A Software Framework for Inversion

William W. Symes

Geophysical Inversion Workshop, Calgary, Aug 06

Inversion: more popular all the time...

Such a popular subject deserves a good software framework...[Thanks: Deschutes Brewing Co. of Bend, OR]

The Agenda

- Optimization formulations of inverse problems combine several natural levels of abstraction.
- It is possible, and advantageous, to design code packages that address each of these abstraction levels separately, and combine these packages to create applications. Possible advantages:
  - reusable code (reuse = no change, not so much as a character!)
  - code organization mirrors mathematics
- Examples: (1) NMO-based velocity inversion, (2) least squares data fitting based on finite difference simulation of the wave equation.

Canonical Optimization Formulation

$$\min_{m \in M} J[m, d]$$

where

- m = model $\in M$ = admissible set of models
- d = data $\in D$ = data “space”
- $J[m, d]$ = objective measuring ”goodness”, eg. data fit error

... but this is just a (family of) optimization problem(s).

Role of Simulation

Simulator or *Forward Map* $F : M \to E$ simulates physical responses to various models, E = "response space".

$$J[m, d] = G(F[m], d)$$

for some function $G : E \times D \to \mathbf{R}$

Granddaddy example: $E = D$ is a Hilbert space, $G[d_1, d_2] = \frac{1}{2}\|d_1 - d_2\|^2$ - least squares formulation of inverse problem.

Inverse problem as *simulation-driven optimization*.

Solving it

Standing assumption: $M \subset H$ = Hilbert space, $J : M \times D \to \mathbf{R}$ differentiable.

Steepest descent iteration: choose $m_0 \in M$ somehow, then until "done"

$$m_{k+1} = m_k - \alpha_k \nabla_m J[m, d]$$

with α_k an approximation to the solution of

$$\min_{0 \leq \alpha \leq \alpha_k^*} J[m_k - \alpha \nabla_m J[m, d], d]$$

where $\alpha_k^* = \inf\{\alpha \geq 0 : m_k - \alpha \nabla_m J[m, d] \notin M\}$.

Similar but more complicated: Newton, quasi-Newtons, Krylov-based,...

NB: Smoothness, Hilbert structure not necessarily natural or best assumptions - but *de facto* underlie most efficient optimization algorithms, formulations of IPs as optimization problems.

What you didn’t see:

- coordinates
- indices
- loops over coordinate indices
- simulation geometry: grids, time steps, etc.
- data geometry: headers etc.
- parallelism, data distribution,...
- ...

but this stuff has to go *somewhere*.... else you can’t do anything!

Calculus vs. Data Management

Abstract IP formulation and (at least some) solution algorithms stated exclusively in terms of vector calculus concepts: vectors, scalar- and vector-valued functions, etc.

Collection of computational types parallel to these = *calculus package*.

Can be *abstract*, like the mathematics - i.e. not in and of itself sufficient to fully define a computation, but rather internally consistent description of a large class of computations. *Behaviour, not implementation!*

Collection of artifacts necessary for implementation of actual computations - coordinate descriptions of vectors, simulation and data geometry, data distribution, ... = *data management package*.

Example of distinction/relation between these categories: vectors (in f. d. vector spaces) *have* coordinate arrays, but are not logically identical with them.

Some Attributes of a Good Calculus Package

Requirement: maintain conceptual parallelism with abstract mathematics while allowing for computational realities - in particular for efficient performance in solution of very large-scale problems.

$\Rightarrow$ design of caculus, data management packages inevitably linked to some extent - but keep linkage as weak as possible!

- Vector spaces as *abstract factories*
- Function evaluations = jet of function at point
- Second order constructs: Cartesian products, compositions, linear combinations,...

