CIEM1110-1: FEM, workshop 1.2

Introduction to pyJive

Frans van der Meer, Iuri Rocha, Martin Lesueur

Introduction to pyJive

Origins:

- Jem/Jive C++ libraries implemented by Dynaflow Research Group
- Used extensively within the Computational Mechanics group

Philosophy:

- Quick coding foundation for numerical simulation software
- Focus on modularity and extensibility
- Flexible data structures

Python version:

- Focus on interactivity through Jupyter notebooks

pyJive — Object Oriented Programming

Jive makes extensive use of classes and objects:

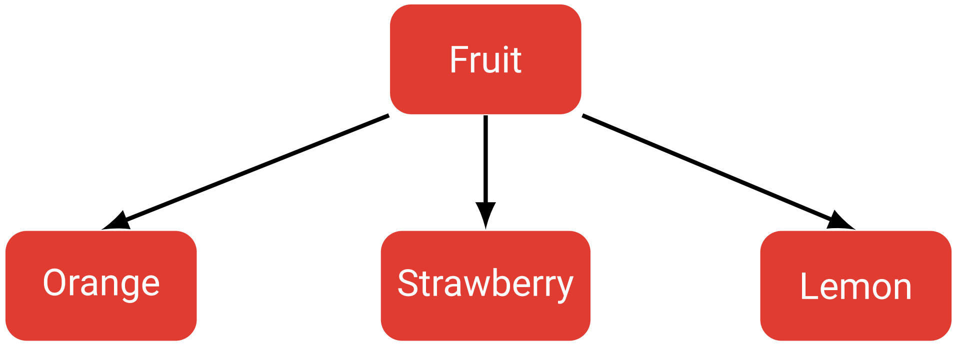
- Class: A collection of functions and variables
- Object: An instance of a class

Class inheritance:

- Relates classes based on a *is-a-type-of* relationship
- Inherited functions promote code reuse

Polymorphism:

- Caller is indifferent as to which class is being used




```
class Fruit:
    def what(self):
            print('Fruit')
    def color(self):
            print('Base implementation')

class Orange(Fruit):
    def color(self):
            print('Yellow')
class Strawberry(Fruit):
    def color(self):
            print('Red')
class Lemon(Fruit):
    def color(self):
            print('Green')

fruits = [Strawberry(), Lemon()]

for fruit in fruits:
    fruit.what()  # prints: 'Fruit' / 'Fruit'
    fruit.color() # prints: 'Red' / 'Green'
```

TU Delft

pyJive — General structure

Modules:

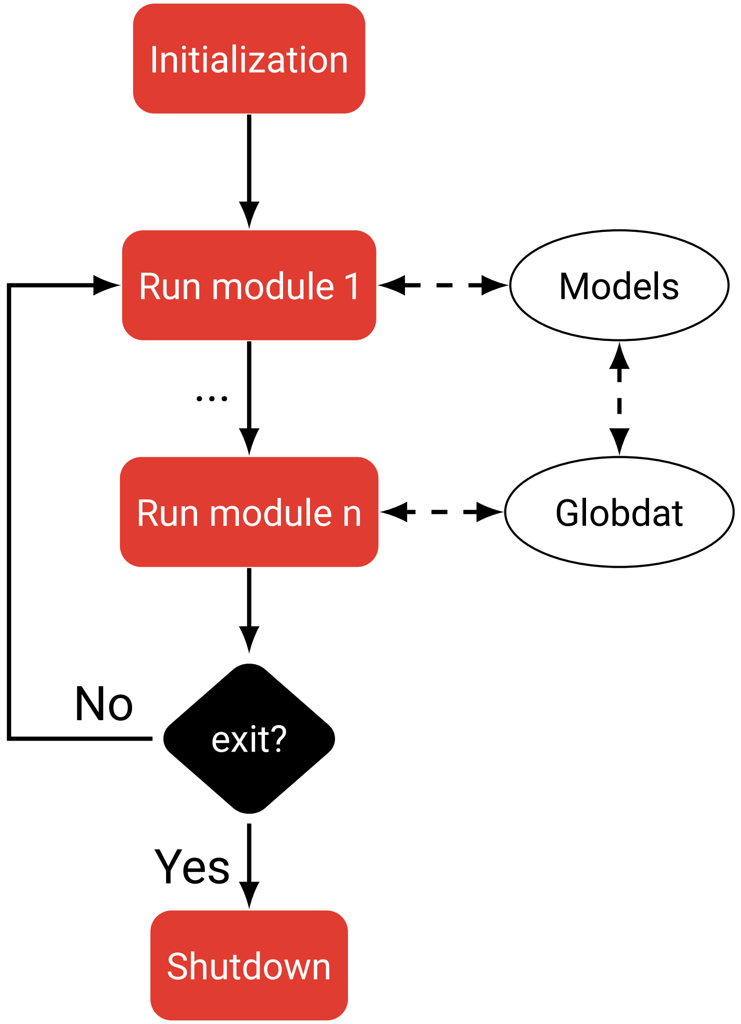
- Applications that define the flow of the simulation
- Module chain: A set of modules run in sequence

