Computational Recipes for EM Inverse Problems: Mathematical Framework and Modular Implementation

Gary D. Egbert
College of Oceanic and Atmospheric Sciences
Oregon State University
Corvallis, OR

1 Introduction

The purpose of this paper is to provide an overview of a modular system of computer codes we have developed over the past few years for solving electromagnetic inverse problems with gradient-based search methods. As the reader will quickly see, there is little here that is by itself novel: the geophysical inversion algorithms discussed have all been previously implemented and described in the literature, and the ideas from computational sciences (e.g., object oriented programming, abstract data types) are conventional. However, to our knowledge, the integrated system we are describing is novel, and we believe it is worth describing in some detail.

There are several motivations for taking the modular approach described here. Our initial motivation was to develop a "test-bed" which could be used for more efficient prototyping of inversion algorithms, for example specialized inversion algorithms that take better advantage of the specific structure of multi-transmitter EM inverse problems. In particular, the modular system simplifies development of inversion codes for 3D problems, allowing initial debugging and testing algorithms on computationally much simpler 2D analogues, before application to the more challenging 3D case. More generally, a modular inversion system allows for more rapid adaptation of inversion codes developed for one purpose (e.g., 3D magnetotellurics (MT)) to other EM problems (e.g., marine controlled source EM (CSEM)) with comparatively minimal modification. 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 parameterization (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 code modularization is to abstract the basic data objects that appear in an inversion scheme (model parameters, EM solution, data vectors). These are treated as abstract data types, with basic methods developed for each class, including creation and destruction, and, as appropriate linear algebra or other vector space methods. Then, the fundamental mappings between classes are developed, along with the linearizations and adjoints, required for gradient computations. 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 attributes of these objects. Our approach could be termed object oriented, although some deviations from a strict adherence to this programming model seem warranted for the computationally intensive problem of 3D EM inversion–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 (such as they are): all code is written using modules, with explicit interfaces for all routines, data hiding through use of the "private" attribute (where practical), operator overloading, and extensive use of derived data types.

Modularity in our inversion system functions at two levels. Some parts of the system are literally generic: the same source code can be used for a wide range of problems. For other parts of the system it is necessary to have separate module instances for different applications. Even here, however, there are significant opportunities for code reuse: modules for a previously developed application serve essentially as templates for variants that must be developed for a second application. Of course this notion of code templates can be formalized, in particular using computer languages other than F95.

Throughout the development we refer to two specific example problems, which have served as test cases in our development of the modular system: 2D and 3D MT. These two EM inverse problems are actually quite different and provide good specific examples of some of the abstractions required of the modular system. For example, in the 2D MT problem, there are two decoupled modes: the transverse electric (TE) and transverse magnetic (TM) modes. The fundamental EM field solutions are scalar electric or magnetic fields, which are obtained by solving different scalar diffusion equations. For the 3D MT problem, there is no comparable decomposition into modes, and the fundamental EM solution consists of a pair of vector fields computed with boundary conditions appropriate to two external source polarizations. To develop a modular system capable of handling both cases, it is necessary to formulate the notion of an EM field solution abstractly, and hide instance specific attributes of the EM field solution object (e.g., TE/TM mode, source polarization) from higher level general inversion routines. Similarly, operators such as forward solvers or data functionals

must have interfaces that make no reference to problem specific attributes of EM solution or data objects. In our discussion we consider implementation details for the 2D and 3D MT cases, to illustrate with concrete examples some of the issues involved in abstraction and development of the modular system.

The paper is organized as follows. We begin with a brief summary of linearized, gradient based EM inversion, writing down the basic equations which define the classic regularized inversion problem in terms minimizing a penalty functional, and then outlining several standard gradient based solution algorithms. A key component in all of these algorithms is computation of the Jacobian of the mapping from model parameters to data. Here we give a general expression for the Jacobian which provides the conceptual basis for our modular scheme. In the third section we overview the modular system as a whole and discuss the basic data types and methods. Further details on specific modules, including implementation specifics for the the 2D and 3D MT cases are provided in section four. In section five we discuss parallelization issues. Some discussion on our experiences in developing the system, prospects for futures developments, and concluding remarks are given in the closing section.

2 Mathematical Background

2.1 Linearized EM inversion

We consider regularized inversion based on gradient-based minimization of a penalty functional of the form

$$\mathcal{J}(\mathbf{m},\mathbf{d}) = (\mathbf{d}-\mathbf{f}(\mathbf{m}))^T {\mathbf{C_d}}^{-1}(\mathbf{d}-\mathbf{f}(\mathbf{m})) + \nu(\mathbf{m}-\mathbf{m}_0)^T {\mathbf{C_m}}^{-1}(\mathbf{m}-\mathbf{m}_0) \qquad (1)$$

to recover, in a stable manner, Earth conductivity which provides an adequate fit to a data vector $\mathbf{d}$ of dimension N_d. In (1) $\mathbf{C_d}$ is the covariance of data errors, $\mathbf{m}$ is the M-dimensional model parameter vector (e.g., $ln(\sigma)$ for discrete blocks), $\mathbf{f}(\mathbf{m})$ defines the forward mapping (which must be computed numerically), $\mathbf{m}_0$ is a prior or first guess model parameter, ν is a regularization parameter, and $\mathbf{C_m}$ (or more properly $\nu^{-1}\mathbf{C_m}$) defines the model covariance or regularization term. In practice $\mathbf{C_d}$ is always taken to be diagonal, so by a simple rescaling of the data and forward mapping (${\mathbf{C_d}}^{-1/2}\mathbf{d}$, ${\mathbf{C_d}}^{-1/2}\mathbf{f}$), we may eliminate ${\mathbf{C_d}}^{-1}$ from the definition of $\mathcal{J}$.

The prior model and model covariance $\mathbf{C_m}$ can also be eliminated from (1) by the affine linear transformation of the model parameter $\tilde{\mathbf{m}} = \mathbf{C_m}^{-1/2}(\mathbf{m} - \mathbf{m_0})$, and forward mapping $\tilde{\mathbf{f}}(\tilde{\mathbf{m}}) = \mathbf{f}(\mathbf{C_m}^{1/2}\tilde{\mathbf{m}})$, reducing (1) to

$$\mathcal{J}(\tilde{\mathbf{m}}, \mathbf{d}) = (\mathbf{d} - \tilde{\mathbf{f}}(\tilde{\mathbf{m}}))^T(\mathbf{d} - \tilde{\mathbf{f}}(\tilde{\mathbf{m}})) + \nu \mathbf{m}^T\mathbf{m} = ||\mathbf{d} - \tilde{\mathbf{f}}(\tilde{\mathbf{m}})||^2 = \nu||\mathbf{m}||^2. \qquad (2)$$

After minimizing (2) over $\tilde{\mathbf{m}}$ we can easily recover $\mathbf{m} = \mathbf{C_m}^{1/2}\tilde{\mathbf{m}}$. Note that this model space transformation is in fact quite practical if instead of following the usual practice of defining $\mathbf{C_m}^{-1} = \mathbf{D}^T\mathbf{D}$, where $\mathbf{D}$ is a discrete representation of a gradient or higher order derivative operator, we formulate the regularization directly in terms of a smoothing operator (i.e., model covariance) $\mathbf{C_m}$. It is relatively easy to construct computationally efficient positive definite discrete symmetric smoothing operators for regularization (e.g., Egbert et al., 1994; Siripunvaraporn and Egbert, 2000; Chua and Bennett, 2001). Although the resulting covariance matrix $\mathbf{C_m}$ will not generally be sparse or practical to invert, all of the computations required for gradient computations and for minimization of the transformed penalty functions require only multiplication by the smoothing operator $\mathbf{C_m}^{1/2}$ (i.e, half of the smoothing of $\mathbf{C_m}$). (And note: it is quite possible to define the covariance so that multiplication by either $\mathbf{C_m}$ or $\mathbf{C_m}^{-1}$ is efficient.)

In the following we consider the simplified "canonical" penalty functional (2), with tildes omitted. We first summarize standard approaches to minimization using a consistent notation. Siripunvaraporn and Egbert (2000), Rodi and Mackie (2001), Newman and Boggs (2004), and Avdeev (2005) provide further details and discussion on these and related methods. Before proceeding we note that many of the computations in frequency domain EM problems are most efficiently implemented (and described) using complex arithmetic, but the model conductivity parameter $\mathbf{m}$ is real. Data might be complex (an impedance) or real (an apparent resistivity and/or phase). For simplicity we will assume that all data are real, i.e., real and imaginary parts of a complex impedance element will be treated as separate elements of the real data vector $\mathbf{d}$, and we assume that all operators have been recast to map real vectors to real vectors. Note, however, that in practice some computations are implemented most efficiently using complex arithmetic, and this must be taken into account to develop a practical and efficient inversion system.

The derivative of $\mathbf{f}$ with respect to the model parameters is the $N_d \times M$ sensitivity matrix (or Jacobian) $\mathbf{J}$, with ij element $J_{ij} = \partial f_i / \partial m_j$. As we show below, for general EM problems the Jacobian can be formally written

$$\mathbf{J} = \mathbf{L}\mathbf{S}_{\mathbf{m}}^{-1}\mathbf{P} + \mathbf{Q}, \tag{3}$$

where $\mathbf{L}$ represents the measurement process (i.e., the linearized data functionals); $\mathbf{S}_{\mathbf{m}}^{-1}$ denotes the EM solution operator for conductivity distribution $\mathbf{m}$; and $\mathbf{P}$ and $\mathbf{Q}$ are, respectively, $N_e \times M$ and $N_d \times M$ matrices related to the model parameterization (of dimension M), numerical EM solution representation (of dimension N_e), and the measurement functionals (of dimension N_d). Newman and Alumbaugh (1997), Spitzer (1998) and Rodi and Mackie (2001) provide more detailed discussions of (3) for some specific EM inverse problems. Further details for the 2D and 3D MT inverse problems are given below. The key points for our discussion are (1) computation of the sensitivity matrix requires application of the forward solver M times (once for each column of $\mathbf{J}$), or the transpose (adjoint) of this solver ${\mathbf{S}_{\mathbf{m}}^T}^{-1}$ N_d times (once for each row of $\mathbf{J}$); and (2) multiplication of an arbitrary model (data) space vector by $\mathbf{J}$ ($\mathbf{J}^T$) requires one forward (adjoint) solution. Because the EM problem is essentially self-adjoint (i.e., in physical terms, reciprocity holds) solving the adjoint problem is essentially equivalent to solving the forward problem.

Gradient-based search for a minimizer of (2) using $\mathbf{J}$ is iterative, as for example in the classical Gauss-Newton (GN) method. Let $\mathbf{m}_n$ be the model parameter at the nth iteration, $\mathbf{J}^n$ the sensitivity matrix evaluated at $\mathbf{m}_n$, and $\mathbf{r}_n = \mathbf{d} - \mathbf{f}(\mathbf{m}_n)$ the data residual. Then linearizing the penalty functional in the vicinity of $\mathbf{m}_n$ for small perturbations $\delta\mathbf{m}$ leads to the $M \times M$ system of normal equations

$$({\mathbf{J}^n}^T\mathbf{J}^n + \nu\mathbf{I})\delta\mathbf{m} = {\mathbf{J}^n}^T\mathbf{r}_n - \nu\mathbf{m}_n, \tag{4}$$

which can be solved for $\delta\mathbf{m}$ to yield a new trial solution $\mathbf{m}_{n+1} = \mathbf{m}_n + \delta\mathbf{m}$. As discussed in Parker (1994) this basic linearized scheme generally requires some form of step length damping for stability (e.g., a Levenberg-Marquardt approach; Marquardt, 1963; Rodi and Mackie, 2001).

There are two variants on the standard GN approach relevant to our development. In the

Occam approach (Constable et al., 1987; Parker, 1994) (4) is rewritten

$$(\mathbf{J}^{n\,T}\mathbf{J}^n + \nu\mathbf{I})\mathbf{m} = \mathbf{J}^{n\,T}\hat{\mathbf{d}}, \tag{5}$$

where $\hat{\mathbf{d}} = \mathbf{d} - \mathbf{f}(\mathbf{m}_n) + \mathbf{J}^n\mathbf{m}_n$. Although $\mathbf{m}_{n+1}$ is obtained directly as the solution to (5) the result is exactly equivalent to solving (4) for the change in the model at step $n+1$ and adding the result to $\mathbf{m}_n$. A more substantive difference is that in the Occam scheme step length control is achieved by varying ν, computing a series of trial solutions $\mathbf{m}_\nu$ to (5), and solving the forward problem for each $\mathbf{m}_\nu$ to evaluate the actual data misfit. An advantage of this approach is that the regularization parameter ν is determined as part of the search process, and at convergence one is assured that the solution attains at least a local minimum of the model norm $||\mathbf{m}|| = (\mathbf{m}^T\mathbf{m})^{1/2}$, subject to the data fit attained (Parker, 1994). In other approaches ν must be varied independently to choose an optimal value (e.g., Newman and Alumbaugh 2000; Newman and Boggs, 2004).