Computational Vector Spaces

Any good textbook: a *vector space* over a *field* $\mathbf{F}$ is a *set* together with an operation (linear combination) satisfying ... (blah, blah, blah)

From which you see that:

- The primary concept is vector *space*, not *vector* - a vector is a member of a vector space (set), and is meaningless without reference to its space.
- Except for dim=0, cardinality of vector space is infinite, so can't realize computationally - the best you can do is a "service window", which provides a member of the set on request. Accepted name for such a thing: *factory*.
- An abstract (nonspecific) type which realizes abstract vector spaces will build abstract vectors, so is an *abstract factory*.

Computational Vector Spaces, cont'd

- To avoid sophomoric mistakes, should be able to compare vector spaces - are they same? Only combine vectors in same space!
- The type also needs to define a linear combination method: $y \leftarrow ax + by$ for scalars $a, b \in \mathbf{F}$ and vectors x, y, asserted to obey the usual axioms (assignment form, overwrite of input: nods to computational efficiency).
- various conveniences (additive identity, scale, negate,...) defined in terms of LC.
- type parametrized by field (NB: true confession - floating point number systems are not fields!).

Not included: *dimension* - abstraction accommodates spaces without dimension, i.e. *infinite-dimensional* spaces. Can actually realize these computationally (cf. infinite precision arithmetic). Good design $\Rightarrow$ don't include attributes which cannot be realized sensibly in every instance.

Vectors

Since *computational* vector spaces are factories, rather than sets, still must specify the structure of vector = product of factory.

Implicit choice of basis $\Rightarrow$ vector determined by its coordinate data - computationally this must be stored somehow, in an instance of a type from the data management package. Many possibilities - native array (eg. in C), or list, or tree, or... For abstract vectors (members of abstract vector spaces) this storage type must be an *abstract data container*. [For what abstract data containers should *do*, stay tuned...]

Upshot: a vector is a pair of

- a reference to a vector space, which provides linear combination and sanity-checking ("vector knows what space it's in"), and
- an abstract data container

Some Design Decisions

Since a vector “owns” an abstract data container, two possibilities exist for the space’s factory function:

- space builds vectors
- space builds (abstract) data containers

I like the second possibility.

Construction of a vector requires a reference to a vector space. During construction, the space provides the data container required by the vector. In this design, the space type need not refer explicitly to the vector type (though it must refer to the abstract data container type).

Some Design Decisions, cont'd

Access: should other program units be permitted direct access to the data container owned by a vector? Alternative: provide *only* an evaluation interface for an *abstract function type*. This allows vectors to keep track of their own modification history - impossible if access is unrestricted.

Abstract function type: another aspect of a satisfactory data management package. Permits arbitrary operations on data.

What abstract data containers do: evaluate instances of abstract function types on themselves. What instances of abstract function do: permit themselves to be evaluated (i.e. use, rather than action, specified!).

Enormously convenient: permit the abstract function type to have *persistent internal state*, so not necessarily a function: result of evaluation can depend on results of previous evaluations. Accepted name for such things: *function object*.

Functions and Evaluation Efficiency

Other main actor in vector calculus: functions of vector variable (**NB**: completely different notion from “function object”).

Functions have: domains, ranges, rules for computing values, derivatives, etc.

Major efficiency issue for functions defined by/through simulation: *intermediate data* may be shared between various related computations (value, derivative,...) and may depend on point of evaluation. Example: data-adapted grids, stiffness matrices, interpolated coefficient arrays...

This data may come at considerable computational cost.

Functions and Evaluation Efficiency

Several options in handling intermediate data:

- Throw it away, who cares, computers are getting faster...
- Keep data associated with $f(x), f'(x), ...$; when asked for $f(y), f'(y), ...$ check whether $x = y$. Requires comparison of vectors.
- Introduce another type which stores the intermediate data needed by values of function f and its derivatives at a point (specific memory reference for x) - *evaluation type*.

Functionally, an evaluation is simply a pair consisting of a function and a point in its domain, equipped to return the value, value of derivative,... on demand.

Evaluations and Updates

Consider fixed point iteration for $f : M \to M$: for $k = 0, 1, 2, ...$

$$x_{k+1} = f(x_k)$$

For storage efficiency, invariably implemented via assignment: until told to stop,

$$x \leftarrow f(x)$$

Clear what this means for one step: given $x \in \mathcal{D}(f)$, form the evaluation of f at x. Overwrite x with the value of f at x.

