pyre 1.0

Michael A. G. Aïvázis

orthologue

michael.aivazis@orthologue.com

January 2015

Introduction

- `pyre` is a strategy for
  - managing code complexity
  - integrating third party tools and libraries into a coherent whole
  - empowering the end-user to make critical decisions about the composition of an application while minimizing the risk of compromising its integrity
- `pyre` extends object oriented ideas
  - abstract base classes become *protocols*
  - appropriately decorated classes become *components*
  - design and implement by contract
- `pyre` is also a powerful computational environment with rich services
  - application configuration
  - launching and staging in serial, parallel, distributed modes
  - logging and monitoring
  - special services for interacting with users via the production of structured documents
    - think `html` for web applications, remote UIs
  - name and filesystem abstractions
  - powerful lazy evaluation mechanisms
  - seamless access to database back-ends without the need for direct access using embedded `SQL` or similar techniques

Monte Carlo integration

- let f be sufficiently well behaved in a region $\Omega \subset \mathbb{R}^n$ and consider the integral

$$I_\Omega(f) = \int_\Omega f \tag{1}$$

- the *Monte Carlo* method approximates the value of the integral in Eq. 1 by sampling f at random points in Ω
- let X_N be such a sample of N points; then the Monte Carlo estimate is given by

$$I_\Omega(f; X_N) = \Omega \cdot \langle f \rangle = \Omega \frac{1}{N} \sum_{x \in X_N} f(x) \tag{2}$$

  where $\langle f \rangle$ is the sample mean of f, and Ω is used as a shorthand for the volume of the integration region. Details in [1, 2]; see [3] for an excellent pedagogical introduction.
- the approximation error falls like $1/\sqrt{N}$
  - rather slow
  - but dimension independent!

Implementation strategy

- computer implementations require a pseudo-random number generator to build the sample
- most generators return numbers in $(0, 1)$ so
  - find a box B that contains Ω
  - generate n numbers to build a point in the unit $\mathbb{R}^n$ cube
  - stretch and translate the unit cube onto B
- the integration is restricted to Ω by introducing

$$\Theta_\Omega = \begin{cases} 1 & x \in \Omega \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

to get

$$I_\Omega(f) = \int_B \Theta_\Omega f \tag{4}$$

Recasting Monte Carlo integration

- there are now two classes of points in the sample X_N
  - those in Ω
  - and the rest
- let $\tilde{N}$ be the number of sample points in Ω; Eq. 2 becomes

$$I_\Omega(f; X_N) = \Omega \frac{1}{\tilde{N}} \sum_{x \in X_{\tilde{N}}} f(x) \tag{5}$$

- let B be the volume of the sampling box; observe that the volume of the integration region can be approximated by

$$\Omega = \frac{\tilde{N}}{N} B \tag{6}$$

and the sum over the points $x \in X_{\tilde{N}}$ can be extended to the entire sample X_N by using the filter Θ_Ω

$$I_\Omega(f; X_N) = B \frac{1}{N} \sum_{x \in X_N} \Theta_\Omega f(x) \tag{7}$$

Requirements

- to summarize, the Monte Carlo approximation is computed from

$$I_\Omega(f; X_N) = B \frac{1}{N} \sum_{x \in X_N} \Theta_\Omega f(x) \tag{8}$$

  using
  - an implementation of the function f to be integrated over Ω
  - an n-dimensional box B that contains Ω
  - a good pseudo-random number generator to build the sample $X_N \in B$
  - a routine to test points $x \in X_N$ and return `false` if they are exterior to Ω and `true` otherwise

  to sum the values of the integrand on points interior to Ω, and scale by the volume of the bounding box B over the sample size N
- essentially a reduction
  - should be straightforward to implement in parallel
- rich enough structure to be a non-trivial `pyre` application

A trivial script

- estimating π using Monte Carlo integration over a quarter disk

```
# get access to the random munber generator functions
import random
# sample size
N = 10**5
# initialize the interior point counter
interior = 0
# integrate by sampling some number of times
for i in range(N):
    # build a random point
    x = random.random()
    y = random.random()
    # check whether it is inside the unit quarter circle
    if (x*x + y*y) <= 1.0: # no need to waste time computing the sqrt
        # update the interior point counter
        interior += 1
# print the result:
print("π: {:.8f}".format(4*interior/N))
```

