# ACM/CS 114
# Parallel algorithms for scientific applications

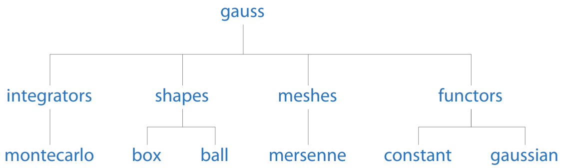
Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

# Namespace design

we are now in a position to assemble the package `gauss`; let's start by laying out the package namespace




and try to use this layout for both the logical and physical structure

- the top level is our package name
- the internal nodes become the names of interfaces and subdirectories
- the leaves are the component family names and the names by which the component factories are accessible

# The shapes package

in order to make the directory `gauss/shapes` a python package, we need to create the special file `gauss/shapes/__init__.py`

```python
"""
Package that contains the implementations of shapes
"""

# the interface
from .Shape import Shape as shape

# the components
from .Box import Box as box
from .Ball import Ball as ball
```

the `import` statements

- use *local* imports to make sure that we are accessing the correct modules
- create local names for the classes declared inside the named modules

the net effect is to simplify access to the components

```python
from gauss.shapes import box, ball
```

# Shapes

## the `Shape` interface in `gauss/shapes/Shape.py`

```python
import pyre

class Shape(pyre.interface, family="gauss.shapes"):
    """
    The obligations of implementations of geometrical shapes
    """

    # my default implementation
    @classmethod
    def default(cls):
        """
        The default {Shape} implementation
        """
        # use a box
        from .Box import Box
        return Box # if you return an instance, it will be shared by all...

    # required interface
    @pyre.provides
    def measure(self):
        """
        Compute my measure: length, area, volume, etc
        """

    @pyre.provides
    def interior(self, points):
        """
        Filter out {points} that are on my exterior
        """
```

# From disks to spheres in $d$ dimensions

- for the simple shapes, such as boxes and disks, it is easy to generalize to arbitrary dimensions
  - for our purposes, this is useful mostly as an exercise in operating on containers
- the volume of a sphere of radius $r$ in $d$ dimensions is given by

$$\mu_d(r) = \frac{\pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2}+1\right)} r^d$$

for even $d$

$$\mu_d(r) = \frac{\pi^{\frac{d}{2}}}{\left(\frac{d}{2}\right)!} r^d$$

for odd $d$

$$\mu_d(r) = \frac{2^{\frac{d+1}{2}} \pi^{\frac{d-1}{2}}}{d!!} r^d$$

# Ball

## the implementation of `Ball` in `gauss/shapes/Ball.py`

```python
import pyre
from .Shape import Shape

class Ball(pyre.component, family="gauss.shapes.ball", implements=Shape):
    """
    A representation of the interior of a spere in $d$ dimensions
    """
    # public state
    center = pyre.properties.array(default=(0,0))
    center.doc = "the location of the center of the ball"

    radius = pyre.properties.float(default=1)
    radius.doc = "the radius of the ball"

    # interface
    @pyre.export
    def measure(self):
        """
        Compute my volume
        """
        # externals
        from math import pi
        import functools, operator
        # compute the dimension of space
        d = len(self.center)
        # for even $d$
        if d % 2 == 0:
            # compute the volume
            normalization = functools.reduce(operator.mul, range(1, d/2+1))
            return pi**(d/2) * self.radius**d / normalization
        # for odd {d}
        normalization = functools.reduce(operator.mul, range(1, d+1, 2))
        return 2**((d+1)/2) * pi**((d-1)/2) / normalization
```

# Ball - continued

```python
    @pyre.export
    def interior(self, points):
        """
        Filter out the members of {points} that are exterior to this ball
        """
        # cache the center of the ball
        center = self.center
        # compute the radius squared
        r2 = self.radius**2
        # for each point
        for point in points:
            # compute the distance from the center
            d2 = sum((p - r)**2 for p,r in zip(point, center))
            # check whether this point is inside or outside
            if r2 >= d2:
                yield point
        # all done
        return
```

# Box

the implementation of `Box` in `gauss/shapes/Box.py`

```python
import pyre
from .Shape import Shape

class Box(pyre.component, family="gauss.shapes.box", implements=Shape):
    """
    A representation of the interior of a $d$-dimensional box
    """
    # public state
    diagonal = pyre.properties.array(default=((0,0),(1,1)))
    diagonal.doc = "a vector that specifies the major diagonal of the box"

    # interface
    @pyre.export
    def measure(self):
        """
        Compute my volume
        """
        # externals
        import functools, operator
        # compute and return the volume
        return functools.reduce(
            operator.mul, ((right-left) for left,right in self.intervals()))
```

