FEniCS, part I

M. M. Sussman

sussmanm@math.pitt.edu

Office Hours: 11:10AM-12:10PM, Thack 622

May 12 – June 19, 2014

Topics

Introduction

Tutorial examples

Solvers and preconditioners
- Direct solvers
- Conjugate Gradients
- Parallel computing
- Preconditioning
- Nonlinear solvers

FEniCS is a collection of “components.”

1. You write a script in high-level Python
   - Uses UFL form language
   - Can use numpy, scipy, matplotlib.pyplot, *etc.*
   - Can use Viper for plotting
2. DOLFIN interprets the script
3. UFL is passed to FFC for compilation
4. Instant turns it into C++ callable from Python (“swig”)
5. Linear algebra is passed to PETSc or UMFPACK

Other capabilities

- You can write your script in C++
- Other components are also available.
- Other linear algebra backends

DOLFIN classes

- `x = Vector()`
- `A = Matrix()`
- `solve(A, x, b)`
- Eigenvalues *via* SLEPc
- Newton solver for nonlinear equations
  - You write a class defining the problem and Jacobian
  - `newton_solver = NewtonSolver()`
  - `newton_solver.solve( ... )`
- Several mesh-generation commands
- Automated mesh-generation is available
- Meshes can come from files
- Easy to write files for ParaView plotting

Linear Algebra and parallel computing

- Linear algebra is where the “rubber meets the road”
- Real problems need good solvers
- Good solvers *must* be used intelligently
- Parallel operation is mostly transparent
- PETSc (plus add-ons SUPERLU, *etc.*)
- Trilinos

Topics

Introduction

Tutorial examples

Solvers and preconditioners
- Direct solvers
- Conjugate Gradients
- Parallel computing
- Preconditioning
- Nonlinear solvers

Resources

- FEniCS book, Chapter 1
- **`dolfin-get-demos`**
  - Creates directory **`$HOME/dolfin-demos`**, but I changed it to **`$HOME/fenics-demos`** for you
  - Demo files in it
  - Move it to where you like, if you like
- Examples for this course on my web pages.

Example 1: Poisson equation

- First tutorial example (**`d1_p2d.py`**)
- Poisson equation in 2D

$$-\Delta u = -\frac{\partial^2 u}{\partial x_0^2} - 2\frac{\partial^2 u}{\partial x_1^2} = -6$$

- Dirichlet boundary conditions

$$u_D = 1 + x_0^2 + 2x_1^2$$

- The solution is $u = 1 + x_0^2 + x_1^2$

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Always start with this

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- Mesh on $[0, 1] \times [0, 1]$
- Uniform 6 cells in x_0, 4 in x_1

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Linear Lagrange shape functions

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- "Expression" causes a compilation
- "Calling FFC just-in-time (JIT) compiler, this may take some time."
- **x** is a "global variable"

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- `on_boundary` is a “global” variable
- Value is True or False
- This is how b.c. are usually done

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- Set Dirichlet b.c.
- Can be more than one boundary
- `u0_boundary` is used

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- This is UFL
- Specify weak form
- “Calling FFC just-in-time (JIT) compiler, this may take some time.”

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Define trial and test function spaces

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Define $L(v) = \int f(x)v(x)\,dx$ with $f = -6$

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Define $a(u, v) = \int \nabla u \cdot \nabla v \, dx$

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

`nabla_grad(u)` is $\partial u_i/\partial x_j$ while `grad` is its transpose. For scalars in Python, they agree, not otherwise.

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

u is *redefined* as a **Function** instead of **TrialFunction**

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Solve the system $L(v) = a(u, v)\ \forall v$ subject to boundary conditions.

example1.py

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- Plot u and mesh in two frames.
- `interactive=True` causes the plot to remain displayed until destroyed by mouse.
- Can also put `interactive()` at the end.

ParaView plotting

Add code:

```
if True:
    # Dump solution to file in VTK format
    file = File('poisson.pvd')
    file « u
```

DEMONSTRATION

`example2.py`: like FEM1D

- Change mesh definition to **`mesh=UnitIntervalMesh(5)`**, nothing else!
- Get Poisson equation in 1D!
- Eliminate Dirichlet b.c.
- Modify weak form

Code comparison

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), \
          nabla_grad(v))*dx

L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)
```

```
from dolfin import *

# Create mesh and define function space
N=10
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, 'Lagrange', 2)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Expression('x[0]+2')
a = (-inner(nabla_grad(u), nabla_grad(v))
    + 2*grad(u)[0]*v \
    + u*v)*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u)
```

Compare with exact solution

```
# exact for comparison
exact=Expression("(1+x[0])*exp(1-x[0])+x[0]*(1-exp(-x[0]))")
```

```
Expression uses C++ syntax!  Use pow(a,b)
instead of a**b!
```

Compare with exact solution

```
# exact for comparison
exact=Expression("(1+x[0])*exp(1-x[0])+x[0]*(1-exp(-x[0]))")

# let's get coordinates, x, at the DOF locations
exF = Expression("x[0]")
exvector = interpolate(exF,V).vector().get_local()
```

Compare with exact solution

```
# exact for comparison
exact=Expression("(1+x[0])*exp(1-x[0])+x[0]*(1-exp(-x[0]))")

# let's get coordinates, x, at the DOF locations
exF = Expression("x[0]")
exvector = interpolate(exF,V).vector().get_local()

