Pyre

An overview of a software architecture for scientific applications

Michael Aivazis
Caltech

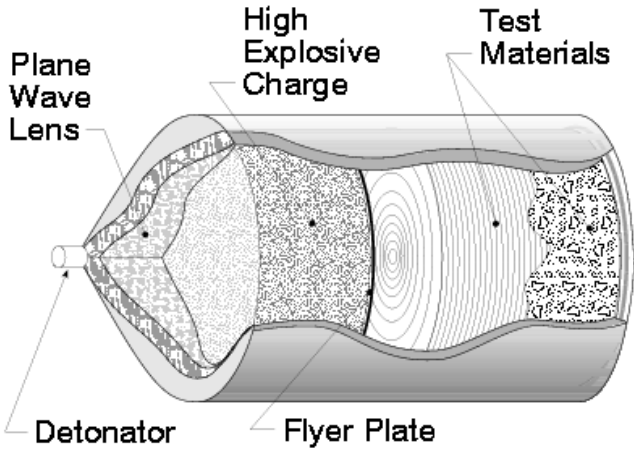
Mantle Convection Workshop
Boulder
19-26 June 2005

Overview

- Projects
  - *Dynamic response of materials: Caltech ASC Center (DOE)*
  - *Geophysics: GeoFramework (NSF ITR), CIG (NSF)*
  - *Neutron scattering data analysis: ARCS(DOE), DANSE (NSF)*
  - *Radar interferometry: ROIPAC (NASA JPL)*
- Usability:
  - *enable the non-expert without hindering the expert*
- Portability:
  - *languages: C, C++, F77, F90*
  - *compilers: all native compilers on supported platforms, gcc, Absoft, PGI*
  - *platforms: all common Unix variants, OSX, Windows*
- Statistics:
  - *1200 classes, 75,000 lines of Python, 30,000 lines of C++*
  - *Largest run: **nirvana** at LANL, 1764 processors for 24 hrs, generated 1.5 Tb*

Projects – VTF

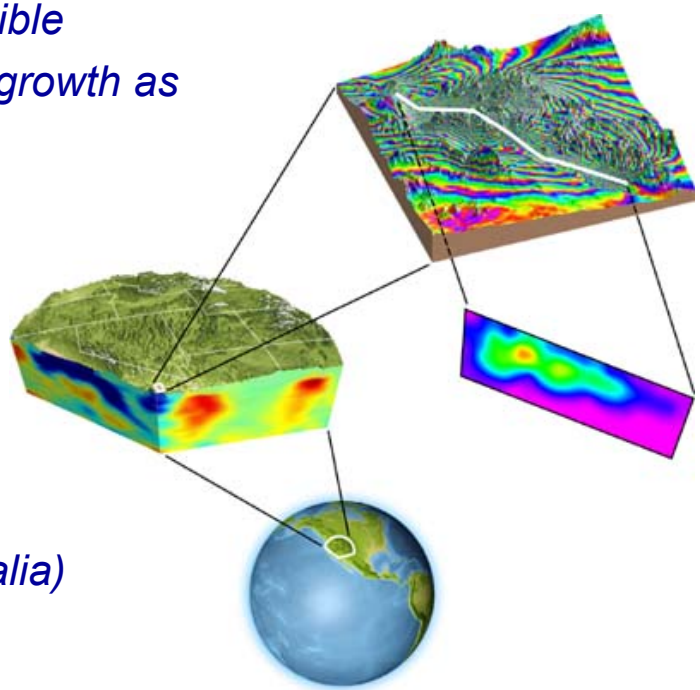
- Virtual Facility for Simulating the Dynamic Response of Materials



- Goals:
  - *simulate experiments where strong shocks and detonation waves impinge on solid targets*
  - *enable validation of such simulations against experimental data*
- Multidisciplinary activities:
  - *modeling and simulation of fundamental processes*
  - *first principles computation of material properties*
  - *compressible turbulence and mixing*
  - *problem solving environment*

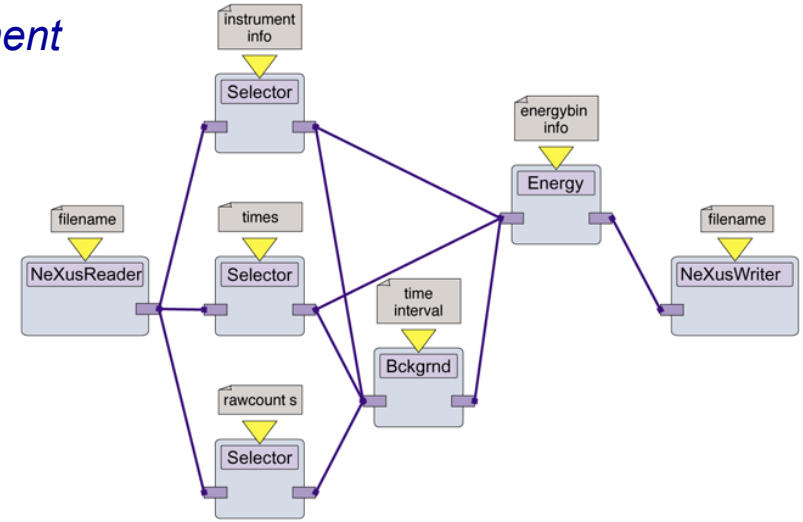
Projects - GeoFramework

- Simulations of multi-scale deformation in solid earth Geophysics (ITR)
- GeoFramework is a modeling package that will
  - *be usable by the entire Earth sciences community*
  - *address the limitations of what is currently feasible*
  - *will be engineered with software evolution and growth as design requirements*
  - *emphasis on validation*
- Pyre is the simulation framework
  - *solver integration and coupling*
  - *uniform access to facilities*
  - *accessible to non-experts*
  - *integrated visualization*
  - *collaboration extended to include VPAC (Australia)*
- More info at www.geoframework.org