Components and protocols

- a design pattern that enables the assembly of applications out of interchangeable parts, under the control of the *end user*
  - *protocols* are abstract specifications of application requirements
  - *components* are concrete implementations that satisfy requirements
- inversion of control:
  - the binding of implementations to specifications happens at runtime, under the control of the end user
- the user
  - controls the application state through configuration files, the user interface, the command line
  - specifies components using simple URIs
- the goal is to isolate contributors from each other as much as possible, and provide a coherent and usable strategy for composing non-trivial applications

Small steps: properties

- let's step back and contemplate a simpler problem

```
class Disk:

    # public state
    radius = 1 # default value
    center = (0,0) # default value

    # interface
    def interior(self, points):
        ...
```

- what do we have to do to tie instances of `Disk` with information in some configuration file?

```
    [ disk1 ]
    center = (-1,1) ; leave {radius} alone

    [ disk2 ]
    radius = .5
    center = (1,1)
```

or, equivalently, from the command line

```
gauss.py --disk1.center=(1,1) --disk2.radius=.5 --disk2.center=(-1,1)
```

Components

- informally, *classes* are software specifications that establish a relationship between *state* and *behavior*
  - we have syntax that allows us to specify these very close to each other
- *instances* are containers of state; there are special rules
  - that grant access to this state
  - allow you to call functions that get easy access to this state
- *components* are classes that specifically grant access to some of their state to the end user
  - the public data are the *properties* of the component
- rule 1: components have properties

A trivial component

- `pyre` provides support for writing components

```
1  import pyre
2
3  class Disk(pyre.component):
4
5      # public state
6      radius = pyre.properties.float(default=1)
7      radius.doc = 'the radius of the disk'
8
9      center = pyre.properties.array()
10     center.default = (0,0)
11     center.doc = 'the location of the center of the circle'
12
13     # interface
14     ...
```

- why bother specifying the type of component properties?
  - command line, configuration files, dialog boxes, web pages: they all gather information from the user as strings
  - we need *meta-data* so we can convert from strings to the intended object

The names of things

- in order to connect components to configurations, we need explicit associations
  - component instances have unique names
  - component classes have unique family names
  - namespace management: components belong to packages

```
1  import pyre
2
3  class Disk(pyre.component, family="gauss.shapes.disk"):
4
5      # public state
6      radius = pyre.properties.float()
7      radius.default = 1
8      radius.doc = 'the radius of the disk'
9
10     center = pyre.properties.array()
11     center.default = (0,0)
12     center.doc = 'the location of the center of the circle'
13     ...
```

- and here are a couple of component instances

```
1  left = Disk(name='disk1')
2  right = Disk(name='disk2')
```

Configuration

- rule 2: components have names
- the package name is deduced from the component family name
  - it is the part up to the first delimiter
- `pyre` automatically loads configuration files whose name matches the name of a package
- there's even a way to override the default values that the developer hardwired into the class declaration

```
1    [ gauss.shapes.disk ]  ; use the family name
2    radius = 2             ; to override the defaults
3    center = (-1,-1)       ; for all instances of disk
4
5    [ disk1 ]              ; the name of an instance
6    center = (-1,1)        ; leave {radius} alone
7
8    [ disk2 ]              ; the name of another instance
9    radius = .5
10   center = (1,1)
```

Recap: what we know so far

- `pyre` components are evolved python objects
  - the factories have family names, the instances have names
  - these names are unique strings in hierarchical namespaces delimited by periods
  - collections of components form packages *implicitly*, based on the topmost level in their namespace
- components have properties that are under the control of the *user*
  - they look and behave like regular attributes
  - they are *typed* to enable conversions from strings
  - they have default values and other metadata
- configuration is partly about assigning values to component properties
  - a requirement for supporting user interfaces
  - intuitive syntax for the command line
  - simple configuration files inspired by the Microsoft Windows `.ini` format
- configuration is automatically handled by the framework and requires no explicit involvement on the part of the component author

Configuration files

- currently, there are two file formats for configuration information
  - `.cfg`: the format in the examples
  - `.pml`: an `XML` based format that is a bit more powerful but not as user friendly
- `pyre` looks for configuration files in the following places
  - explicitly provided on the command line

```
1 gauss.py --config=sample.cfg
```