# remember that u and exact are functions
sumsq0=0.
sumsq1=0.
for i in range(exvector.size):
    print "x=", exvector[i], " u=", u(exvector[i]), \
          " uexact=", exact(exvector[i])
    sumsq0+=(u(exvector[i])-exact(exvector[i]))**2
    sumsq1+=u(exvector[i])**2

sumsq0 = sqrt(sumsq0)
sumsq1 = sqrt(sumsq1)
relerr0 = sumsq0/sumsq1

print "N=",N," relative 2-norm error=",relerr0
```

Convergence in Example2

```
N= 10  relative 2-norm error= 3.21246743999e-07
N= 20  relative 2-norm error= 2.01845561365e-08
N= 40  relative 2-norm error= 1.23303254663e-09
N= 80  relative 2-norm error= 9.04294475874e-11
```

Looks like $O(h^4)$, faster than theory. Probably because mesh is uniform.

Printing the solution

- Code for printing solution and computing error is complicated
- If not printing, don't need coordinates
- could just do:

```
import scipy.linalg as la
u_array = u.vector().array()
u_e = interpolate(exact, V)
u_e_array = u_e.vector().array()
relerr1 = la.norm(u_e_array - u_array) / \
             la.norm(u_e_array)
print "N=",N," relative 2-norm error=",relerr1
```

3D is just as easy, `example3.py`

- Change the Mesh to **`UnitCubeMesh`**
- Change the solver

Exercise 14 (5 points)

`example3` is a 3D Poisson equation for which the solution is known. Add code similar to the 1D code in **`example2`** to compute the error as the norm $\|u - u_{\text{exact}}\|/\|u_{\text{exact}}\|$ and print it at the end of the program. Be sure you are using *quadratic* Lagrange elements, and your error should be of roundoff size because the exact solution is in the approximation space.

The curse of dimensionality

- Solving 50 1D
  - 101×101 system with 401 nonzeros
  - Essentially no time to solve using default LU solver
- Solving 50×50 2D
  - 2601×2601 system, 17801 nonzeros
  - Less than 0.1 sec using default LU solver.
- Solving $50 \times 50 \times 50$ system 3D
  - $132,651 \times 132,651$ system
  - 1,927,951 nonzeros
  - Out of memory with default LU solver (UMFPACK)
  - Conjugate gradient with block jacobi (ilu) preconditioner
  - Less than a second to solve on my laptop

Higher accuracy costs you

- Solving $50 \times 50 \times 50$ system 3D, linear Lagrange
  - $132,651 \times 132,651$ system
  - 1,927,951 nonzeros
  - Conjugate gradient with block jacobi (ilu, "preonly") preconditioner
  - Less than a second to solve on my laptop
- Solving $50 \times 50 \times 50$ system 3D, quadratic Lagrange
  - $1,030,301 \times 1,030,301$ system
  - 29,096,201 nonzeros
  - Conjugate gradient with block jacobi (ilu, "preonly") preconditioner
  - 16.5 seconds to solve on my laptop (109 iterations)
  - 10 seconds to solve using 2 processors (120 iterations)

Topics

Introduction

Tutorial examples

Solvers and preconditioners

- Direct solvers
- Conjugate Gradients
- Parallel computing
- Preconditioning
- Nonlinear solvers

Topics

Introduction

Tutorial examples

Solvers and preconditioners

- Direct solvers
- Conjugate Gradients
- Parallel computing
- Preconditioning
- Nonlinear solvers

Why did LU run out of memory?

$$
\begin{pmatrix}
x & x & & & & x & & & & \\
x & x & x & & & & x & & & \\
& x & x & x & & & & x & & \\
& & x & x & x & & & & x & \\
& & & x & x & x & & & & x \\
x & & & & x & x & x & & & \\
& x & & & & x & x & x & & \\
& & x & & & & x & x & x & \\
& & & x & & & & x & x & x \\
& & & & x & & & & x & x
\end{pmatrix}
$$

Why did LU run out of memory?

$$
\begin{pmatrix}
x & x & & & & x & & & & \\
x & x & x & & & x & x & & & \\
& x & x & x & & x & x & x & & \\
& & x & x & x & x & x & x & x & \\
& & & x & x & x & x & x & x & x \\
x & x & x & x & x & x & x & x & x & x \\
& x & x & x & x & x & x & x & x & x \\
& & x & x & x & x & x & x & x & x \\
& & & x & x & x & x & x & x & x \\
& & & & x & x & x & x & x & x
\end{pmatrix}
$$

Some memory numbers

- $N \times N$ square mesh
  - Matrix is $N^2 \times N^2$, half-bandwidth N
  - N^3 nonzeros each factor
- $N \times N \times N$ cube mesh
  - Matrix is $N^3 \times N^3$, half-bandwidth N^2
  - N^5 nonzeros each factor

Direct or iterative?

- 1D or 2D problem: direct solver
- Maybe run into trouble with large 2D problems
- 3D: debug with direct, go with iterative

Topics

Introduction

Tutorial examples

Solvers and preconditioners
- Direct solvers
- Conjugate Gradients
- Parallel computing
- Preconditioning
- Nonlinear solvers

Conjugate Gradient algorithm

Given SPD matrix A, initial guess x_0

$r^0 = b - Ax^0$
$d^0 = r^0$
for n in range(itmax):
  $\alpha_{n-1} = (d^{n-1}, r^{n-1})/(d^{n-1}, Ad^{n-1}) = (r^n, r^n)/(d^n, Ad^n)$
  $x^n = x^{n-1} + \alpha_{n-1} d^{n-1}$
  $r^n = b - Ax^n = r^{n-1} - \alpha_{n-1} Ad^{n-1}$
  if converged**:**
    return x
  $\beta_n = (r^n, r^n)/(r^{n-1}, r^{n-1})$
  $d^n = r^n + \beta_n d^{n-1}$

Solution as minimization

- Solving $Ax = b$ is the same as
- Minimizing $\|Ax - b\|^2$ is the same as
- Minimizing $2(Ax, x) - (b, y)$ (SPD A)
- Finding $(Ax, y) = (b, y), \quad \forall y$

Iteration as successive minimization

- If A is $N \times N$ SPD and a sequence of subspaces

$$K^n \subset `\mathbb{R}^N$$

  is available.
- An iteration can be defined as

$$x^n = \min_{x \in K^n} (2(Ax, x) - (b, x)).$$

- Clearly, this will converge in N or fewer steps.
- Given an initial vector x^0, you can define a sequence of “Krylov” spaces with r^0=$Ax^0 - b$ with

$$\begin{aligned} r^n &\in r^0 + \operatorname{span}\{Ar^0, A^2r^0, \dots, A^nr^0\}, \text{ and} \\ e^n &\in e^0 + \operatorname{span}\{Ar^0, A^2r^0, \dots, A^nr^0\} \end{aligned}$$

  where $e^n = x^n - x^\infty$.

CG as minimization

- Amazing! CG defined earlier satisfies
- $\|r^n\|_{A^{-1}} = \min_{r \in K^n} \|r\|_{A^{-1}}$
- $\|e^n\|_A = \min_{e \in K^n} \|e\|_A$
- $J(x) = (Ax, x) - (b, x)$, then $J(x^n) = \min_{x \in K^n} J(x)$
- Furthermore,

$$
\begin{aligned}
(r^k, r^j) &= 0, \qquad k \neq j \\
(d^k, d^j)_A &= 0, \qquad k \neq j
\end{aligned}
$$

- Proofs are by induction.

Consequences of CG

- Orthogonality relations $\Longrightarrow A^n r^0$ independent
- Sensitive to roundoff error
- Still rapidly convergent

Important observations

- Never need anything from the matrix except the product Ax
- Sharp contrast with factorization methods.
- “Matrix-free” methods: elementwise multiplication

What if not SPD?

- Replace “automatic orthogonality” with Gram-Schmidt
- GMRES
- Cannot store all the iterates
- “Restart” every 30 or so iterations.

There are many other possibilities, too

How do I know when to stop iterating?

- Watch $\|r^n\|$ and/or $\|x^n - x^{n-1}\|$
- Rates are important
  - Ideally, $\|x^{n+1} - x^n\|/\|x^n - x^{n-1}\| \to \rho < 1$
  - Then $\|x^{n+1} - x^\infty\| \approx \|x^{n+1} - x^n\|/(1 - \rho)$
- Domain-specific knowledge to estimate condition number
- "Model" problems give guidance

Solution methods in FEniCS