Models:

- Implement the specific problem being solved

Globdat:

- Stores input and output
- Facilitates data flow between modules and models



pyJive — General structure

Modules:

- Applications that define the flow of the simulation
- Module chain: A set of modules run in sequence

Models:

- Implement the specific problem being solved

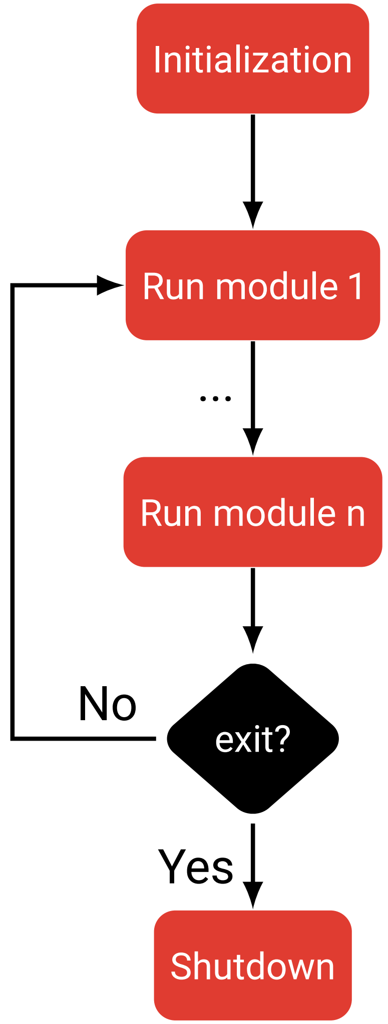
Globdat:

- Stores input and output
- Facilitates data flow between modules and models

```
for module in chain:
    module.init(props, globdat)
```

```
while keep_going:
    for module in chain:
        if 'exit' in module.run(globdat):
            keep_going = False
```

```
for module in chain:
    module.shutdown(globdat)
```

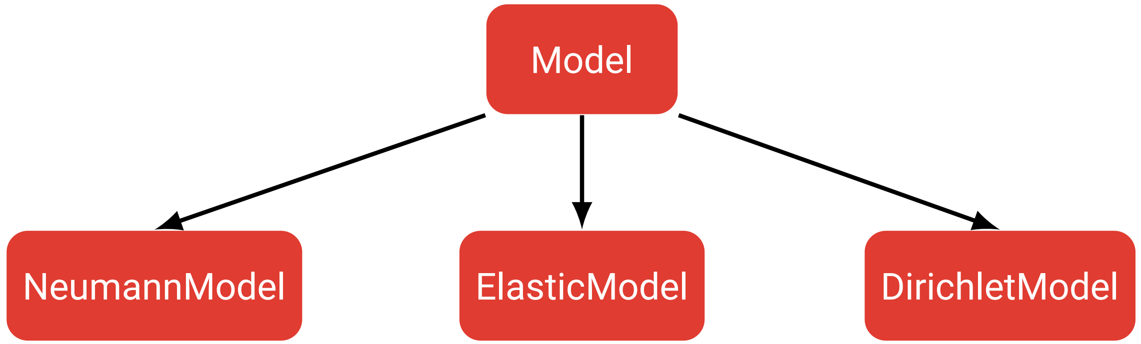


pyJive — Models

Models implement aspects from the physical problem being solved:

