbf{m}), \mathbf{m})$	$: \mathcal{S}_P \mapsto \mathbb{C}$	j'th data functional, in general non-linear
$\mathbf{l}_j$	$\in \mathcal{S}_P$	$\partial\psi_j/\partial\mathbf{e}$, evaluated at $\mathbf{e}_0, \mathbf{m}_0$; j'th row of $\mathbf{L}$
$\mathbf{q}_j$	$\in \mathcal{M}$	$\partial\psi_j/\partial\mathbf{m}$, evaluated at $\mathbf{e}_0, \mathbf{m}_0$; j'th row of $\mathbf{Q}$
$\mathbf{L}$	$: \mathcal{S}_P \mapsto \mathcal{D}$	sparse matrix constructed from $\mathbf{l}_j$
$\mathbf{Q}$	$: \mathcal{M} \mapsto \mathcal{D}$	sparse matrix constructed from $\mathbf{q}_j$
$\mathbf{S}_\mathbf{m}^{-1}$	$: \mathcal{S}_P \mapsto \mathcal{S}_P$	forward solver; $\mathbf{S}_\mathbf{m}\mathbf{e} = \mathbf{b}$
$\mathbf{P}$	$: \mathcal{M} \mapsto \mathcal{S}_P$	operator $-\partial\left(\mathbf{S}_\mathbf{m}\mathbf{e}_0\right)/\partial\mathbf{m}$, evaluated at $\mathbf{m}_0$
$\mathbf{J}$	$: \mathcal{M} \mapsto \mathcal{D}$	full Jacobian $\partial\psi/\partial\mathbf{m}$; $\mathbf{J} = \mathbf{L}\mathbf{S}_{\mathbf{m}_0}^{-1}\mathbf{P} + \mathbf{Q}$
$\mathbf{T}$	$: \mathcal{S}_P \mapsto \mathcal{S}_D$	linear mapping from the primary to dual grid
$\tilde{\mathbf{T}}_{\pi(\mathbf{m}_0),\mathbf{e}_0}$	$: \mathcal{S}_{P,D} \mapsto \mathcal{S}_D$	$\partial\left(\mathbf{T}_{\pi(\mathbf{m})}\mathbf{e}_0\right)/\partial\pi$, evaluated at $\pi(\mathbf{m}_0)$
λ^P	$\in \mathcal{S}_P$	primary grid interpolation coefficients (sparse)
λ^D	$\in \mathcal{S}_D$	dual grid interpolation coefficients (sparse)
Λ_P^T		sparse matrix of primary grid coefficients
Λ_D^T		sparse matrix of dual grid coefficients
$\tilde{\Lambda}_P^T$		$\partial\left(\Lambda_P^T\mathbf{e}_0\right)/\partial\pi$, evaluated at $\pi(\mathbf{m}_0)$
$\tilde{\Lambda}_D^T$		$\partial\left(\Lambda_D^T\mathbf{T}_{\pi(\mathbf{m}_0)}\mathbf{e}_0\right)/\partial\pi$, evaluated at $\pi(\mathbf{m}_0)$
γ_j		dependence of j'th data functional on local fields
a_{jk}^P	$\in \mathbb{C}$	derivative of γ_j with respect to k'th primary field component
a_{jk}^D	$\in \mathbb{C}$	derivative of γ_j with respect to k'th dual field component
$\mathbf{A}^T$		matrix of coefficients a_{jk}^P, a_{jk}^D

Table 1: List of notation from Paper I. Here, $\mathcal{M}$ indicates the model space, $\mathcal{D}$ the data space, and $\mathcal{S}_{P,D}$ stand for the spaces of EM fields defined on primary and dual grids, as defined in Paper I.

be possible to take much of this code from an independent modeling or inversion application. We outline below the structure and dependencies required for these modules to be useful within the system, and briefly discuss implementation of 2D and 3D finite difference forward modeling modules as examples.

Level II modules provide an interface between the generic Level I modules, and the numerical implementation specific Level III modules. Source and receiver details for each specific application are implemented at this level. Multiple EM inverse applications may be developed using the same (or slightly modified) base of numerical discretization modules, through modifications to routines at this level only. As a specific example of this, we discuss below our extension of a 3D MT inversion based on finite difference forward modeling to also treat a simple controlled source problem. Although Level II modules must depend on details of the underlying Level III implementation, even here there are significant opportunities for code reuse: modules for a previously developed application can serve as templates for the variants that must be developed for a new application.

Model and data spaces are treated quite differently in the modular system. For the model space we have adopted an object-oriented approach, minimizing interaction of this fundamental component with the rest of the system. Hiding all details of *model parameter* objects from the rest of the system allows maximal flexibility in model parametrization (and regularization), simplifying modification of this critical part of the inversion. We take a very different approach with *data vector* objects, which have been implemented with a fixed, but quite general structure, suitable for a wide range of EM inverse problems. This "breach" in the abstract object oriented approach allows us to make use of problem specific structure in data vectors to develop more efficient inversion algorithms. As discussed in Paper I, in most cases (e.g., 2D and 3D MT; CSEM) there are multiple receivers (sites) and multiple transmitters (e.g., different frequencies for MT, different source locations or orientations for CSEM). The resulting structure in the data has significant implications for inversion algorithms – e.g., a separate forward problem must be solved for each transmitter, but not for each receiver. And, for CSEM, computations with the Jacobian can be "factored" into components that depend on the receiver and on the transmitter. To simplify development of inversion algorithms that take advantage of such data vector structure, we have built this structure into our implementation of the basic data vector objects, and made all data vector object attributes public.

DataSpace. Two key observations motivate the structure of the basic data space objects, which underpin the Level I inversion modules. First, EM data are generally multivariate. Even considering a simple problem such as 2D MT, data for a single mode and frequency, at a single site, consists of two components: real and imaginary parts of an impedance, or apparent resistivity and phase. For 3D MT, for a single frequency and site the full impedance tensor consists of four

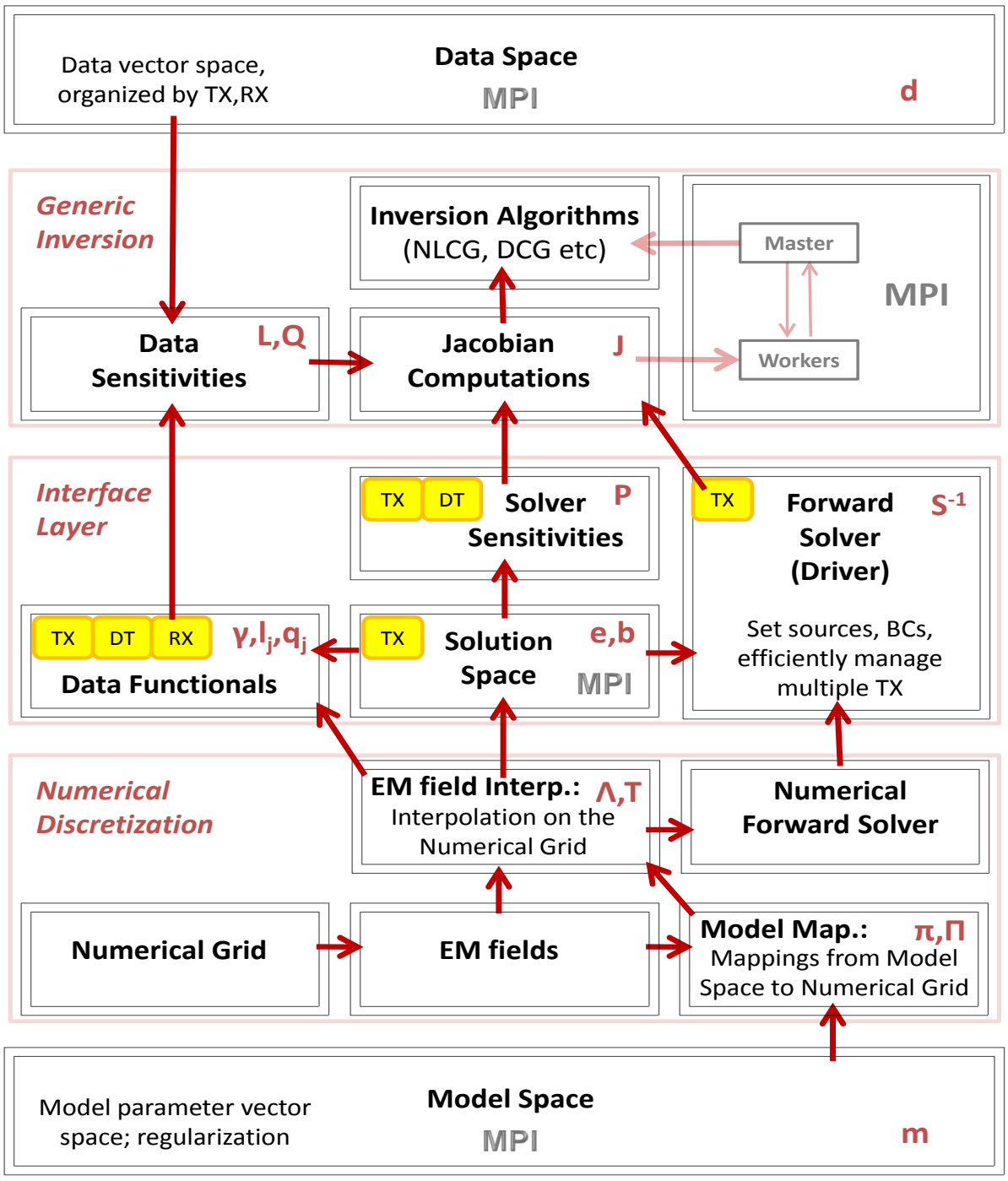

Figure 1: Schematic overview of ModEM. Individual modules, which are described in Sections 3–6, are denoted by boxes, with dependencies indicated by arrows. Some boxes are also marked with symbols to indicate the vectors and operators from *Egbert and Kelbert* (2012) which they implement (see also Table 1). The shaded small boxes indicate which dictionaries are used in each module. The (optional) Message Passing Interface (MPI) module used for the parallel implemen-

Name & Purpose	Example Attributes
transmitters (TX): list of forward problems	Number of polarizations; period / frequency; source configuration; TE or TM mode (for 2D MT).
data types (DT): list of data functional types	Name & integer key to determine transfer function type; number and order of components; units; real vs complex; pointer to the receiver dictionary, if there are several.
receivers (RX): list of site/receiver meta-data	Location (in 2D or 3D); station code; location and code of remote station.

Table 2: **Dictionaries** – lists of *transmitters* (TX), *data types* (DT) and *receivers* (RX), which fully describe the forward problem, site locations, type of data, and any meta-data required to compute predicted responses.

complex, or 8 real components. If vertical field transfer functions are added, there are 12 real data components. As discussed in Paper I, these components couple multiple source polarizations and cannot be evaluated or treated in isolation. Second, these multi-component data can be organized, or indexed, by three attributes which we refer to as *transmitter*, *data type*, and *receiver*. The *transmitter* attribute uniquely defines the forward problem that must be solved, including both the specific partial differential equation (PDE; in general this will depend on frequency) as well as the sources and boundary conditions. The *receiver* attribute is used to define, in conjunction with *data type*, the measurement process that must be applied to the forward solution to allow comparison between model and data.

