Building Robust Scientific Codes

Matthew Knepley

Computation Institute
University of Chicago

HPC³: Workshop on High Performance Computing and Hybrid Programming Concepts for Hyperbolic PDE Codes
KAUST, Saudi Arabia, March 2011
What I Need From You

- Tell me if you do not understand
- Tell me if an example does not work
- Suggest better wording or figures
- Followup problems at petsc-maint@mcs.anl.gov
Ask Questions!!!

- Helps **me** understand what you are missing
- Helps **you** clarify misunderstandings
- Helps **others** with the same question
Simplifying Parallelization of Scientific Codes by a Function-Centric Approach in Python

Jon K. Nilsen, Xing Cai, Bjorn Hoyland, and Hans Petter Langtangen

- **Python** at the application level
- **numpy** for data structures
- **petsc4py** for linear algebra and solvers
- **PyCUDA** for integration (physics) and assembly
New Model for Scientific Software

**Application**

sympy  symbolics  FFC/SyFi

sympy  symbolics  eqn. definition  FFC/SyFi

**numpy**  data structures  integration/assembly

**petsc4py**  solvers  **numpy**  integration/assembly

**PyCUDA**  data structures

**PETSc**  PyCUDA

**CUDA OpenCL**
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls
- Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy

- Depth First Search
  - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

- Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

**Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
  - Boundary conditions
    - Restrict branching to PETSc C numbering and assembly calls

- Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless

- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

- Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
  - Depth First Search
    - Hell if I know

- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
  - Boundary conditions
    - Restrict branching to PETSc C numbering and assembly calls

Audience???
What is Missing from this Scheme?

- Unstructured graph traversal
  - Iteration over cells in FEM
    - Use a copy via numpy, use a kernel via Queue
  - (Transitive) Closure of a vertex
    - Use a visitor and copy via numpy
- Depth First Search
  - Hell if I know
- Logic in computation
  - Limiters in FV methods
    - Can sometimes use tricks for branchless logic
  - Flux Corrected Transport for shock capturing
    - Maybe use WENO schemes which can be branchless
- Boundary conditions
  - Restrict branching to PETSc C numbering and assembly calls

- Audience???
Outline

1. Version Control
2. Configuration and Build
3. PETSc
4. numpy & sympy
5. PyCUDA
6. FEniCS
Version Control
  - Mercurial, Git, Subversion

Hosting
  - BitBucket, GitHub, Launchpad

Community involvement
  - arXiv, PubMed
Distributed Version Control

- CVS/SVN manage a single repository
  - Versioned data
  - Local copy for modification and checkin

- Mercurial manages many repositories
  - Identified by URLs
  - No one Master

- Repositories communicate by ChangeSets
  - Use push and pull to move changesets
  - Can move arbitrary changes with patch queues
Figure: Single Repository
Figure: Master Repository with User Clones
Figure: Project with Release and Bugfix Repositories
Outline

1. Version Control
2. Configuration and Build
3. PETSc
4. numpy & sympy
5. PyCUDA
6. FEniCS
Configuration and Build

“It won’t run on my iPhone”

- Portability
  - PETSc BuildSystem, autoconf

- Dependencies
  - Does this work with UnsupportedGradStudentAMG?

- Configurable build
  - Build must integrate with the configuration system
  - CMake, SCons
BuildSystem

Provides tools for Configuration and Build

- Dependency tracking and analysis
- Package management and hierarchy
- Library of standard tests
- Standard build rules
- Automatic package build and integration

http://petsc.cs.iit.edu/petsc/BuildSystem
http://petsc.cs.iit.edu/petsc/SimpleConfigure
BuildSystem.config.base configures a specific functionality

- **Entry points:**
  - setupHelp()
  - setupDependencies()
  - configure()

- **Built-in capabilities:**
  - Preprocessing, compilation, linking, running
  - Manages languages
  - Checks for executables

- **Output types:**
  - Define, typedef, or prototype
  - Make macro or rule
  - Substitution (old-style)
BuildSystem.config.framework manages the configure run