  - the current directory
  - the `.pyre` subdirectory of the current user's home directory
  - a special subdirectory wherever `pyre` is installed

  in order of priority
- settings on the command line have the highest priority, and override each other from left to right
- when a property is assigned a value multiple times, the highest priority setting wins
  - the framework keeps track of all changes in the value of properties and the source of the assignment, so if a property doesn't end up with the value you expected, you can get its complete history

Properties

- properties make sense for both classes and instances
  - the class holds the default value that gets used in case the component instance does not have explicit configuration
  - each instance gets its own private value when it gets configured
  - identical to regular python attributes
- there is support for
  - simple types: `bool`, `int`, `float`, `str`
  - containers: `tuple`, `array`
  - higher level: `date`, `time`, `inputfile`, `outputfile`, `inet`
  - units: `dimensional`
  - easy enough to implement your own; the requirements are very simple
- metadata:
  - `doc`: a simple and short documentation string
  - `default`: the default value, in case the user doesn't supply one
  - `converters`: a chain of preprocessors of the string representation
  - `normalizers`: a chain of post-processors of the converted value
  - `validators`: a tuple of predicates that get called to ensure the property value satisfies the specified constraints
  - you can add your own; the framework passes them through to your component

Units

- `dimensional` properties have units
- the low level support is in `pyre.units`
  - full support for all SI base and derived units
  - all common abbreviations and names from alternative systems of units
  - correct arithmetic; proper handling of functions from `math`

```
1 from math import cos
2 from pyre.units.SI import meter, second, radian
3
4 A = 2.5 * meter
5 t = 1.5 * second
6 ω = 4.2 * radian/second
7
8 x = A * cos(ω * t)
```

if the units in the argument to `cos` do not cancel, leaving a pure `float` behind, an exception is raised; `x` has dimensions of meters

Connecting components

- the real power is in wiring components to other components

```
1  import pyre
2
3  class MonteCarlo(pyre.component, family="gauss.integrators.montecarlo"):
4      """
5      A Monte Carlo integrator
6      """
7
8      # public state
9      samples = pyre.properties.int(default=10**5)
10     region = ????
11     ...
```

- `MonteCarlo` should be able to specify what constitutes an acceptable region
- `Disk` should be able to advertise itself as being an acceptable region
- the user should have natural means for specifying that she wants to wire an instance of `Disk` as the region of integration
- and be able to configure that particular instance of `Disk` in a natural manner
- the framework should check the consistency of this assignment

Protocols

- the component version of an abstract base class is the *protocol*

```
 9 # access to the framework
10 import pyre
11
12 # declaration
13 class Shape(pyre.protocol, family="gauss.shapes"):
14     """
15     Protocol declarator for geometrical regions
16     """
17
18     # my default implementation
19     @classmethod
20     def pyre_default(cls, **kwds):
21         """
22         The default {Shape} implementation
23         """
24         # use {Ball}
25         from .Ball import Ball
26         return Ball
27
28     # interface
29     @pyre.provides
30     def measure(self):
31         """
32         Compute my measure (length, area, volume, etc)
33         """
34
35     @pyre.provides
36     def contains(self, points):
37         """
38         Filter out {points} that are on my exterior
39         """
```

Declaring compatibility with a protocol

- `Disk` can inform the framework that it intends to implement `Shape`

```
1 import pyre
2 from .Shape import Shape
3
4 class Disk(pyre.component, family="gauss.shapes.disk", implements=Shape):
5     """
6     A representation of a circular disk
7     """
8
9     ...
```

- an exception is raised if `Disk` does not conform fully to `Shape`
  - missing methods or missing attributes
- also, proper namespace design simplifies many things for the user
  - `Shape` declared its family as `gauss.shapes`
  - `Disk` declared its family as `gauss.shapes.disk`
- we'll see how later when it's time to put all this together

Specifying assignment requirements

- `MonteCarlo` can now specify it expects a `Shape` compatible object to be assigned as its region of integration

```
1  import pyre
2  from .Shape import Shape
3
4  class MonteCarlo(pyre.component, family="gauss.integrators.montecarlo"):
5      """
6      A Monte Carlo integrator
7      """
8
9      # public state
10     samples = pyre.properties.int(default=10**5)
11     region = Shape()
12     ...
```

- the default value is whatever `Shape` returns from its `pyre_default` class method