These attributes (*transmitter*, *data type* and *receiver*) are treated abstractly at the level of the data space and inversion modules. This is achieved by storing the actual information associated with these attributes as lists, which we call *dictionaries*. Thus, the *transmitter* (TX) dictionary has an entry for each unique forward problem, providing any data such as the frequency or geometry of the source required to set up and solve each forward problem. Entries in the *data type* (DT) dictionary define general data functional types included in the inversion, such as impedance, vertical field transfer function, apparent resistivity. The same sort of data will of course often be obtained at a number of different site locations, or more generally, from a number of different configurations of an observing system. This information is provided through the *receiver* (RX) dictionary. The dictionaries, which are summarized in Table 2, are each stored in a separate module. These modules define specialized (problem specific) derived data types used for dictionary entries, along with routines for creating and filling the dictionary lists.

The lists of possible dictionary entries, and their explicit meanings, will depend on the application. For example (see Table 2), for 3D MT *transmitter* defines only the frequency (i.e., each TX

corresponds to a pair of linearly independent source polarizations). For 2D MT, *transmitter* also defines the source polarization or mode, TE or TM. For both 2D and 3D MT problems *receiver* refers to the site location, and *data type* can be used to distinguish between cases where vertical field transfer functions are or are not available, or between complex impedances and apparent resistivity/phase. Additional *data types* within the framework of 3D MT inversion would also be possible – e.g., inter-station transfer functions. Note that for such a *data type*, a full description of the *receiver* would require locations for a pair of sites.

The *data vector* objects, then, are no longer burdened with the problem-specific meta-data; instead, they store the actual observations, in conjunction with a set of pointers to the dictionaries, which determine the meta-data, in the form of *transmitter*, *data type* and *receiver*. To maintain generality all data are taken to be real numbers – real and imaginary parts of complex data (e.g., impedances) are treated as separate observations. Complex data are in fact common in frequency domain EM inverse problems, but we want to allow also for data which are more naturally treated as real numbers, e.g., for DC resistivity data (which are intrinsically real) or for apparent resistivity and phase which might not both be available or equally useful for inversion. While generality suggests representation of all data as real numbers, it is still worth keeping track of the cases where pairs of components do in fact correspond to a single complex number. As discussed in Paper I, sensitivities for real and imaginary parts of the complex data can be computed simultaneously, at the cost of a single adjoint solution. To allow for this efficiency, we need to distinguish between the situations when data are actually complex, as opposed to intrinsically real. The stored observations are therefore accompanied by a binary variable (`isComplex`) which is true if the data of this type are complex, with pairs of adjacent real components to be understood as representing a single complex number. It is also useful to store this information in the *data type* dictionary.

The basic building block for data space objects is a derived data type `dataBlock_t`, which we denote here as $\mathbf{d}_j$. Each vector $\mathbf{d}_j$ stores multi-component data of a single *data type* (index j), but for multiple *receivers*. The receivers are kept track of through a list of indices into the receiver dictionary. This list, together with the *transmitter* and *data type* indices, are stored as fields in the `dataBlock_t` structure. Because all data in $\mathbf{d}_j$ correspond to the same transmitter, predicted data for all components (and all receivers) can be computed from the same forward solution. Furthermore, all data stored in a single vector $\mathbf{d}_j$ are of the same *data type*, and hence consist of the same number of components, in the same order. For 3D MT a typical data block $\mathbf{d}_j$ would contain all impedance tensor components for all sites, at a fixed frequency. In the 2D MT case, $\mathbf{d}_j$ consists of TE or TM mode complex impedance data for all sites at one frequency. For the CSEM example, $\mathbf{d}_j$ would consist of all vertical magnetic field components at all receiver locations for a fixed transmitter location/frequency. Optionally, error bars can be stored along with the actual data, in a separate real array.

This basic data block, then, contains data of a single data type, computed for one forward

Module	Level	Type Name	Usage
DataSpace	I	`dataBlock_t`	$\mathbf{d}_j$: basic data block for one type/transmitter
		`dataVector_t`	$\mathbf{d}$: single-transmitter data vector for multiple data types, array of `dataBlock_t` objects
		`dataVectorMTX_t`	$\vec{\mathbf{d}}$: multi-transmitter data vector, array of `dataVector_t` objects
SolnSpace	II	`solnVector_t`	$\mathbf{e}$: solution vector (one transmitter)
		`rhsVector_t`	$\mathbf{b}$: right hand side forcing and boundary conditions
		`sparseVector_t`	$\mathbf{l}$: sparse solution vector for one transmitter
		`solnVectorMTX_t`	$\vec{\mathbf{e}}$: multi-transmitter solution vector, array of `solnVector_t` objects
Grid	III	`grid_t`	the underlying numerical grid; no details of this are used at the upper level
ModelSpace	III	`modelParam_t`	$\mathbf{m}$: model parameter

Table 3: Data Types used by Level I Routines.

problem (*transmitter*). Note that there may be multiple vector $\mathbf{d}_j$ objects for one *transmitter* (corresponding to different *data types*). They are united by the fact that their components can all be computed from the same forward solution. We therefore define the `dataVector_t` as an array of these `dataBlock_t` components for a single transmitter, but allowing for multiple data types. For example, in 3D MT, a variable of derived data type `dataVector_t` may consist of two vectors $\mathbf{d}_j$ for one frequency: one containing impedances, the other inter-station transfer functions. The full data vector, combining data for all transmitters, is then an array of component `dataVector_t` objects, which is again stored in a new derived data type `dataVectorMTX_t` (`MTX` for *multiple transmitters*). For instance in the 2D MT case some elements in the `dataVectorMTX_t` array may be for TE, and some TM, each for a range of frequencies. More generally, objects of type `dataVectorMTX_t` may mix very different kinds of data. For instance, CSEM, DC resistivity, or even some sort of seismic or potential field data could in principal be included, along with MT impedances. Derived data types used to represent data vectors, and other key data objects used by Level I routines, are summarized in Table 3.

In addition to defining the basic derived data types, the `DataSpace` module contains routines for creating, zeroing, copying, reading, writing, and deallocating data space objects. A list of these routines and naming conventions, which are used throughout the modular system when defining data types, is given in Table 4. Routines are also provided for basic algebraic operations including

Routine Name	Implements
`create_dataBlock(nComp, nSite, d, isComplex, errorBar)`	constructor
`deall_dataBlock(d)`	destructor
`zero_dataBlock(d)`	$\mathbf{d} = 0$
`copy_dataBlock(d2, d1)`	$\mathbf{d}_2 = \mathbf{d}_1$
`linComb_dataBlock(a, d1, b, d2, d3)`	$\mathbf{d}_3 = a \times \mathbf{d}_1 + b \times \mathbf{d}_2$
`scMult_dataBlock(a, d1, d2)`	$\mathbf{d}_2 = a \times \mathbf{d}_1$
`scMultAdd_dataBlock(a, d1, d2)`	$\mathbf{d}_2 = a \times \mathbf{d}_1 + \mathbf{d}_2$
`dotProd_dataBlock(d1, d2) result (r)`	$r = \mathbf{d}_1 \cdot \mathbf{d}_2$
`read_dataBlock(d, fname)`	constructor from file
`write_dataBlock(d, fname)`	output to file

Table 4: Basic Routines for dataBlock_t data type. These include the routines for creation, deallocation, I/O and copying, plus routines for linear algebra and dot products. Similar sets of routines have been implemented for all the important data types that are discussed in this paper.

addition of two data vectors, multiplication of a data vector by a scalar, and forming linear combinations of data vectors. Routines for data vector dot products, and for normalization by error bars are also included. Effectively three sets of routines are supported: for basic `dataBlock_t`, and the compound `dataVector_t` and `dataVectorMTX_t` objects, with methods for the latter constructed from those for the former in the obvious ways.

ModelSpace. To decouple the model parametrization and regularization from the remainder of the inversion system, the basic data type (`modelParam_t`) in this module has all object attributes private, and therefore inaccessible to the remainder of the modular system. This guarantees that the rest of the inversion system is independent of any specific details in model parameter implementation. It also requires that all routines that interact with model parameter internal attributes must be defined within this one module. There are essentially three subgroups of such routines. The first group consists of the standard routines for creation, deallocation, I/O and copying, and for linear algebra and dot products. These are largely self explanatory, and follow the same naming and interface conventions used for data vectors, and other basic objects required for the inversion (see Table 4). Note that the model parameter vector space routines are used exclusively (and extensively) by the inversion search algorithms.

The second group of routines in the **ModelSpace** module implement the model covariance $\mathbf{C_m}$. For the approach sketched in Paper I, with regularization based on application of a covariance smoother, essentially two routines are needed. The first initializes covariance parameters, which are stored in a data structure private to this module, and the second applies the symmetric covariance operators $\mathbf{C_m}^{1/2}$ and $\mathbf{C_m}$ to a model parameter $\mathbf{m}$, with output of another model

parameter.

The third group (denoted as **ModelMap** in Figure 1) consists of mappings between the model parameter and the primary or dual EM fields, including the implementations of $\pi(\mathbf{m})$, the Jacobian of this mapping $\Pi = \partial\pi/\partial\mathbf{m}$, and its adjoint Π^T (see Table 1 for a summary of notation used in Paper I). These mappings depend critically on details of both the model parameter and the numerical discretization, and essentially serve as the interface between these components of an application. They are therefore used by many of the other modules including forward modeling, field component evaluation, and multiplication by the matrices $\mathbf{P}$ and $\mathbf{Q}$. Ideally these mappings between the model parameter and the grid would be a separate module, as suggested in Figure 1, or a *submodule*, as would be allowed in Fortran 2003 (*ISO/IEC TR 19767*). However, in Fortran 95 the only way to ensure that the details of the model parametrization stay private outside of **ModelMap**, is to keep this group of routines in **ModelSpace**. Even so, it is still sensible to maintain source code for this group, and for the model covariance group, in separate files, which are included in the main **ModelSpace** module at compile time. This separation is useful since, for example, even with the same model parametrization, one could modify the covariance implementation. And the same base model parameter module could be used with different numerical implementations of the solver, which would generally require modification of the **ModelMap** group.

Our initial model parameter implementations for both 2D and 3D MT problems are very simple, and classical: conductivity or log conductivity is defined independently on each of the cells in the underlying numerical grid. For the 3D case we have also implemented (as an option within the same basic module) parametrization in terms of blocks on a different (generally coarser) grid than that used for the discretization required of the numerical solver. Again, these very specific simple implementations of this module provide a template for alternative parameter representations. Specific forms for the **ModelMap** operators corresponding to the combination of this simple model parametrization and the 3D finite difference numerical solver for the electric field equation are given in Paper I. With regard to the model covariances we have used simple smoothing covariances, implemented as in *Siripunvaraporn and Egbert* (2000) in the 2D case, and *Siripunvaraporn et al.* (2005) in the 3D case. For the 3D MT case, we have also implemented a *recursive autoregressive* (AR) covariance, loosely based on *Purser et al.* (2003a,b). Our implementation has a number of smoothing parameters, which may be adjusted through a configuration file. Horizontal and vertical smoothings may be different; moreover, the extent of horizontal smoothing may vary with depth. The configuration file also allows the user to divide the model domain into disjoint pieces, allowing smoothing to be turned off across domain boundaries, and also allowing model parameters within a domain to be frozen.

