TSOpt: a transient simulation package based on SVL

William W. Symes

Outline

- Transient simulation-driven optimization
- Overview of SVL
- TSOpt: structure and usage
- Example: Acoustic simulation of seismograms
- Conclusion - Parallelism, Components

Transient simulation-driven optimization

Core problem: minimize $J[u, c]$, in which the *state* $u(t)$ and *control* c satisfy the DE constraint (*state equation*)

$$G\left[\frac{du}{dt}, u, c\right] = 0$$

Will focus on reduced problem (”black box”, ”NAND”) approach: solve state equation, minimize $J^{\mathrm{red}}[c] = J[u[c], c]$.

Many components of solution also useful for treatment of original problem as constrained optimization problem (”all at once”, ”SAND”).

If problem properties permit, most efficient approach is some relative of Newton's method $\Rightarrow$ need:

- definition of storage classes for state, control;
- evaluation for LHS of state equation, together with its derivatives ("sensitivity equation") with respect to u, c, possibly second derivatives, and adjoints of these;
- definition of (discrete) time step;
- combine these with an interface defining a *functional*, i.e. a scalar-valued function, to be coupled to optimization algorithms.
- functional interfaces much be embedded in sufficiently rich system of types to support algorihm definition, implementation.

The Standard Vector Library

Aim: provide framework for implementation of simulation driven optimization algorithms.

Design principle: core classes should cleanly emulate *calculus in Hilbert space*. Stratified by level of abstraction; implementation details rigorously confined to strata requiring them.

Example: SVL `Vector`s do not report dimension or length - not required to define the behaviour of vectors!

Successor to HCL (Gockenbach et al., ACM TOMS 1999) - innovations include templated classes, exception handling, namespaces, extensive use of function forwarding, better memory management and encapsulation, plus intrinsic localization of interaction with network in distributed implementation.

The Standard Vector Library

Consists of

- calculus classes: `Space` (vector spaces), `Vector` (vector), `Functional` (scalar valued function), `Operator` (vector valued function), `LinearOp` (linear operator), and supporting classes such as `ProductSpace`, `LeastSquares-Functional`, etc.
- data management classes: `DataContainer` (abstract data container), `Local-DataContainer` (concrete data container), `FunctionObject` (encapsulated function, evaluated by `DataContainer` objects).

Written in ISO C++. Base classes templated (where necessary) on a scalar type.

Data Management Classes

- `LocalDataContainer<Scalar>`: provides access to raw data (`Scalar * getData()`, `int getSize()`). Concrete child classes may also provde *metadata* appropriate to particular data types (grid, FE, seismic,...). Subclass of `DataContainer`.
- `DataContainer`: abstract DC, provides evaluation of `FunctionObject`s, child classes implement by delegation through tree DCs ending in LDC leaves
- `FunctionObject`: templated subclasses provide evaluation on varying numbers of LDC (`void operator()(LocalDataContainer<Scalar> &,...)`) - unary, binary,...
- Various composites such as `ProductDataContainer`, which delegates evaluation to its components, etc.

Calculus Classes

- `Space<Scalar>`: factory class for `DataContainer`s appropriate to vector data type, also repository for linear algebra operations (linear combination, inner product).
- `Vector<Scalar>`: consists of `Space` reference and dynamically allocated `DataContainer`. LA ops delegated to `Space`, evaluation of arbitrary `FunctionObjects` to `DataContainer` member.
- `Functional<Scalar>`: `Scalar`-valued function of vector variable, together with gradient, possibly higher derivatives. Chief user-defined methods are `apply`, `applyGradient`,... Usual implementation: evaluate `FunctionObject`(s).
- `Operator<Scalar>`, `LinearOp<Scalar>`: vector-valued functions, structure similar to that of `Functional`.

`Rn`: A typical SVL class family

Defines standard ”Fortran” or ”Matlab” vectors, no metadata attributes other than length.

- `RnArray`: subclass of `LocalDataContainer`, instance stores length `int n`, array `Scalar * a`:

  ```
  int getSize() { return n; }
  Scalar * getData() { return a; }
  ```

- `RnSpace`: implements `DataContainer * buildDataContainer` (its factory function) by calling `RnArray` constructor, linear algebra ops by calling a packaged set of `FunctionObjects` (`LinearAlgebraPackage`) - standard implementations of linear comb., inner product which can be used in most `Space` implementations.