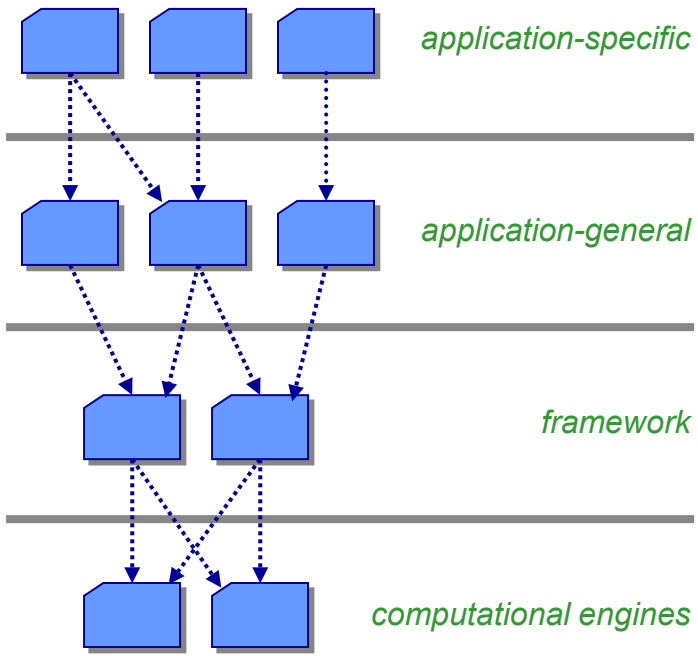
Projects – DANSE

- ARCS will be
  - *a high-resolution, direct-geometry, time-of-flight chopper spectrometer at the Spallation Neutron Source in Oak Ridge.*
  - *optimized to provide a high neutron flux at the sample, and a large solid angle of detector coverage*
- Data analysis for neutron scattering experiments
  - *national capability*
  - *full neutron scattering community engagement*
- Pyre is the data analysis framework
  - *large number of data analysis modules*
  - *standard integration strategy*
  - *distributed*
    - *web services (XMLRPC, SOAP, OGSA,…)*
    - *IO facilities for data transport*
  - *integrated visualization*



Pyre

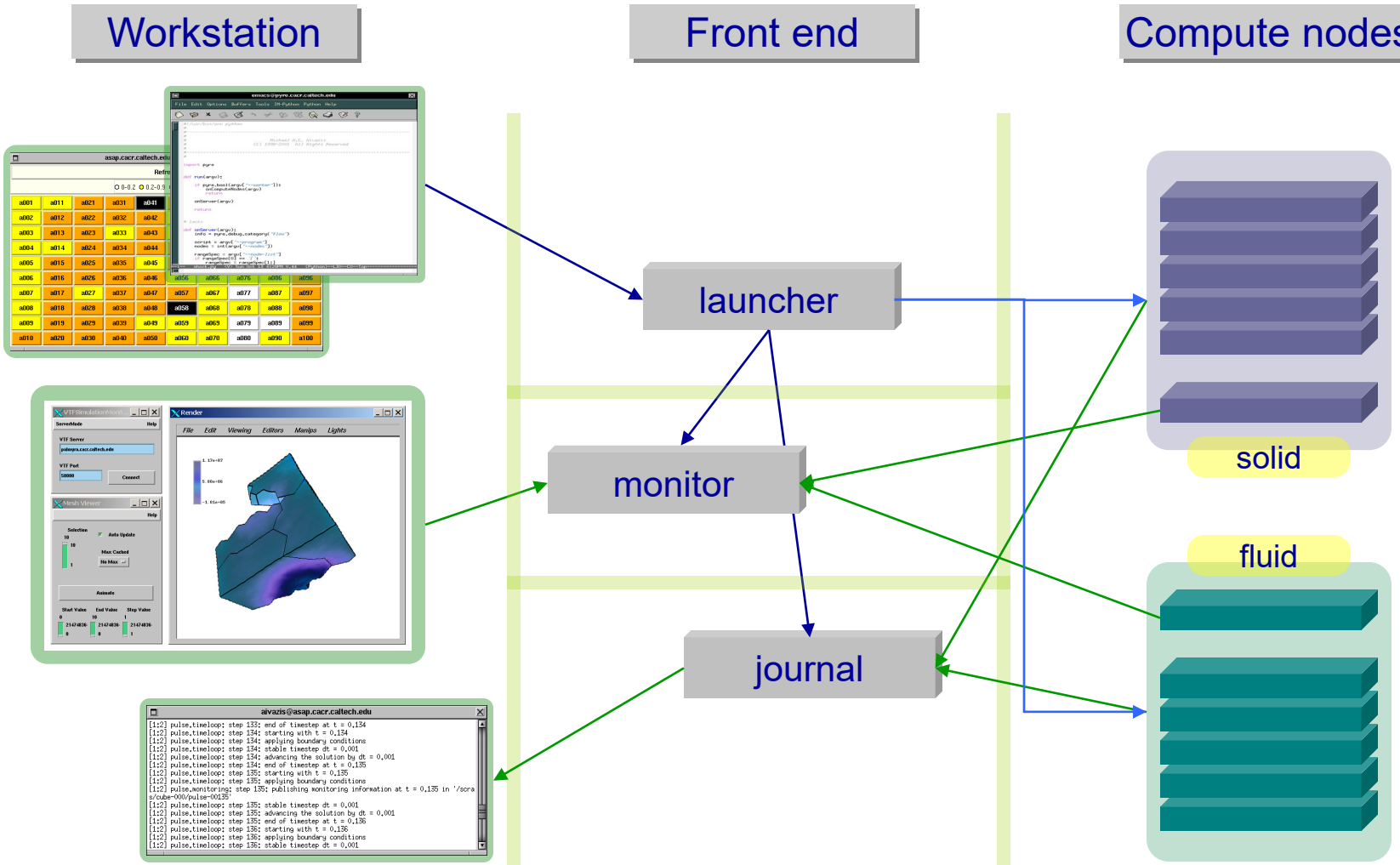
- Pyre is a *software architecture*:
  - *a specification of the organization of the software system*
  - *a description of the crucial structural elements and their interfaces*
  - *a specification for the possible collaborations of these elements*
  - *a strategy for the composition of structural and behavioral elements*
- Pyre is multi-layered
  - *flexibility*
  - *complexity management*
  - *robustness under evolutionary pressures*
- Contemplate the disconnect between a remote, parallel computation and your ability to control it from your laptop