4 Level III: Numerical Solution of the EM Problem

The core of this group of modules (Level III in Figure 1) is the actual solver for the forward and adjoint problems, **EMsolver**. The solver interacts with the rest of the system through objects which represent the discretized electric and/or magnetic fields. In the terminology of Paper I these are elements of the primary and dual field spaces, which we denote $\mathcal{S}_P$ and $\mathcal{S}_D$. These objects, which are defined in **EMfield**, provide inputs and outputs for the solver, representations of the component evaluation functionals λ^P and λ^D, and the outputs of the model parameter mappings $\pi(\mathbf{m})$ and the linearization Π (see Table 1). These basic field objects, and the solver itself, in turn depend on the underlying numerical discretization, defined in the **Grid** module. We also include with this group the module **EMfieldInterp**, which implements the interpolation and transformation functionals required for point evaluation of the EM field components.

All of these modules are specific to a particular numerical approach to discretizing and solving the EM forward problem in a particular geometry. For example, finite difference vs. finite elements, 2D MT vs. 3D MT, spherical geometry vs. Cartesian geometry, all require different instances of this group of modules. On the other hand, the same set of modules can be used for any source/receiver configurations reasonably modeled with a particular numerical approach. For example, we have implemented inversions for both 3D MT and a simple CSEM problem using a common set of Cartesian 3D finite difference modules. With the exception of `grid_t` objects, specific data types and procedures defined in the numerical discretization modules are not referenced by Level I inversion routines. These modules thus must have specific functionality, but details such as routine names, data types, and interfaces are not important. To be concrete in our discussion we consider implementation of 2D and 3D finite difference solvers as examples.

Grid. The underlying numerical grid is stored in a separate module, which defines data type `grid_t`. Attributes of this object need to be accessible to routines in other Level III modules, and are thus declared public. Higher level (I and II) modules reference and pass around instances of `grid_t`, but do not access internal attributes. Thus, the internal structure of the grid is completely flexible, and can be adapted to the specific numerical approach. For the 2D and 3D MT examples based on finite difference modeling the required grid data structure is quite simple. The rectangular grid is completely characterized by the number and spacing of grid nodes, although we have also found it useful to allow for storage of various other pre-computed grid variables, such as the coordinates of nodes and cell centers. Optional grid origin and orientation, to provide geographic context, are also useful. A much more complicated data structure would be required to specify a non-structured grid for a finite element modeling approach. The essential requirement from the perspective of ModEM is that there be a single data structure that provides a self contained complete description of the grid. A pointer to this structure is included in most of the generic objects (*model parameters*, *solution vectors*) that depend on the grid for context.

Public routines to create and deallocate the grid data type are essentially all that is required in this module, although additional functionality (which would be called upon only by other Level III routines) may be useful for more complex grid structures.

EMfield. This module defines the basic data types used to represent objects in the spaces of primary and dual EM fields $\mathcal{S}_P$ and $\mathcal{S}_D$. These objects are the building blocks for the abstract encapsulated solution vector and right hand side objects ($\mathbf{e}$ and $\mathbf{b}$) defined in module **SolnSpace**, as discussed below in Section 5.

Our implementation for the 3D finite difference EM case provides a good example for the functionality that this module should provide. For this case the basic object is a discrete representation of a three-component complex vector field on a staggered grid. In our implementation these are stored as three separate 3D arrays, encapsulated in a single derived data type, together with a pointer to the underlying basic grid object, and a tag to indicate if the vector field is defined on the primary (edges) or dual (faces) grid. Several variants on these basic staggered grid EM vector field objects are also supported. First, it has proved useful to allow also for scalar fields, in this case naturally defined on the cell centers, or the cell nodes, and also to allow for real as well as complex versions of both scalar and vector fields. For example, complex scalars are used to represent electric potentials (used in the solver for the divergence correction; *Smith* (1996)); a real vector can be used to represent electrical conductivity defined on cell edges, i.e., $\sigma(\mathbf{m})$ in the notation of Paper I. Second, we support a sparse vector representation for elements of $\mathcal{S}_P$ and $\mathcal{S}_D$, to allow efficient representation and storage of measurement functionals. For all of the variants of the basic vector fields there are routines for creation, deallocation, linear algebra, dot products, and point-wise multiplication (i.e., the Hadamard product ($\circ$), as in Paper I). Algebraic routines for combinations of types (e.g., dot products of sparse and full vectors, as are used for evaluation of field component functionals; point-wise multiplication of complex and real vector fields, required to compute $i\omega\mu\sigma\mathbf{E}$) are also included. Finally note that we also support a specialized data structure to store data for boundary conditions – e.g., the tangential components of the electric field on the boundaries for 3D MT. We have also implemented basic differential operators such as $\mathbf{S}_0 \equiv \nabla \times \nabla\times$, ∇ and $\nabla\cdot$ in the 3D FD **EMfield** module. Although not essential, defining this functionality in the basic numerical discretization module can be useful, since depending on the application, these operators may be used to implement multiplication by $\mathbf{P}$ and $\mathbf{P}^T$, as well as operators needed by the forward solver.

For the 2D MT case the situation is similar, though somewhat simpler. Since the forward problems for TE and TM mode can be reduced to scalar form (i.e., involving only a single component of the field), no vector field objects are required, only scalar fields defined on a 2D rectangular grid. As for the 3D case we allow for real and imaginary, and sparse and full storage versions of the objects, with a similar set of methods.

EMsolver. EM forward modeling, the core of the inversion, is represented in Figure 1 as a single module, but in general may have more complex structure itself. Indeed, implementations for the 2D and 3D MT cases are quite different, in part reflecting the very different development histories for these two examples. The 2D MT forward modeling code was originally written in Fortran 77 for a different purpose, while for the 3D MT case forward modeling code was developed from scratch in Fortran 95 as an integral part of ModEM. The 3D code thus makes extensive use of the **EMfield** and **Grid** modules, while the 2D code does not. The internal details of the core forward modeling routines are not central to our discussion of ModEM, although interfaces and functionality of these components are important. The basic requirements again can be elucidated by consideration of the 3D and 2D MT examples.

The 3D solution algorithm is essentially identical to that described in *Siripunvaraporn et al.* (2002). The quasi-static frequency domain Maxwell equations are reduced to a second order vector diffusion equation in the electric fields, as given by (8) with the tangential components of **E** defined on domain boundaries. Eq. (8) is discretized using finite differences on a staggered grid (e.g., *Smith* (1996)) with the resulting discrete system of linear equations solved iteratively using a quasi-minimum residual (QMR) scheme, with a level-1 incomplete LU decomposition for pre-conditioning. As in *Smith* (1996) and *Siripunvaraporn et al.* (2002), a divergence correction is applied periodically. In our implementation the 3D forward modeling code itself is quite modular. For example, differential operators required for both (8) and the divergence correction are taken from the basic discretization module **EMfield**. And, the QMR solver and ILU-1 pre-conditioner are modules which can be easily replaced by alternative iterative solvers/pre-conditioners.

To be useful as a core solver for the modular inversion system, several conditions must be met. First, the solver should be general, in the sense that the solution can be computed correctly for arbitrary sources and boundary data. For the usual MT forward problem there are no source terms, except on the boundaries, but adjoint solutions required for sensitivities have sources at the data locations. Second, capability to solve the transposed system is required. As discussed in Paper I, with appropriate scaling to account for non-uniform grid spacing, solution of the forward and transposed problems can be accomplished by the same basic code. However, there are some subtle issues, as discussed in the context of a 3D spherical EM solver by *Kelbert et al.* (2008). Finally, the solver should have a clean interface that allows interaction with higher level routines. At a minimum these must include initialization of the solver, and updating the PDE coefficients (which depend on the model parameter, and the frequency), as well as actually solving the forward or transposed equations for a particular set of sources and boundary conditions. In our implementation of the 3D finite difference solver module we also provide routines to set solver control parameters (convergence criteria, schedules for divergence corrections, etc.) and to retrieve solver diagnostics.

For the 2D MT case we adopted, with minimal modification, the existing 2D MT forward modeling code from the REBOCC inversion of *Siripunvaraporn and Egbert* (2000). This 2D code

was written in Fortran 77, which does not allow dynamic allocation of memory – array sizes for a particular problem had to be set at compile time. We wrapped this code with Fortran 95 routines to provide a cleaner, more clearly defined interface, and to allow dynamic memory allocation. Routines in separate TE and TM mode interface modules are called by other components of the inversion system to initialize and solve the 2D equations. These interface routines in turn make calls to the original REBOCC forward modeling subroutines, which do all of the actual numerical computations. The TE and TM mode interface modules contain, as private data, all coefficient and temporary work arrays (allocated on initialization) needed for solution of the corresponding equations. A simple LU factorization (with pivoting) is used for equation solution, so after forming and factoring the system of equations, solutions for a number of right hand sides can be computed rapidly.

EMfieldInterp. This module implements the interpolation functionals used to evaluate electric and magnetic fields at an arbitrary point within the model domain. These functionals are represented by sparse vectors in the primary field space $\mathcal{S}_P$ corresponding to columns of the matrix $\Lambda = \left[\begin{array}{cc} \Lambda_P & \mathbf{T}^T \Lambda_D \end{array} \right]$, as discussed in Paper I. Here, columns of Λ_P correspond to interpolation on the primary grid, and those of Λ_D to interpolation on the dual grid, while $\mathbf{T}$ is the transformation operator that maps from the primary to the dual EM field. Thus, for our 3D finite difference EM example the primary grid corresponds to cell edges (where electric field components are defined), the dual grid corresponds to cell faces (where the magnetic field components are defined) and $\mathbf{T}$ represents the operator $(i\omega)^{-1}\nabla\times$. Evaluation of electric and magnetic field components from the electric field solutions would then be represented as λ_k^P, and $\mathbf{T}^T\lambda_k^D$, respectively. In our implementation we have separate routines to compute interpolation functionals for primary and dual spaces, and a specialized routine to multiply the sparse vector λ_k^D by the adjoint of the transformation operator. Note that the full sparse differential operator $\mathbf{T}$ is not needed, and is not directly formed. Only the few components needed to form the sparse vector product $\mathbf{T}^T\lambda_k^D$ are actually evaluated.

If data functionals depend explicitly on the model parameter, module **EMfieldInterp** should also contain routines for computing the rows of $\Lambda_D^T \tilde{\mathbf{T}}_{\pi(\mathbf{m}_0),\mathbf{e}_0}$, if the transformation operator depends on $\mathbf{m}$, and $\tilde{\Lambda}_P^T$, $\tilde{\Lambda}_D^T$ if the interpolation operators do; see Table 1 and Paper I for details. All of these are sparse vectors in either $\mathcal{S}_P$ or $\mathcal{S}_D$, the same sorts of objects as the field component evaluation functionals.