- Stiffness matrix and force vector
- Boundary conditions (Dirichlet/Neumann)
- Secondary solution fields (e.g. stresses)

Models are joined together through the MultiModel class.




```python
class Model:
    def __init__(self, name):
        self._name = name

    def take_action(self, action, params, globdat):
        print('Empty model takeAction')

    def configure(self, props, globdat):
        print('Empty model configure')
```

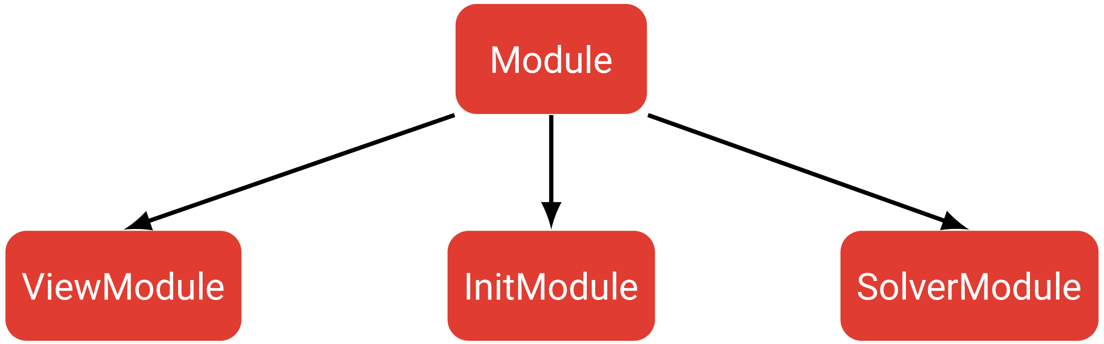
TU Delft

pyJive – Modules

Modules implement the procedure used to solve the problem:

- Read mesh and initialize model
- Solve the final discrete system of equations
- Plot or write output to files

Modules are joined together in a module chain.



