



Stratimikos

Unified Wrapper to Trilinos Linear Solvers and Preconditioners

Roscoe A. Bartlett

Department of Optimization & Uncertainty Estimation

Sandia National Laboratories

April 3rd, 2008

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company,
for the United States Department of Energy under contract DE-AC04-94AL85000.





Motivation for Stratimikos



Nonlinear Algorithms and Applications : Everyone for Themselves?

Nonlinear
ANA Solvers
in Trilinos

NOX / LOCA

Rythmos

MOOCHO

...

Sandia
Applications

Xyce

Charon

Tramonto

Aria

Aleph

...

Trilinos and non-Trilinos
Preconditioner and Linear
Solver Capability

Key Point
• BAD



Nonlinear Algorithms and Applications : Thyra & Model Evaluator!

Nonlinear
ANA Solvers
in Trilinos

NOX / LOCA

Rythmos

MOOCHO

...

Model Evaluator

Trilinos and non-Trilinos
Preconditioner and Linear
Solver Capability

Stratimikos!

Sandia
Applications

Xyce

Charon

Tramonto

Aria

Aleph

...

Key Points

- Provide single interface from nonlinear ANAs to applications
- Provide single interface for applications to implement to access nonlinear ANAs
- Provides shared, uniform access to linear solver capabilities
- Once an application implements support for one ANA, support for other ANAs can quickly follow



Stratimikos Overview



Introducing Stratimikos

- Stratimikos created Greek words "stratigiki" (strategy) and "grammikos" (linear)
- Based on the foundation of abstract interface layer Thyra
- Defines class Stratimikos::DefaultLinearSolverBuilder
 - Provides common access to:
 - Linear Solvers: Amesos, AztecOO, Belos, ...
 - Preconditioners: Ifpack, ML, ...
 - Reads in options through a parameter list (read from XML?)
 - Accepts any linear system objects that provide
 - Epetra_Operator / Epetra_RowMatrix view of the matrix
 - SPMD vector views for the RHS and LHS (e.g. Epetra_[Multi]Vector objects)
- Provides uniform access to linear solver options that can be leveraged across multiple applications and algorithms
- Future: TOPS-2 will add PETSc and other linear solvers and preconditioners!