The Occam scheme can also be implemented in the data space (Siripunvaraporn and Egbert, 2000; Siripunvaraporn et al., 2005). The solution to (5) can be written as

$$\mathbf{m}_{n+1} = \mathbf{C}_\mathbf{m}\mathbf{J}^{n\,T}\mathbf{x} \tag{6}$$

where the coefficients $\mathbf{x}$ satisfy

$$(\mathbf{J}^n\mathbf{J}^{n\,T} + \nu^{-1}\mathbf{I})\mathbf{x} = \hat{\mathbf{d}}, \tag{7}$$

as can be verified by substituting (6) and (7) into (5) and simplifying. This "data space Occam" (DASOCC) approach requires solving an $N_d \times N_d$ system of equations in the data space instead of the $M \times M$ model space system of equations (5). If the model is heavily over-parameterized N_d may be much smaller than M, making the DASOCC scheme significantly more efficient than equivalent model space approaches.

Computing the full Jacobian $\mathbf{J}$ required for the three variants on the GN approach described above is a very demanding computational task for multi-dimensional EM problems, since the equivalent of one forward solution is required for each row (or column) of $\mathbf{J}$. An alternative approach is to solve these equations with an iterative solver such as CG. This requires computation of matrix vector products such as $\mathbf{Am} = [\mathbf{J}^T\mathbf{J} + \nu\mathbf{I}]\mathbf{m}$, which (as (3)

already shows) can be accomplished without forming or storing $\mathbf{J}$ at the cost of two forward solutions (e.g., Mackie and Madden; 1993). Mackie and Madden (1993), Zhang et al. (1995), Newman and Alumbaugh (1996), Rodi and Mackie (2001), and others, have used CG to solve (4), while Siripunvaraporn and Egbert (2007) have applied the same approach to the corresponding data space equations of (6).

Whether a model data space approach is used, CG iteratively solves for $\mathbf{m}_{n+1}$ based on linearization in the vicinity of $\mathbf{m}_n$. This iterative solver is then embedded in a further outer loop (over n) to solve the non-linear inverse problem. Alternatively one can apply CG directly as a non-linear quadratic optimization scheme (e.g., Press et al., 1986) to minimize (1) (e.g., Rodi and Mackie, 2001; Newman and Boggs, 2004; Avdeev, 2005). With this NLCG approach one must evaluate the gradient of (1) with respect to variations in model parameters $\mathbf{m}$:

$$\left.\frac{\partial \mathcal{J}}{\partial \mathbf{m}}\right|_{\mathbf{m}_n} = -\mathbf{J}^{nT}(\mathbf{r}_n) + \nu \mathbf{m}_n. \tag{8}$$

The gradient is then used to calculate a new "conjugate" search direction in the model space. After minimizing the penalty functional along this direction, using a line search which requires at most a few evaluations of the forward operator, the gradient is recomputed. NLCG again utilizes essentially the same basic computational steps as required for solving the linearized equations (4) (i.e., the forward problem must be solved to evaluate $\mathbf{f}(\mathbf{m})$, and for the line search, and (3) implies that multiplication of the residual $\mathbf{r}_n$ by $\mathbf{J}^{nT}$), can be accomplished by solving the adjoint problem (e.g., Newman and Alumbaugh, 2000). Quasi-Newton schemes (QN; e.g., Nocedal and Wright, 1999; Newman and Boggs, 2004; Haber, 2005; Avdeev, 2005) provide an alternative approach to NLCG for direct minimization of (1), with similar advantages with regard to storage and computation of the Jacobian, and similar computational requirements.

The key point here is that all of these gradient based schemes for minimizing (1) can be abstractly expressed in terms of a small number of basic objects (data and model parameter vectors, $\mathbf{d}$ and $\mathbf{m}$), and operators (the forward mapping $\mathbf{f}(\mathbf{m})$, and the corresponding Jacobian $\mathbf{J}$, the data and model covariances $\mathbf{C_m}$ and $\mathbf{C_d}$).

So far we have ignored an important issue that will be fairly central to the structure of

the modular system: in most cases EM data are obtained for a wide range of frequencies (MT), and/or for a number of different transmitter configurations (controlled source problems). Each of these different frequencies or source configurations (generically referred to as "transmitters") requires solving a different forward problem. One can extend this notion further to consider joint inversion of multiple data types, including combining EM and other geophysical data. In general, then, the data vector and forward modeling operator will be decomposed into N_T blocks, one for each transmitter

$$\mathbf{d} = (\mathbf{d}_1^T ... \mathbf{d}_{N_T}^T) \qquad \mathbf{f} = (\mathbf{f}_1^T ... \mathbf{f}_{N_T}^T). \tag{9}$$

The Jacobian, and the matrices in the general expression for $\mathbf{J}$ in (3), will be partitioned similarly into N_T blocks. We return to the issue of multiple transmitters below, but for our initial discussion of Jacobian calculations we consider the simpler case of a single transmitter.

2.2 Data Sensitivities for EM Problems: General Considerations

We first derive the general expression for the sensitivity matrix (3), omitting reference to the iteration index n, and also ignoring multiple transmitter issues. The numerical discretization of the frequency domain EM differential equation is written generically as

$$\mathbf{S_m e} = \mathbf{b} \tag{10}$$

where $\mathbf{b}$ gives appropriate boundary and forcing terms for the particular EM problem, $\mathbf{e}$ is the N_e dimensional vector representing the discretized electric and/or magnetic fields (or perhaps potential functions), and $\mathbf{S_m}$ is an $N_\mathbf{e} \times N_\mathbf{e}$ matrix which depends on the M dimensional model parameter $\mathbf{m}$. Given a solution $\mathbf{e}$ to (10) data can always be written in the general form

$$d_j = f_j(\mathbf{m}) + \epsilon_j = \psi_j(\mathbf{e}(\mathbf{m}), \mathbf{m}) + \epsilon_j \tag{11}$$

where ψ_j is some generally non-linear, but usually simple, function of the components of $\mathbf{e}$ (and possibly $\mathbf{m}$), and ϵ_j represents data error.

With this general setup we have, by the chain rule

$$J_{jk} = \frac{\partial f_j}{\partial m_k} = \frac{\partial \psi_j}{\partial m_k} = \sum_l \frac{\partial \psi_j}{\partial e_l} \frac{\partial e_l}{\partial m_k} + \frac{\partial \psi_j}{\partial m_k} \quad . \tag{12}$$

Let $\mathbf{F}, \mathbf{L}, \mathbf{Q}$ be the partial derivative matrices

$$F_{lk} = \left.\frac{\partial e_l}{\partial m_k}\right|_{\mathbf{m}_0} \qquad L_{jl} = \left.\frac{\partial \psi_j}{\partial e_l}\right|_{\mathbf{e}_0,\mathbf{m}_0} \qquad Q_{jk} = \left.\frac{\partial \psi_j}{\partial m_k}\right|_{\mathbf{e}_0,\mathbf{m}_0}, \tag{13}$$

where $\mathbf{e}_0$ is the solution to (10) for model parameter $\mathbf{m}_0$. Then the Jacobian at $\mathbf{m}_0$ can be written in matrix notation as

$$\mathbf{J} = \mathbf{LF} + \mathbf{Q}. \tag{14}$$

Note that $\mathbf{F}$, $\mathbf{L}$ and $\mathbf{Q}$ in general depend on the model parameter $\mathbf{m}$. The rows of $\mathbf{L}$ (one for each observation) are generally very sparse, supported only on a few nodes surrounding the corresponding data site. When the observation functionals are independent of the model parameters (as they often are) $\mathbf{Q} \equiv \mathbf{0}$. When $\mathbf{Q}$ is required it is also typically sparse, but this depends on the specific nature of the model parameterization. The jth row of $\mathbf{L}$ represents the linearized data functional, which is applied to the perturbation in the EM solution to compute the perturbation in d_j. Although coding $\mathbf{L}$ and $\mathbf{Q}$ can be quite involved for realistic EM data functionals, calculation of $\mathbf{F}$ presents the only real computational challenge.

To derive a general expression for $\mathbf{F}$ take the derivative of (10) with respect to the model parameters $\mathbf{m}$. Assuming $\mathbf{b}$ remains fixed (if this is not true minor modifications are necessary; Newman and Boggs (2004) treat this additional complication), and allowing that as $\mathbf{m}$ is varied the solution $\mathbf{e}$ also varies, we obtain

$$\mathbf{S}_{\mathbf{m}_0}\left[\left.\frac{\partial \mathbf{e}}{\partial \mathbf{m}}\right|_{\mathbf{m}=\mathbf{m}_0}\right] = -\frac{\partial \mathbf{S}_{\mathbf{m}}\mathbf{e}_0}{\partial \mathbf{m}}, \tag{15}$$

or

$$\mathbf{S}_{\mathbf{m}_0}\mathbf{F} = \mathbf{P}. \tag{16}$$

The $N_e \times M$ matrix $\mathbf{P}$ depends on details of both the numerical model implementation and the conductivity parameterization, but is in general inexpensive to calculate.

Putting together (14) and (16) the $N_d \times M$ Jacobian can be expressed as

$$\mathbf{J} = \mathbf{L}\mathbf{S}_{\mathbf{m}}^{-1}\mathbf{P} + \mathbf{Q}. \tag{17}$$

Computing all of $\mathbf{J}$ would appear to require solving the induction equation once for each of the M columns of $\mathbf{P}$. However, simply taking the transpose of (17) we obtain

$$\mathbf{J}^T = \mathbf{P}^T[\mathbf{S}_{\mathbf{m}}^T]^{-1}\mathbf{L}^T + \mathbf{Q}^T, \tag{18}$$

so the sensitivity matrix can in fact be obtained by solving the transposed discrete EM system N_d times, the usual "reciprocity" trick for efficient calculation of sensitivities (e.g., Rodi, 1976, de Lugao *et al.*, 1997).

Note that the EM equations are self-adjoint (except for time reversal) with respect to the usual L^2 inner product (i.e., reciprocity holds), so on a uniform grid $\mathbf{S_m}$ is symmetric (leaving aside issues regarding boundary conditions). For more general grids the fact that the EM operator is self-adjoint implies

$$\mathbf{S}_\mathbf{m}^T = \mathbf{V}\mathbf{S}_\mathbf{m}\mathbf{V}^{-1}, \tag{19}$$

where $\mathbf{V}$ is a diagonal matrix of integration volume elements for the natural discrete representation of the L_2 integral inner product on the model domain. Eq. (19) implies $\mathbf{S}_\mathbf{m}^T\mathbf{V} = \mathbf{V}\mathbf{S}_\mathbf{m}$ is a symmetric (though not Hermitian) matrix. It is easier to compute solutions to this symmetrized problem, so solutions to the forward problem are generally computed as $\mathbf{e} = (\mathbf{V}\mathbf{S}_\mathbf{m})^{-1}\mathbf{V}\mathbf{b}$. The solution for the adjoint problem can also be written in terms of the symmetrized inverse operator as $\mathbf{e} = (\mathbf{S}_\mathbf{m}^T)^{-1}\mathbf{b} = \mathbf{V}(\mathbf{V}\mathbf{S}_\mathbf{m})^{-1}\mathbf{b}$; the the only difference from the forward case is thus the thus order in which multiplication by the diagonal matrix $\mathbf{V}$ and the symmetric solver are called.

2.3 Implementation of P, L, and Q: general considerations and specific examples

To derive more explicit expressions for the operators $\mathbf{L}$, $\mathbf{P}$, and $\mathbf{Q}$ of (17) and (18) we need to be more specific about the numerical implementation of the forward problem (10). To motivate our general development we consider two specific cases: inversion of 2D and 3D MT data, using a finite difference modeling approach, but the results are more broadly applicable. Initially we consider only a single fixed frequency.

In the quasi-static limit appropriate for MT, Maxwell's equations in the frequency domain (i.e., assuming a time dependence of $e^{i\omega t}$) can be expressed as a second order elliptic system of partial differential equations in terms of the electric fields alone

$$\nabla \times \nabla \times \mathbf{E} + i\omega\mu\sigma\mathbf{E} = 0 \qquad (+ \text{ BC}). \tag{20}$$

To solve (20) numerically in 3D, we consider a finite difference approximation on a staggered grid of dimension $N_x \times N_y \times N_z$, as illustrated in Figure 1 (e.g., Yee, 1966; Smith, 1996; Siripunvaraporn et al., 2002). Components of the electric field vector (the basic EM solution object $\mathbf{e}$) are defined on cell edges. We denote the space of these discrete EM solution vectors by $\bar{\mathcal{S}}_P$.

