Code Generation Tools for PDEs

Matthew Knepley
PETSc Developer
Mathematics and Computer Science Division
Argonne National Laboratory

Talk Objectives

- Introduce Code Generation Tools
  - Installation
  - Use
  - Customization
- Full Simulations
  - Simple input language (embedded in Python)
  - Customization
  - Simple Visualization
- Future Work

Motivation

Many parts of a PDE simulation may be handled by a scripting language. However, for maximum effectiveness, the scripting language should be combined with code generation.

- FIAT
  - Element Construction
- FFC
  - Differential form manipulation
- PETSc 3
  - Mesh interaction
  - Differential form integration
  - Operator assembly

Code Generation Tools

Bootstrap Install

1. Install BitKeeper from http://www.bitmover.com
2. Create a root directory, e.g. PETSc3
3. Install BuildSystem in PETSc3/sidl/BuildSystem
   - cd PETSc3; mkdir sidl
   - bk clone bk://sidl.bkbits.net/BuildSystem sidl/BuildSystem
4. Run the bootstrap installer
   - ./sidl/BuildSystem/install/aseBootstrap.py
5. Have beer and wait

General Install

For a general PETSc3 package *name*, you need to:

1. Download the package
   - cd PETSc3; mkdir *type*
   - bk clone bk://*type*.bkbits.net/*name* *type*/*name*
2. Build the package
   - cd PETSc3/*type*/*name*; ./make.py *options*

   NOTE: Configure and build options are both given to make.py

What Can Our Tools Do?

- Parse text into an AST

  We can also create and AST using the API

  Ex. Parse SIDL into the ASE representation

  Ex. Parse C and Python implementation code

- Transform one AST into another

  Ex. Transform a SIDL AST into a C AST for wrapper methods

- Output text (source code) from an AST

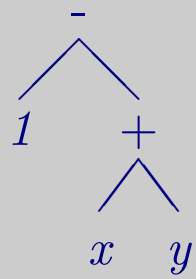
  Ex. Pretty print implementation code with regenerated bindings

What is an AST?

A tree structure encoding the syntatic information in an expression. We may then associate semantic information with individual nodes or the tree itself. For example, our **Expression Trees** represent arithmetic expressions, so that:

- *Leaves* represent variables or constants
- *Internal* nodes represent operations on the leaves

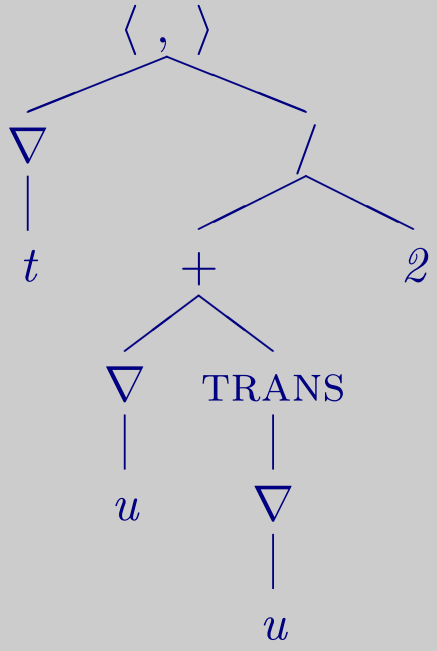
Here is an example tree representing $1 - x - y$



More General Expressions

We allow more general expression than arithmetic in order to incorporate matrix algebra and weak form expressions

Here is another tree representing $\langle \nabla t, \frac{1}{2}\left(\nabla u + \nabla u^T\right)\rangle$



What is a Visitor?

The Visitor is a common CS pattern that arises during tree traversal. We want to separate code for managing tree nodes from code to traverse and decorate the tree.

Each Visitor has:

- A **visitFoo**(Foo f) method for each node type Foo

Each Node has:

- An **accept**(Visitor v) method, which calls the appropriate visit method on v, passing itself

Thus a traversal will bounce back and forth between the visitFoo methods in the visitor, and the accept methods in the nodes. Since these are generic, the nodes need never be recompiled or relinked when a new visitor is added. This is ideal for a slowly changing set of node types, and rapidly changing set of visitors.

Using the Parser

In the ASE Compiler package, we have a sample parser. This may be used to verify that files can be parsed and the SIDL references can be resolved.

```
testerParser.py -help

testerParser.py -resolve=0 -language=SIDL sidl/sidl.sidl

testerParser.py -includes=[../Runtime/sidl/ase.sidl] -language=SIDL sidl/sidl.sidl


testerParser.py -language=Python server-python-sidl/ASE/Compiler/SIDL/Type_impl.py


testerParser.py
-includeDirs="{Cxx:[../Runtime/server-python-ase,server-python-sidl,client-python]}"
-language=Cxx server-python-sidl/ASE/Compiler/SIDL/Type_IOR.c
```

Using the Generator

The FEM package has a sample source code generator for finite element integration routines. The user can chooose any FIAT element type and degree, and the generator will provide C code with a Python wrapper, both in a shared library.

```
sourceGenerator.py --help
```

```
sourceGenerator.py --generationDirectory=test-cxx --degree=3
```

```
sourceGenerator.py --generationDirectory=test-cxx --element=Nonconforming
```

AST Base Class

The base class is `ASE.Compiler.Vertex` which supports tree traversal, the Visitor pattern, an identifier, and simple attributes.

- clone(), copy()
- get/setParent(), get/setChildren()
- accept()
- get/setIdentifier()
- get/setAttribute()

Node Classes

A new language requires only a parser implementing to `ASE.Compiler.Parser`, a pretty printer implementing `ASE.Compiler.Output`, and a set of node classes. The node classes must override the `accept()` function from `ASE.Compiler.Vertex` and handle nay dat specific to the node.

Full Simulation

Input Language

We have a simple text language for input, incorporating:

- Arithmetic, +, -, *, /, ; () abs(x)
- Coordinate functions, cos(x) exp(x)
- Continuum fields (known and unknown)
- Dual pairing, $\langle\ ,\ \rangle$
- Matrix operations, trans(u) det(u) vec(u)
- Differential operators, grad(u) div(u) curl(u)

A Sample Simulator

This sample simulator allows the user to choose a sample domain, an element type, and a problem type. The simulator then:

- parses the equation, boundary conditions, and exact solution
- constructs the dynamic finite element library
- meshes the given domain
- creates and assembles the linear algebra objects
- applies boundary conditions
- solves the algebraic problem
- visualizes the result
- checks the L_2 error of the solution

Using the Simulator

The simulation can be carried out on a series of refined meshes. For the Laplace operator, alternative integration routines are also available.

```
simulation.py Poisson --help

simulation.py Poisson --refinementLevels=5 --visualize

simulation.py Poisson --Lshaped --useFerrari
```

Performance

Dominant computational costs are incurred by optimized code

- Element integration
- Operator assembly (some variations)
- Nonlinear and Linear solve

Parallelism

Dominant communication handled by PETSc

- Vec and Mat assembly
- Matvec

Other communication handled by Python MPI

- Mesh element ghosting
- Vec ghosting (determination and updating)

Visualization

I use **MayaVi** for visualizing the solution fields. It is a Python wrapper for the VTK graphics pipeline. I currently write VTK format files, but I could use it directly from Python as a library.