- Manages configure modules
  - Dependencies with DAG, `require()`
  - Options table
  - Initialization, run, cleanup

- Outputs
  - Configure headers and log
  - Make variable and rules
  - Pickled configure tree
BuildSystem.config.package manages other packages

- BuildSystem/config/packages/* examples (MPI, FIAT, etc.)
- Standard location and install hooks
- Standard header and library tests
- Uniform interface for parameter retrieval
- Special support for GNU packages
A module can declare a dependency using:

```python
fw = self.framework
self.mpi = fw.require('config.packages.MPI', self)
```

so that MPI is configured before `self`. Information is retrieved during `configure()`:

```python
if self.mpi.found:
    include.extend(self.mpi.include)
    libs.extend(self.mpi.lib)
```
A module can declare a dependency using:

```python
fw = self.framework
self.mpi = fw.require('config.packages.MPI', self)
```

so that MPI is configured before `self`. Information is retrieved during `configure()`:

```python
if self.mpi.found:
    include.extend(self.mpi.include)
    libs.extend(self.mpi.lib)
```
A build system can acquire the information using:

```python
class ConfigReader(script.Script):
    def __init__(self):
        import RDict
        argDB = RDict.RDict(None, None, 0, 0)
        argDB.saveFilename = os.path.join('path', 'RDict.db')
        argDB.load()
        script.Script.__init__(self, argDB = argDB)
        return

    def getMPIModule(self):
        self.setup()
        fw = self.loadConfigure()
        mpi = fw.require('config.packages.MPI', None)
        return mpi
```
GNU Make automates a package build

- Has a single predicate, older-than
- Executes shell code for actions
- PETSc has support for
  - configuration integration
  - automatic compilation

Alternatives
- SCons
- CMake
Simple replacement for GNU make

- Excellent configure integration
- User-defined predicates
- Dependency analysis and tracking
- Python actions
- Support for test execution
The simple interface handles the entire build:

`.config/builder.py`

A more flexible front end allows finer control:

`.config/builder2.py help [command]`
`.config/builder2.py clean`
`.config/builder2.py stubs fortran`
`.config/builder2.py build [src/snies/interface/sniesj.c]`
`.config/builder2.py check [src/snies/examples/tutorials/ex10.c]`
Testing

“They are identical in the eyeball norm”

- Unit tests
  - cppUnit

- Regression tests
  - buildbot

- Benchmarks
  - Cigma
Outline

1 Version Control

2 Configuration and Build

3 PETSc
   ● Traditional PETSc
   ● petsc4py

4 numpy & sympy

5 PyCUDA

6 FEniCS
3. **PETSc**
   - Traditional PETSc
   - petsc4py
How did PETSc Originate?

PETSc was developed as a Platform for **Experimentation**

We want to experiment with different

- Models
- Discretizations
- Solvers
- Algorithms
  - which blur these boundaries
The Role of PETSc

Developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort.

PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver, nor a silver bullet.

— Barry Smith
You want to think about how you decompose your data structures, how you think about them globally. [...] If you were building a house, you’d start with a set of blueprints that give you a picture of what the whole house looks like. You wouldn’t start with a bunch of tiles and say, “Well I’ll put this tile down on the ground, and then I’ll find a tile to go next to it.” But all too many people try to build their parallel programs by creating the smallest possible tiles and then trying to have the structure of their code emerge from the chaos of all these little pieces. You have to have an organizing principle if you’re going to survive making your code parallel.

(http://www.rce-cast.com/Podcast/rce-28-mpich2.html)
What is PETSc?

A freely available and supported research code for the parallel solution of nonlinear algebraic equations

Free
- Download from http://www.mcs.anl.gov/petsc
- Free for everyone, including industrial users

Supported
- Hyperlinked manual, examples, and manual pages for all routines
- Hundreds of tutorial-style examples
- Support via email: petsc-maint@mcs.anl.gov

Usable from C, C++, Fortran 77/90, Matlab, Julia, and Python
What is PETSc?