The sparse vectors returned by **EMfieldInterp** are the basic building blocks for the nonlinear data functionals ψ_j (e.g., impedances), as well as the corresponding linearizations–the rows of $\mathbf{L}$ and $\mathbf{Q}$ which are discussed in the next section. The evaluation functionals of course depend critically on details of the numerical grid, while computation of something like an impedance or apparent resistivity from the interpolated electric and magnetic field components does not. Thus,

a different numerical discretization of the problem (e.g., finite elements with a non-structured grid) would require different interpolation routines. However, if these were again represented as sparse vectors in the appropriate EM field spaces, higher level data functional routines (e.g., for calculation of impedance elements), or construction of the corresponding linearized data functionals, would remain unchanged. Conversely, to add new data types (e.g., inter-station magnetic transfer functions) there is no need to revisit the interpolation aspects of the problem.

5 Level II: Interface Modules

Modules in the second level provide an interface between the generic top level modules, and the numerical implementation specific modules already discussed. Each instance of Level II modules will be specific to a particular application, making extensive use of the problem-specific *transmitter*, *data type* and *receiver* dictionaries (see Figure 1 and Table 2). Adapting our implementation of the 3D MT inversion to a simple controlled source problem required modifications to only Level II modules – i.e., new definitions for sources and data functionals had to be developed.

While specific implementations of Level II modules may vary from application to application, all interfaces and interactions with Level I modules must follow a set of rigid rules. All functions or subroutines called from the Level I modules must be instantiated in Level II, with names and interfaces (including data types for all arguments) as in Table 5. Of the data types referenced by Level I (Table 3), all *solution vector* data types (i.e., $\mathbf{e}$, $\mathbf{b}$, $\mathbf{l}$, $\vec{\mathbf{e}}$; see below) need to be declared in Level II. Although the abstract meanings of the data types are fixed, the actual structures declared can be quite different. For example, we have shown in Paper I that the fundamental EM solution object ($\mathbf{e}$) in 3D MT consists of a pair of vector fields on a 3D grid, while for the 2D MT problem this object is a single scalar solution on a 2D grid. These very different data structures must be stored in data types with identical names (`solnVector_t`) defined in two separate instances of the Level II module **SolnSpace**. Similarly, while all routines at this level have fixed names, and fixed abstract functionality, implementation details are application specific. Coding of Level II routines will in fact typically be straightforward, as the same routines from a previous application can be used as a template. In fact, the routines and data objects defined at Level II are mostly just wrappers of lower level objects and methods, that serve to hide application specific details from the generic top level routines. These extra layers of complication are the price of generality.

SolnSpace. This module defines the derived data types used to represent $\mathbf{e}$ and $\mathbf{b}$ in the discrete forward equation $\mathbf{S_m e} = \mathbf{b}$, and is essentially a wrapper of the lower level **EMfield** module. Even though both vectors $\mathbf{e}$ and $\mathbf{b}$ effectively belong to the same space, it is convenient to define a separate derived data type for the right hand side $\mathbf{b}$, to store additional information about

Routine Name	Level	Inputs	Outputs	Used For	Module
initSolver	II	iTx,$\mathbf{m}$	$\mathbf{e}_0$ ($\mathbf{e}$,$\mathbf{b}$)	ALL	ForwardSolver
exitSolver	II	$\mathbf{e}_0$ ($\mathbf{e}$,$\mathbf{b}$)		ALL	ForwardSolver
fwdSolver	II	iTx (FWDorADJ,$\mathbf{b}$)	$\mathbf{e}$	ALL	ForwardSolver
Pmult	II	$\mathbf{e}_0$,$\mathbf{m}_0$,$\mathbf{m}$	$\mathbf{b}$	Jmult	SolverSens
PmultT	II	$\mathbf{e}_0$,$\mathbf{m}_0$,$\mathbf{e}$	$\mathbf{m}$ ($\mathbf{m}'$)	JmultT, Jrows	SolverSens
Qmult	I	$\mathbf{e}_0$,$\mathbf{m}_0$,$\mathbf{d}_0$,$\mathbf{m}$	$\mathbf{d}$	Jmult	DataSens
QmultT	I	$\mathbf{e}_0$,$\mathbf{m}_0$,$\mathbf{d}$	$\mathbf{m}$ ($\mathbf{m}'$)	JmultT	DataSens
Qrows	II	$\mathbf{e}_0$,$\mathbf{m}_0$,iDt,iRx	$\mathbf{m}$	Qmult, QmultT, Jrows	DataFunc
Lmult	I	$\mathbf{e}_0$,$\mathbf{m}_0$,$\mathbf{d}_0$,$\mathbf{e}$	$\mathbf{d}$	Jmult	DataSens
LmultT	I	$\mathbf{e}_0$,$\mathbf{m}_0$,$\mathbf{d}$	$\mathbf{b}$	JmultT	DataSens
Lrows	II	$\mathbf{e}_0$,$\mathbf{m}_0$,iDt,iRx	$\mathbf{l}$	Lmult, LmultT, Jrows	DataFunc
dataResp	II	$\mathbf{e}$,$\mathbf{m}$,iDt,iRx	$\mathbf{Z}$	fwdPred	DataFunc

Table 5: Public Routines Used for Sensitivity Computations. Conventions used in calling arguments: iDt = data type; iRx = receiver; iTx = transmitter. Other symbols are as in Table 3, with subscript zeros denoting background model parameters and solution vectors used for linearization, and for the template data vector. The template data vector $\mathbf{d}_0$ gets overwritten with the computed vector $\mathbf{d}$ on output, where appropriate. $\mathbf{Z}$ denotes an array of real responses. Optional inputs or outputs are in parentheses; in particular, $\mathbf{m}'$ is the optional "imaginary" component of the model parameter output by LmultT and QmultT.

forcing and/or boundary conditions. Thus, we define the two basic data types `solnVector_t` and `rhsVector_t` (Table 3), which correspond to $\mathbf{e}$ and $\mathbf{b}$ in the mathematical development of Paper I. In addition, we define a data type `sparseVector_t`, as a sparse vector representation of the `solnVector_t` type (e.g., $\mathbf{l}_j$, a row of operator $\mathbf{L}$), and finally an array of `solnVector_t` that stores a series of $\mathbf{e}$ objects for multiple transmitters is defined as a separate data type, `solnVectorMTX_t`. All of the basic methods, including creation, destruction, I/O, copying, and basic linear algebra and dot-product operations (following the convention of Table 4) for $\mathbf{e}$, $\mathbf{b}$, $\mathbf{l}$ and $\vec{\mathbf{e}}$ objects are defined in module **SolnSpace**. As with the basic data type definitions, these methods can be built out of the corresponding methods for the lower level objects defined in module **EMfield**. These routines, which are used extensively in the Level I modules, again provide an interface that hides application specific details from the generic inversion system.

For the 3D MT case `solnVector_t`, `sparseVector_t` and `rhsVector_t` objects must contain complex vector field solutions for two independent source polarizations. This is essential, since the basic data type for 3D MT, the impedance tensor, requires solutions for both source polarizations to allow evaluation of the corresponding data functionals. Thus $\mathbf{e}$ in the abstract description of the inverse problem given in Paper I actually refers to the pair of solutions. Similarly $\mathbf{b}$, the abstract representation of the right hand side forcing, refers to the boundary data for a pair of independent source polarizations, sparse vectors ($\mathbf{l}_j$ used to implement data functionals) require components for the two source polarizations, and all of the expressions for sensitivities or gradients of the penalty functional must involve both source polarizations in a consistent fashion. For the 3D MT problem there is thus a clear distinction between objects that represent a single vector field (an EM solution for a single source polarization) and the full two component solution to the 3D MT forward problem.

For 2D MT, $\mathbf{e}$ represents just a single complex scalar field, with data structure essentially identical to that already defined in the lower level **EMfield** module discussed above. In addition to the scalar field, the `solnVector_t` object also contains a pointer to the grid and an index into the transmitter dictionary, where Level II routines obtain the information (e.g., TE or TM mode) needed to define the sources used to generate the solution. Keeping all of this information attached to the generic `solnVector_t` data object greatly simplifies subroutine interfaces in the Level I routines, and hides many bookkeeping details from the inversion algorithm.

Note that for a controlled source problem, data do not couple multiple sources, so the fundamental solution vector object for such a problem would contain a single **EMfield** object. For our simple 3D CSEM example, this would represent a complex 3D vector field, as for 3D MT.

ForwardSolver. The primary purpose of this module is to provide a uniform interface between the application specific forward modeling routines defined in **EMsolver**, and the generic Level I inversion routines. The key public routines in this module include initialization (`initSolver`), deallocation and clean up (`exitSolver`), and the forward solver routine (`fwdSolver`) with inter-

faces as defined in Table 5. Of course, internal functioning of these routines will be application dependent, as our two examples nicely illustrate. In the 3D MT case **e** (the output of `fwdSolver`) consists of solutions for two source polarizations. These require two separate calls to the lower level solver routines with different input boundary conditions or forcing. In the 2D case only a single solution is required, but now there are two possibilities: either a TE or a TM solver must be called, depending on the input *transmitter* index. Thus, routines for different applications may have significant differences internally (a loop over polarizations for the 3D MT implementation vs. a case statement for the 2D case), but **ForwardSolver** routine names, interfaces, and abstract external functionality (i.e., "*compute* **e** *for transmitter* iTx") must always be the same.

Initialization (`initSolver`) is kept explicitly separate from actual solution to enable more efficient computational strategies. For example, in our 2D MT implementation we use a direct matrix LU decomposition approach, so multiple forward solutions can be computed by backward substitution once factorization is complete. The initialization routine is thus coded to keep track of the previous solver call, and only re-initialize and factor the coefficient matrix if necessary. In the 3D case, repetition of operator setup steps can also be minimized by keeping track of attributes (frequency, conductivity parameter) used for the previous solution. The responsibility for these efficiencies lies with the initialization routine. Higher level routines that require solutions of the governing PDE always call the initialization routine first. The cleanup routine `exitSolver` is only called when it is desired to deallocate all operator arrays and return solver module data to the state it had before the first call to `initSolver`. Finally, `fwdSolver` implements the general solver, allowing arbitrary forcing and boundary conditions, and solutions for both the usual forward problem and its transpose, or adjoint. By default, it solves only the forward problem, and does not require explicit specification of the forcing, which is computed internally, with reference to the transmitter dictionary. The forcing may be provided as an optional argument, along with an optional switch to execute the adjoint rather than the default forward solver. These options are both used in the sensitivity computations.

