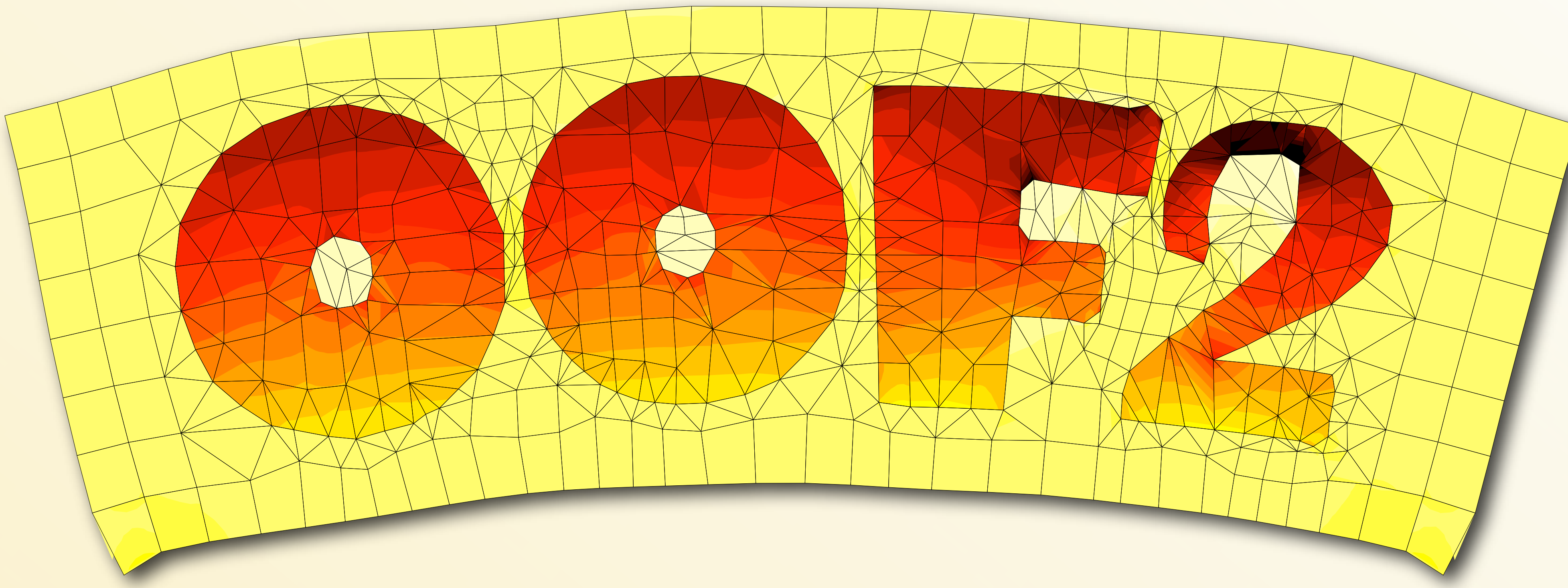


Installing and Extending OOF2

NIST
National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

Outline

- Installing OOF2
  - Hardware & Software Requirements
  - Installation Options
  - Testing
- Extending OOF2
  - Types of extensions
  - Ingredients
  - Examples

Hardware Requirements

- A Unix computer system:
  - Linux, Mac OS X, SGI IRIX, etc.
- About 400 Megabytes of disk space
- Lots of RAM
  - More is better...
  - 100x100 pixel microstructure, 40x40 skeleton requires 110 megabytes real, 410 megabytes virtual memory on Mac OS X.
    - We will try to reduce the memory requirements...