Flexibility through the use of scripting

- Scripting enables us to
  - *Organize the large number of application parameters*
  - *Allow the application to discover new capabilities without the need for recompilation or relinking*
- The python interpreter
  - *The interpreter*
    - *modern object oriented language*
    - *robust, portable, mature, well supported, well documented*
    - *easily extensible*
    - *rapid application development*
  - *Support for parallel programming*
    - *trivial embedding of the interpreter in an MPI compliant manner*
    - *a python interpreter on each compute node*
    - *MPI is fully integrated: bindings + OO layer*
  - No measurable impact on either performance or scalability

Application deployment



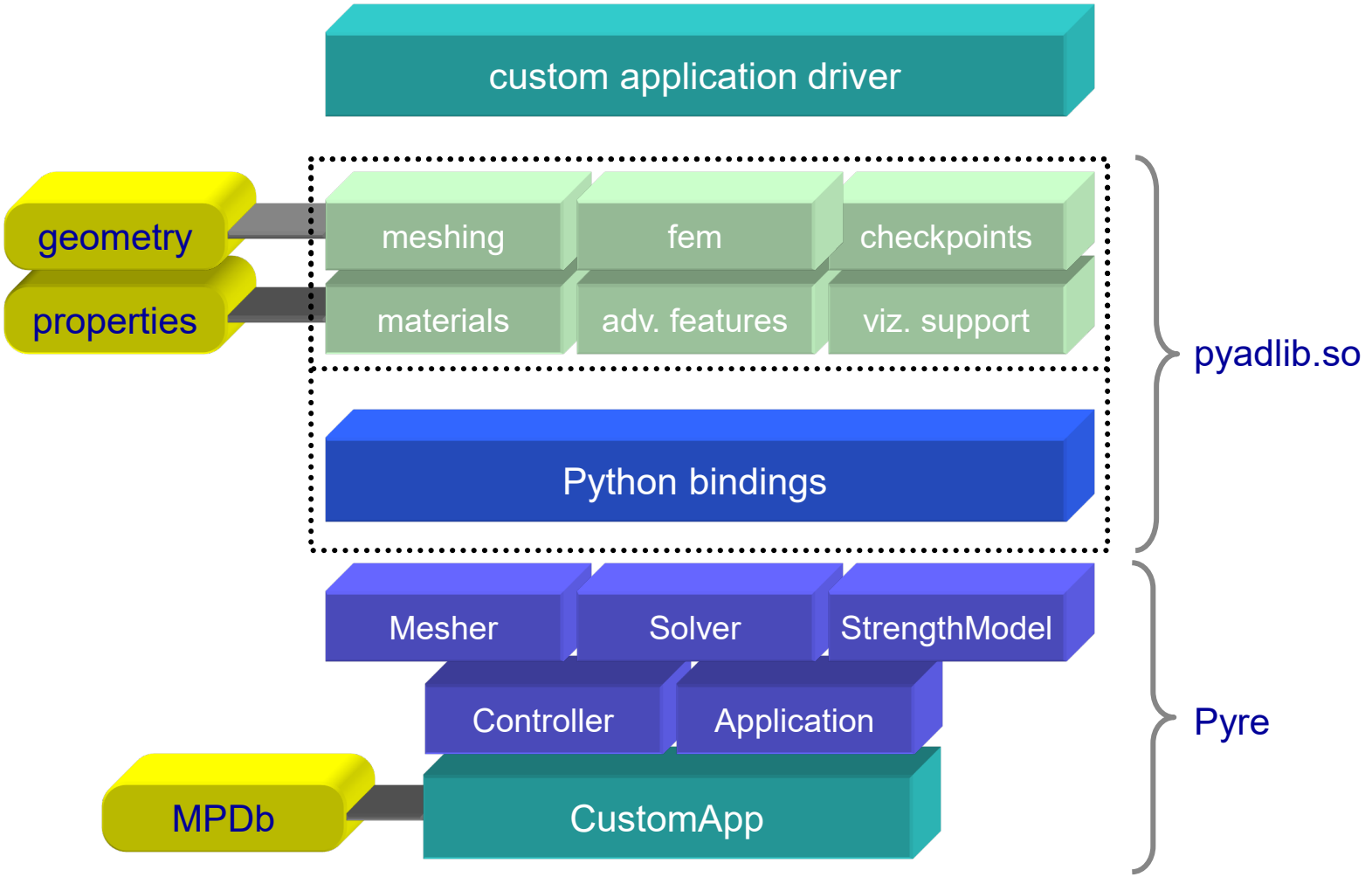
Simulation support

- Problem specification
  - *components and their properties*
- Solid modeling
  - *overall geometry*
  - *model construction*
  - *topological and geometrical information*
- Boundary and initial conditions
  - *high level specification*
  - *access to the underlying solver data structures in a uniform way*
- Materials and constitutive models
  - *materials properties database*
  - *strength models and EOS*
  - *association with a region of space*
- Computational engines
  - *selection and association with geometry*
  - *solver specific initializations*
  - *coupling mechanism specification*
- Simulation driver
  - *initialization*
  - *appropriate time step computation*
  - *orchestration of the data exchange*
  - *checkpoints and field dumps*
- Active monitoring
  - *instrumentation: sensors, actuators*
  - *real-time visualization*
- Full simulation archiving