- Portable to any parallel system supporting MPI, including:
  - Tightly coupled systems
    - Cray XT6, BG/Q, NVIDIA Fermi, K Computer
  - Loosely coupled systems, such as networks of workstations
    - IBM, Mac, iPad/iPhone, PCs running Linux or Windows

PETSc History

- Begun September 1991
- Over 60,000 downloads since 1995 (version 2)
- Currently 400 per month

PETSc Funding and Support

- Department of Energy
  - SciDAC, MICS Program, AMR Program, INL Reactor Program
- National Science Foundation
  - CIG, CISE, Multidisciplinary Challenge Program
PETSc Timeline

- PETSc-1
  - Barry
  - Bill
  - Lois
  - Satish
  - Dinesh
  - Hong
  - Kris
  - Matt
  - Victor
  - Dmitry
  - Lisandro
  - Jed
  - Shri
  - Peter
  - 1991

- PETSc-2
  - MPI-1
  - 1995
  - MPI-2
  - 2000

- PETSc-3
  - 2005
  - 2010

- Traditional PETSc

M. Knepley
Robust
PASI '11
32 / 65
**What Can We Handle?**

- PETSc has run implicit problems with over **500 billion** unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

- PETSc has run on over **290,000** cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

- PETSc applications have run at 23% of peak (**600 Teraflops**)
  - Jed Brown on NERSC Edison
  - HPGMG code
What Can We Handle?

- **PETSc** has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

- **PETSc** has run on over 290,000 cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

- **PETSc applications** have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code
PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

PETSc has run on over 290,000 cores efficiently
  - UNIC on the IBM BG/P Jugene at Jülich
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

PETSc applications have run at 23% of peak (600 Teraflops)
  - Jed Brown on NERSC Edison
  - HPGMG code
Outline

3 PETSc
  • Traditional PETSc
  • petsc4py
**petsc4py** provides Python bindings for PETSc

- Provides **ALL** PETSc functionality in a **Pythonic** way
  - Logging using the Python `with` statement
- Can use Python callback functions
  - `SNESSetFunction()`, `SNESSetJacobian()`
- Manages all memory (creation/destruction)
- Visualization with `matplotlib`
PETSc  petsc4py

petsc4py Installation

- **Automatic**
  - `pip install -install-options=-user petsc4py`
  - **Uses** `$PETSC_DIR` and `$PETSC_ARCH`
  - **Installed into** `$HOME/.local`
  - **No additions to PYTHONPATH**

- **From Source**
  - `virtualenv python-env`
  - `source ./python-env/bin/activate`
  - **Now everything installs into your proxy Python environment**
  - `hg clone https://petsc4py.googlecode.com/hg petsc4py-dev`
  - `ARCHFLAGS="-arch x86_64" python setup.py sdist`
  - `ARCHFLAGS="-arch x86_64" pip install dist/petsc4py-1.1.2.tar.gz`
  - **ARCHFLAGS** only necessary on Mac OSX
petsc4py Examples

- externalpackages/petsc4py-1.1/demo/bratu2d/bratu2d.py
  - Solves Bratu equation (SNES ex5) in 2D
  - Visualizes solution with matplotlib

- src/ts/examples/tutorials/ex8.py
  - Solves a 1D ODE for a diffusive process
  - Visualize solution using -vec_view_draw
  - Control timesteps with -ts_max_steps
4 numpy & sympy
  ● numpy
  ● sympy
**numpy** is ideal for building Python data structures

- Supports multidimensional arrays
- Easily interfaces with C/C++ and Fortran
- High performance BLAS/LAPACK and functional operations
- Python 2 and 3 compatible
- Used by petsc4py to talk to PETSc
Outline