Installing OOF2

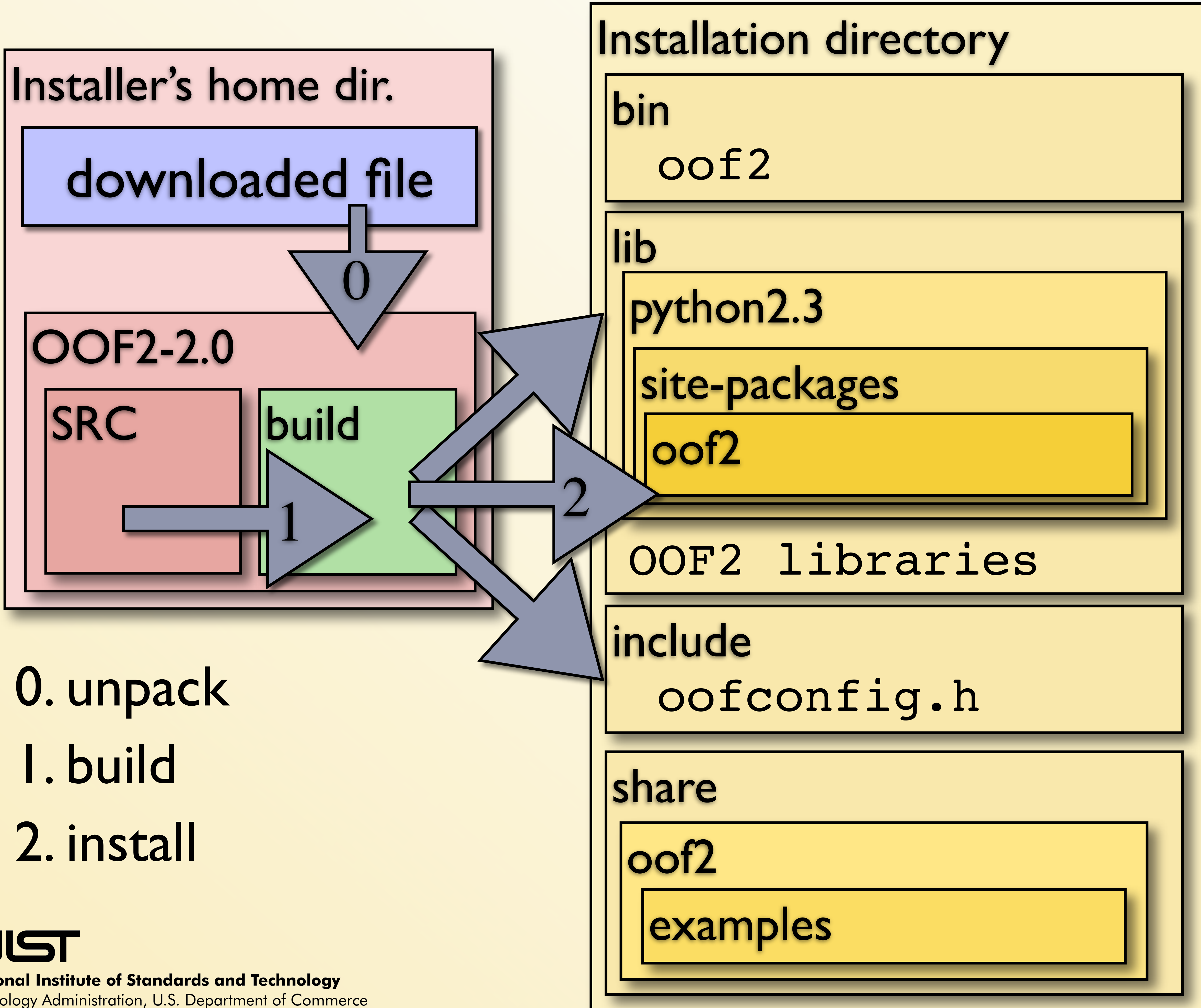
1. Install prerequisite programs and libraries.
2. Download the OOF2 source code.
3. Choose OOF2 installation parameters.
4. Build and install OOF2.
5. Run the test suite.

Prerequisites

- OOF2 requires:
  - Python (version 2.3 or later)
    - usually preinstalled
  - Libraries
    - gtk+2 (version 2.6 or later)
    - pygtk-2.0 (version 2.6 or later)
    - ImageMagick++
    - BLAS and LAPACK libraries
      - not required on Mac
    - ***make sure to install "dev" packages!***
- Building OOF2 extensions requires:
  - SWIG version 1.1 build 883

Get the OOF2 Source Code

- Download `oof2-2.0.tar.gz` from `http://www.ctcms.nist.gov/oof/oof2.html`
- Uncompress and unpack:
  - `tar -xzf oof2-2.0.tar.gz`
  - This creates a directory called `oof2-2.0`, containing
    - `README` file.
    - `SRC` subdirectory containing the oof2 code.
    - `TEST` subdirectory.
    - `examples` subdirectory.
    - `setup.py` and other installation scripts.
    - `shlib` subdirectory with more installation scripts.
- Read the `README` file.
- Read the `README` file.