Component specification

- and now for the real trick: converting some string provided by the user into a live instance of `Disk`, configuring it, and attaching it to some `MonteCarlo` instance
- the syntax is motivated by `URI`, the universal resource identifiers of the web; the general form is

```
<scheme>://<authority>/<path>#<identifier>
```

  where most of the segments are optional
- if your component is accessible from your python path, you could specify

```
import:gauss.shapes.disk
```

- if your component instance is somewhere on your disk, you would specify

```
file:/tmp/shapes.py/disk
```

- in either case, `disk` is expected to be a name that resolves into a component class, a component instance or be a callable that returns one of these

The user does the wiring

- with our definition of `MonteCarlo`, an appropriately structured package `gauss` on the python path, and the following configuration file

```
[ gauss.shapes.disk ] ; change the default values for all disks
radius = 1
center = (0,0)

[ mc ] ; configure our Monte Carlo integrator instance
region = import:gauss.shapes.disk
...
```

- the following python code in some script `sample.py`

```
  ...
  mc = MonteCarlo(name="mc")
  ...
```

builds a `MonteCarlo` instance and configures it so that its region of integration is a `Disk` instance; similarly, from the command line

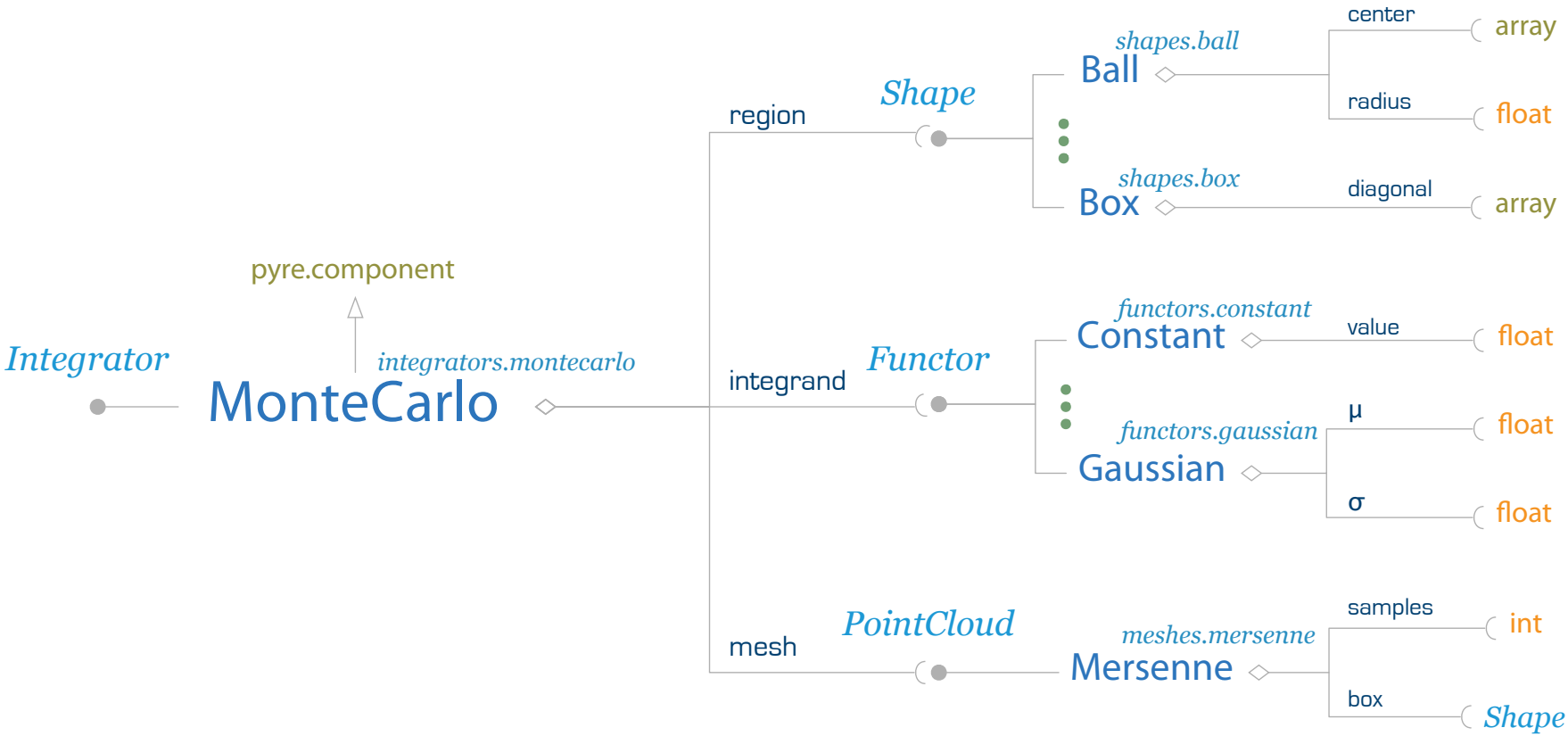
```
  sample.py --mc.region=import:gauss.shapes.disk
```