4. numpy & sympy
   - numpy
   - sympy
sympy is useful for symbolic manipulation

- Interacts with numpy
- Derivatives and integrals
- Series expansions
- Equation simplification
- Small and open source
Create the shifted polynomial

\[ \sum_{i=0}^{\text{order}} \frac{c_i}{i!} (x - a)^i \]

```python
def constructShiftedPolynomial(order):
    from sympy import Symbol, collect, diff, limit
    from sympy import factorial as f
    c = [Symbol('c' + str(i)) for i in range(order)]
    g = sum([c[i]*(x-a)**i/f(i) for i in range(order)])
    # Convert to a monomial
    g = collect(g.expand(), x)
    return c, g
```
Here is the shifted polynomial for order 5:

\[
\begin{align*}
c_0 & - a \cdot c_1 + c_2 \cdot a^{**2}/2 - c_3 \cdot a^{**3}/6 + c_4 \cdot a^{**4}/24 \\
+ x \cdot (c_1 & - a \cdot c_2 + c_3 \cdot a^{**2}/2 - c_4 \cdot a^{**3}/6) \\
+ x^{**2} \cdot (c_2/2 & - a \cdot c_3/2 + c_4 \cdot a^{**2}/4) \\
+ x^{**3} \cdot (c_3/6 & - a \cdot c_4/6) \\
+ c_4 \cdot x^{**4}/24
\end{align*}
\]
Construct matrix transform from

\[
\sum_{i=0}^{\text{order}} \frac{c_i}{i!} (x - a)^i \quad \text{to} \quad \sum_{i=0}^{\text{order}} \frac{c_i}{i!} x^i
\]

def constructTransformMatrix(order = 5):
    from sympy import diff, limit
    c, g = constructShiftedPolynomial(order, debug)
    M = []
    for o in range(order):
        exp = g.diff(x, o).limit(x, 0)
        M.append([exp.diff(c[p]) for p in range(order)])
    return M
Here is the transform matrix $M$:

$$
\begin{pmatrix}
1 & -a & \frac{a^2}{2} & -\frac{a^3}{6} & \frac{a^4}{24} \\
0 & 1 & -a & \frac{a^2}{2} & -\frac{a^3}{6} \\
0 & 0 & 1 & -a & \frac{a^2}{2} \\
0 & 0 & 0 & 1 & -a \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$
Python packages by Andreas Klöckner for embedded GPU programming

- Handles unimportant details automatically
  - CUDA compile and caching of objects
  - Device initialization
  - Loading modules onto card

- Excellent Documentation & Tutorial

- Excellent platform for Metaprogramming
  - Only way to get portable performance
  - Road to FLAME-type reasoning about algorithms
```c
void kernel(float *output) {
    float G[\$\{dim*dim\}] = \$\{', '.join(['3.0']*\(dim*dim\))\};
    float K[\$\{dim*dim\}] = \$\{', '.join(['3.0']*\(dim*dim\))\};
    float product = 0.0;
    const int Ooffset = gridIdx*\$\{numThreads\};

    // Contract G and K
    for n in range(numLocalElements):
        for alpha in range(dim):
            for beta in range(dim):
                gIdx = (n*dim + alpha)*dim + beta
                kIdx = alpha*dim + beta
                product += G[\$\{gIdx\}] * K[\$\{kIdx\}];
            endfor
        endfor
    endfor

    output[Ooffset+idx] = product;
}
```

We render code template into strings using a dictionary of inputs.

```python
args = {'dim': self.dim,
        'numLocalElements': 1,
        'numThreads': self.threadBlockSize}
kernelTemplate = self.getKernelTemplate()
gpuCode = kernelTemplate.render(isGPU = True, **args)
cpuCode = kernelTemplate.render(isGPU = False, **args)
```
```c
__global__ void kernel( float *output) {
    const int gridIdx = blockIdx.x + blockIdx.y * blockDim.x; // This is (i, j)
    const int idx = threadIdx.x + threadIdx.y * 1; // This is (i, j)
    float G[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
    float K[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
    float product = 0.0;
    const int Ooffset = gridIdx * 1;

    // Contract G and K
    product += G[0] * K[0];
    product += G[1] * K[1];
    product += G[4] * K[4];
    product += G[5] * K[5];
    product += G[7] * K[7];
    product += G[8] * K[8];
    output[Ooffset+idx] = product;
    return;
}
```
void kernel(int numInvocations, float *output) {
    for(int gridIdx = 0; gridIdx < numInvocations; ++gridIdx) {
        for(int i = 0; i < 1; ++i) {
            for(int j = 0; j < 1; ++j) {
                const int idx = i + j*1; // This is (i,j)
                float G[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
                float K[9] = {3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0,3.0};
                float product = 0.0;
                const int Ooffset = gridIdx*1;