Because the **ForwardSolver** module sets up sources for the forward problem it provides a simple and natural way to implement a secondary field formulation (e.g., *Alumbaugh et al.*, 1996). It is instructive to consider briefly how our 3D MT inversion was adapted for a simple CSEM inverse problem (with magnetic dipole sources and receivers), where such a modeling approach is appropriate. First, a separate module was developed to solve the forward problem for the 1D background conductivity. This module was then used by the **ForwardSolver** module, which was modified from the one developed for 3D MT. No changes were required for the Level III finite difference solver modules. In the CSEM version of module **ForwardSolver**, routine `initSolver` sets up the background and anomalous conductivity and does preliminary computations for the 1D solution, with all results stored as private saved variables. Then, on each call to `fwdSolver` the appropriate location and frequency is extracted from the *transmitter* dictionary, and the 1D background solution for this transmitter, and then the forcing for the secondary field solution, are

computed. Level III routine (`EMsolver`) is then called to compute the 3D secondary field, which is added to the 1D background solution. The total field solution is returned by `fwdSolver`. Thus, the fact that the solver has been modified is completely hidden from higher level calling routines.

DataFunc. This module defines and implements the data functionals for a specific application. There are three key public procedures, with names and interfaces given in Table 5. The first, `dataResp`, evaluates the non-linear data functional $\psi_j(\mathbf{e}, \mathbf{m})$, e.g., in the 2D and 3D MT inverse problems computing the complex impedance elements from the solution vector $\mathbf{e}$ (and possibly the model parameter $\mathbf{m}$). Thus, in the notation of Paper I and Table 1, `dataResp` evaluates γ_j, using local field values computed from dot products of $\mathbf{e}$ with appropriate evaluation functionals (λ_k^P and $\mathbf{T}^T\lambda_k^D$) returned by calls to Level III `EMfieldInterp` routines. `Lrows` and `Qrows` compute the linearized data functionals, i.e., rows of $\mathbf{L}$ and $\mathbf{Q}$ ($\mathbf{l}_j = \partial\psi_j/\partial\mathbf{e}$ and $\mathbf{q}_j = \partial\psi_j/\partial\mathbf{m}$, see Table 1) for all components corresponding to a specific *data type*, for some *transmitter* and *receiver*. More specifically, `Lrows` returns sparse *solution vector* objects, which are formed from linear combinations of the local field evaluation functionals as

$$\sum_{k=1}^{K_P} a_{jk}^P \lambda_k^P + \sum_{k=1}^{K_D} a_{jk}^D \left(\mathbf{T}^T \lambda_k^D\right). \tag{9}$$

The coefficients in the linear combinations (a_{jk}^P and a_{jk}^D) are the partial derivatives of the data functional with respect to the local primary and dual field components used for its computation (see Paper I for details and examples). As discussed in the previous section, these coefficients are independent of details in the numerical implementation of the forward problem. Note that for the 3D MT problem the basic solution vector object ($\mathbf{e}$) consists of two EM field objects. Similarly the sparse solution vector object used to represent a single row of $\mathbf{L}$ consists of two sparse EM field objects of the sort given in (9). Also note that all components of multivariate data (e.g., all impedance tensor elements) for a single *transmitter/data type/receiver* are handled together with a single call to `Lrows`, with an array of sparse solution vector objects returned on each call.

The function of `Qrows` is similar, but this routine returns an array of *model parameters*, the rows of $\mathbf{Q}$ corresponding to one *transmitter/data type/receiver*. A typical element takes the form

$$\Pi_{\mathbf{m}_0}^T \left[\sum_{k=1}^{K_D} a_{jk}^D \tilde{\mathbf{T}}_{\pi(\mathbf{m}_0),\mathbf{e}_0}^T \lambda_k^D\right] \tag{10}$$

where the coefficients and evaluation functionals are as in (9), and $\tilde{\mathbf{T}}_{\pi(\mathbf{m}_0),\mathbf{e}_0}$, $\Pi_{\mathbf{m}_0}$ are operators defined in Paper I (see also Table 1). As we discuss there, (10) only accounts for the dependence of the transformation operator ($\mathbf{T}$) on the model parameter; if the interpolation functionals themselves depend on $\mathbf{m}$ there will be additional, but similar, terms in the appropriate expression for

the rows of $\mathbf{Q}$. The key point for our discussion here is that the component of (10) in square brackets is a sparse vector in $\mathcal{S}_P$ or $\mathcal{S}_D$. As with `Lrows`, this sparse vector is formed as a linear combination of vectors returned by calls to `EMfieldInterp`. The result is then multiplied by $\Pi^T_{\mathbf{m}_0}$ which maps to the model parameter space. Again, for a problem such as 3D MT with multiple coupled source polarizations, there will be terms such as (10) associated with each polarization; the appropriate output sums over all polarizations. Management of such problem dependent issues is handled by `Qrows` which is itself specific to the particular EM problem. Finally, note that model parameters are real vectors, while in (10) the real matrix $\Pi^T_{\mathbf{m}_0}$ multiplies a complex vector. Thus the left hand side of (10) should really be viewed as a pair of model parameters, corresponding to the real and imaginary parts of the product. As discussed below in the context of module **SensSolve**, both components are used when the full Jacobian is calculated.

In general all of the dictionaries are required to control data functional calculations; indices for *data type* and *receiver* dictionaries are explicit input arguments to these routines. The index into the transmitter dictionary is obtained from the input solution vector object $\mathbf{e}$, which always has a unique *transmitter* associated with it. For example, in the 2D MT case we encode mode (TE or TM) in the *transmitter* dictionary, along with frequency. Case statements within routines of this module are used to select calls to appropriate evaluation functionals, which are different for the two modes. Data type distinctions (e.g., apparent resistivity and phase vs. impedance) would be encoded in the *data type* and would require additional case statements. Adding new data types can be accommodated with modifications to this module, and the dictionaries: the new data types must be encoded in the type dictionary, and code must be added to `dataResp`, `Lrows` and `Qrows` for the new cases. The *receiver* index is used to retrieve meta-data, such as site location, needed to generate evaluation functionals for specific observations. The *receiver* dictionaries may also need to be modified when new data types are added – e.g., to allow for inter-station TFs locations for both local and reference sites would have to be included in the receiver dictionary.

To adapt the 3D MT inversion for our simple CSEM problem, module **DataFunc** also had to be modified. Now observations are just point measurements of magnetic field components, so the data functionals are already linear. For an observation at $\mathbf{x}_j$ these now take the form $\psi_j(\mathbf{e}) = \lambda_j^T \mathbf{e}$ where λ_j, the interpolation functional for evaluation of the appropriate component of the magnetic field at $\mathbf{x}_j$, can be computed by a single call to Level III routine `EMfieldInterp`. `Lrows` thus merely converts this sparse vector to the appropriate type (`sparseVector_t`), and `dataResp` computes the dot product of λ_j with $\mathbf{e}$. Since in our implementation there is no direct dependence of the interpolation functionals on the model parameter, `Qrows` is a dummy routine (for both the 3D CSEM and MT problems). Thus for the CSEM problem **DataFunc** routines are extremely simple, essentially just wrappers for basic evaluation functionals that are actually implemented in Level III modules. As we discuss further below, it would also be possible to mix MT and CSEM, with data of each type referencing a different transmitter. In this case, the simple data functionals for CSEM could be added to those already developed for MT, with the appropriate

case determined from the *transmitter* index of the data vector input to `dataResp` or `Lrows`.

SolverSens. This module implements multiplication by matrices $\mathbf{P}$ and $\mathbf{P}^T$. Viewed as an operator, the input of $\mathbf{P}$ is a *model parameter* object and the output a *RHS vector* object. To multiply a model parameter object ($\delta\mathbf{m}$) by $\mathbf{P}$ in `Pmult` (see (7)), we first call a **ModelMap** routine which multiplies it by $\Pi_{\mathbf{m}_0}$ to produce an EM field object $\Pi_{\mathbf{m}_0}\delta\mathbf{m} \in \mathcal{S}_P$. The problem specific operators ($\mathbf{U}$ and $diag(\mathbf{V}\mathbf{e}_0)$) are then applied, with the results returned as a `rhsVector_t` object, of the proper type to provide forcing for the solver in a computation such as $\mathbf{J}\mathbf{m} = \mathbf{L}\mathbf{S}^{-1}\mathbf{P}\mathbf{m}$. For example, in the 3D MT case, the result of the first step is the conductivity perturbation averaged onto cell edges. The operators denoted by $\mathbf{U}$ and $\mathbf{V}$ are in this case trivial, together amounting to point-wise multiplication by $i\omega\mu\mathbf{e}_{0,k}$, where $\mathbf{e}_{0,k}, k = 1, 2$ are background EM solutions for the two polarizations. The result is a pair of EM field objects defined on the primary grid edges, which together constitute the output `rhsVector_t` object returned by `Pmult`.

`PmultT` multiplies a `solnVector_t` object by $\mathbf{P}^T$, reversing these steps. Again taking the 3D MT case as a specific example, the input $\mathbf{e}$, which encapsulates two separate lower level EM field objects $(\mathbf{e}_1, \mathbf{e}_2)$, is used in the first step to form the EM field object $i\omega\mu(\mathbf{diag}[\mathbf{e}_{0,1}]\mathbf{e}_1 + \mathbf{diag}[\mathbf{e}_{0,2}]\mathbf{e}_2)$. The second step is then to apply the adjoint model parameter mapping $\Pi^T_{\mathbf{m}_0}$ to this. The real and imaginary parts of the result are objects of type `modelParam_t`. When operations such as $\mathbf{J}^T$ are applied in the context of, for example, the NLCG inversion scheme, multiplication by $\mathbf{P}^T$ is the final step in the computation of the penalty functional gradient, and only the real output is required. When the full sensitivity matrix for complex data (e.g., impedance) is to be calculated, the real and imaginary output components, respectively, provide sensitivities for real and imaginary parts of the data, and both should be saved.

In the 2D MT case, multiplication by $\mathbf{P}$ is somewhat different for the TE and TM polarizations, so `Pmult` and `PmultT` must have code for both cases. The appropriate case is determined internally from the mode of the background `solnVector_t` object $\mathbf{e}_0$, an attribute of these objects for this problem. The TE case is essentially like the 3D MT case discussed above, except the `solnVector_t` object involves only a single polarization. The TM case is somewhat more complicated. As the 2D example in Paper I shows, discrete 2D gradient and divergence operators are required. In our implementation we include simple routines for these operations within this module. As the routines are only referenced internally, they can be private to this module.