As illustrated in Figure 1, the magnetic fields, which in continuous form satisfy $\mathbf{B} = (-i\omega)^{-1}\nabla \times \mathbf{E}$, are naturally defined on the discrete grid on cell faces. We denote the space of discrete vector fields defined on faces as $\bar{\mathcal{S}}_S$, and a typical element will be denoted $\tilde{\mathbf{e}}$. The two spaces $\bar{\mathcal{S}}_P$ and $\bar{\mathcal{S}}_S$ are in conjugate, and we will refer to $\mathbf{e}$ (the field that is solved for) as the primary field, and $\tilde{\mathbf{e}}$ the secondary field. In fact, only edges in the interior of the grid are solved for; the tangential components on the boundary edges are provided as input data. We denote the space of these interior nodes (of dimension $N_e = 3N_xN_yN_z - N_xN_y - N_xN_z - N_yN_z$) by $\mathcal{S}_P$, and the corresponding space of interior faces by $\mathcal{S}_S$.

In our 3D example the primary and secondary field components represent electric and magnetic fields, respectively. These are related via

$$\tilde{\mathbf{e}} = (-i\omega)^{-1}\mathbf{C}\mathbf{e}, \tag{21}$$

where $\mathbf{C} : \bar{\mathcal{S}}_P \mapsto \mathcal{S}_S$ is the discrete approximation of the curl of interior cell edge vectors. The discrete operator in (10) can then be expressed as

$$\mathbf{S_m} = \mathbf{C}^\dagger\mathbf{C} + \mathbf{diag}[i\omega\mu\sigma(\mathbf{m})]. \tag{22}$$

Here $\mathbf{C}^\dagger : \mathcal{S}_S \mapsto \mathcal{S}_P$ is the discrete curl mapping cell face vectors to (interior) cell edges. As the notation indicates this operator is the adjoint of $\mathbf{C}$, relative to appropriate inner products on the spaces $\mathcal{S}_S$ and $\mathcal{S}_P$. For completeness, further discussion of these operators are given in the Appendix. In particular we provide details on treatment of interior and boundary nodes, and on how the right hand side $\mathbf{b}$ of (10) is determined from the boundary data.

In (22) the dependence of the operator coefficients on $\mathbf{m}$ is made explicit through $\sigma(\mathbf{m}) \in \mathcal{S}_P$, a real vector which represents the average conductivity surrounding cell edges. As a specific example we consider the simplest model parameterization, with conductivity, or the natural logarithm of conductivity, specified independently for each of the $M = N_xN_yN_z$

cells in the numerical grid. We denote by $\mathbf{W}$ the $N_e \times M$ matrix which defines the volume weighted average from the four surrounding cells to each of the interior edges. Then a physically consistent (current conserving) mapping of conductivity to cell edges is given by $\sigma(\mathbf{m}) = \mathbf{Wm}$ or $\mathbf{Wexp}(\mathbf{m})$, for the cases of linear and log conductivity.

It is also instructive to consider the 2D MT inverse problem. Now there are effectively two distinct modeling problems: for TE and TM modes, with electric and magnetic fields, respectively, parallel to the geologic strike. The TE mode case is essentially identical to the 3D case already discussed. The TM mode case, which is solved in terms of the magnetic field instead of the electric field, is more instructive with regard to generalization. In the full 3D case, the quasi-static magnetic induction equation takes the form

$$\nabla \times \rho \nabla \times \mathbf{B} - i\omega\mu\mathbf{B} = 0 \qquad (+ \text{ BC}). \tag{23}$$

In the TM mode the magnetic field parallels the geological strike (x) and (23) can be reduced to a scalar partial differential equation (PDE) in the $y - z$ plane

$$\partial_y \rho \partial_y B_x + \partial_z \rho \partial_z B_x + i\omega\mu B_x = 0 \quad (+BC). \tag{24}$$

As for the 3D MT problem, with the second order PDE formulated in terms of the electric fields, we can define finite dimensional spaces of primary ($\mathcal{S}_P$) and secondary ($\mathcal{S}_S$) EM fields. Now the the primary field is B_x, defined on the nodes (corners) of the 2D grid, and the secondary fields are the electric field components E_y and E_z defined on the vertical and horizontal cell edges (Figure 2). A natural centered finite difference approximation of (24) can be written in terms of a discrete 2D gradient operator $\mathbf{G} : \mathcal{S}_P \mapsto \mathcal{S}_S$ and a 2D divergence operator $\mathbf{D}\mathcal{S}_S \mapsto \mathcal{S}_P$. Using $\mathbf{e} \in \mathcal{S}_P$ to denote the primary discrete EM field solution (B_x) we have a more explicit form for (10) for this TM mode modeling implementation

$$\left[\mathbf{Ddiag}\left[\rho(\mathbf{m})\right]\mathbf{Ge} - i\omega\mu\mathbf{I}\right]\mathbf{e} = \mathbf{b}. \tag{25}$$

Again, there are some minor technical issues with regard to boundary conditions (which effectively define $\mathbf{b}$), which are discussed for completeness in the Appendix.

In (25) the PDE coefficients depend on the model parameters through $\rho(\mathbf{m}) \in \mathcal{S}_S$, i.e., the resistivity defined on cell edges. To be specific, we again consider the simplest model

parameterizations, with conductivity or log conductivity for each cell in the numerical grid an independent parameter. From physical considerations, it is most reasonable to compute the required edge resistivities from cell conductivities by first transforming to resistivity, and then computing the area weighted average of resistivities of the two cells on either side of the edge. Representing the averaging operator from cells to cell edges as $\mathbf{W}_{TM}$, we then have

$$\rho(\mathbf{m}) = \mathbf{W}_{TM}(\mathbf{m})^{-1} \tag{26}$$

$$\rho(\mathbf{m}) = \mathbf{W}_{TM}\mathbf{exp}(-\mathbf{m}) \tag{27}$$

for linear and log conductivity respectively. In (26) $(\mathbf{m})^{-1}$ denotes the component-wise inverse of the model parameter vector.

The two specific examples discussed here illustrate key points which will hold for many other EM modeling approaches. In particular, there are two spaces of discrete vectors on conjugate grids $\mathcal{S}_P$ and $\mathcal{S}_S$, defining primary and secondary fields, respectively. The equations are reduced to a second order equation in the primary fields, but for several reasons it is worthwhile to explicitly consider the secondary fields defined on the conjugate grid as well. First, in both of the examples considered the secondary fields correspond to electric or magnetic fields, which the data functionals discussed below will depend on. Second, the dependence of the discrete PDE operator coefficients on the model parameter can generally be represented explicitly through a mapping $\pi(\mathbf{m})$ from the model parameter space $\mathcal{M}$ to either $\mathcal{S}_P$ or $\mathcal{S}_S$.

We emphasize here that our formulation would apply to any basis functions used to represent conductivity or log conductivity in the model domain–only the matrices $\mathbf{W}$ or $\mathbf{W}_{TM}$ which define the appropriate volume or area averages of conductivity or resistivity would change. Other mappings from a model parameter space (e.g., a sharp boundary inversion, where model parameters define the position of a layer interface) also fit within this general framework, although the explicit form for the mapping $\pi(\mathbf{m})$ would become more complicated.

In other finite difference modeling approaches, e.g. if Maxwell's equations are cast in terms of vector potentials, similar (although somewhat more complicated) sets of conjugate

spaces can be defined, the differential operator can be decomposed into discrete approximations to first order linear differential operators which map between conjugate grids, and the dependence of discrete operator coefficients on an abstract model parameter space can be described through a mapping $\pi : \mathcal{M} -> \mathcal{S}_{P,S}$. Finite element approaches to EM modeling will result in similar structures, with conjugate grids now representing space of element nodes and element integration points.

2.3.1 Implementation of P

To derive a general form for the matrix $\mathbf{P}$ we write the discrete EM operator as

$$\mathbf{S_m} = \mathbf{S}_0 + \tilde{\mathbf{R}}\mathbf{diag}[\pi]\mathbf{R}. \tag{28}$$

The 3D finite difference staggered grid electric field ($\mathbf{S}_0 = \mathbf{C}^\dagger\mathbf{C}$, $\tilde{\mathbf{R}} = i\omega\mu\mathbf{I}$, $\mathbf{R} = \mathbf{I}$, $\pi(m) = \sigma(m)$), and the 2D TM mode case ($\mathbf{S}_0 = i\omega\mu\mathbf{I}$, $\tilde{\mathbf{R}} = \mathbf{G}$, $\mathbf{R} = \mathbf{D}$, $\pi(\mathbf{m}) = \rho(\mathbf{m})$) are special cases. Then, writing out the components of $\mathbf{P}$ (see (15))

$$P_{ji} = \frac{\partial[\mathbf{S_m}\mathbf{e}_0]_j}{]}\partial m_i = \frac{\partial\left[\sum_k \tilde{R}_{jk}\pi_k(\mathbf{m})\sum_l R_{kl}e_{0l}\right]}{\partial m_i} = \sum_k \tilde{R}_{jk}\left[\sum_l R_{kl}e_{0l}\right]$$
$$\frac{\partial \pi_k(\mathbf{m})}{\partial m_i} = \left[\tilde{\mathbf{R}}\mathbf{diag}[\mathbf{R}\mathbf{e}_0]\frac{\partial \pi}{\partial \mathbf{m}}\right]_{ji}. \tag{29}$$

Thus we have the explicit matrix expression for $\mathbf{P}$ for the general case (28)

$$\mathbf{P} = \left[\tilde{\mathbf{R}}\mathbf{diag}[\mathbf{R}\mathbf{e}_0]\right]\frac{\partial \pi}{\partial \mathbf{m}}. \tag{30}$$

For the 3D MT case with the model parameterized using log conductivity for each cell in the numerical discretization we find

$$\mathbf{P} = \left[\mathbf{diag}(i\omega\mu\mathbf{e}_0)\right]\left[\mathbf{W}\mathbf{diag}(\mathbf{m}_0)\right]. \tag{31}$$

For the 2D TM case with the comparable simple logarithmic conductivity parameterization we have

$$\mathbf{P} = \left[\mathbf{D}\mathbf{diag}\left[\mathbf{G}\mathbf{e}_0\right]\right]\left[\mathbf{W}_{TM}\mathbf{diag}[\mathbf{exp}(-\mathbf{m})]\right]. \tag{32}$$

Slightly simpler expressions are obtained in the case of a linear conductivity parameterization.

Note that in all cases multiplication by $\mathbf{P}$ involves application of simple diagonal, local averaging, and finite difference operators. Multiplication by the transpose $\mathbf{P}^T$ requires adjoints of the finite difference and averaging operators, but otherwise presents no significant complications. For example, for (32) the adjoint is given by

$$\mathbf{P}^T = \left[\mathbf{diag}[\mathbf{exp}(-\mathbf{m})]\mathbf{W}_{\mathbf{TM}}^{\mathbf{T}}\right]\left[\mathbf{diag}\left[\mathbf{G}\mathbf{e}_0\right]\mathbf{D}^T\right]. \tag{33}$$

Note the adjoint of the averaging operators considered here ($\mathbf{W}$ and $\mathbf{W}_{TM}$) represent mappings from cell edges to cells, summing contributions from all edges that bound a cell.

Eq. (30) provides a broadly applicable recipe for construction of the operators $\mathbf{P}$ and $\mathbf{P}^T$. A key point to emphasize is that these operators divide into two components, indicated by the brackets in (31)-(33). Only the second part depends on details of the model parameterization.