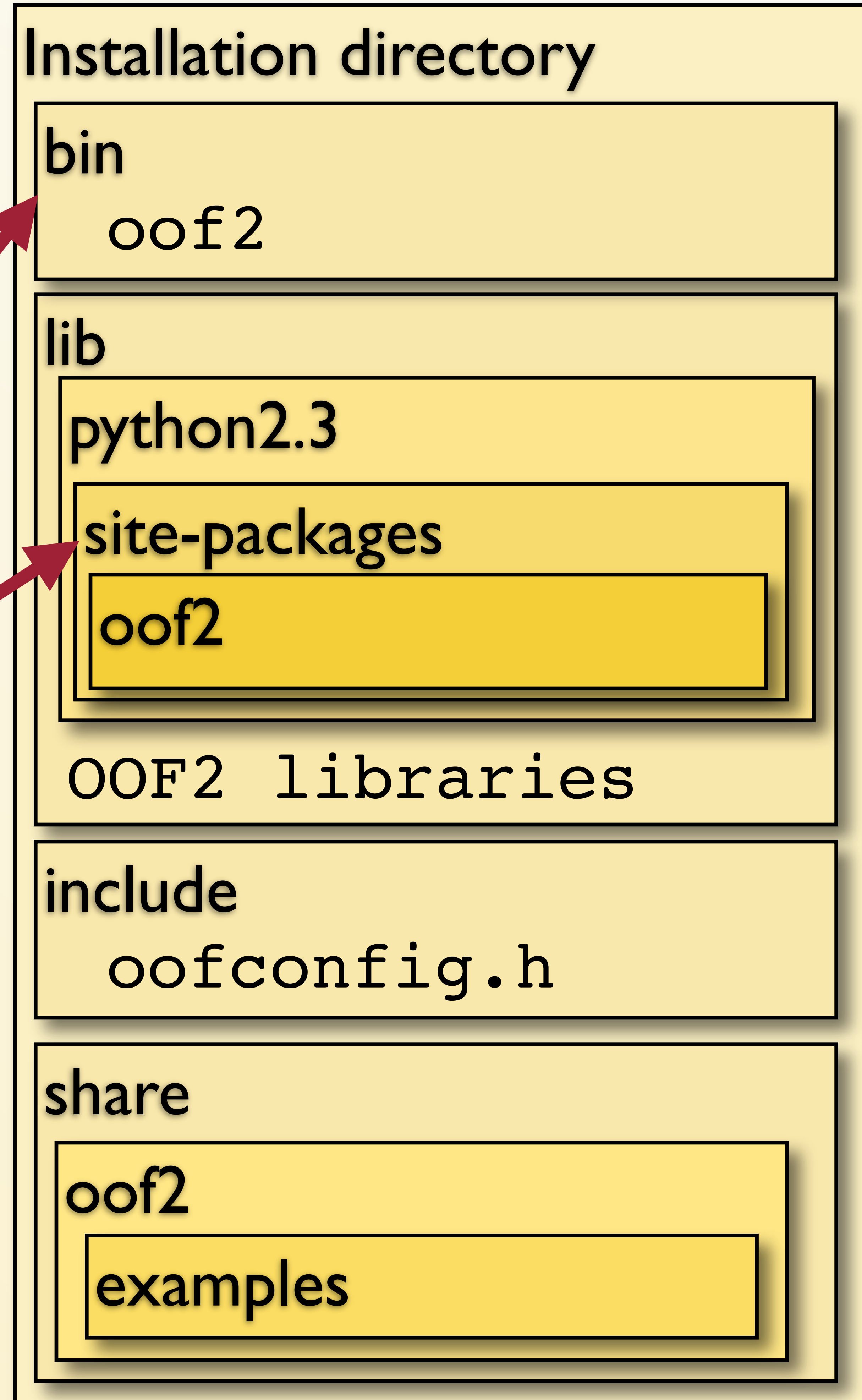


0. unpack
1. build
2. install

You choose the installation directory when you build oof2.

Should be in your shell's `PATH`

Must be in Python's `sys.path`



The paths can be changed via Unix environment variables if you need to install in a non-standard place.