```
list_linear_solver_methods()
```

Solver	description
default	default linear solver (UMFPACK)
umfpack	UMFPACK
mumps	MUMPS
petsc	PETSc builtin LU solver
cg	Conjugate gradient
gmres	Generalized minimal residual
minres	Minimal residual
tfqmr	Transpose-free quasi-minimal residual
richardson	Richardson
bicgstab	Biconjugate gradient stabilized

- UMFPACK: Unsymmetric MultiFrontal sparse LU factorization
- MUMPS: MUltifrontal Massively Parallel Sparse direct Solver

Topics

Introduction

Tutorial examples

Solvers and preconditioners

- Direct solvers
- Conjugate Gradients
- Parallel computing
- Preconditioning
- Nonlinear solvers

References on parallel computing

Three texts are recommended. All are excellent sources.

- William Gropp, Ewing Lusk, Anthony Skjellum, *Using MPI, Portable Parallel Programming with the Message-Passing Interface*, Second Edition, MIT Press, Cambridge, MA, 1999, ISBN 0-262-57134-X.
- Ian Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1994.
  `http://www-unix.mcs.anl.gov/dbpp/text/book.html`
- Craig C. Douglas, Gundolf Haase, Ulrich Langer, *A Tutorial on Elliptic PDE Solvers and Their Parallelization,* SIAM, Philadelphia, PA, 2003, ISBN 0-89871-541-5

Computers are never big enough or fast enough!

- Moore's Law: Chip density doubles every 18 months.
- Speed increases with chip density in part because signals have less distance to travel.
- Even with PCs, the faster the CPU chip, the more stuff you cram into the operating system.
- When you run a problem, you always pick it so that it includes everything you can think of but runs in acceptable time. In order to do so, you always leave something out. Some problems end up being perhaps ten times too small, others end up being 10^7 times too small.
- Daily weather forcasts *must* run in less than a day. If you get a new computer and it is twice as fast, you increase your local coverage just far enough so it runs in just less than a day. You get more accuracy, but you still miss things.
- Some problems (turbulent fluid flow) require mesh sizes small enough to resolve the turbulence details (fraction of a millimeter) but enough mesh elements to cover, say, a whole airplane (tens of meters).

Hook a bunch of computers together

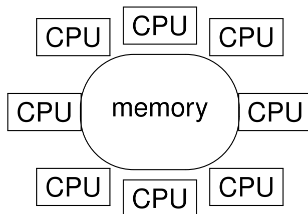
- Gropp, Lusk, Skjellum: “To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic ox.” This is true even when oxen double in capacity every eighteen months.
- The fastest and most advanced single-CPU computers are the most expensive, too. (*ibid.*, “Large oxen are expensive.”)
- Why not hook a bunch of smaller, cheaper computers together and have them all work together?
- Good idea! Except *how* can you make them all work together?
- Parallel computing.

There are two major classes of parallel computer

1. Shared memory
2. Distributed memory

Classification by memory access: shared memory

A "shared memory" parallel computer (SMP) is a set of CPUs, all sharing the same memory space.



Current "multi-core" chips have adopted this architecture with several cpus on a single chip, all accessing the same memory.

Advantages and disadvantages of shared memory (following Douglas, Haase, Langer)

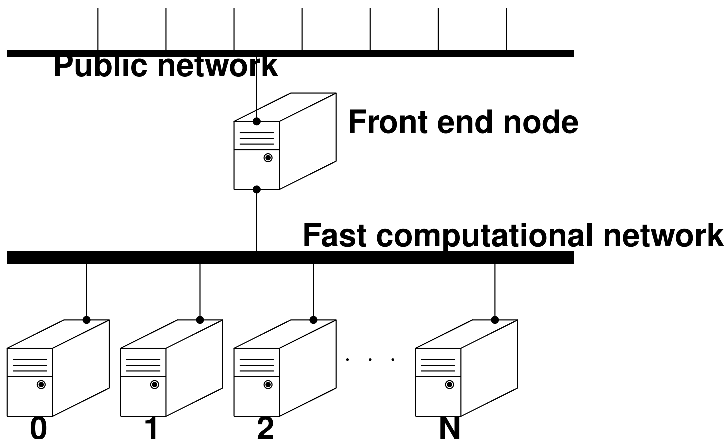
+ Each process has access to all the data.
+ Sequential code can easily be ported.
+ Speedup factors of a few 10s
- Memory bandwidth per CPU can suffer.
- Cache-coherence is a stumbling block.
- Poor scalability
- Memory subsystem is very expensive.

Shared memory for you

- Today, numerical libraries are optimized to use several processors (cores) if available.
- You don't have to do much yourself to take advantage of them.

Classification by memory access: distributed memory

Ideally speaking, a “distributed memory” parallel computer is a set of serial computers connected together with a communication facility such as a network:



Advantages and disadvantages of distributed memory (from Douglas, Haase, Langer)

+ There are no access conflicts since all data is locally stored.
+ Inexpensive hardware. (Network)
+ Code for such computers scales very well.
+ Need more memory? Buy another node!
- Sequential code does not run because one processor cannot see another's data.
- Special parallel code is difficult.
- ▶ The ratio between arithmetic and communication must be balanced.
- ▶ Packages such as **`PETSc`** can do much of the work for you.

“Message passing” computing model

- Very like distributed memory computer
- Easily implemented on shared memory computers
- Each process has local memory and a way to send data to others.
- Data transfer requires cooperation between sender and receiver.

“Single Program Multiple Data”

- Same program runs on all processors
- Relatively few special cases

Message Passing Interface (MPI) (Foster, Chapter 8)

- Set of Fortran (C, C++) subroutines to implement message passing.
- Not ISO/ANSI standard because of the cost, but is standardized.
- `http://www.mpi-forum.org` for standards
- `http://www-unix.mcs.anl.gov/mpi/www` for descriptions of all MPI functions and subroutines.
- MPI: The Complete Reference by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra is at `http://www.netlib.org/utk/papers/mpi-book/mpi-book.ht`
- William Gropp, Ewing Lusk, Anthony Skjellum, *Using MPI, Portable Parallel Programming with the Message-Passing Interface*, Second Edition, MIT Press, Cambridge, MA, 1999, ISBN 0-262-57134-X.

MPI Basics