# Box - continued

```
    def interior(self, points):
        """
        Filter out the members of {points} that are exterior to this box
        """
        # form the list of intervals alomg each cordinate axis
        intervals = tuple(self.intervals()) # expand and store
        # now, for each point
        for point in points:
            # for each cordinate
            for p, (left,right) in zip(point, intervals):
                # if this point is outside the box
                if p < left or p > right:
                    # move on to the next point
                    break
            # if we got here, all tests passed, so
            else:
                # this one is on the interior
                yield point
        # all done
        return

    # utilities
    def intervals(self):
        """
        Re-pack the diagonal vector as a list of the intervals along each axis
        """
        return zip(*self.diagonal)
```

# The `meshes` package

again, we need the special file `gauss/meshes/__init__.py` in order to turn `gauss/meshes` into a python package

```python
"""
Package that contains the implementations of point clouds
"""

# the interfaces
from .PointCloud import PointCloud as cloud

# the components
from .Mersenne import Mersenne as mersenne
```

# Point clouds

the `PointCloud` interface in `gauss/meshes/PointCloud.py`

```python
import pyre

class PointCloud(pyre.interface, family="gauss.meshes"):
    """
    The abstract base class for point generators
    """

    # the default implementation
    @classmethod
    def default(cls):
        """
        The default {PointCloud} implementation
        """
        from .Mersenne import Mersenne
        return Mersenne

    # required interface
    @pyre.provides
    def points(self, n, box):
        """
        Generate {n} random points on the interior of {box}
        parameters:
            {n}: the number of points to generate
            {box}: the major diagonal of the computational domain
        """
```

# Generating points with the Mersenne Twister RNG

in `gauss/meshes/Mersenne.py`

```
import pyre, random, itertools
from .PointCloud import PointCloud

class Mersenne(pyre.component, family="gauss.meshes.mersenne",
               implements=PointCloud):
    """
    A point generator implemented using the Mersenne Twister random number
    generator that is available as part of the python standard library
    """

    # interface
    @pyre.export
    def points(self, n, box):
        """
        Generate {n} random points in the interior of {box}
        """
        # unfold the bounding box
        intervals = tuple(box.intervals()) # realize, so we can reuse in the loop
        # loop over the sample size
        while n > 0:
            # make a point and yield it
            yield tuple(itertools.starmap(random.uniform, intervals))
            # update the counter
            n -= 1
        # all done
        return
```

# The `functors` package

the package initialization file in `gauss/functors/__init__.py`

```python
"""
Package with functor definitions
"""

# the interface
from .Functor import Functor as functor

# the components
from .Constant import Constant as constant
from .Gaussian import Gaussian as gaussian
```

# Functors

the `Functor` interface in `gauss/functors/Functor.py`

```python
import pyre

class Functor(pyre.interface, family="gauss.functors"):
    """
    The abstract base class for function objects
    """

    # the default implementation
    @classmethod
    def default(cls):
        """
        The default {Functor} implementation
        """
        from .Constant import Constant
        return Constant

    # required interface
    @pyre.provides
    def eval(self, points):
        """
        Evaluate the function at the supplied points
        """
```

# The `Constant` functor

in `gauss/functors/Constant.py`

```python
import pyre
from .Functor import Functor

class Constant(pyre.component, family="gauss.functors.constant",
               implements=Functor):
    """
    A representation of constant functions
    """

    # public state
    value = pyre.properties.float(default=1)
    value.doc = "the value of the constant functor"""

    # interface
    @pyre.export
    def eval(self, points):
        """
        Compute the value of the function
        """
        # cache the constant
        constant = self.value
        # return the constant regardless of the evaluation point
        for point in points: yield constant
        # all done
        return
```

# A non-trivial functor

```python
import pyre
from .Functor import Functor

class Gaussian(pyre.component, family="gauss.functor.gaussian",
               implements=Functor):
    """
    An implementation of the normal distribution with
    mean μ and variance σ²
    """

    # public state
    mean = pyre.properties.array(default=[0])
    mean.doc = "the mean of the gaussian distribution"
    mean.aliases.add("μ")

    spread = pyre.properties.float(default=1)
    spread.doc = "the variance of the gaussian distribution"
    spread.aliases.add("σ")
```

# A non-trivial functor – continued

```
    # interface
    @pyre.export
    def eval(self, points):
        """
        Compute the value of the gaussian
        """
        # access the math symbols
        from math import exp, sqrt, pi
        # cache the shape information
        mean = self.mean
        spread = self.spread
        # precompute the normalization factor and the exponent scaling
        normalization = 1 / sqrt(2*pi) / spread
        scaling = 2 * spread**2
        # loop over points and yield the computed value
        for p in points:
            # compute the norm |p - mean|^2
            # this works as long as {p} and {mean} have the same length
            r2 = sum((p_i - mean_i)**2 for p_i, mean_i in zip(p, mean))
            # yield the value at the current p
            yield normalization * exp(- r2/scaling)
        # all done
        return
```