Building and Installing OOF2

- Choose the installation (prefix) directory
  - Do you have superuser privileges?
    - Yes: install in `/usr/local/`
    - No: install in your home directory
- Do you have swig 1.1 build 883 installed (required for extending OOF2)?
  - Yes:

```
python setup.py build
python setup.py install --prefix=prefix
```

  - No:

```
python setup.py build --skip-swig install --prefix=prefix
```

- See the `README` file if you have a non-standard system configuration.

Special Installation Instructions

- Some systems require additional arguments to setup.py.
- See the `README` file for the format of additional arguments.
- Details for specific systems are at **http://www.ctcms.nist.gov/oof/oof2install.html**
  - Redhat (Fedora) Linux
  - Suse Linux
- Please let us know what additional arguments you needed to use, and we'll add them to the list.
- Please let us know if you have trouble installing oof2.

Running the TEST Suites

- Non-GUI tests check the core commands
  - `cd OOF2-2.0/TEST`
  - `python regression.py`
  - The last line printed should read `OK`.
- GUI tests check the user interface
  - `cd OOF2-2.0/TEST/GUI`
  - `python guitests.py`
  - The last line printed should be `All tests ran successfully!`
- Read the `README` files in both directories!
- Report test failures to `oof_bugs@ctcms.nist.gov`

Extending OOF2

- What can be added?
- How? (Briefly)

What Can Be (Easily) Added to OOF2?

- New Fields
  - Temperature, Displacement, ...
- New Fluxes
  - Heat Flux, Stress, ...
- New Equations
  - Heat Equation, Force Balance Equation, ...
- New Material Properties
  - Thermal Conductivity, Elasticity, Body Forces, ...
- New Output Quantities
  - Energy Densities, Arbitrary Combinations of Fields, ...

How to Extend OOF2

- Read Chapters 7 & 8 of the Manual:
  - `http://www.ctcms.nist.gov/~langer/oof2man/index.html`
  - Chapter 7: required files and compilation methods.
  - Chapter 8: contents of the required files.
- Write the required files.
- Build and install ...
  - ... all of OOF2 for *internal* extensions.
  - ... only new files for *external* extensions.
- Run OOF2 ...
  - ... normally for internal extensions.
  - ... with `--import` for external extensions.
    - `oof2 --import myextensions.propertyA`
    - directory `myextensions` must be in Python's `sys.path`

Adding a new Field

- In a Python file:

```
from oof2.engine import problem
from oof2.SWIG.engine import field

newfield = field.ScalarField('MyField')
problem.advertise(newfield)
```

- This creates a new Field object and makes it available in all scripts and GUI locations where other fields can be used.

- Scalar, Vector, and (soon) Tensor Fields can be created

Adding a new Flux

- Similarly:

```
from oof2.engine import problem
from oof2.SWIG.engine import flux

newflux = flux.VectorFlux('MyFlux')
problem.advertise(newflux)
```

- Can create Vector and Symmetric Tensor fluxes (more types on demand).

Adding a new Divergence Equation

- Similarly:

```
from oof2.engine import problem
from oof2.SWIG.engine import equation

neweqn = equation.DivergenceEquation(
              'MyEquation', newflux, 1)
problem.advertise(neweqn)
```

- Divergence of Flux = applied forces
- Arguments:
  - Equation name, flux, and dimension of divergence
  - applied forces *aren't* specified
    — they're Material Properties.

Adding a new Plane-Flux Equation

- Similarly:

```
from oof2.engine import problem
from oof2.SWIG.engine import equation

neweqn2 = equation.PlaneFluxEquation(
            'MyFlux_plane', newflux, 1)
problem.advertise(neweqn2)
```

- Out-of-plane components of Flux = 0
- Arguments:
  - Equation name, flux, and number of out-of-plane components

Adding a new Material Property

- Requires more work, unfortunately.
- Can be done in either C++ or Python.
- See the manual for all of the details...

- Create a C++ or Python class with methods that perform these roles:
  - Identification
  - Cross reference with other Properties of the same Material
  - Precomputation
  - Computation: contributions to the FE stiffness matrix & right hand side
  - Postcomputation
  - Output
  - Many of these functions are optional!