2.3.2 Implementation of L and Q

The matrices $\mathbf{L}$ and $\mathbf{Q}$ represent linearizations of the data functionals. $\mathbf{L}$ connects perturbations in the primary EM solution ($\delta\mathbf{e}$) to perturbations in the data; $\mathbf{Q}$ gives the component of variations in the observations due to possible dependence of the data functionals on the model parameters. In general, EM data used in an inversion are functions of electric and magnetic field components observed at one or more locations. These functionals can thus always be expressed as

$$\psi_j(\mathbf{e}(\mathbf{m}),\mathbf{m}) = \zeta_j(\vec{\theta}(\mathbf{e},\mathbf{m})) = \zeta_j(\theta_1(\mathbf{e},\mathbf{m}),...,\theta_K(\mathbf{e},\mathbf{m})) = \zeta_j(\lambda_1(\mathbf{m})^T\mathbf{e},...,\lambda_K(\mathbf{m})^T\mathbf{e}), \tag{34}$$

where $\lambda_k(\mathbf{m}), k = 1,...,K$ represent more basic functionals which evaluate an electric or magnetic component at a point in the model domain. From the definition in (13) row j of $\mathbf{L}$ and of $\mathbf{Q}$ are then given by

$$\mathbf{L}_{j\cdot} = \frac{\partial\psi_j}{\partial\mathbf{e}} = \sum_k \frac{\partial\zeta_j}{\partial\theta_k}\bigg|_{\vartheta(\mathbf{e}_0,\mathbf{m}_0)} \lambda_k(\mathbf{m})^T = \sum_k \alpha_{jk}\lambda_k(\mathbf{m})^T, \tag{35}$$

$$\mathbf{Q}_{j\cdot} = \frac{\partial\psi_j}{\partial\mathbf{m}} = \sum_k \alpha_k \frac{\partial\lambda_k(\mathbf{m})^T\mathbf{e}_0}{\partial\mathbf{m}}. \tag{36}$$

The evaluation functionals λ_k are represented by elements of the primary solution space $\mathcal{S}_P$ which are sparse, with non-zero values only on grid elements surrounding the evaluation

location. Eq. (35) implies that we can decompose $\mathbf{L}$ into two sparse matrices as

$$\mathbf{L} = \mathbf{A}^T \Lambda^T, \tag{37}$$

with the non-zero elements in row j of $\mathbf{A}$ just the coefficients α_{jk}, and Λ constructed from the full set of evaluation functionals. Λ depends only on the observation locations (and possibly the model parameter $\mathbf{m}$, as we discuss below). Observation functionals for any sort of EM data will be constructed from the same basic evaluation functionals, which must be closely tied to the specific numerical discretization scheme used. $\mathbf{A}$ on the other hand, depends on details of the observation functionals, and will also depend in general on the EM solution $\mathbf{e}_0$. However, $\mathbf{A}$ does not depend on details of the numerical discretization of the problem. We take advantage of this division in our development of the modular system.

The basic evaluation functionals can depend on $\mathbf{m}$ through the values of resistivity or conductivity on one of the conjugate grids, defined by the mapping we have denoted generically by $\pi(\mathbf{m})$. Then we have

$$\mathbf{Q} = \mathbf{A}^T \frac{\partial \Lambda(\pi(\mathbf{m}))^T \mathbf{e}_0}{\partial \mathbf{m}} = \mathbf{A}^T \left[\left. \frac{\partial \Lambda(\pi))^T \mathbf{e}_0}{\partial \pi} \right|_{\pi(\mathbf{m}_0)} \right] \frac{\partial \pi(\mathbf{m})}{\partial \mathbf{m}} = \mathbf{A}^T \mathbf{B} \frac{\partial \pi(\mathbf{m})}{\partial \mathbf{m}}. \tag{38}$$

The matrices $\mathbf{A}$ and $\mathbf{B}$ depend on the background model parameter $\mathbf{m}_0$ through the EM solution $\mathbf{e}_0$ and the grid conductivity/resistivity $\pi(\mathbf{m}_0)$, but they do not depend on details of how the model is parameterized. Thus, as for $\mathbf{P}$ (see (30)) the part of $\mathbf{Q}$ which depends on details of the model parameter implementation can generally be separated from those parts which depend only on the specific data functionals and the numerical grid.

As specific examples, we consider implementation of data functionals for 2D and 3D MT impedances. Data functionals for other sorts of EM data (including apparent resistivities and phases for MT) can be derived by a similar procedure, with the linearized functionals expressed as linear combinations of basic evaluation functionals for electric and magnetic field components. First consider the evaluations functionals (Λ) for these problems. In both cases we need to evaluate primary and secondary field components (electric and magnetic fields) to compute impedances from the primary EM solution $\mathbf{e}$. The secondary fields (magnetics for our 3D example, and for the 2D TE mode, electrics for the 3D TM mode), can be written

in general as $\tilde{\mathbf{e}} = \mathbf{Te}$, where $\mathbf{T} : \mathcal{S}_P \mapsto \mathcal{S}_S$ is an appropriate differential operator. Then

$$\Lambda = \left[\begin{array}{cc} \Lambda_P & \Lambda_S \end{array} \right] = \left[\begin{array}{c} \Lambda_P \mathbf{T} \tilde{\Lambda}_S \end{array} \right], \tag{39}$$

where columns of Λ_P and $\tilde{\Lambda}_S$ are, respectively, interpolation operators on the primary and conjugate (secondary) grids. For the 3D MT example $\mathbf{T} = (i\omega)^{-1}\mathbf{C}$ maps from edges to faces, and interpolation from edges and faces to an arbitrary location within the model domain can be based on something simple such as tri-linear splines (but see discussion below). Then Λ is independent of $\mathbf{m}$, and $\mathbf{Q} \equiv 0$. Note that in some cases the evaluation functionals can be somewhat more complicated than expression (39) suggests. These details are not central to our general discussion, but an example of such complications, which is relevant to the TM MT problem, is discussed in the Appendix.

For the 2D TE mode case, $\mathbf{T} = (-i\omega)^{-1}\mathbf{OG}$, where $\mathbf{O}$ is a diagonal operator with entries $+1$ and -1 for components corresponding to y and z edges respectively. Columns of Λ_P and $\tilde{\Lambda}_S$ represent bi-linear spline interpolation from nodes and faces, respectively, to the data sites. Again $\mathbf{Q} \equiv 0$. For 2D TM are the same, but now $\mathbf{T} = \mathbf{diag}[\rho(\mathbf{m})]\mathbf{OG}$. Then we find

$$\mathbf{Q} = \mathbf{A}^T \Lambda^T = \mathbf{A}^T \left[\begin{array}{c} 0 \\ \left(\mathbf{diag}[\mathbf{OGe}_0] \tilde{\Lambda}_S \right)^T \end{array} \right] \frac{\partial \rho}{\partial \mathbf{m}} = \tilde{\mathbf{Q}} \frac{\partial \rho}{\partial \mathbf{m}}. \tag{40}$$

The matrix $\tilde{\mathbf{Q}}$ is sparse, and is independent of details of the model parameterization. In the context of the modular system multiplication by $\mathbf{Q}$ (or $\mathbf{Q}^T$) can thus be broken into two separately implemented steps, as noted above.

Finally, we give further details on $\mathbf{L}$ and $\mathbf{Q}$ for the case of 2D and 3D MT impedances. In the first case we have $Z = f(\mathbf{e}) = L_E\mathbf{e}/L_B\mathbf{e}$. From (35) we can write down the linearized data functional for a scalar impedance at a single location, i.e., one row of $\mathbf{L}$ is

$$L_j = \lambda_Z = (\lambda_B \mathbf{e}_0)^{-1} \lambda_E - \lambda_E \mathbf{e}_0 / (\lambda_B \mathbf{e}_0)^2] \lambda_B. \tag{41}$$

In the TE case λ_E and $\lambda_B = (-i\omega)^{-1}\mathbf{G}\tilde{\lambda}_B$ are, respectively, columns of Λ_P and Λ_S, and λ_E and $\tilde{\lambda}_B$ represent bi-linear spline interpolation functionals on node and edge spaces. For the TM mode the role of primary and secondary fields is reversed, with λ_B just interpolation of the primary field and $\lambda_E = \mathbf{diag}[\rho(\mathbf{m})]\mathbf{G}\tilde{\lambda}_E$. The corresponding row $\tilde{\mathbf{Q}}$ in the TM mode case

is just

$$\tilde{Q}_j = (\lambda_B \mathbf{e}_0)^{-1}[\mathbf{diag}[\mathbf{OGe}_0]\tilde{\lambda}_E]^T. \tag{42}$$

This can be viewed as a functional acting on the secondary field space $\mathcal{S}_S$, i.e., the space of cell edges. $\tilde{Q}_j$ multiplies $\partial\rho/\partial\mathbf{m}$; the product is then a data functional on the model parameter space.

In the 3D MT case the impedance is a 2×2 tensor, the computation of which requires two independent solutions computed for different source polarizations

$$\mathbf{Z} = \mathcal{E}\mathcal{B}^{-\mathbf{1}}, \tag{43}$$

where $\mathcal{E}$ and $\mathcal{B}$ represent 2×2 matrices, consisting of two components (x, y) of the electric or magnetic fields evaluated for the two solutions

$$\mathcal{E}_{ij} = L_{Ei}\mathbf{e}_j \qquad \mathcal{B}_{ij} = L_{Bi}\mathbf{e}_j \qquad i,\, j = 1, 2. \tag{44}$$

Here the index i is used to distinguish between the x and y components of the fields, and j distinguishes solutions for the two source polarizations. The easiest way to derive the specific form for the linearization (35) of impedance component Z_{ij} is to note that perturbations to the computed EM fields $\delta\mathcal{E}$ and $\delta\mathcal{B}$ result in the first order perturbation to the impedance tensor

$$\delta\mathbf{Z} = \delta\mathcal{E}\mathcal{B}^{-1} - \mathbf{Z}^{-1}\delta\mathcal{B}\mathcal{B}^{-1}. \tag{45}$$

Writing (45 out in components we find for the perturbation to Z_{ij}

$$\delta Z_{ij} = L_{Zij}(\delta\mathbf{e}_1, \delta\mathbf{e}_2) = \sum_{k=1}^{2} \left([\mathcal{B}^{-1}]_{kj}(L_{Ei} - Z_{i1}L_{B1} - Z_{i2}L_{B2}) \right) \delta\mathbf{e}_k. \tag{46}$$

The key point here is that the functionals for the impedance (both non-linear and linearized) act on the pair of solutions $\mathbf{e}_1$, $\mathbf{e}_2$, so that for the 3D MT problem this pair together must be viewed as the fundamental EM solution object, i.e., what is represented by $\mathbf{e}$ in our abstract treatment of sensitivity computations. Thus the general expression for the Jacobian (17) really looks like

$$\mathbf{J} = \begin{bmatrix} \mathbf{L}_1 & \mathbf{L}_2 \end{bmatrix} \begin{bmatrix} \mathbf{S_m} & 0 \\ 0 & \mathbf{S_m} \end{bmatrix} \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{A}_1^T & \mathbf{A}_2^T \end{bmatrix} \begin{bmatrix} \Lambda & 0 \\ 0 & \Lambda \end{bmatrix} \begin{bmatrix} \mathbf{S_m} & 0 \\ 0 & \mathbf{S_m} \end{bmatrix} \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{bmatrix}. \tag{47}$$

Solution operator ($\mathbf{S_m}$), and the EM component evaluation functionals (Λ) are the same for both modes, but the operators $\mathbf{A}_k$ and $\mathbf{P}_k$ are different. Eq. (47) shows that multiplying a model space object by $\mathbf{J}$, or a data space object by $\mathbf{J}^T$, requires two calls to the forward solver.

All of these complications are hidden in the general expression (17), provided we interpret the component operators with enough generality. This is an important constraint on a general modular system: basic objects such as EM solutions and data functionals should be constructed in such a way that extraneous details (such as the multiplicity of source polarizations required for 3D MT) can be hidden completely from the general routines which implement Jacobian and gradient calculations.

Note, however, that a more careful examination of (41) reveals that the impedance is degenerate in some sense, so that if the full Jacobian is to be calculated some efficiencies are possible. These efficiencies are potentially obscured in an abstract treatment of the sensitivity calculation, as they depend on specific features of a particular problem. This is discussed in the Appendix, maybe.

2.4 Multiple Transmitters

Finally we offer some further comments on the common *multiple transmitter* case, e.g., with data from a range of different frequencies, or different geometrical transmitter configurations. In both of these cases one has to solve separate forward problems for each transmitter. Partition $\mathbf{J}$ $\mathbf{L}$ and $\mathbf{Q}$ consistent with the data vector in (9). Then the sensitivity matrix is

$$\mathbf{J} = \begin{pmatrix} \mathbf{J}_1 \\ \vdots \\ \mathbf{J}_{N_T} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_1\mathbf{S}_{1,\mathbf{m}}^{-1}\mathbf{P}_1 + \mathbf{Q}_1 \\ \vdots \\ \mathbf{L}_{N_T}\mathbf{S}_{N_T,\mathbf{m}}^{-1}\mathbf{P}_{N_T} + \mathbf{Q}_{N_T} \end{pmatrix}. \tag{48}$$

The matrices $\mathbf{P}_l$ and $\mathbf{Q}_l$ generally depend on the solution for transmitter l, and are in general different for each transmitter. Furthermore, the differential operator for the PDE $\mathbf{S}_{l,\mathbf{m}}$ may also depend on the transmitter, as, for example, in the MT cases.

In all of our detailed discussions of sensitivity calculations we have focused on the operators for a single transmitter $\mathbf{J}_l, \mathbf{P}_l, \mathbf{Q}_l$ and $\mathbf{S}_{l,\mathbf{m}}$, while in our discussions of inversion algorithms $\mathbf{J}$ refers to the full multi-transmitter Jacobian. However, multiplication by $\mathbf{J}$ or $\mathbf{J}^T$ involves

only simple concatenations or sums over the single transmitter components, so all of our discussion of single transmitter calculations remain relevant to the full problem.