# The `integrators` package

the package initialization file in `gauss/integrators/__init__.py`

```python
"""
Package with integrator implementations
"""

# the interface
from .Integrator import Integrator as integrator

# the component
from .MonteCarlo import MonteCarlo as montecarlo
```

# Integrators

## in gauss/integrators/Integrator.py

```python
import pyre

class Integrator(pyre.interface, family="gauss.integrators"):
    """
    Interface declarator for integrators
    """

    # access to the required interfaces
    from ..shapes import shape
    from ..functors import functor

    # required public state
    region = pyre.facility(interface=shape)
    region.doc = "the region of integration"

    integrand = pyre.facility(interface=functor)
    integrand.doc = "the functor to integrate"

    # my preferred implementation
    @classmethod
    def default(cls):
        # use {MonteCarlo} by default
        from .MonteCarlo import MonteCarlo
        return MonteCarlo

    # required interface
    @pyre.provides
    def integrate(self):
        """
        Compute the integral of {integrand} over {region}
        """
```

# The Monte Carlo integrator

## in `gauss/integrators/MonteCarlo.py`

```python
# externals
import pyre
from ..meshes import cloud
from ..functors import functor
from ..shapes import shape, box, ball
from .Integrator import Integrator

class MonteCarlo(pyre.component, family="gauss.integrators.montecarlo",
             implements=Integrator):
    """
    A Monte Carlo integrator
    """

    # public state
    samples = pyre.properties.int(default=10**5)
    samples.doc = "the number of integrand evaluations"

    box = pyre.facility(interface=shape, default=box)
    box.doc = "the bounding box for my mesh"

    mesh = pyre.facility(interface=cloud)
    mesh.doc = "the generator of points at which to evaluate the integrand"

    region = pyre.facility(interface=shape, default=ball)
    region.doc = "the shape that defines the region of integration"

    integrand = pyre.facility(interface=functor)
    integrand.doc = "the functor to integrate"
```

# The Monte Carlo integrator – continued

```
    # interface
    @pyre.export
    def integrate(self):
        """
        Compute the integral as specified by my public state
        """
        # compute the overall normalization
        normalization = self.box.measure()/self.samples
        # get the set of points
        points = self.mesh.points(n=self.samples, box=self.box)
        # select the points interior to the region of integration
        interior = self.region.interior(points)
        # sum up and scale the integrand contributions
        integral = normalization * sum(self.integrand.eval(interior))
        # and return the value
        return integral
```

# Top level – the `gauss` package

## the package initialization file in `gauss/__init__.py`

```python
"""
This is the implementation for the interfaces and components of {gauss},
a sample pyre application. See gauss.license() for terms of use.
"""

# access to the package contents
from . import functors, integrators, meshes, shapes

# factories
montecarlo = integrators.montecarlo


# misc
def copyright():
    """
    Return the {gauss} copyright note
    """
    return _gauss_copyright

def license():
    """
    Return the {gauss} license
    """
    return _gauss_license

def version():
    """
    Return the {gauss} version
    """
    return _gauss_version
```

# Checking that all is ok

assuming that the directory `gauss` is somewhere on the python path, we are now ready to check that everything works

```
mga@pythia:~/dv/acm114/2012-spring/lectures>python
Python 3.2.3 (default, Apr 19 2012, 01:32:56)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
enabling readline
>>> import gauss
>>> mc = gauss.montecarlo()
>>> mc.samples
10000
>>> mc.box.diagonal
((0, 0), (1, 1))
>>> mc.region
<gauss.shapes.Ball.Ball object at 0x1083c5910>
>>> mc.region.radius
1.0
>>> mc.region.center
(0, 0)
>>> mc.integrand
<gauss.functors.Constant.Constant object at 0x1083c59d0>
>>> mc.integrand.value
1.0
>>> 4 * mc.integrate()
3.12672
```

# More on configuration files

there are a few more pieces of functionality that we haven't covered

- assignments involving expressions and references
- wiring shortcuts for properly designed package namespaces
- having multiple configurations for the same property in a given file
- wiring a facility to a specific, perhaps preëxisting component

here is a configuration file that uses all of them

```
one = 1

[ mc ] ; configure our Monte Carlo integrator instance
samples = 10**6
region = ball#frisbee ; equivalent to import:gauss.shapes.ball#frisbee
integrand = constant ; equivalent to import:gauss.functors.constant

[ gauss.functors.constant # mc.integrand ] ; if mc.integrand is a constant
value = {one}

[ gauss.functors.gaussian # mc.integrand ] ; if mc.integrand is a gaussian
mean = (0, 0)
spread = {one}/3
```