Adding a new Material Property, Cont'd

- Create a Python *PropertyRegistration*
  - Makes the Property known to the rest of OOF2
  - Provides information (meta-data) about how to construct the Property, and what it can do.
  - Example — Cubic Elasticity:

```
PropertyRegistration(
    'Mechanical:Elasticity:Anisotropic:Cubic',
     CubicElasticityProp,
    "oof2.SWIG.engine.property.elasticity.aniso.aniso",
    11,
    [anisocijkl.CubicCijklParameter('cijkl',
                                    anisocijkl.CubicRank4TensorCij(c11=1.0,
                                                                   c12=0.5,
                                                                   c44=0.25),
                                    tip=parameter.emptyTipString)],
    fields=[problem.Displacement],
    fluxes=[problem.Stress],
    outputs=["Energy"],
    propertyType="Elasticity",
    tip="Cubic linear elasticity."
    )
```

Adding a new Material Property, Cont'd

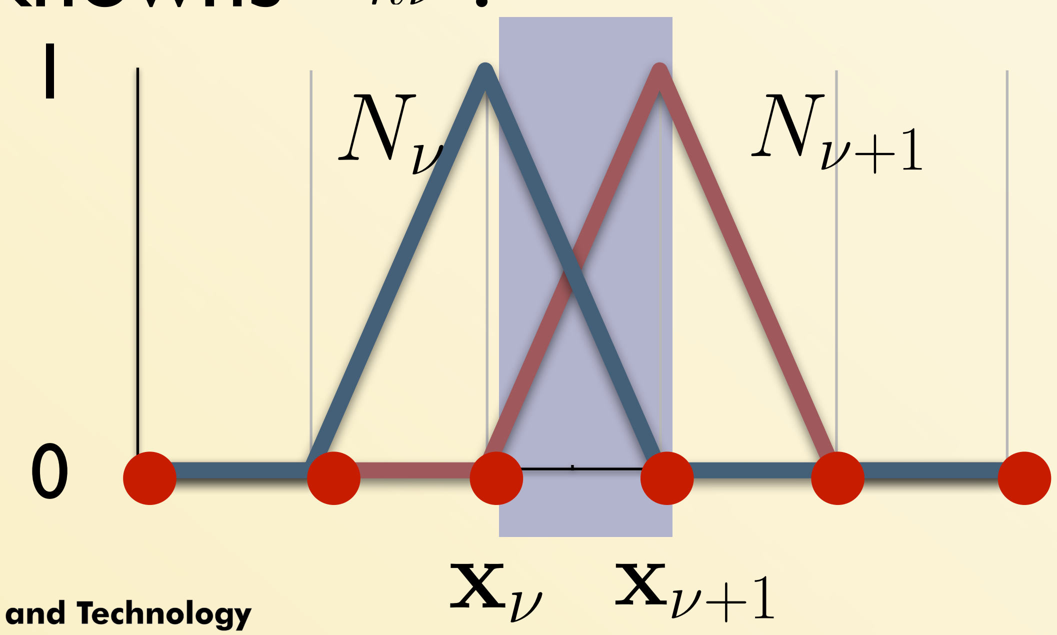
- Example of a Property method:
  - `Property::fluxmatrix` is used to compute the Property's contribution to the finite element stiffness matrix.
  - But first, a little math...

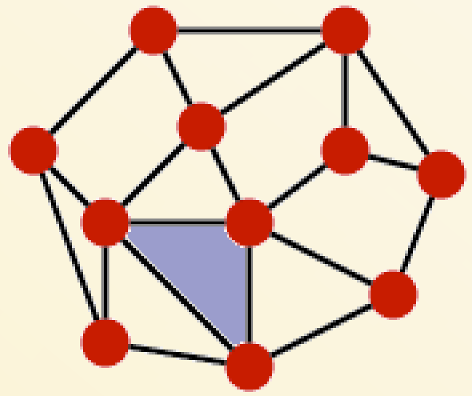
Finite Elements in 50 Words or Less

- Divide space into *elements*.
- Evaluate fields at *nodes* between elements: $u_{n\nu} = u_n(\mathbf{x}_\nu)$
- Interpolate fields in elements via *shape functions* $N_\nu(\mathbf{x})$

$$u_n(\mathbf{x}) = \sum_\nu u_{n\nu} N_\nu(\mathbf{x})$$

- Substitute expansion into equations, multiply by a test function, integrate by parts, and solve the resulting system of linear equations for the unknowns $u_{n\nu}$.