Support for concurrent applications

- Python as the driver for concurrent applications that
  - *are embarrassingly parallel*
  - *have custom communication strategies*
    - *sockets, ICE, shared memory*
- Excellent support for MPI
  - **`mpipython.exe`**: *MPI enabled interpreter (needed only on some platforms)*
  - **`mpi`**: *package with python bindings for MPI*
    - *support for staging and launching*
    - *communicator and processor group manipulation*
    - *support for exchanging python objects among processors*
  - **`mpi.Application`**: *support for launching and staging MPI applications*
    - *descendant of* **`pyre.application.Application`**
    - *auto-detection of parallelism*
    - *fully configurable at runtime*
    - *used as a base class for user defined application classes*

Integrating existing codes



Writing python bindings

- Given a “low level” routine, such as

```
double adlib::stableTimeStep(const char *);
```

- and a wrapper

```
char pyadlib_stableTimestep__name__[] = "stableTimestep";
PyObject * pyaldib_stableTimestep(PyObject *, PyObject * args)
{
    double dt = adlib::stableTimeStep("deformation");

    return Py_BuildValue(“d”, dt);
}
```

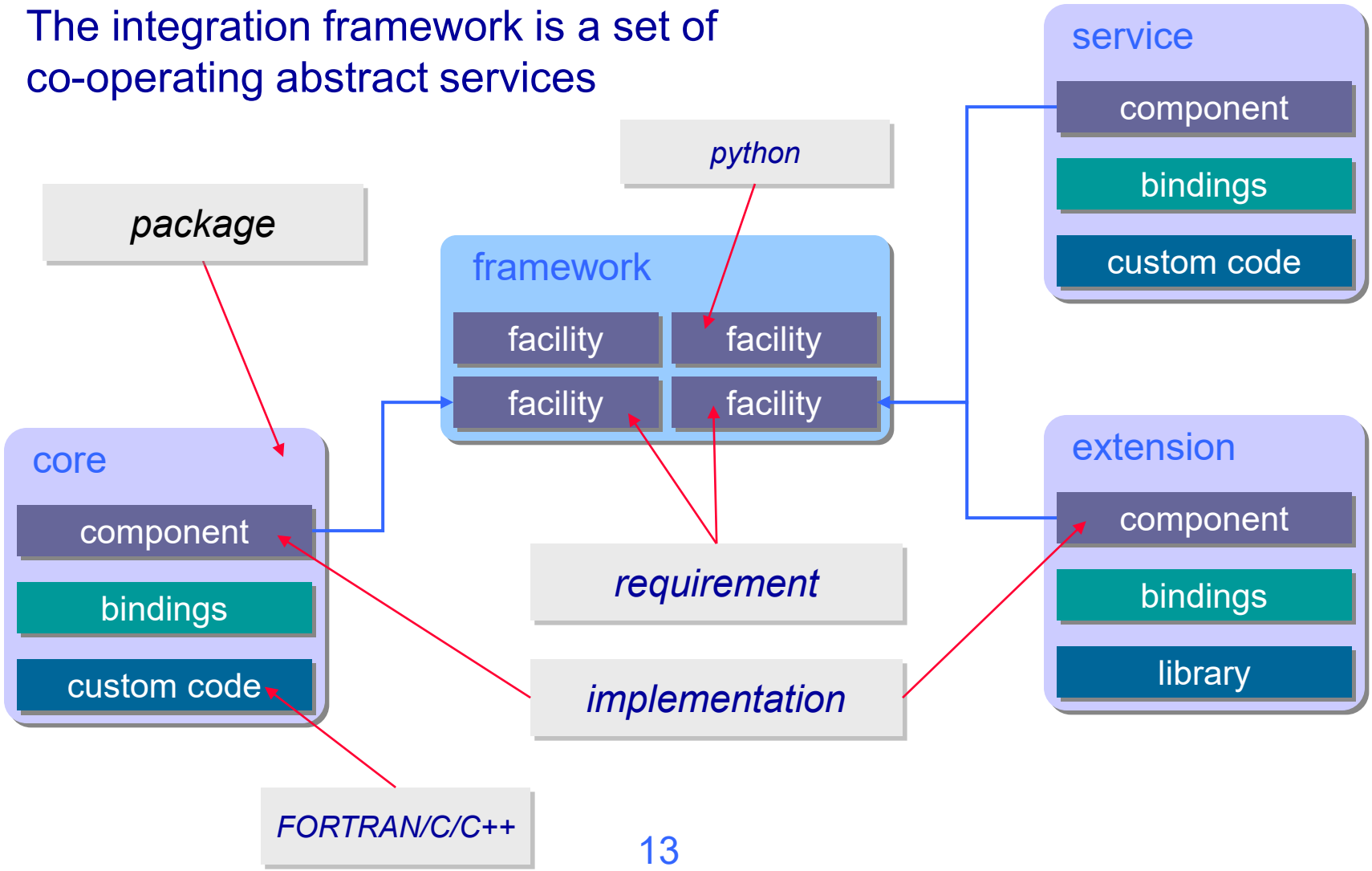
- one can place the result of the routine in a python variable

```
dt = pyadlib.stableTimestep()
```

- The general case is not much more complicated than this

Component architecture

- The integration framework is a set of co-operating abstract services



Facilities and components

- A design pattern that enables the assembly of application components at run time under user control
- Facilities are named abstract application requirements
- Components are concrete named engines that satisfy the requirements

- Dynamic control:
  - *the application script author provides*
    - *a specification of application facilities as part of the* **`Application`** *definition*
    - *a component to be used as the default*
  - *the user can construct scripts that create alternative components that comply with facility interface*
  - *the end user can*
    - *configure the properties of the component*
    - *select which component is to be bound to a given facility at runtime*