Question: *now* what is the value of $f(x)$?

Answer: it should be the value of f at the *new* x - i.e. the evaluation should update as soon as the evaluation point updates.

Evaluations and Updates, cont'd

How can you assure the synchronization of evaluation (output) and evaluation points (input)?

This is *easy*, if vectors have been structured to permit access to their defining data only via evaluation of function objects, *and* provided with a *version count*. Then version count is updated whenever such evaluation takes place.

Copy ($x \leftarrow y$, implemented via a function object) alters the target (x) so the assignment $x \leftarrow f(x)$ updates the version count of x. Ask the evaluation type for the vector function f to retain a reference version count: comparison now shows that x has been updated, so all intermediate data and final results should be recomputed. This can be done on demand, i.e. whenever the results are needed.

$\Rightarrow$ the evaluation type for vector functions can honor the semantics of "$f(x)$ for variable x".

Calculus Package Overview

[Apologies to anyone whose favorite software is omitted from my list...]

Since no high-level language defines vector space, vector, ... must *introduce new types*, including abstract ("polymorphic") types. For example, any instruction involving an abstract vectors must execute properly when supplied with any concrete realization of "vector".

Massively convenient: use a source language which directly supports user-defined polymorphic types - "object oriented". Examples: C++, Java, Python,... Most work so far in C++.

C++ is the worst object-oriented high-level language in the world... except for all the others (apologies to Churchill).

NB: You don't *have* to use an "object oriented" language to do this. Proof: the compiler codes it in assembler. Q. E. D.

Calculus Package Overview, cont'd

- Early-mid 90's: CLOP (Nichols, Dunbar, Claerbout), COOOL (Scales, Deng) - abstract linear operators, vectors; OPT++ (Meza), OptSolve++ (TechX): abstract functions and vectors, aimed at optimization.
- HCL (Gockenbach & WWS, 1996): introduced vector space and evaluation types.
- RTOp (Bartlett, 2000): function objects
- Trilinos project at SNL (Heroux): interoperable collection of packages for many common NA tasks, including optimization. Subpackages: Thyra (Bartlett) defines abstract vector spaces, linear operators, and "problems"; NOX (Kolda & Pawlowski) extends Thyra vector space type and defines an evaluation type ("group"), aimed at optimization, nonlinear eqns.
- RVL (Scott, Padula, & WWS): evolution of HCL, general role for function objects, automatic updating of evaluations.

How Parallel Can You Get?

Examples: translations of a few common algorithmic fragments using RVL types.

A is a linear operator,
p, r are vectors in $\mathcal{D}(A)$
	`cgstep(LinearOp<float> & A,`
	`Vector<float> & p,`
	`Vector<float> & r) {`

Let q be a vector in $\mathcal{D}(\mathcal{A})$	`Vector<float> q(A.getDomain());`
$q \leftarrow Ap$ (presumes that $\mathcal{D}(A) = \mathcal{R}(A)$)	`A.applyOp(p,q);` (`applyOp` code checks that `p.getSpace() == A.getDomain()`, `q.getSpace() == A.getRange()`)
$\alpha = \langle r, r \rangle / \langle p, q \rangle$	`float alpha =` `r.inner(r)/p.inner(q);`
$r \leftarrow r - \alpha q$	`r.linComb(-alpha,q)`
...	...}

Data Management Packages

Required to implement “concrete” (usable) subtypes of the abstract calculus types.

Example: computational vector spaces are factories that make data containers - what sort of data containers?

Simplest (maybe) choice: a type which provides access to::

- a size or dimension
- the address of the first element of an array of this size

Everyone has one of these “simple data containers”: eg. `HCL::Vector`, `Thyra/RTOp::subvector`, `RVL::LocalDataContainer`.