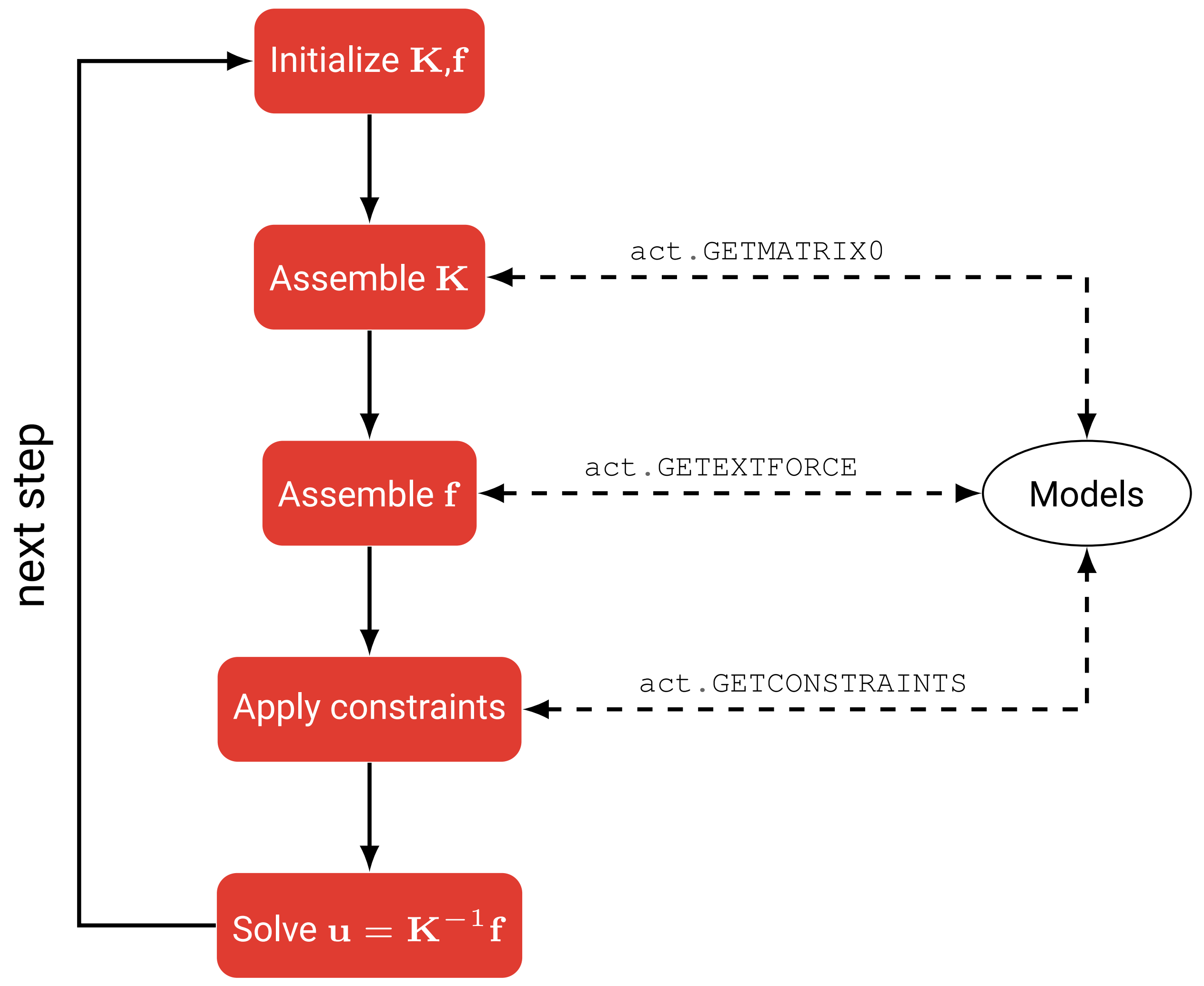

```python
class Module:
    def __init__(self, name):
        self._name = name

    def init(self, props, globdat):
        print('Empty module init')

    def run(self, globdat):
        print('Empty module run')
        return 'exit'

    def shutdown(self, globdat):
```

pyJive — Model/module interaction

The take_action paradigm:

- Pre-defined actions in *names.py*
- Modules ask, models answer
- Data exchange through *Globdat* and *params*

Example: the basic workflow of *SolverModule*:



pyJive — Jupyter notebooks and modularity

Functionality can be added/modified directly on notebooks:

- Change model input on the fly

```
props = pu.parse_file('model.pro')

# Run with original properties
globdat1 = main.jive(props)

# Change stiffness and run again
props['model']['bar']['EA'] = 10000.0
globdat2 = main.jive(props)

# Compare results
print('Displacement before',globdat1[gn.STATE0][0])
print('Displacement after',globdat2[gn.STATE0][0])
```

model.pro

```
model =
{
  type = Multi;

  [...]

  bar =
  {
    type = Bar;

    elements = all;

    EA = 1.0;

    shape =
    {
      type = Line2;
      intScheme = Gauss2;
    };
  };
};
```

pyJive — Jupyter notebooks and modularity

Functionality can be added/modified directly on notebooks:

- Change model input on the fly
- Customized postprocessing routines

```
lnode = globdat[gn.NGROUPS]['left'][0]
ldof = globdat[gn.DOFSPACE].get_dof(lnode,'dx')

rnode = globdat[gn.NGROUPS]['right'][0]
rdof = globdat[gn.DOFSPACE].get_dof(rnode,'dx')

reldisp = globdat[gn.STATE0][rdof] - globdat[gn.STATE0][ldof]
print('Relative displacement:', reldisp)
```

pyJive — Jupyter notebooks and modularity

Functionality can be added/modified directly on notebooks:

- Change model input on the fly
- Customized postprocessing routines
- Create and use a standalone module

```
myoutput = globdat[gn.MODULEFACTORY].get_module('VTKOut','myout')

props['myout']['file'] = 'results.out'

myoutput.init(props,globdat)
myoutput.run(globdat)
myoutput.shutdown(globdat)

# Here we could read 'results.out'
```

TU Delft