Rn: A typical SVL class family

There is no `RnVector`*!*

Access to a vector in a `RnSpace`: via `Vector` constructor - concrete, not subclassed:

```
RnSpace<double> sp(n); // n-diml RnSpace
Vector<double> v(sp);  // get a vector in sp
```

Major consequence: for algorithms formulated exclusively in terms of SVL calculus classes, need for dynamic memory management is dramatically reduced. Usual access to workspace, eg. in domain of an operator rep. by `Operator A`:

```
Vector<double> v(A.getDomain());
```

SVL: interaction with Data

The only way to interact with data in SVL: via evaluation of `FunctionObjects`!

Example: define a vector in R^{10}, read data into it from an ascii file.

```
try {
  RnSpace<double> sp(10);
  Vector<double> v(sp);
  ASCIIReader<double> ar("myfile.dat");
  v.eval(ar);
}
catch (SVLException & e) {
  ...
}
```

TSOpt: SVL-based time domain simulation for optimization

Aim: define abstract interfaces for the elements of time-domain simulation, concrete simulation operators based on these - isolate the roles of various elements to facilitate code re-use. Evolution of *FDTD* package based on HCL (Gockenbach et al., ACM TOMS 2002).

Example: common definition of time step (Euler, BDF,...) applicable w/o loss of efficiency across wide range of problems - like Fortran ODE libraries, but with better data encapsulation and built-in access to derivatives (”sensitivities”), adjoints, and other optimization artifacts.

Provide platform for structure use of Automatic Differentiation: user defines only single step operations, rest of simulation (including eg. Griewank checkpointing scheme for adjoint state method) is built-in.

TSOpt: Design Goals

- top-level interface is SVL `Operator` - useful for both NAND and SAND formulations; objective, constraints defined in terms of operator output
- allow arbitrary product structure for control, state
- accomodate adaptive discretization - adaptive time stepping, spatial meshing
- internal, external discretizations of control, state may differ
- both implicit, explicit, single- and multi-step schemes, implicit and explicit state equations
- accommodate multisimulations, in which output of simulator is multicomponent vector, each component of which is itself the output of a simulation.
- implement using `FunctionObject` interface - natural relation to component frameworks - must be `Unary` to allow for arbitrary numbers of control, state components

TSOpt: User-supplied Classes

- `Statics`: defines state, control at single time step, also translation between internal, external representations of these
- `Dynamics`: defines LHS of state equation, compatibly with `Statics` definition of state and control and with enough functionality to implement arbitrary...
- `Step`: time step class (Euler, Leapfrog, Crank-Nicholson,...), application regulated by...
- `Clock`: control of simulation time

TSOpt::Statics

```
/** return reference to factory, which builds control and
    state instances  (i.e. internal representations) */
virtual ModelBuilder<Scalar> & getModelBuilder() = 0;
/** return reference to factory, which builds translators
    between internal, external control representations */
virtual SamplerFactory<Scalar> &
    getControlSamplerFactory() = 0;
/** return reference to factory, which builds translators
    between internal and external state representations */
virtual SamplerFactory<Scalar> &
    getDataSamplerFactory() = 0;
```

TSOpt::ModelBuilder

Factory with two products:

```
/** dynamically allocate control LDC */
virtual LocalDataContainer<Scalar> * buildControl() = 0;
/** dynamically allocate state LDC */
virtual LocalDataContainer<Scalar> * buildState() = 0;
```

Allocation is dynamic - memory is managed by calling object (part of library, no worry for user!).

Return type may include whatever metadata is useful in defining particular dynamics to be used (grid information, notably).

TSOpt::SamplerFactory

Also factory with two products:

```
/** use auxliary data of LDC to initialize internal data..*/
virtual void initialize(LocalDataContainer<Scalar> & d) = 0;
/** dynamically allocate forward sampler FO  */
virtual FwdSampler<Scalar> * buildFwd() = 0;
/** dynamically allocate adjoint sampler FO */
virtual AdjSampler<Scalar> * buildAdj() = 0;
```

`SamplerFactory::initialize` = opportunity to pass any data not proper to the `SamplerFactory` itself, eg. to component simulations of a multisimulation. May be no-op for single simulation implementations, when `SamplerFactory` instance data is complete.