Data Management Packages, cont'd

Characteristics of simple data container: (1) the data is local, i.e. is stored in the address space of a single process; (2) easy to implement functions which manipulate this data, in any high-level language; (3) easy to make subtypes by adding more attributes.

Example of subtyping: *regularly gridded data* type: add to the simple data container described above the ability to return a description of a grid or finite hypercubical lattice - number of axes, number of points on each axis, step along each axis, coordinate of first point on each axis [SEP's `n1, d1, o1, n2, d2, o2,...`].

Another example: the SEGY trace = a simple data container, plus a *trace header*.

NB: Each of these fancier data containers *is* a simple data container, and can be used in any context which requires only size and array access - but can be used in additional contexts as well.

Function Objects and Abstract DCs

A *tuple* of simple data containers is not a simple data container (eg. data is in many arrays, rather than one), but wish to treat tuple as a data container.

This and other examples $\Rightarrow$ need for abstract data container type.

Recall: data containers evaluate function objects. Subtypes of abstract data container type have specific evaluation rules: for example, tuple data containers evaluate function objects in loop over components. Ultimately delegate to evaluation on DCs with direct data access (array, list,...).

Function object evaluation on simple DC (encapsulated array) can be implemented in any high-level language. Example: FO implementing finite difference time step evaluated by regular grid DC, can be implemented in Fortran.

Examples from Reflection Seismology

- DSNMO Velocity Analysis
- Finite difference least squares inversion

Shared features:

- RVL implementation of LBFGS (Nocedal);
- regular grid and seismic trace data container implementations; seismic trace DCs use SU library functions;
- non-OO code implements basic simulation functions as “guts” of function object - C for DSNMO, Fortran for FD.

Example 1: DSNMO Velocity Analysis

Differential semblance velocity estimation for CMP-sorted seismic reflection data $d(t, \mathbf{x}_h, \mathbf{x}_m)$,

$$\mathbf{x}_h = \frac{\mathbf{x}_r - \mathbf{x}_s}{2}, \ \mathbf{x}_m = \frac{\mathbf{x}_r + \mathbf{x}_s}{2}$$

$\mathbf{x}_r$ = horizontal receiver position, $\mathbf{x}_s$ = horizontal source position. [Position in flat Earth: $(z, \mathbf{x})$, z = depth, $\mathbf{x}$ = horizontal coords.

Somewhat lengthy analysis, many approximations: if compressional wave velocity $v(z, \mathbf{x})$ is chosen correctly, then "moveout corrected data"

$$F[v, d](t_0, \mathbf{x}_h, \mathbf{x}_m) \equiv d(T[v](t_0, \mathbf{x}_h, \mathbf{x}_m), \mathbf{x}_h, \mathbf{x}_m)$$

will be approximately independent of $\mathbf{x}_h$. Map $t_0 \rightarrow T[v](...)$ is v-parametrized change of variables related to the time of travel of waves, and t_0 is the vertical time of travel.

Example 1: DSNMO Velocity Analysis, cont'd

Suggests optimization for extraction of v: minimize "differential semblance" objective

$$J[v;d] = \frac{1}{2}\left\|\nabla_{\mathbf{x}_h} F[v,d]\right\|^2$$

Fits sim-driven opt model

$$\min_{m\in M} J[m,d],\;\; J[m,d] = G(F[m],d)$$

with $M = \{(v,d') : d' = d\}$, $E = D$, and $G(d_1,d_2) = \frac{1}{2}\left\|\nabla_{\mathbf{x}_h} d_1\right\|^2$

WWS, Stolk, Kim,...: (1) J as defined here is smooth jointly in v, d, wrt appropriate metrics; (2) J is (essentially) the only such smooth v-dependent quadratic form in d; (3) all stationary points are global mins.

Example 2: Finite Difference Least Squares Inversion

Fits sim-driven opt model: M = "suitable" set of seismic compressional velocity models, D = seismic traces at specified source and receiver locations for given sources, F = FD approximation to solution of wave equation solution operator + sampling at receiver locations, G = mean square error functional.