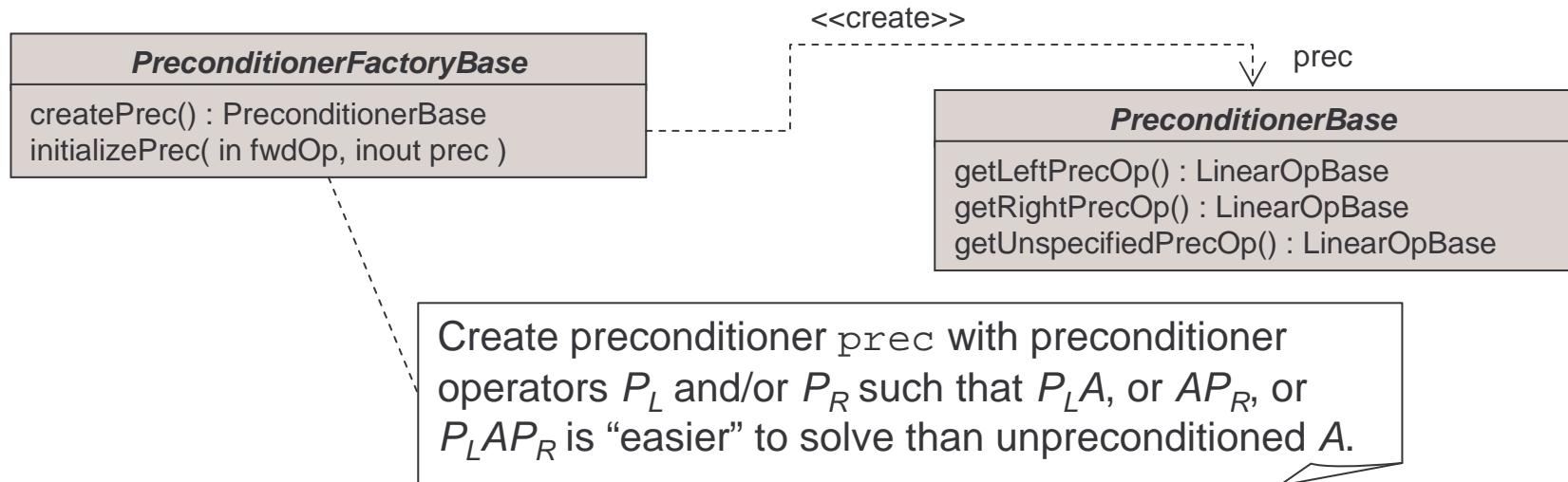
Key Points

- Stratimikos is an important building block for creating more sophisticated linear solver capabilities!



Preconditioners and Preconditioner Factories

PreconditionerFactoryBase : Creates and initializes **PreconditionerBase** objects



- Allows unlimited creation/reuse of preconditioner objects
- Supports reuse of factorization structures
- Adapters currently available for Ifpack and ML
- New Stratimikos package provides a single parameter-driver wrapper for all of these

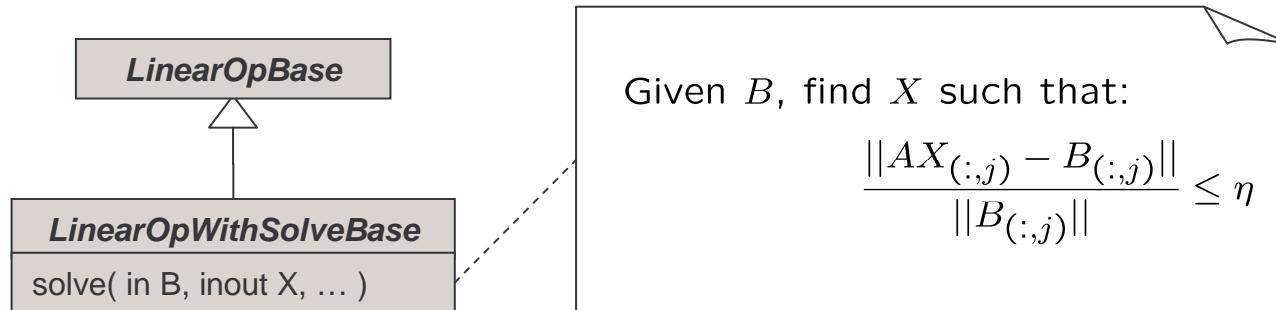
Key Points

- You can create your own **PreconditionerFactory** subclass!



Linear Operator With Solve and Factories

LinearOpWithSolveBase : Combines a linear operator and a linear solver

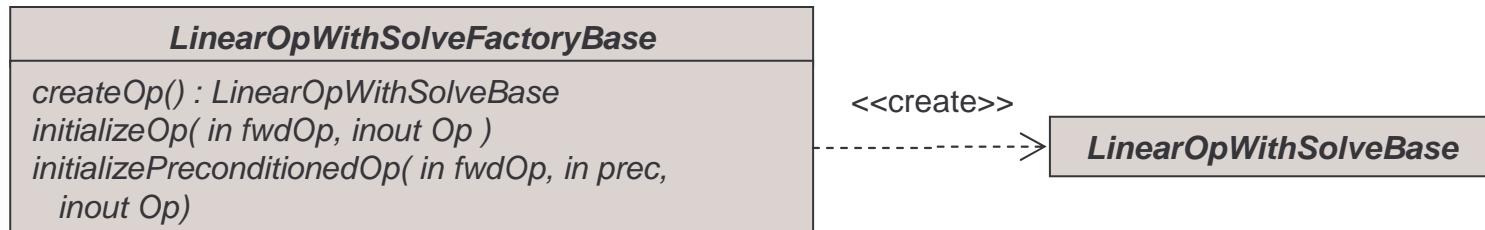


- Appropriate for both direct and iterative solvers
- Supports multiple simultaneous solutions as multi-vectors
- Allows targeting of different solution criteria to different RHSs
- Supports a “default” solve

Key Points

- You can create your own subclass!