Inversion of control

- A feature of component frameworks
  - *applications require facilities and invoke the services they promise*
  - *component instances that satisfy these requirements are injected at the latest possible time*

- The pyre solution to this problem
  - *eliminates the complexity by using "service locators"*
  - *takes advantage of the dynamic programming possible in python*
  - *treats components and their initialization state fully symmetrically*
  - *provides simple but acceptable persistence (performance, scalability)*
    - *XML files, python scripts*
    - *an object database on top of the filesystem*
  - *can easily take advantage of other object stores*
  - *is ideally suited for both parallel and distributed applications*
    - *gsl: "grid services lite"*

Summary

- Existing services:
  - *Well understood strategy for code integration*
    - *legacy codes, community efforts*
    - *interfaces to MATLAB, IDL, ACIS*
  - *Flexible environment for composing applications*
    - *component lifecycle management*
    - *decoupling from user interfaces*
- Under development
  - *database access*
  - *enhanced support for distributed computing*
    - *XMLRPC, SOAP, web services*
    - *CCA?*
  - *web portals*
  - *looking for a suitable workflow GUI*

Overview of selected services

- Services described here
  - *application structure*
  - *properties: types and units*
  - *parallelism and staging*
  - *facilities, components*
  - *application monitoring*
  - *geometry specification*
  - *simulation control: controller, solver*
  - *support for integrated visualization*
  - *enabling distributed computing*

HelloApp

```
from pyre.application.Script import Script

class HelloApp(Script):

    def main(self):
        print "Hello world!"
        return

    def __init__(self):
        Script.__init__(self, "hello")
        return

# main
if __name__ == "__main__":
    app = HelloApp()
    app.run()
```

access to the base class

- Output

```
> ./hello.py
Hello world!
```


Properties

- Named attributes that are under direct user control
  - *automatic conversions from strings to all supported types*
- Properties have
  - *name*
  - *default value*
  - *optional validator functions*
- Accessible from pyre.properties
  - *factory methods:* **`str`**, **`bool`**, **`int`**, **`float`**, **`sequence`**, **`dimensional`**
  - *validators:* **`less`**, **`greater`**, **`range`**, **`choice`**

```
import pyre.inventory

flag = pyre.inventory.bool("flag", default=True)
name = pyre.inventory.string("name", default="pyre")
scale = pyre.inventory.float(
                "scale", default=1.0, validator=pyre.inventory.greater(0))
```

- The user can derive from Property to add new types

Units

- Properties can have units:
  - *framework provides:* **`dimensional`**
- Support for units is in **`pyre.units`**
  - *full support for all SI base and derived units*
  - *support for common abbreviations and alternative unit systems*
  - *correct handling of all arithmetic operations*
    - *addition, multiplication, functions from* **`math`**

```
import pyre.inventory
from pyre.units.time import s, hour
from pyre.units.length import m, km, mile

speed = pyre.inventory.dimensional("speed", default=50*mile/hour)

v = pyre.inventory.dimensional(
                            "velocity", default=(0.0*m/s, 0.0*m/s, 10*km/s))
```


HelloApp: adding properties

```
from pyre.application.Script import Script

class HelloApp(Script):
    …

    class Inventory(Script.Inventory):

        import pyre.inventory

        name = pyre.inventory.str("name", default="world")

    …
```

framework property factories

HelloApp: using properties

```
from pyre.application.Script import Script

class HelloApp(Script):

    def main(self):
        print "Hello %s!" % self.inventory.name
        return

    def __init__(self):
        Application.__init__(self, "hello")
        return

    …
```

accessing the property value

- Now the name can be set by the user interface

```
> ./hello.py --name="Michael"
Hello Michael!
```


Support for concurrent applications

- Python as the driver for concurrent applications that
  - *are embarrassingly parallel*
  - *have custom communication strategies*
    - *sockets, ICE, shared memory*
- Excellent support for MPI
  - **`mpipython.exe`**: *MPI enabled interpreter (needed only on some platforms)*
  - **`mpi`**: *package with python bindings for MPI*
    - *support for staging and launching*
    - *communicator and processor group manipulation*
    - *support for exchanging python objects among processors*
  - **`mpi.Application`**: *support for launching and staging MPI applications*
    - *descendant of* **`pyre.application.Application`**
    - *auto-detection of parallelism*
    - *fully configurable at runtime*
    - *used as a base class for user defined application classes*

Parallel Python

- Enabling parallelism in Python is implemented by:
  - *embedding the interpreter in an MPI application:*

```
int main(int argc, char **argv) {
    int status = MPI_Init(&argc, &argv);
    if (status != MPI_SUCCESS) {
        std::cerr << argv[0]
             << ": MPI_Init failed! Exiting ..." << std::endl;
        return status;
    }
    status = Py_Main(argc, argv);
    MPI_Finalize();
    return status;
}
```

  - *constructing an extension module with bindings for MPI*
  - *providing an objected oriented veneer for easy access*

Access to MPI through Pyre

```
import mpi

# get the world communicator
world = mpi.world()

# compute processor rank in MPI_COMM_WORLD
rank = world.rank

# create a new communicator
new = world.include([0])
if new:
    print "world: %d, new: %d" % (rank, new.rank)
else:
    print "world: %d (excluded from new)" % rank
```

creates a new communicator by manipulating the communicator group

Parallel HelloApp

```
from mpi.Application import Application

class HelloApp(Application):

    def main(self):
        import mpi
        world = mpi.world()
        print "[%03d/%03d] Hello world" % (world.rank, world.size)
        return

    def __init__(self):
        Application.__init__(self, "hello")
        return

# main
if __name__ == "__main__":
    app = HelloApp()
    app.run()
```

new base class

Staging

- The new base class **`mpi.Application`**
  - *overrides the initialization protocol*
  - *gives the user access to the application launching details*

excerpt from mpi/Application/Application.py

```
from pyre.application.Application import Application as Base

class Application(Base):

    …
    class Inventory(Base.Inventory):

        import pyre.inventory
        from LauncherMPICH import LauncherMPICH

        mode = pyre.inventory.str("mode", default="server",
                                  validator=pyre.inventory.choice(["server", "worker"])

        launcher = pyre.inventory.facility("launcher", factory=LauncherMPICH)
    …
```