Least squares has long and successful history in geosciences (and other areas of science & engineering).

Application to seismic inverse problem - proposed in late 70's, 80's (notable exponent: A Tarantola). Not particularly successful - not smooth in the same sense as DS, apparently many spurious local minima.

Data fitting essential goal $\Rightarrow$ should be component of successful seismic velocity inversion algorithms yet to be formulated.

Example 2: FD LS Inversion, cont'd

Our implementation:

- 1D, 2D, 3D simulation;
- Admissible set of models defined by upper, lower velocity bounds; these plus desired data passband determine FD grids - construction is automated;
- source, receiver locations, recording time step decoupled from FD grid via interpolation;
- (2,4) FD scheme *single time step* coded in Fortran;
- Fortran for linearized, linearized adjoint single time steps automatically generated using TAMC;
- TSOpt package takes single time step functions, some other code describing model, state space, etc., and makes full-blown simulator, with RVL `Operator` (vector-valued vector function) interface.

Example 2: FD LS Inversion, cont'd

- Within admissible set, grid is *constant*;
- $\Rightarrow$ discrete simulator is *differentiable* as function of parameters (velocity field);
- $\Rightarrow$ straightforward verification of derivative, adjoint derivative calculations,

"The adjoint test": choose m in domain, (pseudo)random vectors $\delta m, \delta d$ in (tangent space of) domain, range of discrete forward operator, show that

$$|\langle DF[m]\delta m, \delta d\rangle - \langle \delta m, DF[m]^*\delta d\rangle|$$

be small in suiitable sense.

[Implementation in RVL: simulator constructed by TSOpt is an `RVL::Operator`, has functions which return domain, range `RVL::Space`s, use these to construct vectors, evaluate randomizing function object on these, etc. This test is *built into* the `RVL::Operator` class.]

Parallelism, Clusters, other HPC considerations

Two categories of distributed execution relevant (*RVL implementation*):

- loop-level - decompose individual simulations by segmenting loops ("domain decomposition"), distribute pieces. *Hidden in internal structure of time step function objects.*
- task-level - distribute entire simulations to nodes or subclusters (now in 2D, soon in 3D). *Implemented generically in definition of function object evaluation for distributed data containers - uses MPI structs.*

Expl of task level parallelism: RVL/TSOpt FD simulation of shot gather (single experiment) from IFP "Marmousi" 2D synthetic data set w. accurate 5–40 Hz band requires 10 min on recent vintage Opteron; simulation of entire line (240 shots) on cluster of 240 Opterons takes only a few seconds longer.

Conclusions

- Within limits imposed by nature of von Neumann machines, can write code that closely mimics *abstract* mathematics - just like the mathematics, this code applies *ipso facto* to wide variety of contexts *without alteration*.
- A number of such software frameworks for simulation-driven optimization have been or are being constructed, from academic experiments like RVL to near-production environments like Trilinos - *take a look!*
- Abstract programming occupies vanishingly small number of cycles compared to inner loops of large simulations, even makes parallelism more accessible.

The (hoped-for) end result: programming tools that (like all successful programming tools) make computation exploration of complex, large-scale inverse problems easier and more productive.

Acknowledgements

I am deeply grateful to my former students Mark Gockenbach, Shannon Scott, Hala Dajani, Tony Padula, and Eric Dussaud for all they have taught me about scientific computing, OOP, and related topics.

The work reported in this talk owes much to many people, but particularly to Jon Claerbout and his students (particularly Dave Nichols and Matt Schwab), Lester Dye, Amr el Bakri, Mike Gertz, Ross Bartlett, Juan Meza, and Mike Heroux.

The work reported here was supported in part by grants from the National Science Foundation, the Department of Energy, ExxonMobil Upstream Research Co., and by the sponsors of The Rice Inversion Project.

Inversion: more popular all the time...

Such a popular subject deserves a good software framework...[Thanks: Deschutes Brewing Co. of Bend, OR]