```
MPI.COMM_WORLD  Communications group
comm.Get_size() Get the number of processors
comm.Get_rank() Get the rank (number) of this process
comm.Barrier() Synchronize processes
```

MPI sending and receiving

`comm.Send(ndarray,dest,tag)`
    `Send a message`
`comm.Recv(ndarray,source,tag)`
    `Receive a message (wait)`
`comm.Bcast(ndarray,root)`
    `Broadcast data from root to all processes`
`comm.Reduce(ndarray to send,ndarray to recv,op,root)`
    `Arithmetic reduction over all processes`

My first parallel program

Here is a program to print “Hello world” from each process running in parallel.

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print "hello from process ",rank," of total of ",size
```

Get MPI module

My first parallel program

Here is a program to print “Hello world” from each process running in parallel.

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print "hello from process ",rank," of total of ",size
```

Use all available processes

My first parallel program

Here is a program to print “Hello world” from each process running in parallel.

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print "hello from process ",rank," of total of ",size
```

What is the number of *this process*?

My first parallel program

Here is a program to print “Hello world” from each process running in parallel.

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print "hello from process ",rank," of total of ",size
```

What is the total number of processes?

My first parallel program

Here is a program to print “Hello world” from each process running in parallel.

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print "hello from process ",rank," of total of ",size
```

Each process prints something!