Facilities and components

- A design pattern that enables the assembly of application components at run time under user control
- Facilities are named abstract application requirements
- Components are concrete named engines that satisfy the requirements

- Dynamic control:
  - *the application script author provides*
    - *a specification of application facilities as part of the* **`Application`** *definition*
    - *a component to be used as the default*
  - *the user can construct scripts that create alternative components that comply with facility interface*
  - *the end user can*
    - *configure the properties of the component*
    - *select which component is to be bound to a given facility at runtime*

Auto-launching

```
...

def execute(self, *args, **kwds):
    if self.inventory.mode == "worker":
        self.onComputeNodes(*args, **kwds)
        return

    self.onServer(*args, **kwds)
    return

def onComputeNodes(self, *args, **kwds):
    self.run(*args, **kwds)
    return

def onServer(self, *args, **kwds):
    launched = self.inventory.launcher.launch()
    if not launched:
        self.onComputeNodes(*args, **kwds)

    return

...
```

excerpt from mpi/Application/Application.py

invokes **`mpirun`** *or the batch scheduler*

Launcher properties

- **`LauncherMPICH`** defines the following properties
  - **`nodes`**: *the number of processors (int)*
  - **`dry`**: *do everything except the actual call to* **mpirun** *(bool)*
  - **`command`**: *specify an alternative to* **`mpirun`** *(str)*
  - **`extra`**: *additional command line arguments to* **`mpirun`** *(str)*
- Running the parallel version of HelloApp:

```
./hello.py --launcher.nodes=4 --launcher.dry=on
```

- Specifying an alternate staging configuration
  - *assuming that a properly constructed asap.py is accessible*

```
./hello.py --launcher=asap --launcher.nodes=4
```

- or the equivalent

```
./hello.py --launcher=asap --asap.nodes=4
```


Inversion of control

- A feature of component frameworks
  - *applications require facilities and invoke the services they promise*
  - *component instances that satisfy these requirements are injected at the latest possible time*

- The pyre solution to this problem
  - *eliminates the complexity by using "service locators"*
  - *takes advantage of the dynamic programming possible in python*
  - *treats components and their initialization state fully symmetrically*
  - *provides simple but acceptable persistence (performance, scalability)*
    - *XML files, python scripts*
    - *an object database on top of the filesystem*
  - *can easily take advantage of other object stores*
  - *is ideally suited for both parallel and distributed applications*
    - *gsl: "grid services lite"*

Services for solid and fluid modeling

- Problem specification
  - *components and their properties*
- Solid modeling
  - *overall geometry*
  - *model construction*
  - *topological and geometrical information*
- Boundary and initial conditions
  - *high level specification*
  - *access to the underlying solver data structures in a uniform way*
- Materials and constitutive models
  - *materials properties database*
  - *strength models and EOS*
  - *association with a region of space*
- Computational engines
  - *selection and association with geometry*
  - *solver specific initializations*
  - *coupling mechanism specification*
- Simulation driver
  - *initialization*
  - *appropriate time step computation*
  - *orchestration of the data exchange*
  - *checkpoints and field dumps*
- Active monitoring
  - *instrumentation: sensors, actuators*
  - *real-time visualization*
- Full simulation archiving

Simulation monitoring

- **journal**: a component for managing application diagnostics
  - *error, warning, info*
  - *debug, firewall*
- Named categories under global dynamic control for
  - *python, C, C++, FORTRAN*