LinearOpWithSolveFactoryBase : Uses LinearOpBase objects to initialize LOWSB objects



- Allows unlimited creation/reuse of LinearOpWithSolveBase objects
- Supports reuse of factorizations/preconditioners
- Supports client-created external preconditioners (which are ignored by direct solvers)
- Appropriate for both direct and iterative solvers
- Concrete adaptors for Amesos, AztecOO, and Belos are available
- New Stratimikos package provides a single parameter-driven wrapper to all of these!



Simple Stratimikos Example



Simple Stratimikos Example

simple_stratimikos_example.cpp

- Demonstrates how to solve single-RHS linear systems expressed as Epetra objects

Typical steps for “using a solver that accepts Thyra objects”:

- A) Setup the environment for the program
 - => Read in command-line options
- B) Create the input objects using your data structures (**independent of Thyra**)
 - => Read in linear system objects in terms of Epetra objects
- C) Wrap your objects as Thyra objects (**The “glue”**)
 - => Use Epetra-to-Thyra wrapper functions
- D) Call the solver through Thyra interfaces (**Thyra-specific**)
 - => Create the LinearOpWithSolve[Factory]Base objects and solve ...
- E) Post process the solution (if needed) using your objects (**independent of Thyra**)
 - => Post process solution and check in terms of Epetra objects (Not Thyra!!!)



A) Setup the environment for the program

```
//  
// A) Program setup code  
  
//  
// Read options from command-line  
  
std::string matrixFile = "";  
  
...  
  
Stratimikos::DefaultLinearSolverBuilder linearSolverBuilder;  
  
CommandLineProcessor clp(false); // Don't throw exceptions  
  
// Set up command-line options for the linear solver that will be used!  
linearSolverBuilder.setupCLP(&clp);  
  
clp.setOption( "matrix-file", &matrixFile  
              , "Defines the matrix and perhaps the RHS and LHS for a linear system to be solved." );  
  
...  
  
CommandLineProcessor::EParseCommandLineReturn parse_return = clp.parse(argc,argv);  
if( parse_return != CommandLineProcessor::PARSE_SUCCESSFUL ) return parse_return;
```

See the file simple_stratimikos_example.cpp for more details ...



B) Create the input objects using your data structures

```
//  
// B) Epetra-specific code that sets up the linear system to be solved  
//  
// While the below code reads in the Epetra objects from a file, you can  
// setup the Epetra objects any way you would like. Note that this next  
// set of code has nothing to do with Thyra at all, and it should not.  
//  
*out << "\nReading linear system in Epetra format from the file '"<<matrixFile<<"' ...\\n";  
  
#ifdef HAVE_MPI  
    Epetra_MpiComm comm(MPI_COMM_WORLD);  
#else  
    Epetra_SerialComm comm;  
#endif  
    RCP<Epetra_CrsMatrix> epetra_A;  
    RCP<Epetra_Vector> epetra_x, epetra_b;  
    EpetraExt::readEpetraLinearSystem( matrixFile, comm, &epetra_A, NULL, &epetra_x, &epetra_b );  
  
    if(!epetra_b.get()) {  
        *out << "\\nThe RHS b was not read in so generate a new random vector ...\\n";  
        epetra_b = rcp(new Epetra_Vector(epetra_A->OperatorRangeMap()));  
        epetra_b->Random();  
    }  
  
    if(!epetra_x.get()) {  
        *out << "\\nThe LHS x was not read in so generate a new zero vector ...\\n";  
        epetra_x = rcp(new Epetra_Vector(epetra_A->OperatorDomainMap()));  
        epetra_x->PutScalar(0.0); // Initial guess is critical!  
    }
```

Create the Epetra objects any way that you would like ...