How a Property Contributes to the Finite Element Stiffness Matrix

- A "Property" is a term in a flux: $\sigma = \sum_i k_i \nabla \phi_i$

  SCHEMATIC

- Define $\sigma = \mathbf{M} \cdot u$
  - u is the vector of all field values at all nodes of an element
  - $\mathbf{M}$ is the "flux matrix"
- Developer must provide a routine to compute an element's contribution to $\mathbf{M}$ at $\mathbf{x}$ for node ν.
  - This can be done with no explicit knowledge of the element geometry.

Example: Elasticity

- Displacement component *l* at point **x**: $u_l(\mathbf{x})$
- Stress component *ij* at **x**: $\sigma_{ij}(\mathbf{x}) = C_{ijkl}\partial_k u_l(\mathbf{x})$
- Expand in shape functions: $u_l(\mathbf{x}) = N_\nu(\mathbf{x})u_{l\nu}$
  - $u_{l\nu}$ is displacement component *l* at node ν.
  - $\sigma_{ij}(\mathbf{x}) = C_{ijkl}\partial_k N_\nu(\mathbf{x})\, u_{l\nu}$
- Compare to $\sigma_{ij}(\mathbf{x}) = M_{ij}^{k\nu}u_{k\nu}$
- Find $M_{ij}^{k\nu}(\mathbf{x}) = C_{ijkl}\partial_l N_\nu(\mathbf{x})$

Example: Elasticity

```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, FluxData *fluxdata,
                            const MasterPosition &x) const {
  if(*flux != *stress_flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
  }

  const Cijkl modulus = cijkl(mesh, element, x);
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x);
  double dsf1 = nu.dshapefunction(1, x);

  for(SymTensorIndex ij; !ij.end(); ++ij) {
    for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
      SymTensorIndex ell0(0, ell.integer());
      SymTensorIndex ell1(1, ell.integer());
      fluxdata->matrix_element(mesh, ij, displacement, ell, nu) +=
                               modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
    }
 if(!displacement->in_plane(mesh)) {
      Field *oop = displacement->out_of_plane();
      for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
          fluxdata->matrix_element(mesh, ij, oop, ell, nu)
                      += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
      }
    }
  }
}
```

Example: Elastic

Node ν

Flux σ

Position **x**

```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, FluxData *fluxdata,
                            const MasterPosition &x) const {
  if(*f          _flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__,
  }

  const Cijkl modulus = cijkl(mesh, element, x);
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x);
  double dsf1 = nu.dshapefunction(1, x);

  for(SymTensorIndex ij; !ij.end(); ++ij) {
    for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
      SymTensorIndex ell0(0, ell.integer());
      SymTensorIndex ell1(1, ell.integer());
      fluxdata->matrix_element(mesh, ij, displacement, ell, nu) +=
                        modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
    }
 if(!displacement->in_plane(mesh)) {
      Field *oop = displacement->out_of_plane();
      for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
          fluxdata->matrix_element(mesh, ij, oop, ell, nu)
                  += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
      }
    }
  }
}
```

Example: Elasticity

```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, FluxData *fluxdata,
                            const MasterPosition &x) const {
```

```
if(*flux != *stress_flux) {
  throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
}
```

```
  const Cijkl modulus = cijkl(mesh, element, x);
  double sf = nu.shapefun
  double dsf0 = nu.dshape
  double dsf1 = nu.dshapefunction(1, x);

  for(SymTensorIndex ij; !ij.end(); ++ij) {
    for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
      SymTensorIndex ell0(0, ell.integer());
      SymTensorIndex ell1(1, ell.integer());
      fluxdata->matrix_element(mesh, ij, displacement, ell, nu) +=
                            modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
    }
   if(!displacement->in_plane(mesh)) {
      Field *oop = displacement->out_of_plane();
      for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
          fluxdata->matrix_element(mesh, ij, oop, ell, nu)
                      += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
      }
    }
  }
}
```