3 A Modular System for EM Inversion: Overview

We now turn to implementation or a modular system of computer codes for inversion of EM data with gradient based search algorithms, guided by the general mathematical developments in the previous section. First we outline the basic structure of the system, which is represented schematically in Figure 3, where important general features are illustrated. The specific modules shown in this figure are then discussed in more detail in section 4.

In Figure 3 red lettering is used to denote modules which define the principal public data types, i.e., those that are used extensively in the modular system for all applications. The arrows between modules define the use dependency: arrowheads point at modules which use the module that the tail is touching, or inside of. Thus modules in the upper parts of the diagram depend directly or indirectly on data objects and methods from modules in the lower parts of the figure. In fact, the routines at the top level in the figure are the most general, intended to be used for multiple problems. A key challenge is to ensure that these general purpose upper level inversion modules do not depend on specific details of how the lower level modules, which may be much more application specific, have been implemented. Further discussion of model use dependencies is thus warranted.

If an arrow originates from within the box representing a module, then at least some attributes of data objects within the scope of the originating module need to be public to the using module. If the tail of the arrow does not extend into the interior, then lower level data types or methods used are more abstract: the using module does not reference internal attributes of the data objects. The only module where a data type is defined with purely private attributes is ModelParameter. However, there is generally only one "layer" of surrounding modules that reference specific attributes of lower level data objects. Higher level routines (for which the use arrows start outside of the boxes) treat used data objects abstractly, and do not, by convention, reference internal attributes. However, except for the case of ModelParam, derived data type attributes are public, and because in Fortran usage

is inherited, the compiler cannot enforce this convention.

There are three groups of modules delineated by dashed lines in Figure 3. The uppermost layer consists of the fully generic modules intended to be used for a wide range of EM inverse problems. In particular, these "Level I" modules have been used for implementation of inversion codes for both the 2D and 3D MT examples discussed in section 2. Although details in the design may in practice have to be modified for other applications, these modules are intended to be generic. Routines in these modules make no reference to specific attributes of the abstract data objects they manipulate. Of course, this generality makes the code more complicated than would be required if only a single specific application (e.g., 2D MT) were allowed for. The actual inversion algorithms are implemented at this top generic level; two example modules that have been implemented are shown in the upper right corner of Figure 3. Other algorithms developed and tested within the modular framework for a specific application (e.g., 2D MT) should be immediately applicable to other problems as well.

Modules in the second level in Fig. 1 provide an interface between the generic top level modules, and the numerical implementation specific modules below. This group of modules is also very specific to a particular EM application. Problem specific source and receiver details are implemented at this level. In particular, any EM inverse problem (MT or active source) which could be reasonably modeled with the same discrete numerical scheme could be accommodated by appropriate changes to routines at this level–i.e., by redefining sources and data functionals, without any changes to higher or lower level modules.

Aside from model parameter objects and methods (which are defined in module modelParameter) only data types and methods defined in these Level II modules are referred to by the generic top level inversion modules. All data types referenced by the generic modules must obviously have fixed names. Similarly, functions or subroutines called from the generic level one modules must have fixed names and interfaces, including having the same data types for all arguments. All of these data types and methods must be instantiated for each specific application with new versions of the level two modules.

Although the abstract meaning of the data types are fixed, the actual structures declared can be quite different. For example, we have shown in section 2 that the fundamental EM

solution data object 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 defined in two separate instances of the level 2 module SolnRHS. Similarly, while all routines at this level have fixed names, and fixed functionality, implementation details are application specific. Coding of level 2 routines is in fact straightforward, as the same routines from a previous application can be used as a template. As we shall see when we discuss our specific applications to 2D and 3D MT, the routines and data objects defined at level two 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 apparent complication are the price of generality.

The third group of modules delineated in Figure 3 are the numerical discretization modules. These modules are specific to the fundamental numerical discretization used by the forward modeling modules. This is where, in the terminology of section 2, the numerical grid, the primary and secondary EM field data objects, and the methods needed to manipulate them are defined. The basic EM field interpolation functionals, which depend only on the numerical discretization of the PDE, are also a part of this group of modules. Modules at this level are not specific to a particular EM application; the same implementations could be used for any source/receiver configurations which could be reasonably modeled with the particular numerical approach.

The forward modeling module, the core workhorse of the inversion, is represented as a single module, but in general may have more complex structure itself. Indeed, the 2D and 3D MT cases discussed in detail below are quite different, in part reflecting the very different development histories for these two examples. The 2D forward modeling code was originally written in F77, and was the core of the REBOCC inversion of Sripunvaraporn et al. (2000). For the 3D MT case, forward modeling code was developed from scratch in F95 using a modular approach, based on the forward solver described by Sripunvaraporn et al. (2002). The 3D code thus makes extensive use of the numerical discretization modules, while the 2D code does not. In fact, the internal details of the core forward modeling routines are not so relevant to our discussion of the modular system, and will not be discussed in detail here.

However, interfaces and functionality of these components are important, and these aspects will be discussed further in the next section.

Three fundamental data objects appear in the general abstract development of EM inversion of section 2: data vectors ($\mathbf{d}$), EM solutions ($\mathbf{e}$), and model parameters ($\mathbf{m}$). In addition, we have found it useful to define a "right hand side" (RHS) object ($\mathbf{b}$) which generally defines sources and boundary conditions that determine an EM solution. Although in principal RHS and EM solution objects might be taken to be in the same space, it turns out to be useful to distinguish these objects explicitly. EM solutions, RHS objects, and model parameters are viewed as abstract data types (ADTs) with no specific attributes referenced by the generic Level I routines.

Our treatment of data vectors is rather different, as the Level I routines make specific reference to the specific structure of these objects, and all attributes of this data type are public. Indeed, the data space module (where the basic data vector types and methods are defined) is in Level I. The major rationale for this approach is to take advantage of the very common "multi-transmitter" structure of EM datasets. In a large number of cases (in particular for 2D and 3D MT, or for a large fraction of CSEM applications) 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. To simplify development of inversion algorithms that take advantage of this data vector structure, we have built this into our implementation of the basic data space objects, and we make use of this specific structure throughout the Level I modules. Further details on this multi-transmitter structure are provided in the next section.

It is also worth singling out the model parameter module (lower right in Fig. 1). The fundamental data type defined in this module is ModelParam. In contrast to other derived data types defined in other modules, all attributes of this data type are private to this module. This means that the rest of the code does not know about, and cannot make explicit use of, any details in the way the model parameter has been defined. Furthermore, since all ModelParam attributes are private, it is necessary to include within the modelParameter

module all functions and methods that need to make reference to any model parameter attributes. This includes functions which map from the model parameter to the numerical grid (i.e., $\pi(\mathbf{m})$ or the corresponding linearization or adjoint), and the model parameter covariance, along with any needed basic model parameter methods, including linear algebra and dot products required by inversion routines. The advantage of keeping all ModelParam attributes private is that we can guarantee that no other parts of the code depend explicitly on how we have defined the model parameterization. Thus, the model parameterization, defined broadly to include regularization, constraints, etc., can be modified without touching any other modules. We view modification of the model parameterization as something that should be relatively easy, and accessible to the broader community of research scientists. Making this part of the inversion system comparatively easy to modify (at least in the sense that it is decoupled from the rest of the inversion code) has thus been given high priority.

In summary, our modular system is built around a very specific structure for data space objects, with much more general structures allowed for representation of model parameters and EM solution objects. Inversion routines are generic, and their implementations can be coded in terms of ADTs, much as our discussion of EM inversion in the previous section was abstract and non-specific. Much of the rest of the system, represented by the lower level modules in Fig. 1, must be coded explicitly based on the specific intended application, and the specific forward modeling approach used. However, as we shall show with specific examples, the modular system as developed for our specific examples provides a fairly complete template for development and testing of modules suitable for a wide range of specific inversion applications.

4 Module Details

Here we discuss each of the modules in Figure 1 in greater detail, focusing in particular on implementation for the 2D and 3D MT inverse problems. We begin with two key modules which define the most basic objects required of any inversion: data vectors ($\mathbf{d}$) and model parameters ($\mathbf{m}$). Note that order of discussion of individual modules will be changed!

4.1 DataSpace

The modular inversion is built on a very specific structure for data vectors, which have public attributes available to the generic Level I inversion routines. Two key observations underlie our organization of the data vector, which we nonetheless believe is flexible enough to accommodate a wide range of EM (and more general geophysical) data sets. First, EM data is 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 complex, or 8 real components. If vertical field transfer functions are added, there are 12 real data components. Second, these multi-component data can be organized, or indexed, by three attributes which we refer to abstractly as *data type*, *transmitter*, and *receiver*. In the context of the 2D and 3D MT problems *receiver* refers to the site location, and *transmitter* refers to the frequency or period. For 2D MT, *data type* can be taken to distinguish between TE and TM mode data, although this could also reasonably be taken to be part of the full *transmitter* description. For 3D MT, *data type* could be used to distinguish between cases where vertical field transfer functions are or are not available. Looking ahead, we could define additional data types within the framework of 3D MT inversion: e.g., inter-station transfer functions. Note that in this case, a full description of the *receiver* would require locations for a pair of sites.

The key point here is that these attributes are treated abstractly at the level of the data space and inversion modules. The list of possible values for these attributes, and their explicit meaning, will depend on the specific application. In general the *transmitter* attribute should define, possibly in conjunction with *data type*, the forward problem that must be solved for these data. The *receiver* attribute is used to define (again, possibly in conjunction with *data type*) the measurement process that must be applied to the forward solution to allow comparison between model predictions and observations. To allow abstract treatment of these data description attributes in Level I modules, we use dictionaries to carry the actual descriptive meta-data (site location or locations, mode, frequency, etc.) Data vector objects refer to specific entries in these dictionaries through three integer indices. The dictionaries

and integer indices allow for a flexible and general treatment of the meta-data needed to fully characterize complex multi-component EM data sets, while still allowing a fixed structure for data vector objects, and the generic inversion routines which must manipulate these objects.

Note that to maintain generality all data are taken to be real numbers–complex data (e.g., impedances) are represented as separate observations, for real and imaginary parts. For while complex data are common in frequency domain EM data, 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 or phase by themselves (the pair together are virtually real and imaginary parts of complex data, but both might not be available or useful for inversion). However, while generality demands a representation of data in terms of real components, it is still worth keeping track of the cases where pairs of components do in fact correspond to a single complex number. This is because certain efficiencies are possible in sensitivity calculations when complex arithmetic is used. In particular, the expression (18) for the transposed Jacobian yields a complex matrix. The real (imaginary) part of this complex Jacobian is the derivative of the real (imaginary) part of the data with respect to the model parameters. Thus, sensitivities for real and imaginary parts of the data can be computed simultaneously, at the cost of a single adjoint solution. To allow for this, we need to distinguish between when data are actually complex, or intrinsically real. This is done through the *data type* dictionary. An essential attribute of every data type is whether the data are complex, with pairs of real components to be treated as a single complex number, or real.

The basic building block for data space objects is the derived data type **dvec**, which contains multi-component data of a single *data type* for one *transmitter*, but for multiple *receivers*. The corresponding receivers are kept track of through a list of indices into the receiver dictionary. Because all of these data correspond to the same transmitter, predicted data for all components in a **dvec** object can be computed from the same forward solution. Furthermore, data for all receivers in one **dvec** are of the same data type, and hence consist of the same number of components, in the same order. For 3D MT a **dvec** object would generally contain all impedance tensor components for all sites, at a fixed frequency. In the

2D MT case, a typical **dvec** object would contain TE or TM mode complex impedance data for all sites at one frequency. The full data vector (**d**) is essentially an array of **dvec** objects, which is stored in a data type named **dvecMTX** (dvec, multiple transmitter). These objects may mix objects of different data types with different numbers of components or sites. Thus in the 2D MT case some elements in the dvecMTX array can be for TE, and some TM. More generally, a completely different data type such as DC resistivity (or even some sort of seismic or potential field data) could be included in the array. The structure of the complete data vector, with different data types/transmitters grouped together in separate structures is intended to simplify coding of inverse algorithms which require solution of multiple forward problems (possibly of very different sorts) for data subsets, as is common in EM inversion. Note that except for the highest level modules (SensMatrix and the inversion modules) only single transmitter data objects are used.

In addition to defining the basic derived data types the module contains basic routines for creating, zeroing, copying, reading, writing, and deallocating data space objects. Routines are also provided for basic algebraic operations including 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. Similar basic methods for creation, deallocation and I/O, as well as for basic arithmetic and vector space operations, are part of all of the modules where the basic data objects are defined. Effectively two sets of routines are supported: for **dvec** and **dvecMTX** objects. The latter are simply constructed from the former in the obvious way.

Need some discussion of error bar implementation????