Run it

```
$ python mpihello.py
hello from process  0  of total of  1
```

Run it

```
$ python mpihello.py
hello from process  0  of total of  1

$ mpirun -np 2 python mpihello.py
hello from process  0  of total of  2
hello from process  1  of total of  2
```

Run it

```
$ python mpihello.py
hello from process  0  of total of  1

$ mpirun -np 2 python mpihello.py
hello from process  0  of total of  2
hello from process  1  of total of  2

$ mpirun -np 4 python mpihello.py
hello from process  2  of total of  4
hello from process  3  of total of  4
hello from process  0  of total of  4
hello from process  1  of total of  4
```

Run it

```
$ python mpihello.py
hello from process  0  of total of  1

$ mpirun -np 2 python mpihello.py
hello from process  0  of total of  2
hello from process  1  of total of  2

$ mpirun -np 4 python mpihello.py
hello from process  2  of total of  4
hello from process  3  of total of  4
hello from process  0  of total of  4
hello from process  1  of total of  4

$ mpirun -np 4 python mpihello.py
hello from process  0  of total of  4
hello from process  3  of total of  4
hello from process  1  of total of  4
hello from process  2  of total of  4
```

Remarks on `mpi_hello.py`

```
mpirun -np 4 python mpi_hello.py
```

- 4 separate copies of python
- Variable `rank` is distinct on each copy
- No order of execution

Approximating π with numerical integration

The value of π can be found many different ways. One way is to compute the integral

$$I = \int_0^1 \frac{4}{1+x^2}\,dx = 4\tan^{-1}(1)$$

by some numerical method, such as the rectangle rule with n intervals

$$I \approx \sum_{i=1}^{n} \frac{4h}{1+x_i^2}$$

where $h = 1/n$ and $x_i = h(2i-1)/2$.

A parallel program for approximating π

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

NumIntervals = 100000

h = 1.0/NumIntervals
myPieceOfPi = np.array([0.0])
for i in range(rank,NumIntervals,size):
    x= h * (i - 0.5)       #center of interval
    myPieceOfPi += 4.0*h / (1.0 + x**2)

wholePi = np.empty(1)
comm.Reduce(myPieceOfPi, wholePi, op=MPI.SUM, root=0)

if rank == 0:
    print "Pi=", wholePi, " error=", np.abs(wholePi - np.pi)
```

Splits up the work among `size` processes

A parallel program for approximating π

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

NumIntervals = 100000

h = 1.0/NumIntervals
myPieceOfPi = np.array([0.0])
for i in range(rank,NumIntervals,size):
    x= h * (i - 0.5)      #center of interval
    myPieceOfPi += 4.0*h / (1.0 + x**2)

wholePi = np.empty(1)
comm.Reduce(myPieceOfPi, wholePi, op=MPI.SUM, root=0)

if rank == 0:
    print "Pi=", wholePi, " error=", np.abs(wholePi - np.pi)
```

Sum individual `myPieceOfPi` into `wholePi`

A parallel program for approximating π

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

NumIntervals = 100000

h = 1.0/NumIntervals
myPieceOfPi = np.array([0.0])
for i in range(rank,NumIntervals,size):
    x= h * (i - 0.5)      #center of interval
    myPieceOfPi += 4.0*h / (1.0 + x**2)

wholePi = np.empty(1)
comm.Reduce(myPieceOfPi, wholePi, op=MPI.SUM, root=0)

if rank == 0:
    print "Pi=", wholePi, " error=", np.abs(wholePi - np.pi)
```

- Only want one print
- Only valid on processor 0 anyhow

A parallel program for approximating π

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

NumIntervals = 100000

h = 1.0/NumIntervals
myPieceOfPi = np.array([0.0])
for i in range(rank,NumIntervals,size):
    x= h * (i - 0.5)       #center of interval
    myPieceOfPi += 4.0*h / (1.0 + x**2)

wholePi = np.empty(1)
comm.Reduce(myPieceOfPi, wholePi, op=MPI.SUM, root=0)

if rank == 0:
    print "Pi=", wholePi, " error=", np.abs(wholePi - np.pi)
```

Variables in messages need to be contiguous arrays

`mpi_pi_sr.py` with sends and receives

It is far more common to send and receive messages than to broadcast them. The following example replaces the reduce with sends and receives.

Each worker sends result to leader.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

NumIntervals = 100000

h=1.0/NumIntervals
myPieceOfPi=np.array([0.0])
for i in range(rank,NumIntervals,size):
    x=h*(i-0.5) #center of interval
    myPieceOfPi+=4.0*h/(1.0+x**2)

leader=0
if rank == leader:
    # assuming leader is 0,
    wholePi = myPieceOfPi.copy()
    for n in range(1,size):
        comm.Recv(myPieceOfPi,source=n)
        wholePi += myPieceOfPi
    print "Pi=",wholePi," error=",np.abs(wholePi-np.pi)
else:
    comm.Send(myPieceOfPi,dest=leader)
```

`mpi_pi_sr.py` with sends and receives

It is far more common to send and receive messages than to broadcast them. The following example replaces the reduce with sends and receives.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

NumIntervals = 100000

h=1.0/NumIntervals
myPieceOfPi=np.array([0.0])
for i in range(rank,NumIntervals,size):
    x=h*(i-0.5) #center of interval
    myPieceOfPi+=4.0*h/(1.0+x**2)

leader=0
if rank == leader:
    # assuming leader is 0,
    wholePi = myPieceOfPi.copy()
    for n in range(1,size):
        comm.Recv(myPieceOfPi,source=n)
        wholePi += myPieceOfPi
    print "Pi=",wholePi," error=",np.abs(wholePi-np.pi)
else:
    comm.Send(myPieceOfPi,dest=leader)
```

Leader receives messages from other processes *in order*, then prints combined result.

mpi_pi_sr.py with sends and receives

It is far more common to send and receive messages than to broadcast them. The following example replaces the reduce with sends and receives.