6 Level I: Generic Sensitivity and Inversion Modules

Level I modules implement actual inversion search algorithms, and manage computations with sensitivity matrices, such as the matrix-vector products $\mathbf{J}^T\mathbf{d}$ and $\mathbf{J}\mathbf{m}$. A goal of the modular

Routine Name	Computes	Inputs	Outputs
`fwdPred`	$\mathbf{e} = \mathbf{S}_{\mathbf{m}}^{-1}\mathbf{b}; \mathbf{d} = \psi(\mathbf{e}, \mathbf{m})$	$\mathbf{m}$, $\mathbf{d}_0$	$\mathbf{d}$ ($\mathbf{e}$)
`calcJ`	$\mathbf{J}^T = \mathbf{P}^T(\mathbf{S}_{\mathbf{m}_0}^T)^{-1}\mathbf{L}^T + \mathbf{Q}^T$	$\mathbf{m}_0$, $\mathbf{d}_0$	$\mathbf{J}$
`Jmult`	$\delta\mathbf{d} = \mathbf{J}\delta\mathbf{m}$	$\delta\mathbf{m}$, $\mathbf{m}_0$, $\mathbf{d}_0$ ($\mathbf{e}$)	$\delta\mathbf{d}$
`JmultT`	$\delta\mathbf{m} = \mathbf{J}^T\delta\mathbf{d}$	$\mathbf{m}_0$, $\delta\mathbf{d}$ ($\mathbf{e}$)	$\delta\mathbf{m}$

Table 6: Level I sensitivity computations. General symbol conventions for input and output arguments are as in Table 3, but here $\mathbf{d}_0$ denotes a **template** data vector that provides transmitter/receiver/data type, but no actual data values. Optional inputs or outputs are in parentheses. Single-transmitter versions of `fwdPred`, `Jmult` & `JmultT` are also provided to simplify parallelization.

scheme it to make these top-level modules generic in the most literal sense, so that they may be directly applied to any EM (or even non-EM) problem, as long as the interface modules and the basic data types are consistent with Tables 5 and 3. However, as we have discussed in Paper I, data vectors for some EM problems have special structure (in particular, with regard to dependencies on *transmitter* and *receiver*) that can be exploited to develop more efficient schemes for computations with the Jacobian. Our strategy has been to develop generic versions of all routines at this level, and to then modify key routines involving Jacobian calculations to treat special cases more efficiently. In our discussion here we follow a similar strategy, focusing on the generic case, and then briefly describing modifications needed for particular cases. Parallelization of the inversion over transmitters is handled in a similar fashion, by modifying Level I routines which manipulate or calculate the Jacobian, to distribute computational tasks over a number of computational nodes. In both cases the actual inversion routines are essentially unmodified, in particular, the same inversion routines are used for parallel and serial versions of the code.

DataSens. This module implements multiplication by $\mathbf{L}$, $\mathbf{L}^T$, $\mathbf{Q}$ and $\mathbf{Q}^T$ through the four public routines listed in Table 5. The first, `Lmult`, is the linearized counterpart of the non-linear response $\psi(\mathbf{e}, \mathbf{m})$, $\partial\psi/\partial\mathbf{e}$, returning the perturbation in the data for small perturbation in $\mathbf{m}$ and $\mathbf{e}$, i.e., $\mathbf{L}\delta\mathbf{e} = \delta\mathbf{d}$. The second, `LmultT`, implements the transpose of this operator, returning $\mathbf{L}^T\delta\mathbf{d} = \delta\mathbf{e}$. The third routine, `Qmult`, computes the component of the data perturbation due to direct dependence of the data functional ψ on the model parameter $\mathbf{m}$, which we have represented as $\mathbf{Q}\delta\mathbf{m} = \delta\mathbf{d}$. Finally, `QmultT` provides the transpose of this operator, $\mathbf{Q}^T\delta\mathbf{d} = \delta\mathbf{m}$.

In all cases, computations are done for a single *transmitter* in each call to a **DataSens** routine. For each *data type* and *receiver*, appropriate routines in module **DataFunc** are called to set up representations of the linearized data functionals. For `Lmult` and `LmultT`, these are the sparse *solution vector* objects returned by `Lrows`. For `Lmult`, these objects are used to form the dot products with $\mathbf{e}$ and then to assemble the `dataVector_t` object $\mathbf{d} = \mathbf{L}\mathbf{e}$, while `LmultT` multiplies

these sparse rows of $\mathbf{L}$ by the respective data components, and sums to form the forcing for the adjoint system $\mathbf{L}^T\mathbf{d}$, output as an object of type `rhsVector_t`. For `Qmult` and `QmultT`, the linearized functionals are *model parameter* objects, the outputs of `Qrows`. In the first routine, dot products are formed with the input model parameter and assembled into the `dataVector_t` object $\mathbf{d} = \mathbf{Q}\mathbf{m}$. For the transpose, the model parameters returned by `Qrows` are multiplied by the respective data components and summed to obtain $\mathbf{m} = \mathbf{Q}^T\mathbf{d}$.

For the forward calculations (i.e., `Lmult`) the calls to linearized and nonlinear data functionals (implemented in the application specific `DataFunc` module) result in complex values. Routines in this module convert these for storage as pairs of reals in the `dataBlock_t` basic objects. In the adjoint case (`LmultT`), real and imaginary parts, which are stored separately in the input $\mathbf{d}$, are combined into complex coefficients to construct the proper complex forcing $\mathbf{L}^T\mathbf{d}$ for the adjoint equation.

SensComp. This module puts all of the pieces ($\mathbf{L}$, $\mathbf{S}_\mathbf{m}^{-1}$, $\mathbf{P}$, $\mathbf{Q}$) together to implement full sensitivity computations. In contrast to all previously discussed modules, `SensComp` deals with *multiple transmitter* objects. The key public routines are summarized in Table 6. `Jmult` multiplies $\mathbf{m}$ by the Jacobian $\mathbf{J}$, to produce a `dataVectorMTX_t` object $\mathbf{d}$. `JmultT` implements the transpose of this operation, multiplying $\mathbf{d}$ by $\mathbf{J}^T$ to produce a `modelParam_t` object $\mathbf{m}$. There is also a routine (`fwdPred`) that implements the full forward problem, looping over transmitters to solve all component problems and returning the full vector of predicted data for a given model parameter ($\mathbf{d} = \mathbf{f}(\mathbf{m})$). Note that this routine can optionally return $\mathbf{e}$, the array of EM solutions objects computed for all unique transmitters. These can then be passed, as optional inputs, to routines `Jmult` and `JmultT`. This makes it possible to use the forward solutions (calculated, for example, for evaluation of data misfit) for subsequent sensitivity calculations, e.g., those that are needed to evaluate the gradient of the penalty functional. Single transmitter versions of these three routines (`fwdPred_TX`, `Jmult_TX` and `JmultT_TX`) also exist; these make parallelization over frequencies straightforward (see Section 6.2). In fact, the multiple-transmitter routines loop over the list of transmitters, and call the single transmitter versions to do the actual calculations.

Finally, `calcJ` provides a routine for computation of the full sensitivity matrix $\mathbf{J}$, for all *transmitters*, *data types* and *receivers*. Rows of $\mathbf{J}$ (one for each real observation) are model parameter (`modelParam_t`) objects. The full matrix is stored using a structure which mirrors the data vector (see Table 3). Thus, the full Jacobian is an array of single transmitter sensitivity matrices, each of which is turn an array of model parameters giving the sensitivities for all *receivers* for one *transmitter* and *data type*. The computations are done using the adjoint form (Paper I), taking advantage of redundancy associated with complex data where possible. In this case, parallelization may easily make use of a much larger number of processors, by computing a single sensitivity value for each data point. The actual sensitivity calculation is implemented through routine `Jrows`, which computes the sensitivities for all data components corresponding to a single

transmitter, *data type* and *receiver*. Routine `calcJ` then loops over transmitters, receivers and data types, calling `Jrows` to compute the full sensitivity matrix.

For both `fwdPred` and `Jmult`, vector $\mathbf{d}$ must be input to provide a template for the output data vector. For `calcJ`, this template is only used to provide the meta-data needed to define the data functionals, through the indices into the *transmitter*, *data type* and *receiver* dictionaries.

6.1 Inversion Algorithms

The basic sensitivity computations described above, and summarized by Table 6, provide the necessary tools for development of a wide range of gradient based inversion algorithms. Building on this framework, we have so far implemented and tested several distinct algorithms, including non-linear conjugate gradients (NLCG, e.g., *Press et al.*, 1992), and the data-space conjugate gradient scheme (DCG) of *Siripunvaraporn and Egbert* (2007), as well as the hybrid CG/Occam, and multi-frequency hybrid schemes described in *Egbert* (2010).

For NLCG, we adopted the standard Polak-Ribière scheme (see Figure 2 for pseudo-code). For the line search, which is based on *Press et al.* (1992) and *Nocedal and Wright* (2006, Chapter 3), we first interpolate using a quadratic approximation; if the solution does not satisfy the sufficient decrease (Armijo) condition (ignoring the curvature condition), we backtrack using cubic interpolation. This strategy only requires one gradient evaluation and is very efficient when these computations are expensive. The initial step size is set outside of the line search, and adjusted automatically based on the first gradient computation; it can also be specified by the user. We have also implemented an automatic criterion to decrease the damping parameter (ν, see Paper I) when convergence of the inversion stalls. User control of the inversion includes specifying the initial damping parameter, the sufficient decrease condition, and the stopping criterion.

The DCG scheme of *Siripunvaraporn and Egbert* (2007) is an iterative Gauss-Newton algorithm. Briefly, the penalty functional is linearized about a trial value of the model parameter $\mathbf{m}_n$, and the minimum of the linearized functional is sought. This leads to a system of normal equations involving cross-products of $\mathbf{J}$. In the data-space variant we consider here, these equations are

$$(\mathbf{J}\mathbf{C}_\mathbf{m}\mathbf{J}^T + \nu\mathbf{I})\mathbf{b}_n = \hat{\mathbf{d}}_n \tag{11}$$

$$\mathbf{m}_{n+1} = \mathbf{C}_\mathbf{m}\mathbf{J}^T\mathbf{b}_n. \tag{12}$$

In the DCG algorithm Eq. (11) is solved for $\mathbf{b}_n$ using conjugate gradients, and the model update is computed as in (12). The whole procedure is iterated over n to achieve the desired level of misfit. Pseudo-code for this scheme is provided in Figure 3. Choice of the regularization parameter, and