                // Contract G and K
                product += G[0] * K[0];
                product += G[1] * K[1];
                product += G[4] * K[4];
                product += G[5] * K[5];
                product += G[7] * K[7];
                product += G[8] * K[8];
                output[Ooffset+idx] = product;
            }
        }
    }
    return;
}
Creating a Module

CPU:

```python
# Output kernel and C support code
self.outputKernelC(cpuCode)
self.writeMakefile()
out, err, status = self.executeShellCommand('make')
```

```
\bigskip
```

GPU:

```python
\begin{minted}{python}
from pycuda.compiler import SourceModule

mod = SourceModule(gpuCode)
self.kernel = mod.get_function('kernel')
self.kernelReport(self.kernel, 'kernel')
```

```
import pycuda.driver as cuda
import pycuda.autoinit

blockDim = (self.dim, self.dim, 1)
start = cuda.Event()
end = cuda.Event()
grid = self.calculateGrid(N, numLocalElements)
start.record()
for i in range(iter):
    self.kernel(cuda.Out(output),
                block = blockDim, grid = grid)
end.record()
end.synchronize()
gpuTimes.append(start.time_till(end)*1e-3/iters)
Outline

1. Version Control
2. Configuration and Build
3. PETSc
4. numpy & sympy
5. PyCUDA
6. FEniCS
Finite Element Integrator And Tabulator by Rob Kirby

http://www.fenics.org/fiat

FIAT understands

- Reference element shapes (line, triangle, tetrahedron)
- Quadrature rules
- Polynomial spaces
- Functionals over polynomials (dual spaces)
- Derivatives

User can build arbitrary elements specifying the Ciarlet triple \((K, P, P')\)

FIAT is part of the FEniCS project, as is the PETSc Sieve module
FFC is a compiler for variational forms by Anders Logg.

Here is a mixed-form Poisson equation:

$$ a((\tau, w), (\sigma, u)) = L((\tau, w)) \quad \forall (\tau, w) \in V $$

where

$$ a((\tau, w), (\sigma, u)) = \int_{\Omega} \tau \sigma - \nabla \cdot \tau u + w \nabla \cdot u \, dx $$

$$ L((\tau, w)) = \int_{\Omega} w f \, dx $$
shape = "triangle"

BDM1 = FiniteElement("Brezzi–Douglas–Marini", shape, 1)
DG0 = FiniteElement("Discontinuous Lagrange", shape, 0)

element = BDM1 + DG0
(tau, w) = TestFunctions(element)
(sigma, u) = TrialFunctions(element)

a = (dot(tau, sigma) - div(tau)*u + w*div(sigma)) * dx

f = Function(DG0)
L = w*f*dx
Here is a discontinuous Galerkin formulation of the Poisson equation:

\[ a(v, u) = L(v) \quad \forall v \in V \]

where

\[
\begin{align*}
a(v, u) &= \int_{\Omega} \nabla u \cdot \nabla v \, dx \\
+ &\sum_{S} \int_{S} - \langle \nabla v \rangle \cdot [[u]]_n - [[v]]_n \cdot \langle \nabla u \rangle - \left(\alpha/h\right) vu \, dS \\
+ &\int_{\partial\Omega} -\nabla v \cdot [[u]]_n - [[v]]_n \cdot \nabla u - \left(\gamma/h\right) vu \, ds \\
L(v) &= \int_{\Omega} vf \, dx
\end{align*}
\]
DG1 = FiniteElement("Discontinuous Lagrange", shape, 1)
v = TestFunctions(DG1)
u = TrialFunctions(DG1)
f = Function(DG1)
g = Function(DG1)
n = FacetNormal("triangle")
h = MeshSize("triangle")
a = dot(grad(v), grad(u)) * dx
   - dot(avg(grad(v)), jump(u, n)) * dS
   - dot(jump(v, n), avg(grad(u))) * dS
   + alpha / h * dot(jump(v, n) + jump(u, n)) * dS
   - dot(grad(v), jump(u, n)) * ds
   - dot(jump(v, n), grad(u)) * ds
   + gamma / h * v * u * ds
L = v * f * dx + v * g * ds
\[ \int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) d\mathbf{x} \]  

\begin{align*}
\text{element} & = \text{FiniteElement('Lagrange', tetrahedron, 1)} \\
v & = \text{TestFunction(element)} \\
u & = \text{TrialFunction(element)} \\
a & = \text{inner(grad(v), grad(u))}\times dx
\end{align*}
\[ \int_T \nabla \phi_i(x) \cdot \nabla \phi_j(x) dx \]  

\begin{verbatim}
element = FiniteElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(grad(v), grad(u))*dx
\end{verbatim}
\[ \frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : \left( \nabla \phi_j(x) + \nabla^T \phi_j(x) \right) \, dx \]  