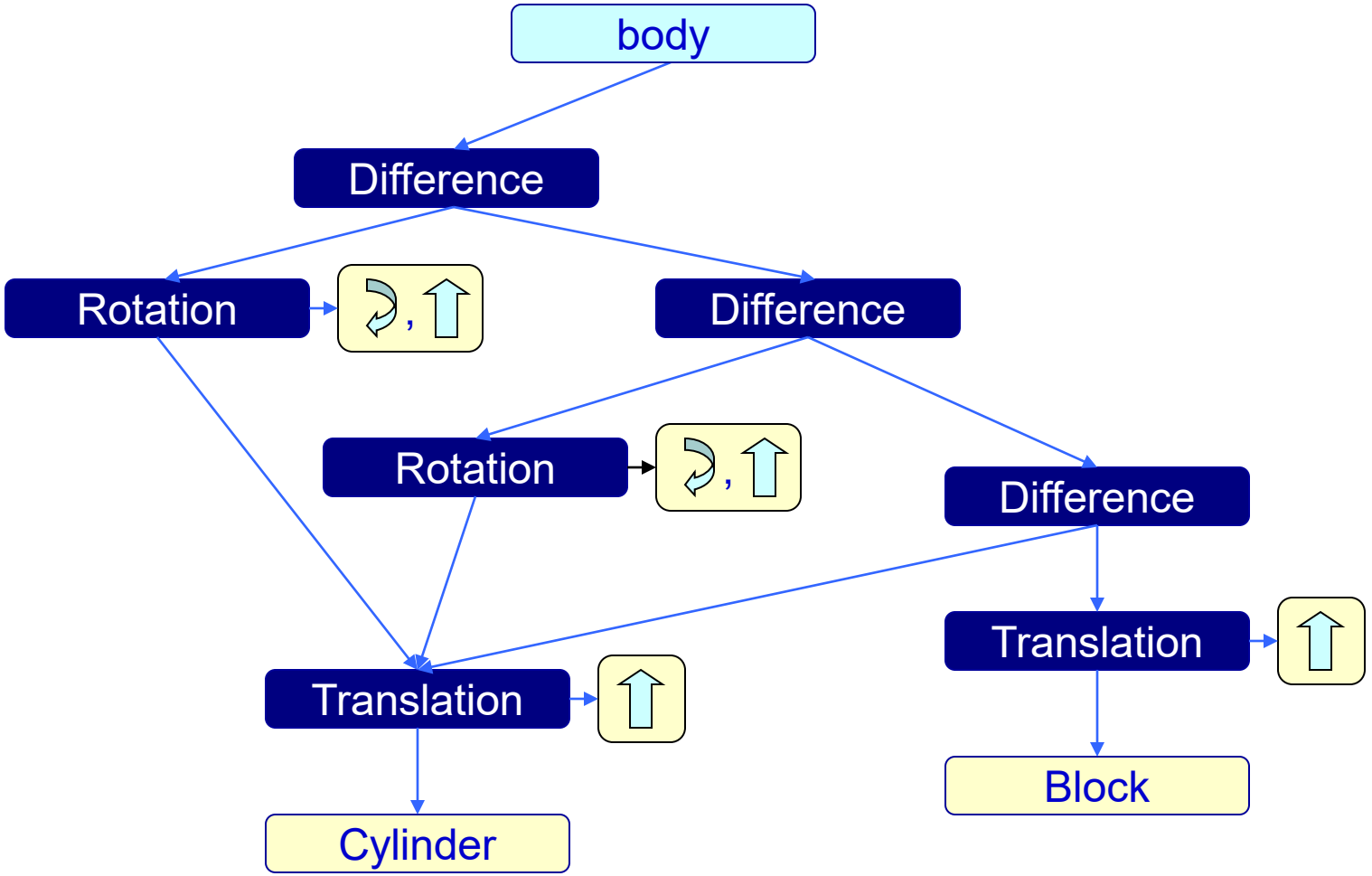
```
import journal

debug = journal.debug("hello")
debug.activate()
debug.log("this is a diagnostic")
```

- Customizable
  - *message content*
  - *meta-data*
  - *formatting*
  - *output devices: console, files, sockets for remote transport*
- **journal** is a component!

Geometry specification

```
def geometry():
    from pyre.units.length import mm
    side = 50.0*mm
    diameter = 25.0*mm
    scale = 5

    from pyre.geometry.solids import block,cylinder
    from pyre.geometry.operations import rotate,subtract,translate

    cube = block((side, side, side))
    cube = translate(cube, (-side/2, -side/2, -side/2))
    hole = cylinder(height=2*side, radius=diameter/2)
    z_hole = translate(hole, (0*side, 0*side, -side)
    y_hole = rotate(z_hole, (1,0,0), pi/2)
    x_hole = rotate(z_hole, (0,1,0), pi/2)

    body = subtract(body, x_hole)
    body = subtract(body, y_hole)
    body = subtract(body, z_hole)

    ils = min(radius, side - diameter)/scale

    return pyre.geometry.body(body, ils)
```