4.2 ModelParameter

Model parameters (**m**) sit at the opposite end of the inversion from the data **d**, outputs opposed to inputs. Although both sorts of objects are manipulated directly by Level I inversion routines, our implementation of model parameter objects in the modular system is also quite opposite to that of data vectors. In contrast to the data space, where the basic objects have a fixed specific form with public attributes referenced by high level inversion routines, we treat model parameters as a purely abstract data type, with all attributes private to the defining

module, There is no reference to the specific structure of the model parameter representation by routines in other modules, which can manipulate these objects only indirectly by calling routines within the ModelParameter module. Because external reference to internal attributes of the ModelParam data type are not allowed, we can guarantee that the rest of the inversion system is independent of any specific details of our implementation of this module. Thus the system should work with any model parameter module which provides definition of a ModelParam data type, together with the methods necessary to manipulate these data objects in the inversion modules. The underlying rationale for this approach is that definition of the model parameterization, which in a broad sense may embed prior assumptions, constraints and regularization, is a critical scientific input to geophysical inversion, and should thus be reasonably accessible to as broad a class of users as possible. Our approach minimizes interaction of the model parameterization with the rest of the modular system, and in this way simplifies and streamlines modification of this critical part of the inversion.

The price of keeping all data type attributes private is that all procedures which actually modify or access internal model parameter attributes must be merged into a single module, and all of the functionality provided by these routines must be provided by every specific instance of this module. Furthermore, for any instantiation, all routines must have the same names and same interfaces, as well as the same abstract functionality. Our initial model parameter implementations for both 2D and 3D MT problems are very simple, and classic: as discussed in section 2, conductivity or log conductivity is defined independently on each of the cells in the numerical grid. For the 3D case we have also implemented (as an option within the same basic module) parameterization 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.

There are essentially three subgroups of routines in the ModelParameter module. The first group consists of the usual routines for creation, deallocation, I/O and copying, plus routines for linear algebra and dot products. These are largely self explanatory, and follow the usual naming and interface conventions. Note that the vector space routines are heavily used by

the actual inversion algorithms. The second group consists of mappings between the model parameter and the primary or secondary fields defined in module EMfield. These mappings including $\pi(\mathbf{m})$, the linearization $\partial\pi/\partial\mathbf{m}$, and the adjoint $[\partial\pi/\partial\mathbf{m}]^T$. Note that $\partial\pi/\partial\mathbf{m}$ is a matrix which represents a linear operator mapping from model parameter objects to EMfield objects. Specific forms for these matrices are given for the 3D and 2D MT examples discussed in section 2 by the second bracketed terms in (31) and (32). An example of the adjoint, which is a linear mapping from EMfield objects to model parameter vectors, is given by the first bracketed expression in (33). These linear mappings (not construction of the matrix which represents them) are implemented in this module, which must thus use the EMfield module. The mapping routines are in turn used by FWD, InterpEB, and Pmult modules to set up coefficients of the differential operators or interpolation functionals, and to implement multiplication by the matrices $\mathbf{P}$ and $\mathbf{Q}$. We have found it useful for the InterpEB model to implement also a functional form of the primary mapping, which returns an individual component $pi_k(\mathbf{m})$ of the full EMfield parameter mapping.

The final group of routines in the modelParameter module implement the model covariance, which in principal may also be used to impose *a priori* constraints. In our example applications we have used simple smoothing covariances, implemented as in Siripunvaraporn and Egbert (2000) in the 2D case, and Siripunvaraporn et al. (2004) in the 3D case. Essentially two routines are needed: one to initialize the covariance parameters (which are stored in a data structure that is private to this module) and one to apply the symmetric covariance operators $\mathbf{C_m}^{1/2}$ and $\mathbf{C_m}$ to a model parameter $\mathbf{m}$, to compute the product.

4.3 Grid

The grid module forms the basis for the specific numerical implementation of the forward problem, defining the data structure used to store the grid geometry. In our 2D and 3D examples, we use finite difference modeling approach with a comparatively simple rectangular grid, which can be completely characterized by the grid dimensions, and the spacing of grid nodes in 2 or 3 directions. A much more complicated data structure would be required to specify a non-structured grid (e.g., for finite element modeling), but a similar approach would be used, with a single data structure encapsulating a complete description of the grid.

In our applications with finite difference grids We have found it useful to precompute and store various derived grid quantities, such as the coordinates of nodes and cell centers. The essential requirement, from the perspective of the modular system, is that there be a single data structure (with public attributes) that provides a complete description of the grid. A pointer to this structure is included in most of the basic data objects (e.g., model parameters, EM solutions). Key public routines in this module are to create and deallocate the grid data type. Note that the basic grid module, together with the core EM Field, forward modeling, and interpolation modules can in principal be applied to a variety of EM inverse problems, with suitable changes to other modules which define source terms, boundary conditions, and measurement functionals.

4.4 EMfield

The EM field module defines the basic data types used to represent objects in the space of primary and secondary EM fields $\mathcal{S}_P$ and $\mathcal{S}_S$. Objects defined in this implementation specific module are generally not directly referred to by the generic Level I inversion modules, which instead manipulate the more abstract encapsulated EM solution objects defined in module EMsoln. EMfield data types have public attributes which are explicitly referred to by the modules for forward modeling or interpolation that use them.

In the 3D MT case the basic EM field object is a discrete representation of a three component complex vector field on a staggered grid. These are stored as three separate three dimensional arrays, encapsulated in a single derived data type (**cvec**), together with array sizes and a pointer to the underlying basic grid object which provides geometric Context. As discussed in section 2, for our modeling approach electric field components are defined on grid cell edges, and with magnetic field components are defined as normal to the cell faces. Both sorts of staggered grid vector field are supported by the **cvec** data type. Several variants on this basic staggered grid EM vector field representation are supported. First, it has proven 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 potentials; a real vector can be used to represent electrical conductivity defined on cell edges–i.e., $\sigma(\mathbf{m})$. Second, we support

a sparse vector representation, to save storage space when only a small number of components are non-zero. This arises for representing measurement functionals, and also could be used to represent a localized source. For all of the variants of the basic vector fields there are routines for creation, deallocation, linear algebra, and vector space operations, including for combinations Of types (e.g., dot products of sparse and full vectors). Finally note that we also support a specialized data structure to store data for boundary conditions–i.e., the tangential components of the electric field on the boundaries.

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. As for the 3D case we do allow for sparse as well as full array representations of the EM fields, with the usual basic and combined methods routines. In the 3D case we divided the different basic EM field data types into several distinct modules. The 2D case is much simpler, and all routines are in a single EM Field module.

4.5 ForwardSolver

The forward solver is the core of the inversion routine. Although not the focus of our discussion in this paper, there are some essential issues concerning the forward solver which deserve mention.

The solver core is quite different for the the two cases we consider as examples. For the 3D MT case, we have written code from scratch, making extensive use of the EMfield and Grid modules. The 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 (20), and these equations are 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, the QMR solver, and the ILU-1 preconditioner can be easily replaced by alternative iterative solvers, or pre-conditioners, respectively.

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 conditions. For the usual MT forward problem there no source terms, except on the boundaries, but to compute sensitivities one must be able to solve the forward problem for arbitrary sources also. Second, capability to solve the transposed system (as in (18) is required. As discussed in section 2, 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. (2007). Finally, the solver should have a clean interface. For example, our forward solver driver module has public routines which can be called to initialize the solver and modify PDE coefficients (which depend on the model parameter, and the frequency), and to actually solve the forward or transposed equations for a particular set of sources and boundary conditions. Additional public routines are provided to set solver control parameters (convergence criteria, schedules for divergence corrections, etc.) or to retrieve solver diagnostics.

In contrast to the 3D MT solver, which we coded to meet the specific requirements of the modular system, 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 Fortan77, which did not allow dynamic allocation of memory–array sizes for a particular problem had to be set at compile time. We wrapped this code with Fortran95 routines which provide a cleaner, more clearly defined interface, and allow dynamic memory allocation. Routines in separate TE and TM mode modules are called by higher level inversion routines to initialize and solve the 2D equations. The rest of the modular system makes no reference to the original REBOCC subroutines, though these still do all of the actual numerical computations. The TE and TM mode interface modules contain, as private data, all coefficient and temporary work arrays 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.

Subroutines that were originally part of the REBOCC 2D MT inversion code were placed

in a single module. In the original code integer parameters to set array sizes were set in an include file, and array sizes were fixed at compile time. In the wrapped modular version all arrays are dynamically allocated at run time. To minimize changes to the code, the same array size parameter names were used, but these are now static module variables, set by an initialization routine before the first use of any routines in this module. A further aspect of the wrapping is that the arrays that were originally declared in driver programs for REBOCC are private variables in the wrapper modules, with the save attribute. Arrays are allocated with the appropriate size by TE and TM mode interface module setup routines. These arrays (still with the names used in REBOCC) are then passed to the old REBOCC subroutines where the actual computation is done. With minimal effort we have thus created a 2D forward modeling core with general capabilities (including solution of the transposed systems), and with a clean interface. Initialization and calling of the modeling routines is functionally similar to that developed for the 3D case.

4.6 InterpEB

This module provides interpolation functionals to evaluate both electric and magnetic fields at an arbitrary point within the model domain. Interpolation functionals provide the basic building blocks for more complex data functionals, as well as the linearizations which are required by the inversion. Any conceivable EM data, such as an impedance tensor, inter-station magnetic transfer function, or the electric potential difference between two points, is a function of the electric and magnetic fields at one or more points in the model domain, and hence can be written in terms of these basic EM interpolation functionals. Routines in this module create sparse vector representations of these basic EM field interpolation functionals. These sparse vectors are then used by routines in module DataFunc to actually apply the measurement operators, for example to compute a predicted impedance tensor component. The sparse vectors for linearized measurement operators are also used in the sensitivity calculation to force the transposed system of (18). Note that the sparse vector representations of the interpolation functionals depend critically on details of the numerical grid, while the application of the measurement operator (e.g., to compute a predicted impedance) does not. Thus, if a different numerical discretization were applied, such as finite elements with a non-

structured grid, the interpolation routines would have to be modified, but not the higher level data functional routines. Conversely, to add new data types, there is no need to revisit the interpolation aspects of the problem.

In the 2D MT case separate routines are provided for the TE (electric field) and for the TM (magnetic field) solutions. In both cases evaluation of the other field component (i.e., magnetic for TE mode, electric for TM) are supported. In addition to the fundamental EMfield and Grid modules, this module requires access to the ModelParameter module for TM mode computations. This is because calculation of electric fields from the TM magnetic field solution requires knowledge of the resistivity in the vicinity of the evaluation point. This module uses only the abstract data type ModelParam (passed as an argument to some routines in this module), and a function rhoC (defined in the modelParameter module) which computes resistivity for any cell in the 2D grid. NOTE: this is how dependence of evaluation functionals on model parameters is handled now, but this will almost certainly be modified, as it is not very generalizable. At present basic evaluation routines return sparse vectors needed to compute rows of the matrix **Q**. Again details of this implementation will be revised in future.

For the 3D MT case there only two evaluation routines, one for direct interpolation of electric field to an arbitrary point, and the other for evaluation of magnetic fields from the electric field solution vector. For improved accuracy the electric field interpolation may also depend on nearby conductivity, in which case the same considerations as discussed above in conjunction with the TM mode apply. Details on this *might* be discussed in an Appendix!

4.7 SolnRHS

The basic objects used to represent a generic EM solution for some set of forcing and boundary conditions were defined for specific applications in module EMfield. This module essentially provides wrappers for these more primitive objects in order to hide application specific data type names and attributes from the generic Level I inversion modules. In particular, the full EM solution and right hand side forcing data types **EMsoln** and **EMrhs**, which correspond to **e** and **b** in section 2, are defined in this module. In addition to **EMsoln** and **EMrhs** data types it is also useful to define a data type **EMsparse**, a sparse vector representation of the

EMsoln data type. All of the basic methods, including creation, destruction, I/O copying, and basic linear algebra operations for these data types are defined in this module. As with the basic data type definitions, these methods are built out of the corresponding methods for the lower level objects defined in module EMfield. These routines effectively provide an interface that hides application specific details of a specific instance of EMfield from generic Level I routines.

For the 3D MT case an **EMsoln** object 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 for evaluation of the corresponding data functionals. Thus $\mathbf{e}$ in the abstract description of the inverse problem given in section 2 actually refers to the pair of solutions. Similarly $\mathbf{b}$, the abstract representation of the RHS for these solutions, refers to the boundary data for a pair of independent source polarizations. 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. Similarly, all of the expressions for sensitivities or gradients of the penalty functional must involve both source polarizations in a consistent fashion.