C) Wrap your objects as Thyra objects (The “glue”)

```
//  
// C) The "Glue" code that takes Epetra objects and wraps them as Thyra  
// objects  
//  
// This next set of code wraps the Epetra objects that define the linear  
// system to be solved as Thyra objects so that they can be passed to the  
// linear solver.  
  
RCP<const Thyra::LinearOpBase<double> >  
    A = Thyra::epetraLinearOp( epetra_A );  
RCP<Thyra::VectorBase<double> >  
    x = Thyra::create_Vector( epetra_x, A->domain() );  
RCP<const Thyra::VectorBase<double> >  
    b = Thyra::create_Vector( epetra_b, A->range() );
```

Epetra-to-Thyra wrapper functions are found in `Thyra_EpetraThyraWrappers.hpp`

Turning Epetra objects into Thyra objects is easy!





D) Call the solver through Thyra interfaces

```
//  
// D) Thyra-specific code for solving the linear system  
//  
// Note that this code has no mention of any concrete implementation and  
// therefore can be used in any use case.  
//  
  
// Reading in the solver parameters from the parameters file and/or from  
// the command line. This was setup by the command-line options  
// set by the setupCLP(...) function above.  
linearSolverBuilder.readParameters(out.get());  
  
// Create a linear solver factory given information read from the  
// parameter list.  
RCP<Thyra::LinearOpWithSolveFactoryBase<double> >  
lowsFactory = linearSolverBuilder.createLinearSolveStrategy( "" );  
  
// Setup output stream and the verbosity level  
lowsFactory->setOutputStream(out);  
lowsFactory->setVerbLevel(Teuchos::VERB_LOW);  
  
// Create a linear solver based on the forward operator A  
RCP<Thyra::LinearOpWithSolveBase<double> >  
lows = Thyra::linearOpWithSolve(*lowsFactory,A);  
  
// Solve the linear system (note: the initial guess in 'x' is critical)  
Thyra::SolveStatus<double>  
status = Thyra::solve(*lows,Thyra::NOTRANS,*b,&x);  
*out << "\nSolve status:\n" << status;  
  
// Write the linear solver parameters after they were read  
linearSolverBuilder.writeParamsFile(*lowsFactory);
```

- This part of the code has nothing to do with Epetra!
- This is really only a trivial interaction with Thyra!



E) Post process the solution (if needed) using your objects

```
//  
// E) Post process the solution and check the error  
//  
// Note that the below code is based only on the Epetra objects themselves  
// and does not in any way depend or interact with any Thyra-based  
// objects. The point is that most users of Thyra can largely gloss over  
// the fact that Thyra is really being used for anything.  
//  
// Wipe out the Thyra wrapper for x to guarantee that the solution will be  
// written back to epetra_x!  
x = Teuchos::null;  
  
*out  
    << "\nSolution ||epetra_x||^2 = " << epetraNorm2(*epetra_x) << "\n";  
  
*out << "\nTesting the solution error ||b-A*x|| / ||b|| computed through the Epetra objects ... \n";  
  
// r = b - A*x  
Epetra_Vector epetra_r(*epetra_b);  
{  
    Epetra_Vector epetra_A_x(epetra_A->OperatorRangeMap());  
    epetra_A->Apply(*epetra_x,epetra_A_x);  
    epetra_r.Update(-1.0,epetra_A_x,1.0);  
}  
  
const double  
    nrm_r = epetraNorm2(epetra_r),  
    nrm_b = epetraNorm2(*epetra_b),  
    rel_err = ( nrm_r / nrm_b );  
const bool  
    passed = (rel_err <= tol);  
  
*out  
    << " ||b-A*x|| / ||b|| = " << nrm_r << "/" << nrm_b << " = " << rel_err  
    << " < tol = " << tol << " ? " << ( passed ? "passed" : "failed" ) << "\n";
```