Geometry specification graph



Creating the model

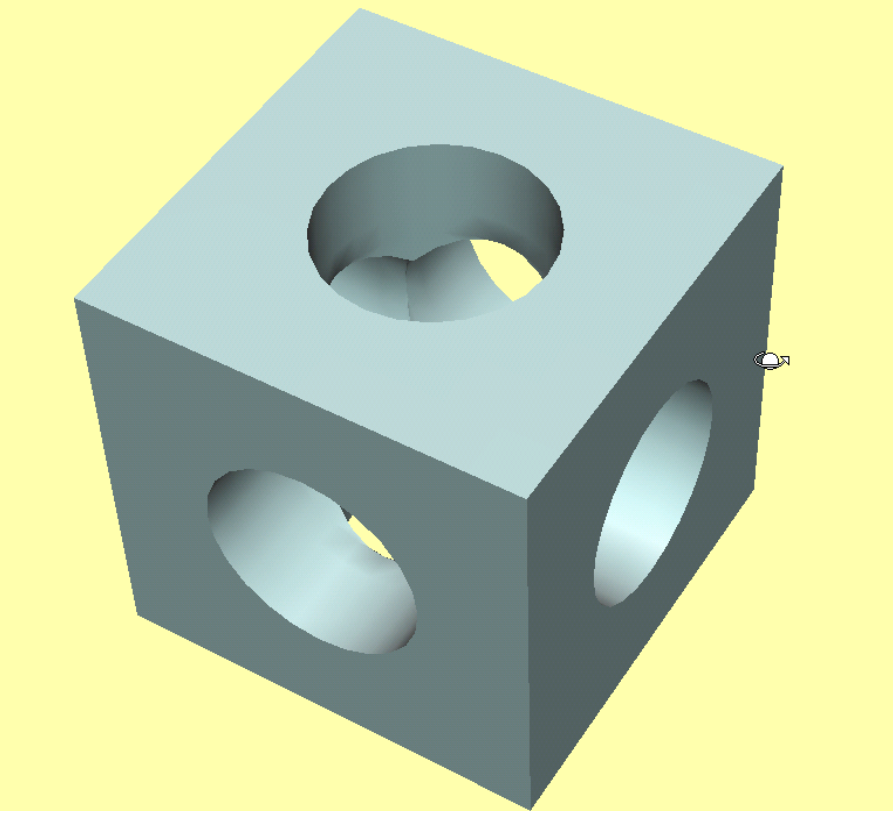
abstract representation of the model

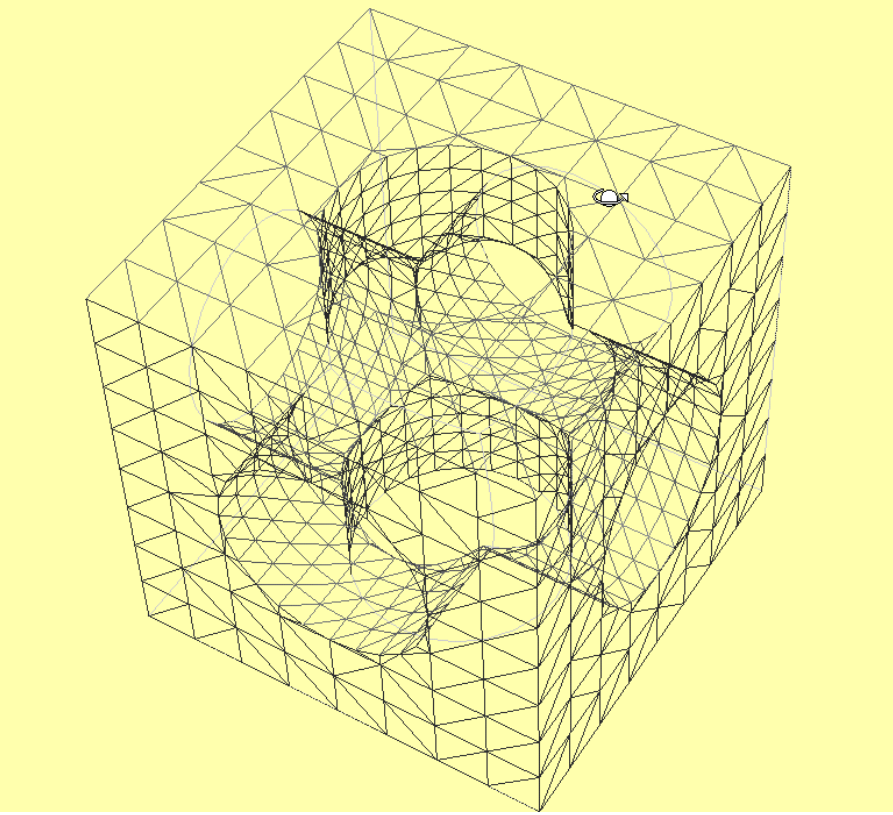
```
def mesh(model):
    body = createBody(model)

    import acis
    faceter = acis.faceter()

    properties = faceter.properties
    properties.gridAspectRatio = 1.0
    properties.maximumEdgeLength = body.ils

    boundary = faceter.facet(acis.create(body.geometry))
    bbox = boundary.boundingBox()

    return boundary, bbox
```

the Python bindings for the solid modeler

convert the abstract geometrical description into an actual instance of the ACIS class **BODY**

The finished model





Simulation control

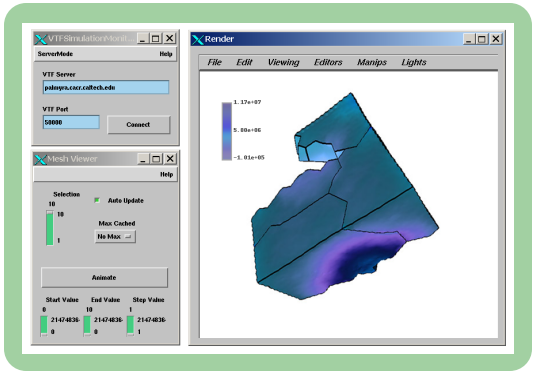
- Achieved through the collaboration of two components
  - **`SimulationController`** *(controller)*
  - **`Solver`** *(solver)*
- SimulationController expects Solver to conform to the following interface
  - **`initialize()`**, **`launch()`**
  - **`startTimestep()`**, **`endTimestep()`**
  - **`applyBoundaryConditions()`**
  - **`stableTimestep()`**, **`advance()`**
  - *save:* **`publishState()`**, **`plotFile()`**, **`checkpoint()`**
- Both components provide (trivial but usable) default implementations
- The facility/component pattern enables the selection and initialization of solvers by the end user

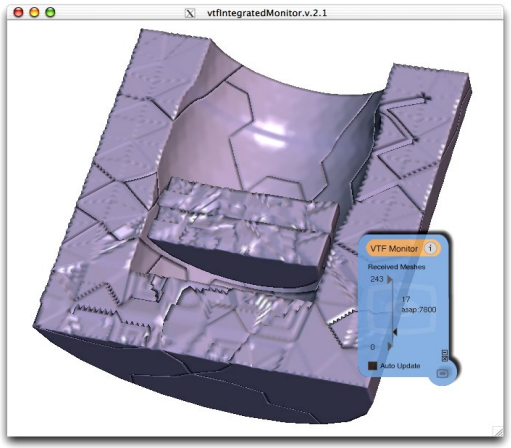
Advancing the solution in time

```
from pyre.components.Component import Component

class SimulationController(Component):

    def march(self, totalTime=0, steps=0):
        while 1:
            solver.startTimestep()
            solver.applyBoundaryConditions()
            solver.save()
            dt = solver.stableTimestep()
            solver.advance(dt)

            self.clock += dt
            self.step += 1
            if totalTime and self.clock >= totalTime: break
            if steps and self.step >= step: break

        solver.endSimulation()
        return
```


Visualization

- Support for integrated remote visualization is provided by simpleviz





- Three tier system:
  - *a data source embedded in the simulation*
  - *a data server*
  - *a client: visualization tools (such as IRIS Explorer)*
- The server
  - *is a daemon that listens for socket connections from*
    - *data sources (mostly pyre simulations)*
    - *visualization tools*
- The client
  - *is a default facility for vtf.Application*
  - *has the hostname and port number of the server*

Summary

- We have covered many framework aspects
  - *application structure*
  - *support for parallelism*
  - *properties, facilities and components*
  - *geometry specification*
  - *the controller-solver interface*
  - *support for embedded visualization*
  - *support for distributed computing*
- There is support for much more…