$\mathbf{m}_{prior}$ = prior model
ν = initial damping parameter
a = initial line search step α scaled by model norm
$n = 0$
$\tilde{\mathbf{m}}_0 = 0$
$\mathbf{m}_0 = \mathbf{C}_\mathbf{m}^{1/2}\tilde{\mathbf{m}}_0 + \mathbf{m}_{prior}$ = starting model
Evaluate $\mathcal{P}_0 = (\mathbf{d} - f(\mathbf{m}_0))^T\mathbf{C}_\mathbf{d}^{-1}(\mathbf{d} - f(\mathbf{m}_0)) + \nu\tilde{\mathbf{m}}_0^T\tilde{\mathbf{m}}_0$
Evaluate $\mathcal{P}'(\tilde{\mathbf{m}})|_{\tilde{\mathbf{m}}_0} = -2\mathbf{C}_\mathbf{m}^{1/2}\mathbf{J}^T\mathbf{C}_\mathbf{d}^{-1}(\mathbf{d} - f(\mathbf{m}_0)) + 2\nu\tilde{\mathbf{m}}_0$
$\mathbf{g}_0 = -\mathcal{P}'(\tilde{\mathbf{m}})|_{\tilde{\mathbf{m}}_0}$
$\mathbf{h}_0 = \mathbf{g}_0$
Set $\alpha_0 = a/\mathbf{g}_0^T\mathbf{g}_0$
While $(\mathbf{d} - f(\mathbf{m}_n))^T\mathbf{C}_\mathbf{d}^{-1}(\mathbf{d} - f(\mathbf{m}_n)) > T$ do
    $n = n + 1$
    Line search: choose $\hat{\alpha}$ to minimize $\mathcal{P}(\mathbf{C}_\mathbf{m}^{1/2}(\tilde{\mathbf{m}}_{n-1} + \alpha\mathbf{h}_{n-1}) + \mathbf{m}_{prior})$
    $\tilde{\mathbf{m}}_n = \tilde{\mathbf{m}}_{n-1} + \hat{\alpha}\mathbf{h}_{n-1}$
    $\mathcal{P}_n = (\mathbf{d} - f(\mathbf{m}_n))^T\mathbf{C}_\mathbf{d}^{-1}(\mathbf{d} - f(\mathbf{m}_n)) + \nu\tilde{\mathbf{m}}_n^T\tilde{\mathbf{m}}_n$
    $\mathbf{g}_n = -\mathcal{P}'(\tilde{\mathbf{m}})|_{\tilde{\mathbf{m}}_n}$
    $\alpha = 2(\mathcal{P}_n - \mathcal{P}_{n-1})/\mathbf{g}_{n-1}^T\mathbf{h}_{n-1}$
    Adjust α for super-linear convergence: $\alpha = \min(1.00, 1.01 * \alpha)$
    If α too small
        $\nu = 0.1 * \nu$
        $\alpha = a/\mathbf{g}_n^T\mathbf{g}_n$
        Restart: $\mathbf{h}_n = \mathbf{g}_n$
        Cycle do loop
    End if
    $\beta = \mathbf{g}_n^T(\mathbf{g}_n - \mathbf{g}_{n-1})/\mathbf{g}_{n-1}^T\mathbf{g}_{n-1}$
    If $(\mathbf{g}_n^T\mathbf{g}_n + \beta\mathbf{g}_n^T\mathbf{h}_{n-1} > 0)$
        $\mathbf{h}_n = \mathbf{g}_n + \beta\mathbf{h}_{n-1}$
    Else
        Restart to restore orthogonality: $\mathbf{h}_n = \mathbf{g}_n$
    End if
End do

Figure 2: Pseudo-code for non-linear conjugate gradient (NLCG) algorithm. Notation: $f(\mathbf{m})$ represents responses obtained from forward modeling; $\mathcal{P}_n$ and $\mathcal{P}'_n$ are values of the penalty functional and its derivative at nth iteration; $\mathbf{m}_n$, $\mathbf{h}_n$, $\mathbf{g}_n$ are vectors in the model space; α and β represent real scalars; other symbols are as in the text.

stopping criteria for the conjugate gradient iterations are discussed in *Siripunvaraporn and Egbert* (2007). For sensitivity computations we use multiplication routines `Jmult` and `JmultT`, described in Table 6.

6.2 Parallel Implementation

Parallelization has been implemented by making minor additions and modifications to a few modules, and adding one new Level I module. As with the inversion search algorithms, the parallelization does not depend on the details of a specific EM problem or the implementation of the numerical forward solver. The parallelization is very coarse-grained, distributing computations for independent forward problems over processors, using the Massage Passing Interface (MPI) communication library. As the name suggests, MPI is based on exchanging messages between processors. Messages can consist of any standard data type in the programming language used (i.e., in Fortran 95: Integer, Real, etc) dimensioned as scalars or arrays. Passing derived data types in messages requires some care.

To develop the parallel version of ModEM an MPI module has been developed (**MPI_Main**). This module is considered to be in Level I, in that it does not reference details of the underlying EM problem. Because the MPI library cannot directly pass derived data types, additional routines have been implemented in the **ModelSpace**, **SolnSpace** and **DataSpace** modules. These routines create MPI structured data types from the Fortran derived data types defined in these modules, allowing the communication routines in **MPI_Main** to also treat solution and model objects abstractly while sending and receiving. Thus, it is not necessary for the **MPI_Main** routines to reference internal details of the derived data types. The MPI specific source code is kept in separate files, which are included in the appropriate modules when compiled. In addition to the new **MPI_Main** module and the derived data type conversion routines, very minor modifications are required to inversion (e.g., NLCG, DCG) routines, as discussed below. The relationship of the **MPI_Main** module to the rest of ModEM is illustrated in Figure 1.

To minimize interactions with the rest of the system, we applied the following general scheme for parallelization. After processor initialization, one processor is assigned as the master, and the rest as workers. The workers enter a queue inside **MPI_Main** and await messages from the master. The master alone executes the actual inversion algorithm steps. To execute any statement in parallel, the master enters **MPI_Main** and distributes messages to all workers indicating what tasks to perform (see the pseudo-code in Figure 4a). Thus, all MPI communication subroutines are hosted inside **MPI_Main**. This module consists of separate sets of routines for the master, and for workers. For the master there are separate routines corresponding to each parallelized version of the routines in **SensComp**. These routines have the same name as the equivalent serial versions, with the addition of the prefix *"Master_Job"*. For example, the parallel version of

$\mathbf{m}_{prior}$ = prior model
$\mathbf{m}_0$ = starting model
Outer loop: For $n = 0, 1, 2, \ldots$
    $\hat{\mathbf{d}}_n = \mathbf{d} - f(\mathbf{m}_n) + \mathbf{J}[\mathbf{m}_n - \mathbf{m}_{prior}]$
    Solve: $[\mathbf{C}_\mathbf{d}^{-1/2}\mathbf{J}\mathbf{C}_\mathbf{m}\mathbf{J}^T\mathbf{C}_\mathbf{d}^{-1/2} + \nu\mathbf{I}]\mathbf{b}_n = \hat{\mathbf{d}}_n$ using CG
    $\mathbf{x}_0 = 0$
    $\mathbf{r}_0 = \mathbf{p}_0 = \hat{\mathbf{d}}_n$
    Inner loop (CG): For $k = 1, 2, \ldots$
        $\mathbf{y}_k = \mathbf{C}_\mathbf{d}^{-1/2}\mathbf{J}\mathbf{C}_\mathbf{m}\mathbf{J}^T\mathbf{C}_\mathbf{d}^{-1/2}\mathbf{x}_k + \nu\mathbf{x}_k$
        $\alpha = \mathbf{r}_k^T\mathbf{r}_k / \mathbf{p}_k^T\mathbf{y}_k$
        $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k$
        $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k\mathbf{y}_k$
        If $||\mathbf{r}_{k+1}|| < r_{tol}$
            $\mathbf{b}_n = \mathbf{x}_{k+1}$
            Exit
        Else
            $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} - \beta_k\mathbf{p}_k$
        End if
    End inner loop
    $\mathbf{m}_{n+1} = \mathbf{m}_{prior} + \mathbf{C}_\mathbf{m}\mathbf{J}^T\mathbf{C}_\mathbf{d}^{-1/2}\mathbf{b}_n$
    If $(\mathbf{d} - f(\mathbf{m}_{n+1}))^T\mathbf{C}_\mathbf{d}^{-1}(\mathbf{d} - f(\mathbf{m}_{n+1})) < T$ exit
End outer loop

Figure 3: Pseudo-code for data space conjugate gradient (DCG) algorithm, with normal equations solved in the data space. Notation: $f(\mathbf{m})$ represents responses obtained from forward modeling; $\hat{\mathbf{d}}_n$, $\mathbf{b}_n$, $\mathbf{x}_k$, $\mathbf{y}_k$, $\mathbf{p}_k$ and $\mathbf{r}_k$ are all vectors in the data space; α_k and β_k represent real scalars; other symbols are as in the text.

`fwdPred` in **SensComp** is `Master_Job_fwdPred` in **MPI_Main**. Consequently, the parallelized version of any inversion scheme calls `Master_Job_fwdPred` instead of `fwdPred`. To avoid having two copies of the inversion module (one for serial and one for parallel computations) we make use of compiler directives, so that lines of code appropriate for either the serial or parallel case are compiled, depending on the setting of a flag in the makefile (see the pseudo code in Figure 4b,c).

The worker part of **MPI_Main** consists of a single subroutine called `Worker_Job`. This subroutine contains a main loop that is executed while awaiting messages from the master (Figure 4a), so that all processors assigned as workers remain inside **MPI_Main** until they receive a stop message. Once a message is received from the master, the worker executes the requested job, typically for a single transmitter, by calling the appropriate subroutines inside **SensComp**. For example, `fwdPred_TX` is called to solve the forward problem and compute the model responses for a single transmitter (see Table 6). Thus, connections between the inversion module (e.g., NLCG, effectively running only on the master node) and the routines in **SensComp** that do the actual modeling computations (on worker nodes) are always through the `Worker_Job` subroutine in **MPI_Main**.

Our parallel scheme is designed to minimize communication between processors. Consider for example the matrix-vector multiplication implemented in `JmultT`. As noted in Table 6, this routine takes the background EM solution (which typically will already have been computed to evaluate model misfit) as an optional input. We keep the solution for a particular transmitter on the processor used to compute it, and assign the same transmitter to this processor for the task of computing the product $\mathbf{J}^T\mathbf{d}$ (for a single transmitter). Thus it is not necessary to gather EM solutions for all transmitters on the master processor and then re-distribute these to the workers. The master sends only a transmitter index and the data vector ($\mathbf{d}$) to each worker to perform the multiplication in parallel. Of course the resulting model parameters must still be returned to the master to be summed to complete computation of $\mathbf{J}^T\mathbf{d}$ for the full data vector.

We have also parallelized the computation of the full Jacobian over transmitter, data type and receiver. Here, we decompose and compute the matrix block-wise, where each block contains the sensitivities for one particular *transmitter*, *data type* and *receiver*. To minimize the storage required for inversion schemes that requires an explicit use of this matrix, each block is stored on the processor where it is computed. However, computations such as the data and model space cross products $\mathbf{J}\mathbf{J}^T$ or $\mathbf{J}^T\mathbf{J}$ require heavy communications between processors. Several strategies and schemes have been tested to minimize the communication and the idle time on each processor. For a comprehensive discussion of these, as well as more details on the block-wise computation of the full Jacobian, see *Meqbel* (2009).

a)

MPI_Main

Master:

```
                                    :
                                    :
Subroutine Master_Job_fwdPred (m0,d,e0)
     1- Task = 'Distribute Model Parameters'
        -Send a copy of model parameters to all workers.
     2- Task = 'Solve Forward Problem '
        - Send the index of one transmitter to each worker
        - Receive the predicted data
End subroutine
                                    :
```

Worker:

```
Do
                Receive task from Master

   if  task = 'Distribute Model Parameters'  then
      Receive a copy of model parameters from Master

   if  task = 'Solve Forward Problem '  then
      Receive the index of one transmitter from Master
        Call fwdPred_TX (m0,d,e0)
      Send the predicted data for one transmitter to Master

   if  task = 'Stop' then
      EXIT loop

Loop
```

b)

Serial version: i.e. inside NLCG

```
        :
        :
        :
        :
        :
        :
Call  fwdPred   (m0,d,eAll)
        :
        :
        :
```

c)

Serial/Parallel version: i.e. inside NLCG

```
        :
        :
        :
#if def=MPI then
  Call  Master_Job_fwdPred (m0,d,eAll)
#else
  Call  fwdPred   (m0,d,eAll)
#end if
        :
        :
        :
```

6.3 Problem Dependent Efficiencies

As we have shown in Paper I, one row of the Jacobian can be effectively "factored" into two components: one that depends on the receiver (site location) and one that depends on the source geometry. This special structure might in some cases be exploited to increase efficiency of manipulations with the Jacobian. A simple example is provided by our simple vertical magnetic dipole CSEM example, with N_{RX} receivers each collecting data for N_{TX} transmitters, which for simplicity we take to be operating at a fixed frequency. Then, as shown in Paper I, and previously in *Newman and Alumbaugh* (1997), the full Jacobian for all $2N_{TX}N_{RX}$ real data can be computed by solving the forward problem for each source ($\mathbf{S}\mathbf{e}_j = \mathbf{s}_j, j = 1, N_{TX}$) and the adjoint problem for each receiver ($\mathbf{S}^T\mathbf{b}_k = \mathbf{l}_k, k = 1, N_{RX}$) – a total of $N_{TX} + N_{RX}$ solver calls. The sensitivity for observation jk can then be computed as $\mathbf{P}(\mathbf{e}_j)^T\mathbf{b}_k$, where the dependence on the transmitter enters through $\mathbf{P}$. If we use the "generic" approach described above to implement the Jacobian products, each call to `Jmult` ($\mathbf{Jm}$) and to `JmultT` ($\mathbf{J}^T\mathbf{d}$) requires the same number of solver calls. For a search algorithm such as DCG, where repeated calls to `Jmult` and `JmultT` are required for a single inner-loop iteration (*Siripunvaraporn and Egbert* (2007); Figure 3), this generic approach would be highly inefficient.