Stratimikos Parameter Lists



Stratimikos Parameter List and Sublists

```
<ParameterList name="Stratimikos">
  <Parameter name="Linear Solver Type" type="string" value="AztecOO"/>
  <Parameter name="Preconditioner Type" type="string" value="Ifpack"/>
  <ParameterList name="Linear Solver Types">
    <ParameterList name="Amesos">
      <Parameter name="Solver Type" type="string" value="Klu"/>
      <ParameterList name="Amesos Settings">
        <Parameter name="MatrixProperty" type="string" value="general"/>
        ...
        <ParameterList name="Mumps" > ... </ParameterList>
        <ParameterList name="Superludist" > ... </ParameterList>
      </ParameterList>
    </ParameterList>
  </ParameterList>
  <ParameterList name="AztecOO">
    <ParameterList name="Forward Solve">
      <Parameter name="Max Iterations" type="int" value="400"/>
      <Parameter name="Tolerance" type="double" value="1e-06"/>
      <ParameterList name="AztecOO Settings">
        <Parameter name="Aztec Solver" type="string" value="GMRES" />
        ...
      </ParameterList>
    </ParameterList>
    ...
  </ParameterList>
  <ParameterList name="Belos" > ... </ParameterList>
</ParameterList>

<ParameterList name="Preconditioner Types">
  <ParameterList name="Ifpack">
    <Parameter name="Prec Type" type="string" value="ILU"/>
    <Parameter name="Overlap" type="int" value="0"/>
    <ParameterList name="Ifpack Settings">
      <Parameter name="fact: level-of-fill" type="int" value="0"/>
      ...
    </ParameterList>
  </ParameterList>
  <ParameterList name="ML" > ... </ParameterList>
</ParameterList>
</ParameterList>
```

Top level parameters

Linear Solvers

Preconditioners

Sublists passed
on to package
code!

Every parameter
and sublist **not in**
red is handled by
Thyra code and is
fully validated!



Automatically Generated Parameter List Documentation

“Human readable” automatically generated documentation for Stratimikos

```

Linear Solver Type : string = Amesos
  # Determines the type of linear solver that will be used.
  # The parameters for each solver type are specified in the sublist "Linear Solver Types"
  #   Valid values: "Belos", "Amesos", "AztecOO"

Preconditioner Type : string = ML
  # Determines the type of preconditioner that will be used.
  # This option is only meaningful for linear solvers that accept preconditioner factory objects!
  # The parameters for each preconditioner are specified in the sublist "Preconditioner Types"
  #   Valid values: "None", "Ifpack", "ML"

Linear Solver Types ->
  AztecOO ->
    Output Every RHS : bool = 0
      # Determines if output is created for each individual RHS (true or 1) or if output
      # is just created for an entire set of RHSs (false or 0).

    Forward Solve ->
      Max Iterations : int = 400
        # The maximum number of iterations the AztecOO solver is allowed to perform.

      Tolerance : double = 1e-06
        # The tolerance used in the convergence check (see the convergence test
        # in the sublist "AztecOO Settings")

    AztecOO Settings ->
      Aztec Solver : string = GMRES
        # Type of linear solver algorithm to use.
        #   Valid values: "CG", "GMRES", "CGS", "TFQMR", "BiCGStab", "LU"

      Convergence Test : string = r0
        # The convergence test to use for terminating the iterative solver.
        #   Valid values: "r0", "rhs", "Anorm", "no scaling", "sol"

    ...

```

See Doxygen documentation for Stratimikos::DefaultLinearSolverBuilder!





Output of Default Parameter Values

Example input parameter (sub)list to Stratimikos (use simple_stratimikos_example.exe)

```
<ParameterList name="Stratimikos">
  <Parameter name="Linear Solver Type" type="string" value="AztecOO" />
  <Parameter name="Preconditioner Type" type="string" value="Ifpack" />
</ParameterList>
```

Output (augmented) parameter (sub)list

```
<ParameterList>
  <Parameter name="Linear Solver Type" type="string" value="AztecOO" />
  <ParameterList name="Linear Solver Types">
    <ParameterList name="AztecOO">
      <ParameterList name="Forward Solve">
        <ParameterList name="AztecOO Settings">
          <Parameter isDefault="true" name="Aztec Solver" type="string" value="GMRES" />
          <Parameter isDefault="true" name="Convergence Test" type="string" value="r0" />
          ...
        </ParameterList>
        <Parameter isDefault="true" name="Max Iterations" type="int" value="400" />
        <Parameter isDefault="true" name="Tolerance" type="double" value="1e-06" />
      </ParameterList>
      <Parameter isDefault="true" name="Output Every RHS" type="bool" value="0" />
    </ParameterList>
  </ParameterList>
  <Parameter name="Preconditioner Type" type="string" value="Ifpack" />
  <ParameterList name="Preconditioner Types">
    <ParameterList name="Ifpack">
      <Parameter isDefault="true" name="Overlap" type="int" value="0" />
      <Parameter isDefault="true" name="Prec Type" type="string" value="ILU" />
    </ParameterList>
  </ParameterList>
</ParameterList>
```