Without **copy()**, there is a bug.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

NumIntervals = 100000

h=1.0/NumIntervals
myPieceOfPi=np.array([0.0])
for i in range(rank,NumIntervals,size):
    x=h*(i-0.5) #center of interval
    myPieceOfPi+=4.0*h/(1.0+x**2)

leader=0
if rank == leader:
    # assuming leader is 0,
    wholePi = myPieceOfPi.copy()
    for n in range(1,size):
        comm.Recv(myPieceOfPi,source=n)
        wholePi += myPieceOfPi
    print "Pi=",wholePi," error=",np.abs(wholePi-np.pi)
else:
    comm.Send(myPieceOfPi,dest=leader)
```

Comments on `mpi_pi_sr.py`

- Note that we have `size-1` sends in an explicit loop, but the *same number of receives* are not in a loop. The sends and receives *must* come in pairs.
- If you try to receive a message you have not sent, the receiving process will "block" until the message is sent (possibly forever).
- You will not notice if an extra message is sent.
- If one of the MPI functions fails and its process stops, the other processes won't automatically know about it. Ultimately, the processes will end up waiting for messages the stopped process never sends.

Exercise 15 (10 points)

Write `mpi4py` program that runs with two processes. Have process 0 first print the word "ping" and then send a message to process 1. When process 1 gets its message, have it print the word "pong" and send a message back to process 0. Repeat the cycle 5 times. The output should be the words "ping" and "pong" in sequence, 5 times.

Topics

Introduction

Tutorial examples

Solvers and preconditioners
- Direct solvers
- Conjugate Gradients
- Parallel computing
- Preconditioning
- Nonlinear solvers

Preconditioning

- Instead of solving $Ax = b$, solve $M^{-1}Ax = M^{-1}b$
- Really solving $(M^{-1}AM)(M^{-1}x) = M^{-1}b$
- Pick M^{-1} "close to" A^{-1}
- Can substantially reduce the number of iterations
- M^{-1} must be "easy" to gain anything

ICCG

- Suppose A is SPD
- A Cholesky factorization $A = LL^T$ exists where L is lower-triangular
- Lower-triangular matrices are easy to invert
- Fill-in eats you up
- Only keep entries in L corresponding to nonzeros in A (HOW?)
- “Incomplete” Cholesky
- Use it if you can!

Preconditioned CG

Given a SPD matrix A, preconditioner M, initial guess vector x^0, right side vector b, and maximum number of iterations **itmax**

$r^0 = b - Ax^0$
Solve $Md^0 = r^0$
$z^0 = d^0$
for n in range(itmax):
  $\alpha_n = (r^n, z^n)/(d^n, Ad^n)$
  $x^{n+1} = x^n + \alpha_n d^n$
  $r^{n+1} = b - Ax^{n+1}$
  if converged**:**
    return x
  Solve $Mz^{n+1} = r^{n+1}$
  $\beta_{n+1} = (r^{n+1}, z^{n+1})/(r^n, z^n)$
  $d^{n+1} = z^{n+1} + \beta_{n+1} d^n$

- Solve $Mz = r$ *iteratively*
- Iterations inside iterations

Preconditioners in parallel context

- Problem partitioned generally reside on single processor
- Common preconditioner strategy: respect the parallel partitioning
- “Block Jacobi” preconditioning
- “Additive Schwarz” preconditioning

How does Block Jacobi preconditioning work?

Suppose there are 3 processes. Write M as

$$M = \begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{pmatrix}$$

Block Jacobi iteration is

$$\begin{pmatrix} M_{11} & 0 & 0 \\ 0 & M_{22} & 0 \\ 0 & 0 & M_{33} \end{pmatrix} \begin{pmatrix} X_1^{n+1} \\ X_2^{n+1} \\ X_3^{n+1} \end{pmatrix} =$$

$$\begin{pmatrix} 0 & M_{12} & M_{13} \\ M_{21} & 0 & M_{23} \\ M_{31} & M_{32} & 0 \end{pmatrix} \begin{pmatrix} X_1^{n} \\ X_2^{n} \\ X_3^{n} \end{pmatrix} + \begin{pmatrix} B_1 \\ B_2 \\ B_3 \end{pmatrix}$$

On each process, solve a system

- Off-process variables are computed once each iteration
- Diagonal block M_{kk} equations are solved
- All solver arithmetic is on-block
- Might not solve with M_{kk} but with an incomplete factorization
- **`bjacobi`** using **`ilu`** (default in FEniCS)
- Additive Schwarz is similar
- Native PETSc precondioners respect the partitioning

Alternating Schwarz method

- Want to solve $N \times N$ system $Au = f$
- Decompose whole domain into J *overlapping* sub-domains.
- Diagonal block on domain j is $n \times n$ matrix A_j
- Split iteration into steps

$$u^{(n+j/J)} = u^{(n+(j-1)/J)} + R_j^T A_j^{-1} R_j (f - Au^{(n+(j-1)/J)})$$

  where R_j represents the restriction to domain j.
- Whole step can be written

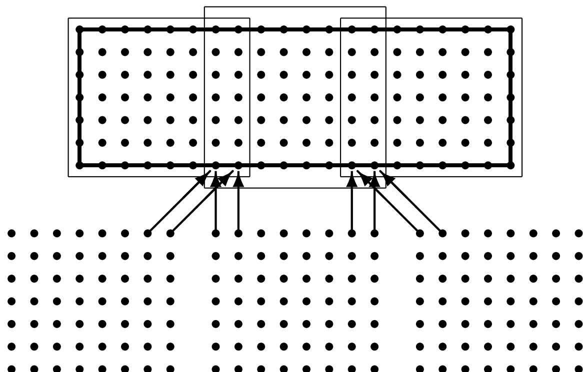
$$u^{(j+1)} = u^{(j)} + M^{-1}(f - Au^{(j)})$$

- "Multiplicative Schwarz" method amounts to

$$M_{\mathrm{MS}}^{-1} = [I - \prod_j (I - R_j^T A_j^{-1} R_j A)] A^{-1}$$

Visualizing overlapping blocks

Imagine a long problem broken horizontally into overlapping blocks with each block assigned to a processor.



Additive Schwarz theory

- Additive Schwarz

$$M_{\mathrm{AS}}^{-1} = \sum_j R_j^T A_j^{-1} R_j$$

- Won't usually converge because common (overlapping) values updated repeatedly
- Restricted Additive Schwarz

$$M_{\mathrm{RAS}}^{-1} = \sum_j \overline{R}_j^T A_j^{-1} R_j$$

- Makes sense *either* with subdomains defined physically or according to parallel data distribution
- Amount of overlap can be a parameter.
- Overlapped points often called "ghost" points in the redundant block.

Other preconditioners

- Multigrid
  - Presented later
- Incomplete LU across all processes

Preconditioners in FEniCS