Sanity Check

National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

Example: Elasticity

```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, FluxData *fluxdata,
                            const MasterPosition &x) const {
  if(*flux != *stress_flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
  }

  const Cijkl modulus = cijkl(mesh, element, x);
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x);
  double dsf1 = nu

  for(SymTensorInd
    for(FieldItera                                                    l) {
      SymTensorInd
      SymTensorInd
      fluxdata->matrix_element(mesh, ij, displacement, ell, nu) +=
                              modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
    }
  if(!displacement->in_plane(mesh)) {
      Field *oop = displacement->out_of_plane();
      for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
          fluxdata->matrix_element(mesh, ij, oop, ell, nu)
                      += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
      }
    }
  }
}
```

Elastic modulus computed by virtual function call to derived class (eg. CubicElasticity)

Example: Elasticity

```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, FluxData *fluxdata,
                            const MasterPosition &x) const {
  if(*flux != *stress_flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
  }

  const Cijkl modulus = cijkl(mesh, element, x);
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x);
  double dsf1 = nu.dshapefunction(1, x);

  for(SymTensorIndex ij; !ij.end(); ++ij) {
    for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
      SymTensorIndex ell0(0
      SymTensorIndex ell1(1
      fluxdata->matrix_elem
                                                                ll1)*dsf1;
    }
 if(!displacement->in_plane(mesh)) {
      Field *oop = displacement->out_of_plane();
      for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
          fluxdata->matrix_element(mesh, ij, oop, ell, nu)
                        += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
      }
    }
  }
}
```

Shape function evaluation for node ν at point **x**

Example: Elasticity

```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, FluxData *fluxdata,
                            const MasterPosition &x) const {
  if(*flux != *stress_flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
  }

  cons
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x
  double dsf1 = nu.dshapefunction(1, x

  for(SymTensorIndex ij; !ij.end(); ++ij) {
    for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
      SymTensorIndex ell0(0, ell.integer());
      SymTensorIndex ell1(1, ell.integer());
      fluxdata->matrix_element(mesh, ij, displacement, ell, nu) +=
                        modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
    }
  if(!displacement->in_pl
      Field *oop = displ
      for(FieldIterator e                                nd(); ++ell) {
          fluxdata->matri
                     +=                                ger())) * sf;
      }
    }
}
```

For all stress components ij

For all displacement components l

$$M_{ij}^{l\nu}(\mathbf{x}) = \sum_k C_{ijkl}\partial_k N_\nu(\mathbf{x})$$

$l\nu \Rightarrow$ degree of freedom

$ij \Rightarrow$ stress component

Example: Elasticity

```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, FluxData *fluxdata,
                            const MasterPosition &x) const {
  if(*flux != *stress_flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
  }

  const Cijkl modulus = cijkl(mesh, element, x);
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x);
  double dsf1 = nu.dshapefunction(1, x);

  for(SymTensorIndex ij; !ij.end(); ++ij) {
    for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
      SymTensorIndex ell0(0, ell
      SymTensorIndex ell1(1, ell
      fluxdata->matrix_element(m
                               modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
    }
```

Contribution from out-of-plane strains

```
if(!displacement->in_plane(mesh)) {
     Field *oop = displacement->out_of_plane();
     for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
         fluxdata->matrix_element(mesh, ij, oop, ell, nu)
                     += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
     }
   }
 }
```

Example: Elasticity

```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, FluxData *fluxdata,
                            const MasterPosition &x) const {
  if(*flux != *stress_flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
  }

  const Cijkl modulus = cijkl(mesh, el
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x
  double dsf1 = nu.dshapefunction(1, x

  for(SymTensorIndex ij; !ij.end(); ++
    for(FieldIterator ell=displacement
      SymTensorIndex ell0(0, ell.intege
      SymTensorIndex ell1(1, ell.intege
      fluxdata->matrix_element(mesh, i
                             modulus(
    }
 if(!displacement->in_plane(mesh)) {
      Field *oop = displacement->out_of_plane();
      for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
          fluxdata->matrix_element(mesh, ij, oop, ell, nu)
                      += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
      }
    }
  }
}
```

- ***No*** explicit dependence on:
  - Element geometry
    - triangle, quadrilateral
  - Element order
    - linear, quadratic...
  - Equation
    - divergence, plane-stress
  - Other material properties

For more details, see the on-line manual.

`http://www.ctcms.nist.gov/~langer/oof2man/index.html`