=> Ifpack is not showing its default parameters!



Parameter List Validation



Error Messages for Improper Parameters/Sublists

Example: User misspells "Aztec Solver" as "ztec Solver"

```
<ParameterList>
  <Parameter name="Linear Solver Type" type="string" value="AztecOO" />
  <ParameterList name="Linear Solver Types">
    <ParameterList name="AztecOO">
      <ParameterList name="Forward Solve">
        <ParameterList name="AztecOO Settings">
          <Parameter name="ztec Solver" type="string" value="GMRES" />
        </ParameterList>
      </ParameterList>
    </ParameterList>
  </ParameterList>
</ParameterList>
```

Error message generated from PL::validateParameters(...) with exception:

```
Error, the parameter {name="ztec Solver",type="string",value="GMRES"}
in the parameter (sub)list "RealLinearSolverBuilder->Linear Solver Types->AztecOO->Forward
Solve->AztecOO Settings"
was not found in the list of valid parameters!
```

```
The valid parameters and types are:
{
  "Aztec Preconditioner" : string = ilu
  "Aztec Solver" : string = GMRES
...
}
```



Error Messages for Improper Parameters/Sublists

Example: User specifies the wrong type for "Aztec Solver"

```
<ParameterList>
  <Parameter name="Linear Solver Type" type="string" value="AztecOO"/>
  <Parameter name="Preconditioner Type" type="string" value="Ifpack"/>
  <ParameterList name="Linear Solver Types">
    <ParameterList name="AztecOO">
      <ParameterList name="Forward Solve">
        <ParameterList name="AztecOO Settings">
          <Parameter name="Aztec Solver" type="int" value="GMRES" />
        </ParameterList>
      </ParameterList>
    </ParameterList>
  </ParameterList>
</ParameterList>
```

Error message generated from PL::validateParameters(...) with exception:

```
Error, the parameter {paramName="Aztec Solver",type="int"}
in the sublist "DefaultRealLinearSolverBuilder->Linear Solver Types->AztecOO->Forward Solve-
>AztecOO Settings"
has the wrong type. The correct type is "string"!
```



Error Messages for Improper Parameters/Sublists

Example: User specifies the wrong value for "Aztec Solver"

```
<ParameterList>
  <Parameter name="Linear Solver Type" type="string" value="AztecOO" />
  <Parameter name="Preconditioner Type" type="string" value="Ifpack" />
  <ParameterList name="Linear Solver Types">
    <ParameterList name="AztecOO">
      <ParameterList name="Forward Solve">
        <ParameterList name="AztecOO Settings">
          <Parameter name="Aztec Solver" type="string" value="GMRESS" />
        </ParameterList>
      </ParameterList>
    </ParameterList>
  </ParameterList>
</ParameterList>
```

Error message generated from PL::validateParameters(...) with exception:

```
Error, the value "GMRESS" is not recognized for the parameter "Aztec Solver" in the sublist "".
```

```
Valid selections include: "CG", "GMRES", "CGS", "TFQMR", "BiCGStab", "LU".
```



Future Plans for Stratimikos

- TOPS-2 PETSc/Trilinos Interoperability
 - Fill PETSc matrices & vectors and use Trilinos preconditioners and linear solvers
 - Access PETSc preconditioners and solvers through a parameter list in Stratimikos?
- Fortran 2003 interface through Stratimikos
 - Built on Fotran/Epetra wrappers
- GUI to manipulate Parameter List XML files
- “Smart” linear solver & preconditioner builders?