```
list_krylov_solver_preconditioners()
```

preconditioner	name
default	default preconditioner (bjacobi + ilu)
none	No preconditioner
ilu	Incomplete LU factorization
icc	Incomplete Cholesky factorization
sor	Successive over-relaxation
petsc_amg	PETSc algebraic multigrid
jacobi	Jacobi iteration
bjacobi	Block Jacobi iteration
additive_schwarz	Additive Schwarz
amg	Algebraic multigrid
hypre_amg	Hypre algebraic multigrid (BoomerAMG)
hypre_euclid	Hypre parallel incomplete LU factorization
hypre_parasails	Hypre parallel sparse approximate inverse

Topics

Introduction

Tutorial examples

Solvers and preconditioners

Direct solvers

Conjugate Gradients

Parallel computing

Preconditioning

Nonlinear solvers

Newton's method

$$u^{(k+1)} = u^{(k)} - J^{-1} f(u^{(k)})$$

Quick and dirty derivation

$$\frac{f(u^{(k)} + \Delta u) - f(u^{(k)})}{\Delta u} \approx f'(u^{(k)})$$

If f were linear, this would be true equality, and the next iterate would satisfy $f(u^{(k+1)}) = f(u^{(k)} + \Delta u) = 0$. Rearrainging terms gives

$$\Delta u = u^{(k+1)} - u^{(k)} = -f(u^k)/f'(u^{(k)})$$

Facts about Newton iterations

- Convergence is (usually) quadratic: $\|\Delta u^{(k+1)}\| \approx \|\Delta u^{(k)}\|^2$
- "Radius of convergence" can be small
- Must have the Jacobian!

newton.py for Newton iterations

```
import numpy as np
import scipy.linalg as la
import copy

def ex1(x):
    """
    intersection of circle
    and parabola
    """
    assert(len(x) == 2)
    f=np.empty(2)
    fprime=np.empty([2,2])
    f[0]=x[0]**2+x[1]**2-1.0
    f[1]=x[1]-x[0]**2
    fprime[0,0]=2.0*x[0]
    fprime[0,1]=2.0*x[1]
    fprime[1,0]=-2.0*x[0]
    fprime[1,1]=1.0
    return f,fprime
```

```
def newton(f,xin):
    """
    Newton's method
    the function f returns the pair (f,fprime)
    """
    EPSILON = 1.0e-10
    x = copy.deepcopy(xin)
    # usually converges in <= 100 iterations
    for n in range(100):
        value,derivative = f(x)
        increment = la.solve(derivative,value)
        x -= increment
        errorEstimate = la.norm(increment)/la.norm(x)
        print "errorEstimate = ",errorEstimate
        if errorEstimate < EPSILON:
            return x,n
    assert(False)

y,i = newton(ex1,np.array([10.,10.]))
print "y = ",y," i= ",i
```

`newton.py` for Newton iterations

```
import numpy as np
import scipy.linalg as la
import copy

def ex1(x):
    """
    intersection of circle
    and parabola
    """
    assert(len(x) == 2)
    f=np.empty(2)
    fprime=np.empty([2,2])
    f[0]=x[0]**2+x[1]**2-1.0
    f[1]=x[1]-x[0]**2
    fprime[0,0]=2.0*x[0]
    fprime[0,1]=2.0*x[1]
    fprime[1,0]=-2.0*x[0]
    fprime[1,1]=1.0
    return f,fprime
```

```
def newton(f,xin):
    """
    Newton's method
    the function f returns the pair (f,fprime)
    """
    EPSILON = 1.0e-10
    x = copy.deepcopy(xin)
    # usually converges in <= 100 iterations
    for n in range(100):
        value,derivative = f(x)
        increment = la.solve(derivative,value)
        x -= increment
        errorEstimate = la.norm(increment)/la.norm(x)
        print "errorEstimate = ",errorEstimate
        if errorEstimate < EPSILON:
            return x,n
    assert(False)

y,i = newton(ex1,np.array([10.,10.]))
print "y = ",y," i= ",i
```

newton.py for Newton iterations