For 2D MT there might appear to be little reason to distinguish between the SolnRHS objects, and the corresponding lower level EMfield objects. In this case **EMsoln** (i.e., $\mathbf{e}$) is just a single complex scalar field, with an essentially identical data structure to that already defined in the lower level EMfield module discussed above. One might wonder if for at least the 2D MT problem it might be possible to dispense with the two distinct levels of EM solution type objects and methods. For several reasons this is probably not a good idea. Even in the 2D case the **EMsoln** object is intrinsically more complex than what is desirable at the lowest levels of the modular system. For example, it turns out to be useful to include pointers to the grid and model parameter, as well as an index into the transmitter dictionary, and any other 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 **EMsoln** data object greatly simplifies subroutine interfaces in the Level I routines, and hides many bookkeeping

details from the inversion algorithm.

At the same time it is useful to have basic objects which can define scalar (2D) or vector (3D) fields on the grid defined at a low level in the module hierarchy. These basic objects are in fact useful for many purposes, and in most of these cases the extra information outlined above is not even meaningful. Furthermore, there are issues with regard to module dependencies which must be considered. For example, the modelParameter module must generally have access to the basic objects used to represent EM fields on the grids. In particular, the mappings from parameter space to grid require access to such objects. Including a pointer to a ModelParam object in the basic EMfield objects, and then using the EMfield module in the modelParameter module is impossible, due to the circularity of module definitions this causes. It thus appears to be simplest to maintain two levels of definition of EM field solution (and related) objects. A lower level one, of relatively simple form, developed explicitly for the specific application, and this higher level SolnRHS module which wraps the application specific EMfield objects and methods into a more generic form in which all data types have fixed names, and all methods have fixed interfaces.

4.8 DataFunc

This module defines and implements the data functionals needed for a specific application. There are two key public procedures in this module with names and interfaces given in Table 2¿ The first, **nonLinDataFunc**, evaluates the non-linear data functional, e.g. computing the complex impedance elements from the computed EM solutions in the 2D and 3D MT inverse problems. The second, **linDataFunc**, creates linearized data functional, the rows of **L**, and if appropriate **Q** associated with an individual transmitter-receiver pair. All components of multivariate data (e.g., all impedance tensor elements) are handled together. The linearized data functional are returned as sparse vectors, which can then be used to evaluate the linearized data functional or to force the adjoint system, e.g., for computation of the penalty functional gradient as in (8).

The module also contains the *dataType*, *transmitter* and *receiver* dictionaries, as well as procedures for dictionary initialization. The receiver dictionary is only used within this module and can be private. Each multivariate observation associated with a specific receiver

is identified by the receiver index, an attribute of [**dvec** objects. These are used to look up any needed meta-data associated with this receiver. In the context of 2D and 3D MT, the meta-data is essentially just the site coordinates, 2 or 3 real numbers. Other types of data may require additional parameters to define the specific observation. For example, for inversion of network MT data, with very long dipoles (e.g., Siripunvaraporn et al, 200x) one would need to define both ends of a dipole. Similarly, inter-station horizontal magnetic field transfer functions would require two site locations, one for the local and one for the normal reference site. It would be possible to have more than one receiver dictionary within the DataFunc module. In this case the *dataType* dictionary would have to include information about which receiver dictionary to use for elements of a particular **dvec** object. E.g., if one wanted to mix MT and network MT in a single inversion, one approach would be to have a dictionary for the MT site locations, and a separate one for the long dipoles. Neither the specific nature of the meta-data, nor how it is organized within this module, is referenced by Level I inversion modules.

In general the dataType dictionary will be used to control DataFunc calculations. For example, in the 2D MT case we encode mode (TE or TM) in the dataType dictionary. Case statements within module routines are used to select calls to appropriate evaluation functionals, which are different for the two modes. Other data type distinctions (e.g., apparent resistivity and phase vs. impedance) would require additional case statements. Indeed, adding new types of data can be accomplished with modifications to this module alone: the new data types must be encoded in the type dictionary, and code must be added for the new cases.

When data functionals depend on model parameters, there are additional terms in the linearizations of the data functionals (i.e., the corresponding rows of matrix **Q** are non=zero). The data func routines produce sparse vectors to represent these components. At present this has only been implemented for the 2D MT case, and the sparse vectors are the full row of **Q**–i.e., each represents a functional on the model parameter space. With our choice of model parameters as numerical grid cell conductivities, these functionals can be represented as sparse vector objects, though this will not be true for all model parameterizations. The

present implementation is inelegant as well.

The proper approach is not clear at present. From a purely theoretical perspective, rows of QQ are model parameters. A concern (perhaps not well founded) is that the simplest approach of taking these to be model parameters is wasteful of memory, since they usually will be sparse vectors. It would be possible to allow for sparse representations for model parameters, but this might not always be sensible or at least worth the trouble. In our approach, where routines outside the ModelParam module can't know if this sparse representation is or isn't implemented, implementation would represent some challenges. Note also that according to (40) we can divide QQ into three terms as in (40). The middle term should be implemented in module InterpEB, and this can be returned as a sparse EMfield object. Then, integration with the other two terms to create a model Parameter object would be completed in this module.

4.9 EMsolver

EMsolver is the Level II module which provides a uniform interface between to the application specific forward modeling routines, and the generic Level I inversion routines. The key public routines in this module must have names and interfaces consistent with the conventions of Table 2. However internal functioning of these routines will be application dependent, as our two examples nicely illustrate. In the 3D MT case an EMsoln object (the output of fwdSolve and genSolve) consists of solutions for two source polarizations. These require two separate calls to the lower level (module FWD) solver routines. In the 2D case, only a single solution is required, but now there are two possibilities: either a TE or a TM equation solver must be called, depending on the dataType of the input dvec object. 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 routine names, interfaces, and abstract functionality are identical.

The key public routines in this module include initialization, deallocation and clean up, and two separate solver routines, fwdSolve and genSolve. The second of these implements the general solver, allowing arbitrary forcing and boundary conditions, and solutions for both the usual forward problem and its transpose, or adjoint. The first solves only the

forward problem, and does not require (or allow) explicit specification of the forcing, which is computed internally with reference to the transmitter dictionary. Clearly the first routine can be coded to call the second, after RHS initialization.

Initialization 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 back-substitution once factorization is complete. The initialization routine is thus coded to keep track of the previous solver call, and only reinitialize 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 (if any). The responsibility for these efficiencies lies with the developer of the initialization routine. Higher level routines that require solutions of the governing PDE always call the initialization routine first. The cleanup routine exitSover 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.

4.10 Pmult

This module implements multiplication by the matrices $\mathbf{P}$ and $\mathbf{P}^T$, for a single transmitter. There are two essential routines (Table 2), whose functions should be self explanatory. Note that the general expression (30) for $\mathbf{P}$ can be divided into two pieces: the first involves applying sparse operators to the background solution $\mathbf{e}_0$, and the second is the linearized parameter mapping $\partial\pi/\partial\mathbf{m}$. Multiplication by $\mathbf{P}$ thus involves first calling the appropriate modelParameter routine, to produce the appropriate (lower level) primary or secondary EMfield object. Then the appropriate (problem specific) operators are applied (yielding a modified EMfield object or objects), and the output Level II EMsoln object is constructed from these. In the 3D MT example, the result of the first step is conductivity averaged onto edges. The second step multiplies this (point-wise) by $i\omega\mu\mathbf{e}_{0,k}$, where $\mathbf{e}_{0,k}, k = 1, 2$ are the "background" EM solutions for the two polarizations. The result is a pair of EMfield edge objects, which together constitute the output EMsoln object. Multiplication by $\mathbf{P}^T$ is similar. Now the input is an EMsoln object consisting of two separate EMfield objects $(\mathbf{e}_1, \mathbf{e}_2)$,

and the first step is to form the EMfield 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 to apply the adjoint model parameter mapping $[\partial\sigma/\partial\mathbf{m}]^T$. The real and imaginary parts of the output are ModelParam objects. When operations such as $\mathbf{J}^T$ are applied in the context of, for example, a NLCG inversion scheme, multiplication by $\mathbf{P}^T$ is the final step in the computation. In this case 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.

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 EMsoln $\mathbf{e}_0$, an attribute of these objects in the MT 2D case. The TE case is essentially like the 3D MT case discussed above, except of course the EMsoln involves only a single polarization. The TM case is somewhat more complicated. As(32) 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.

It does not appear that multiplication by $\mathbf{Q}$ and $\mathbf{Q}^T$ should really be part of this module. See discussion in connection with module DataFunc.

4.11 MeasComb

This module merges the single receiver data functionals from module DataFunc for a single transmitter. There are three essential public routines (Table 3). The first, *dataMeas* computes $d = \psi(\mathbf{e}, \mathbf{m})$. The second, *linDataMeas*, is the linearized counterpart, returning the perturbation in the data for small perturbation in $\mathbf{m}$ and $\mathbf{e}$, i.e., in the notation of (13) $\delta\mathbf{d} = \mathbf{L}\delta\mathbf{e} + \mathbf{Q}\delta\mathbf{m}$. The third, *linDataComb*, implements the adjoint of the second, returning $\mathbf{b} = \mathbf{L}^T\mathbf{d}$ and, optionally, $\mathbf{Q}^T\mathbf{d}$. In all cases these computations are for a single dvec object, and computations involving $\mathbf{Q}$ are only invoked if the corresponding data type has value true for the required *calcQ* attribute.

Routines in this module also manage issues related to the *isComplex*, pairing real and imaginary parts in the real data vectors as appropriate. For the forward calculations (*dataMeas* and *linDataMeas*) 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 dvec object. In the adjoint case, real and imaginary parts are combined into complex coefficients to construct the proper complex forcing $\mathbf{L}^T\mathbf{d}$ for the adjoint equation. Note also that in the linearized and adjoint cases dataFunc module routines only set up sparse vector representations of the data functionals (rows of $\mathbf{L}$). Actually forming the dot products between these sparse vectors and the EMsoln objects–i.e., computing $\mathbf{Le}$–is accomplished in *linDataMeas*. Similarly, adding up the component sparse vectors in the computation $\mathbf{L}^T\mathbf{d}$ is accomplished in *linDataComb*.

4.12 SensMatrix

This module puts all of the pieces ($\mathbf{L}$, $SS_{\mathbf{m}}^{-1}$, $\mathbf{P}$, $\mathbf{Q}$) together to implement Jacobian calculations. In contrast to all modules discussed so far, this module deals with multiple transmitter objects. The key public routines are summarized in Table 3. Jmult multiplies a model space object by $\mathbf{J}$, to produce an array of dvec objects, one for each transmitter. JmultT, implements the transpose of this operation, multiplying a dvecMTX object by $\mathbf{J}^T$ to produce a model parameter object. There is also a routine that implements the full forward problem, looping over transmitters to solve all component problems and returning the predicted data in a dvecMTX object for a given model parameter ($\mathbf{d} = \mathbf{f}(\mathbf{m})$). Finally, there is a routine which will compute the full sensitivity matrix $\mathbf{J}$, again looping over all transmitters. Note that the sensitivity matrix is returned as an array of model parameter objects, each corresponding to the full sensitivity for a single real data point. The computations are done using the adjoint form (18), taking advantage of redundancy associated with complex data where possible. Note that for both Jmult and calcJ a dvecMTX object must be input to provide a template for the output data vector. This template is used to provide the meta-data needed to define data functionals.

For parallelization over frequencies (probably a reasonable strategy for EM inversion with modest numbers of processors) it appears that it will be useful to have single transmitter versions of these four routines. These would have as input a **dvec** object instead of a **dvecMTX** object, but would otherwise be the same.

Finally, we have implemented some other variants (e.g., JmultMTX) which take an array

Table 1: Data Types used by Level I Routines.

Module	Type Name	Usage
ModelParameter	modelParam_t	**m**: model parameter
DataSpace	dvec_t	**d**:data vector for one type/transmitter
	dvecMTX_t	**dMtx**:multi-transmitter data vector, array of dvec_t objects
SolnRHS	EMsoln_t	**e**:EM solution, for one transmitter
	EMsolnMTX_t	**eMtx**:multi-transmitter EM solution, array of EMsoln_t objects
	EMrhs_t	**b**:right hand side forcing and boundary conditions
	EMsparse_t	λ:sparse EM solution object

Table 2: Level II Public Routines. Conventions used in calling arguments: iDt = data type; iRx = receiver; iTx = transmitter; others are consistent with Table 1.