A more efficient approach, similar to the scheme described in *Newman and Alumbaugh* (1997), would be to pre-compute and store the (receiver dependent) adjoint solutions $\mathbf{b}_k$, $k = 1, N_{RX}$, along with the (transmitter dependent) forward solutions $\mathbf{e}_j$, $j = 1, N_{TX}$, and then use these to compute the required Jacobian products without any further solver calls, i.e.,

$$(\mathbf{Jm})_{jk} = \mathbf{b}_k^T\mathbf{P}(\mathbf{e}_j)\mathbf{m} \tag{13}$$

$$\mathbf{J}^T\mathbf{d} = \sum_{jk} d_{jk}\mathbf{P}(\mathbf{e}_j)^T\mathbf{b}_k. \tag{14}$$

We implement this approach in ModEM by creating a new module, which we only sketch here. The module includes variants of the single transmitter Jacobian routines (say `JmultFac_TX` and `JmultTFac_TX`) which implement the simple calculations of (13-14). On the first call to these routines (or after any change in the model parameter) an initialization routine is called to compute the adjoint solutions $\mathbf{b}_k$, $k = 1, N_{RX}$; these are saved in the module for use in subsequent calls. The top level Jacobian routines in **SensComp**, `Jmult` and `JmultT` are then modified to call `JmultFac_TX` and `JmultTFac_TX` for any transmitter for which this computational approach is appropriate. Note that the actual inversion routine (e.g., DCG) does not need to be modified–indeed, the way in which the Jacobian calculations are implemented remains completely hidden from this routine. Note also that this approach allows mixing of controlled source and MT data–based on information provided in the transmitter dictionary `Jmult` and `JmultT` can use the appropriate implementation of the Jacobian product calculation to maximize efficiency.

Other efficiencies, such as those applied to computation of the full Jacobian for multi-polarization transfer functions (including 3D MT; see Paper I), can be implemented by developing specialized variants on the appropriate **SensComp** routines.

7 Discussion and Conclusions

ModEM has been built around a very specific structure for data vector objects, with very general implementations allowed for representation of solution vectors and model parameters. Inversion routines are generic, and implemented in terms of ADTs, much as our discussion of EM inversion in Paper I was abstract and non-specific. The rest of the system will depend (in detail at least) on the specific intended application, and the specific forward modeling approach used. Although we have discussed explicitly only MT and the simplest CSEM problems as examples, ModEM can be extended to treat a wide range of other EM inversion applications. In fact, much of the code developed for our specific examples could be reused in, or would at least provide a fairly complete template for, such additional applications. Modifications and extensions to an existing application (e.g., to parametrize or regularize the model in a different way, or to allow for additional sorts of data, such as inter-site transfer functions in the 3D MT inversion) are also comparatively simple, generally requiring changes or additions to only a single module.

It also should be apparent that it would be straightforward to implement additional inversion algorithms (e.g., quasi-Newton instead of NLCG (e.g., *Newman and Boggs*, 2004) or any of a number of Gauss-Newton variants such as OCCAM (implemented in either the model (*Constable et al.*, 1987), or data (*Siripunvaraporn et al.*, 2005) space). Translating pseudo-code of the sort shown in Figures 2 and 3 into an actual ModEM inversion module is relatively simple, since all of the key operators and data objects are readily available, and can be manipulated without reference to problem specific details. Furthermore, the transmitter/receiver structure built into the data space objects, and mirrored in our implementation of the Jacobian routines, allows more complex schemes such as the multi-frequency hybrid inversion of *Egbert* (2010) to be developed within ModEM. ModEM thus provides a natural test-bed for development, refinement, and comparison of inversion search algorithms.

The data vector transmitter/receiver structure has also been exploited to develop a flexible coarse-grained MPI parallelization (over forward problems) for ModEM. Only very trivial modifications to specific inversion routines are required, essentially to route calls to forward modeling and Jacobian calculations through parallelized MPI versions of these routines. The parallelization is thus effectively decoupled from other sorts of code modifications and extensions (new inversion algorithms, model parametrizations, data types). As our simple CSEM example illustrates, the

specialized data space structure can in some cases be further exploited to develop specialized, more efficient, code for Jacobian calculations.

In some respects ModEM has been designed specifically for frequency domain EM applications – e.g., the structure of data vectors is based on the multiplicity of transmitters and receivers common in EM geophysical inverse problems. However, ModEM should be more broadly useful, in particular as a good starting point for development of joint inversion applications. Inversion of multiple EM data types is in essence already implemented, as the data space structure supports simultaneous handling of a multiplicity of data types, with potentially very different characteristics. Our treatment of the 2D MT problem, where different forward solvers are called for TE and TM modes already provides a prototype for cases which would require a multiplicity of very different forward problems.

Joint inversion of EM and some other sort of geophysical data (e.g., seismic or gravity) could also be accommodated. In this case there might be essentially two separate sets of Level III modules, with possibly different grids, numerics, interpolation functionals, etc. The Level II interface routines would then merge and encapsulate these, hiding the details, and even the multiplicity of distinct physics involved. For example, the Level II solution vectors would essentially be a container, of fixed type, which could hold actual forward solutions for either sort of problem. The two sorts of data would of course have to be coupled through the model parameter, either through some sort of constitutive relation, or through the covariance.

It must be acknowledged that our implementation has limits in terms of generality. For example, we have so far designed inversion algorithms under the assumption that the regularization can be represented as a norm on the model space, defined by the quadratic form $\mathbf{C_m}$. If a model regularization term which could not be so represented were used, some modification to the inversion algorithms would be required. Even then the modifications would be relatively minor. In this regard we emphasize that while we have developed the Level I modules to be as generic as possible, this does not preclude further modification and extension of these routines for specialized applications, or for improved efficiency. Thus, at all levels, we view ModEM as a work in progress: already quite capable, but inherently designed for further modification and extension.

References

Alumbaugh, D. L., G. A. Newman, L. Prevost, and J. N. Shadid (1996), Three-dimensional wide band electromagnetic modeling on massively parallel computers, *Radio Science*, *33*, 1–23.

Constable, S. C., R. L. Parker, and C. G. Constable (1987), Occam's inversion: A practical

algorithm for generating smooth models from electromagnetic sounding data, *Geophysics*, *52*(3), 289–300.

Egbert, G. D. (2010), Efficient inversion of multi-frequency and multi-transmitter EM data, *in preparation*.

Egbert, G. D., and A. Kelbert (2012), Computational Recipes for Electromagnetic Inverse Problems, *Geophys. J. Int.*, *in publica*.

ISO/IEC TR 19767 (2005), Technical Report: Enhanced module facilities in Fortran., *Tech. rep.*, JTC1/SC22/WG5, ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1602.pdf.

Kelbert, A., G. D. Egbert, and A. Schultz (2008), Non-linear conjugate gradient inversion for global EM induction: resolution studies, *Geophysical Journal International*, *173*(2), 365–381, doi:10.1111/j.1365-246X.2008.03717.x.

Meqbel, N. (2009), The electrical conductivity structure of the Dead Sea Basin derived from 2D and 3D inversion of magnetotelluric data, Ph.D. thesis, Free University of Berlin, Berlin, Germany.

Newman, G. A., and D. L. Alumbaugh (1997), Three-dimensional massively parallel electromagnetic inversion – I. Theory, *Geophysical Journal International*, *128*, 345–354, doi:10.1111/j.1365-246X.1997.tb01559.x.

Newman, G. A., and P. T. Boggs (2004), Solution accelerators for large-scale three-dimensional electromagnetic inverse problems, *Inverse Problems*, *20*, 151–170, doi:10.1088/0266-5611/20/6/S10.

Nocedal, J., and S. J. Wright (2006), *Numerical Optimization*, Springer.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992), *Numerical Recipes in Fortran 77 – The Art of Scientific Computing*, second ed., Cambridge University Press.

Purser, R. J., W.-S. Wu, D. F. Parrish, and N. M. Roberts (2003a), Numerical Aspects of the Application of Recursive Filters to Variational Statistical Analysis. Part I: Spatially Homogeneous and Isotropic Gaussian Covariances, *Monthly Weather Review*, *131*(8), 1524–1535, doi: 10.1175//1520-0493(2003)131¡1524:NAOTAO¿2.0.CO;2.

Purser, R. J., W.-S. Wu, D. F. Parrish, and N. M. Roberts (2003b), Numerical Aspects of the Application of Recursive Filters to Variational Statistical Analysis. Part II: Spatially Inhomogeneous and Anisotropic General Covariances, *Monthly Weather Review*, *131*(8), 1536–1548, doi:10.1175//2543.1.

Siripunvaraporn, W., and G. D. Egbert (2000), An efficient data-subspace inversion method for {2-D} magnetotelluric data, *Geophysics*, *65*(3), 791–803.

Siripunvaraporn, W., and G. D. Egbert (2007), Data space conjugate gradient inversion for 2-D magnetotelluric data, *Geophysical Journal International*, *170*, 986–994, doi:10.1111/j.1365-246X.2007.03478.x.

Siripunvaraporn, W., G. D. Egbert, and Y. Lenbury (2002), Numerical accuracy of magnetotelluric modeling: A comparison of finite difference approximations, *Earth Planets and Space*, *54*(6), 721–725.

Siripunvaraporn, W., G. D. Egbert, Y. Lenbury, and M. Uyeshima (2005), Three-dimensional magnetotelluric inversion: data-space method, *Physics of the Earth and Planetary Interiors*, *150*(1-3), 3–14.

Smith, J. T. (1996), Conservative Modeling of {3-D} Electromagnetic Fields: 2. {B}iconjugate gradient solution and an accelerator, *Geophysics*, *61*(5), 1319–1324.