```
import numpy as np
import scipy.linalg as la
import copy

def ex1(x):
    """
    intersection of circle
    and parabola
    """
    assert(len(x) == 2)
    f=np.empty(2)
    fprime=np.empty([2,2])
    f[0]=x[0]**2+x[1]**2-1.0
    f[1]=x[1]-x[0]**2
    fprime[0,0]=2.0*x[0]
    fprime[0,1]=2.0*x[1]
    fprime[1,0]=-2.0*x[0]
    fprime[1,1]=1.0
    return f,fprime
```

```
def newton(f,xin):
    """
    Newton's method
    the function f returns the pair (f,fprime)
    """
    EPSILON = 1.0e-10
    x = copy.deepcopy(xin)
    # usually converges in <= 100 iterations
    for n in range(100):
        value,derivative = f(x)
        increment = la.solve(derivative,value)
        x -= increment
        errorEstimate = la.norm(increment)/la.norm(x)
        print "errorEstimate = ",errorEstimate
        if errorEstimate < EPSILON:
            return x,n
    assert(False)

y,i = newton(ex1,np.array([10.,10.]))
print "y = ",y," i= ",i
```

`newton.py` for Newton iterations

```
import numpy as np
import scipy.linalg as la
import copy

def ex1(x):
    """
    intersection of circle
    and parabola
    """
    assert(len(x) == 2)
    f=np.empty(2)
    fprime=np.empty([2,2])
    f[0]=x[0]**2+x[1]**2-1.0
    f[1]=x[1]-x[0]**2
    fprime[0,0]=2.0*x[0]
    fprime[0,1]=2.0*x[1]
    fprime[1,0]=-2.0*x[0]
    fprime[1,1]=1.0
    return f,fprime
```

```
def newton(f,xin):
    """
    Newton's method
    the function f returns the pair (f,fprime)
    """
    EPSILON = 1.0e-10
    x = copy.deepcopy(xin)
    # usually converges in <= 100 iterations
    for n in range(100):
        value,derivative = f(x)
        increment = la.solve(derivative,value)
        x -= increment
        errorEstimate = la.norm(increment)/la.norm(x)
        print "errorEstimate = ",errorEstimate
        if errorEstimate < EPSILON:
            return x,n
    assert(False)

y,i = newton(ex1,np.array([10.,10.]))
print "y = ",y," i= ",i
```

`newton.py` for Newton iterations

```
import numpy as np
import scipy.linalg as la
import copy

def ex1(x):
    """
    intersection of circle
    and parabola
    """
    assert(len(x) == 2)
    f=np.empty(2)
    fprime=np.empty([2,2])
    f[0]=x[0]**2+x[1]**2-1.0
    f[1]=x[1]-x[0]**2
    fprime[0,0]=2.0*x[0]
    fprime[0,1]=2.0*x[1]
    fprime[1,0]=-2.0*x[0]
    fprime[1,1]=1.0
    return f,fprime
```

```
def newton(f,xin):
    """
    Newton's method
    the function f returns the pair (f,fprime)
    """
    EPSILON = 1.0e-10
    x = copy.deepcopy(xin)
    # usually converges in <= 100 iterations
    for n in range(100):
        value,derivative = f(x)
        increment = la.solve(derivative,value)
        x -= increment
        errorEstimate = la.norm(increment)/la.norm(x)
        print "errorEstimate = ",errorEstimate
        if errorEstimate < EPSILON:
            return x,n
    assert(False)

y,i = newton(ex1,np.array([10.,10.]))
print "y = ",y," i= ",i
```

Output from `newton.py`

```
errorEstimate =  0.990068105443
errorEstimate =  0.961414155537
errorEstimate =  0.860078869547
errorEstimate =  0.586994090496
errorEstimate =  0.214892321847
errorEstimate =  0.0279885646798
errorEstimate =  0.000496297688535
errorEstimate =  1.56654175525e-07
errorEstimate =  1.55344805293e-14

y =  [ 0.78615138  0.61803399]  i=  8
```

Mistake in Jacobian

A mistake in the Jacobian destroys quadratic convergence

$$J = \begin{pmatrix} 2x_1 & 2x_2 \\ -2x_1 & x_2 \end{pmatrix}$$

```
errorEstimate =  0.990714987357
errorEstimate =  0.968455531242
errorEstimate =  0.934784964882
errorEstimate =  0.776980402552
errorEstimate =  0.491063275932
errorEstimate =  0.212353923457
errorEstimate =  0.0250723487329
errorEstimate =  0.00186382569352
errorEstimate =  0.000328673565462
errorEstimate =  6.7936322317 6e-05
errorEstimate =  1.39892028329e-05
errorEstimate =  2.88221015768e-06
errorEstimate =  5.9375625e-07
errorEstimate =  1.223210110 7e-07
errorEstimate =  2.51994931177e-08
errorEstimate =  5.19138209013e-09
errorEstimate =  1.06948346144e-09
errorEstimate =  2.20325693152e-10
errorEstimate =  4.53895086651e-11
y =  [-0.78615138  0.61803399]  i= 1
```

Debugging hint

If quadratic convergence is not observed, check that function and Jacobian are consistent.

Newton is not the only possibility

- Broyden's method
  - Multidimensional generalization of secant
  - Superlinear convergence
  - Need storage for approximate Jacobian
- Picard iteration (successive substitution)
  - Often slow convergence
  - Usually linear convergence