Module	Routine Name	Inputs	Outputs	Optional
DataFunc	nonLinDataFunc	**e**,iDt,iRx	**Z**	various
	LinDataFunc	$\mathbf{e}_0$,iDT,iRX	λ	$\mathbf{m}_q$
EMsolver	initSolver	iDt,iTx,**m**	$\mathbf{e}_0$	**e**,**b**
	exitSolver		$\mathbf{e}_0$	**e**,**b**
	fwdSolve	iTx,iDt	**e**	
	genSolve	iTx,iDt,ForA,**b**	**e**	
Pmult	Pmult	$\mathbf{e}_0$,**m**	**e**	
	PmultT	$\mathbf{e}_0$,**e**	**m**	$\Im\mathbf{m}$

of model parameters (one for each transmitter) as inputs. The intended use of these is beyond the scope of this document ... and they are in any event probably made obsolete by the single transmitter versions needed for parallelization.

4.13 Inversion Modules

So far there is a single inversion module that has been minimally debugged. This implements a non-linear conjugate gradients scheme, and I am not prepared to say anything about this, as I didn't do this part, and have not had time to even look too closely at it.

Table 3: Level I Public Routines. Conventions used in calling arguments are as in Table 2.

Module	Routine Name	Inputs	Outputs	Optional
MeasComb	dataMeas	**e**	**d**	
	linDataMeas	$\mathbf{e}_0$,**e**	**d**	**m**
	linDataComb	$\mathbf{e}_0$,**d**	**b**	$\mathbf{m}_q$
SensMatrix	fwdPred	**m**,**dMtx**	**dMtx**	**eMtx**
	calcJ	**dMtx**, **m**	**J**	
	Jmult	**m**, $\mathbf{m}_0$,	**dMtx**	**eMtx**
	JmultT	$\mathbf{m}_0$, **dMtx**	**m**	**eMtx**

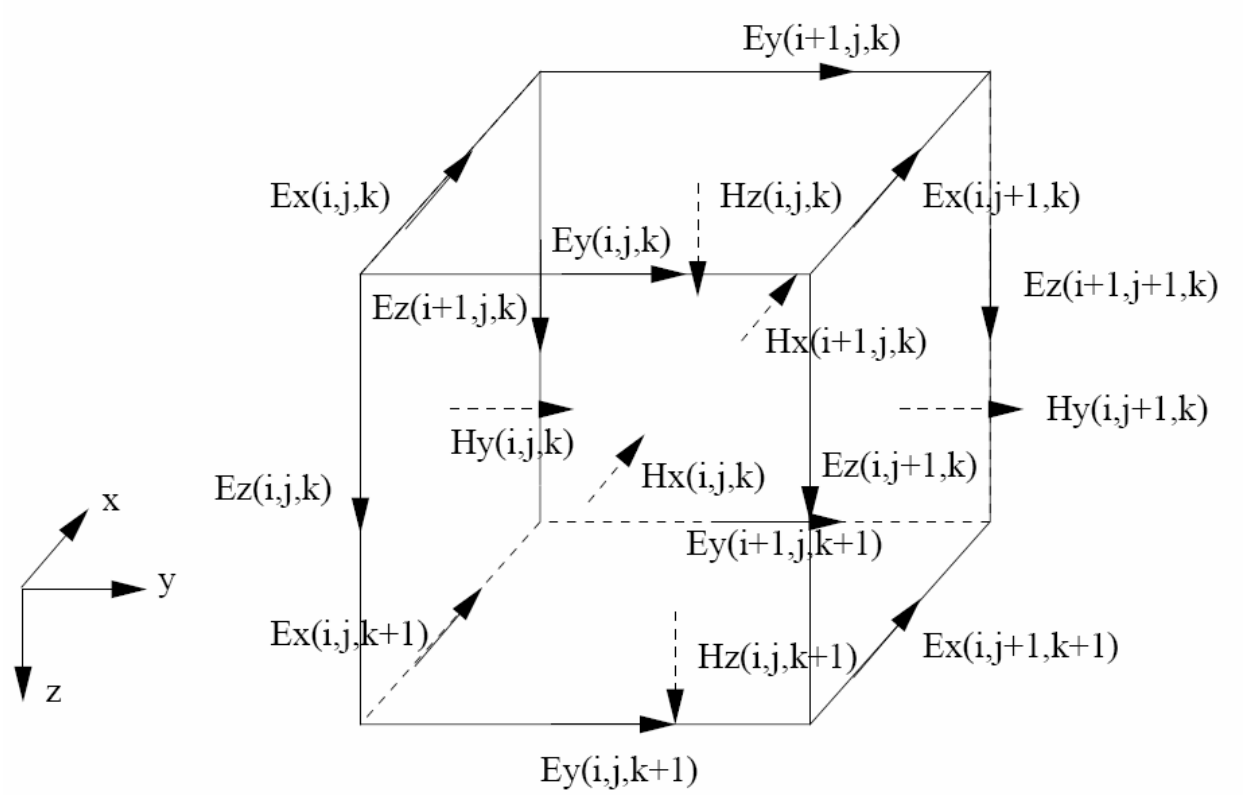

Figure 1: Staggered finite difference grid for the 3D MT forward problem. Electric field components defined on cell edges are the primary EM field component, which the PDE is formulated in terms of. The magnetic field components can be defined naturally on the cell faces; these are the secondary EM field in this numerical formulation.

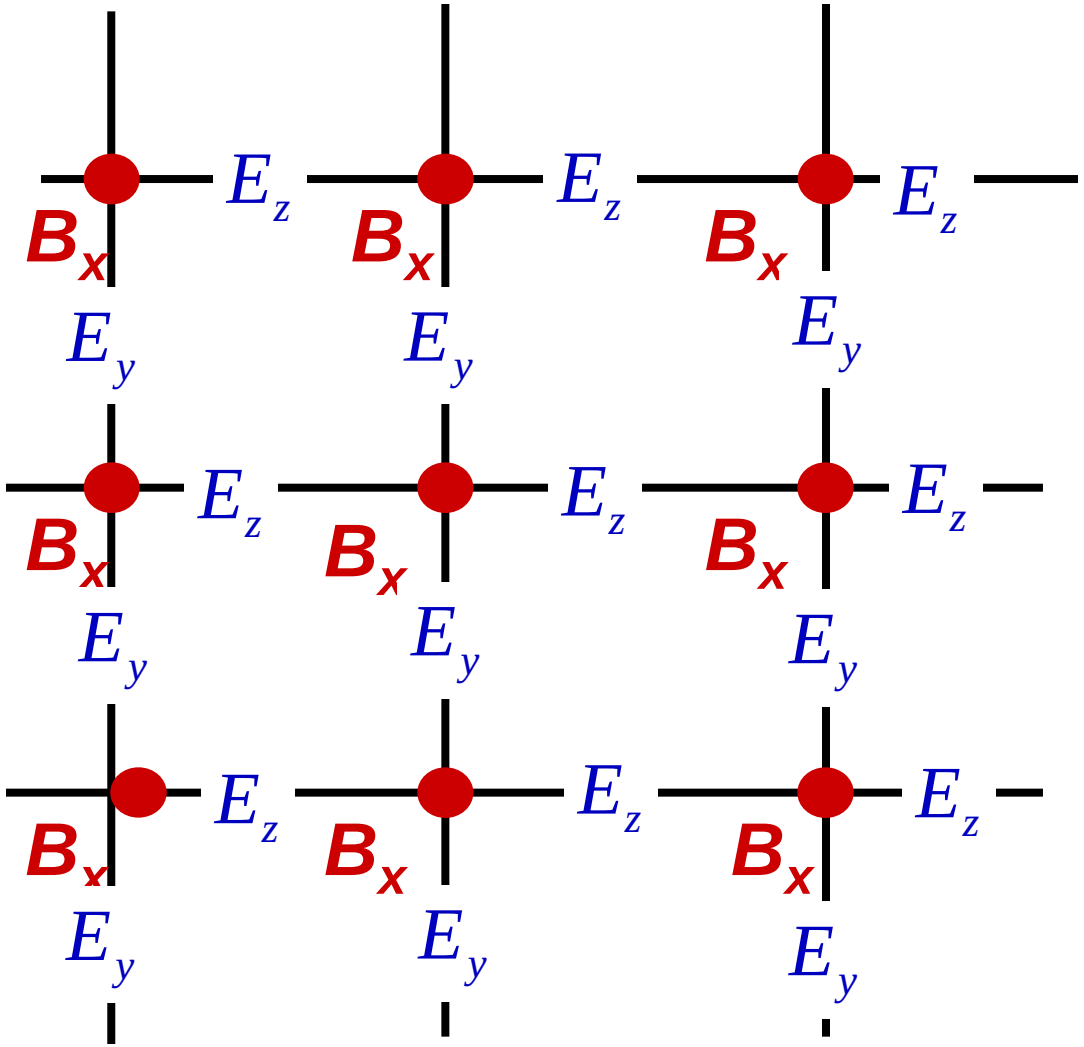

Figure 2: Finite difference grid for the 2D MT TM mode forward problem. The scalar B_x magnetic field, defined on 2D cell corners is the primary field. The secondary field components are E_y and E_z, defined on vertical and horizontal cell edges, respectively.

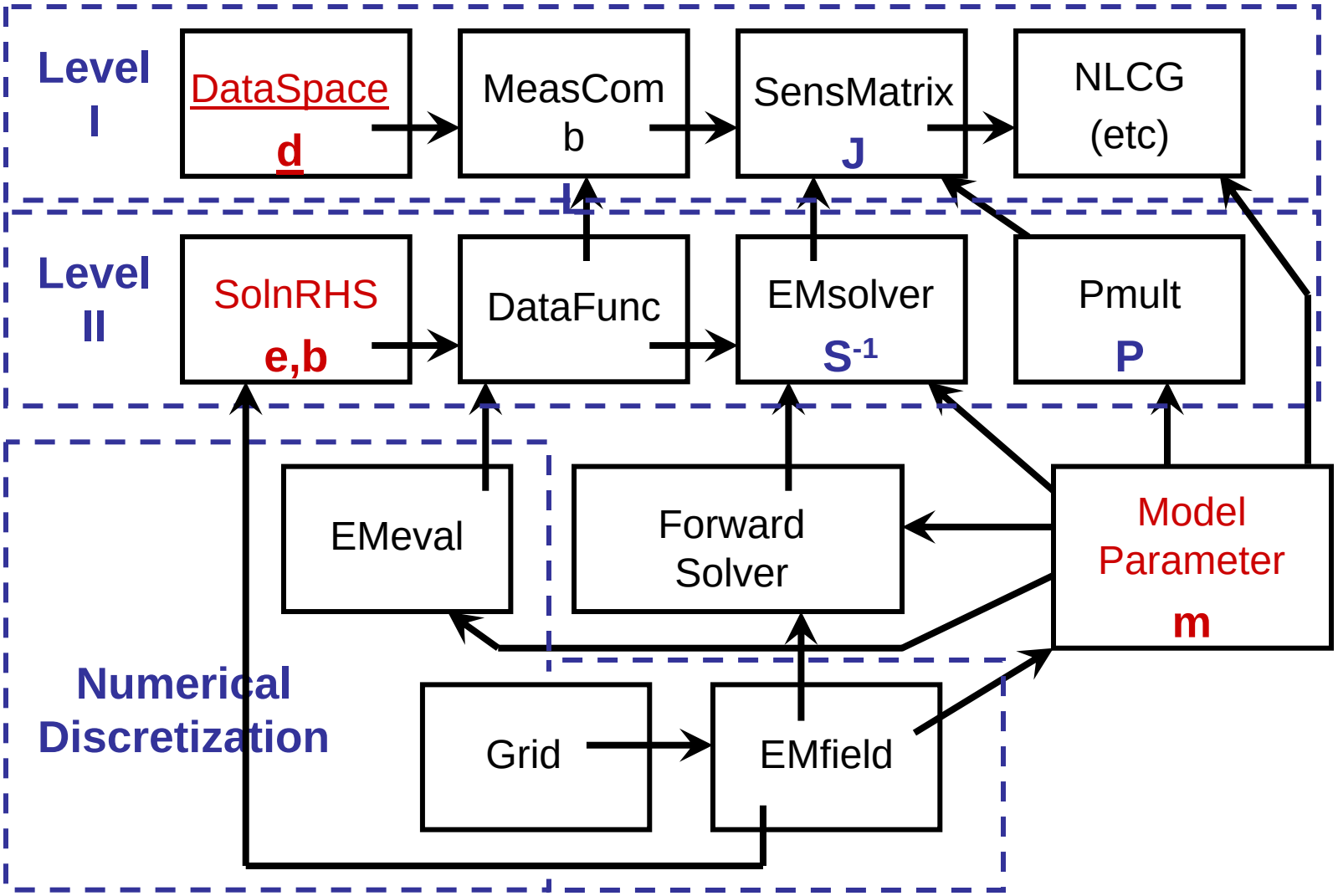

Figure 3: Overview of modular inversion system, including module usage dependency (arrows). Dashed lines delineate three major groups: (a) Level I, generic inversion routines; (b) Level 2, application specific interface to (c) numerical discretization module, which are applicable to multiple EM problems with similar numerical treatment.