TSOpt::[Fwd][Adj]Sampler

Translation between internal $\longrightarrow$ external data reps - subclass of `UnaryFunction-Object`

```
/** Provides access to sample time */
virtual void setTime(Scalar t) = 0;
/** number of output LDCs  */
virtual int getNumberOfData() = 0;
/** write method - output of internal buffer onto output
  LDC, index 0 <= i < getNumberOfData() */
virtual void save(LocalDataContainer<Scalar> & d, int i)=0;
/** sample to internal buffer */
virtual void operator()(LocalDataContainer<Scalar> & x)=0;
```

`AdjSampler`: `load` instead of `save`, `operator()` samples *from* internal buffer.

TSOpt::Dynamics

Expresses LHS of state equation, together with derivatives and adjoint. Main method:

```
virtual void rhse(LocalDataContainer<Scalar> & u,
                  LocalDataContainer<Scalar> & c,
                  LocalDataContainer<Scalar> & up,
                  Scalar a, Scalar b_0, Scalar b,
                  Scalar t) = 0;
```

which expresses

$$u_p = b_0 u_0 + bu + aH(u, c, t)$$

For explicit schemes, u_p = next time level of state. For 1-step, H = RHS, $b_0 = 0$. For multistep, u_0 = lin comb of previous time levels. For implicit equations or schemes, $H = G$ = LHS of state equation, u_p = residual in Newton loop.

TSOpt::Step

Base encapsulates time stepping methods:

```
virtual void fwdStep(Model<Scalar> & mdl,
                     Dynamics<Scalar> & dyn,
                     Clock<Scalar> & clk) = 0;
...
```

Uses `Model`, implemented in the base library, which uses `ModelBuilder` (user-supplied as part of `Statics`) to construct and return references to storage for state, control, and their perts. on as-needed basis.

Also uses...

TSOpt::Clock

Keeps track of time (!).

```
virtual Scalar getTime() = 0;
virtual void setTime(Scalar t) = 0;
virtual void setTimeStep(Scalar dt) = 0;
...
```

Child class `ConstClock` provided in base library, for fixed step methods:

```
ConstClock(Scalar tbeg, Scalar tend, Scalar dt);
```

Adaptive stepping can record time step history.

Example: a Simple Seismic Simulator

Acoustic 2D multisimulation using (2,4) leapfrog FD scheme.

- basic data structures: `GridData` (LDC) for control (velocity), internal state (pressure field), `SeismicDataContainer` (DC) and `SeismicBin` (LDC) for external state (seismogram - implemented using Seismic Unix)
- `Statics`: `ModelBuilder` uses constructors for grid, seismic DC classes; `Sampler`s use cubic spline interpolators, `SamplerFactory`s use their constructors.
- `Dynamics`: implement standard (2,4) scheme in F77, use AD to build derivative, adjoint subroutines, wrap.
- `Step`, `Clock`: Leapfrog with overwrite (in base library), `ConstClock`.

Example: Command Source Code

- Build `Statics`, `Dynamics`, `Step`, and `Clock` instances using data (filenames, parameters) read from input data file. NB: this includes both definition of internal (simulation) grids and construction of operators to interpolate / adjoint interpolate between these and archival representations of fields. `Statics` also includes definitions of `Space` instances specifying domain, range of simulation operator (external representations of control, state).
- Construct TSOp:

```
TSOp<float> op(sta, dyn, stp, clk);
```

- Use *built-in unit tests* provided by SVL (`Operator::checkDeriv`, `LinearOp::checkAdjointRelation`) to verify code.
- **Efficiency: execution times within a 1-5% of pure F77 implementation - 96-99%+ of execution time spent in F77 time stepping subroutines.**

Conclusion

- SVL provides clean ISO C++ framework for simulation driven optimization.
- TSOpt implements common code underlying all transient sim-driven opt apps: user implements only code defining particular app
- For even modest-sized problems (2D acoustics), virtual function overhead is negligible.
- Parallelism via client-server arch - server implements only data management layer of SVL, i.e. server-side subclasses of LDCs and FOs. These subclasses are free to use MPI or any other parallel execution runtime library, the artifacts of which are kept separate from base library.
- RTSOpt - remote execution version of TSOpt, based on SVLRemote (simple socket-based client-server framework).

Thanks to...

- Mark Gockenbach
- Roscoe Bartlett
- Tony Padula, Eric Dussaud, Hala Dajani, Peng Shen
- The Rice Inversion Project
- National Science Foundation
- Department of Energy - LACSI