```python
element = VectorElement( 'Lagrange' , tetrahedron , 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(sym(grad(v)), sym(grad(u))) * dx
```
\[
\frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : \left( \nabla \phi_j(x) + \nabla^T \phi_j(x) \right) \, dx
\]  

\text{element} = \text{VectorElement}(\text{'Lagrange'}, \text{tetrahedron}, 1) \\
v = \text{TestFunction}(\text{element}) \\
u = \text{TrialFunction}(\text{element}) \\
a = \text{inner} \left( \text{sym} \left( \nabla v \right), \text{sym} \left( \nabla u \right) \right) \ast dx
\[ \frac{1}{4} \int_{\mathcal{T}} \left( \nabla \vec{\phi}_i(x) + \nabla^T \vec{\phi}_i(x) \right) : C : \left( \nabla \vec{\phi}_j(x) + \nabla \vec{\phi}_j(x) \right) \, dx \] (3)

element = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
(dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i, j] * C[i, j, k, l] * sym(grad(u))[k, l] * dx

Currently broken in FEniCS release
\[
\frac{1}{4} \int_{\mathcal{T}} \left( \nabla \vec{\phi}_i(x) + \nabla^T \vec{\phi}_i(x) \right) : C : \left( \nabla \vec{\phi}_j(x) + \nabla^T \vec{\phi}_j(x) \right) \, dx
\]  

\begin{align*}
\text{element} & = \text{VectorElement}('\text{Lagrange}', \text{tetrahedron}, 1) \\
\text{cElement} & = \text{TensorElement}('\text{Lagrange}', \text{tetrahedron}, 1, (\text{dim}, \text{dim}, \text{dim}, \text{dim})) \\
v & = \text{TestFunction(element)} \\
u & = \text{TrialFunction(element)} \\
C & = \text{Coefficient(cElement)} \\
i, j, k, l & = \text{indices}(4) \\
a & = \text{sym(grad(v))[i, j]} \* C[i, j, k, l] \* \text{sym(grad(u))[k, l]} \, dx
\end{align*}
\[
\frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : C : \left( \nabla \phi_j(x) + \nabla \phi_j(x) \right) \, dx \tag{3}
\]

```python
element = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                        (dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i, j]*C[i, j, k, l]*sym(grad(u))[k, l]*dx
```

Currently broken in FEniCS release
from ffc.analysis import analyze_forms
from ffc.compiler import compute_ir

parameters = ffc.default_parameters()
parameters['representation'] = 'tensor'
analysis = analyze_forms([a,L], {}, parameters)
ir = compute_ir(analysis, parameters)

a_K = ir[2][0]['AK'][0][0]
a_G = ir[2][0]['AK'][0][1]

K = a_K.A0.astype(numpy.float32)
G = a_G
Usability is paramount
- Need community by-in
- Need complete workflow

Leverage existing systems
- Adoption is much easier with the familiar
  - arXiv, package managers