or, thanks to the consistency in our namespace layout, simply

```
  sample.py --mc.region=disk
```

A non-trivial example

- `gauss`: an extensible numeric integration package
  - specify protocols and components
  - identify the configurable state
  - implement component behavior in terms of the specified protocols

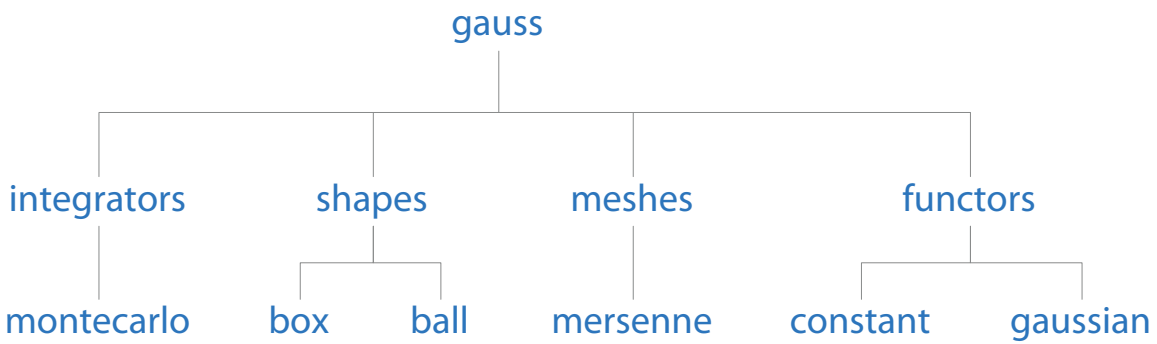


Implementation strategy

- key abstractions:
  - **functor:** encapsulation of a function in a class; our integrands will be functors
  - **shape:** a spatial domain; we will use these to specify the region of integration
  - **mesh:** a discretization of the domain of integration; meshes provide the locations at which we sample integrands
  - **integrator:** the implementation of a particular integration algorithm
- each one of these will turn into a *protocol*
  - that spells out all the obligations imposed on concrete realizations
- the various package *components*
  - will provide concrete implementations of all the obligations
  - and specify their public state: what can be delegated to the end user

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`

```
9  """
10 Package that contains definitions of geometrical shapes
11 """
12
13 # the interfaces
14 from .Shape import Shape as shape
15
16 # the components
17 from .Ball import Ball as ball
18 from .Box import Box as box
```

- 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

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

Shapes

- the `Shape` protocol in `gauss/shapes/Shape.py`

```
 9 # access to the framework
10 import pyre
11
12 # declaration
13 class Shape(pyre.protocol, family="gauss.shapes"):
14     """
15     Protocol declarator for geometrical regions
16     """
17
18     # my default implementation
19     @classmethod
20     def pyre_default(cls, **kwds):
21         """
22         The default {Shape} implementation
23         """
24         # use {Ball}
25         from .Ball import Ball
26         return Ball
27
28     # interface
29     @pyre.provides
30     def measure(self):
31         """
32         Compute my measure (length, area, volume, etc)
33         """
34
35     @pyre.provides
36     def contains(self, points):
37         """
38         Filter out {points} that are on my exterior
39         """
```

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 - traits

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

```
 9 # access the framework
10 import pyre
11 # my protocol
12 from .Shape import Shape
13
14 # declaration
15 class Ball(pyre.component, family="gauss.shapes.ball", implements=Shape):
16     """
17     A representation of the interior of a spere in $d$ dimensions
18     """
19
20     # public state
21     center = pyre.properties.array(default=(0,0))
22     center.doc = "the location of the center of the ball"
23
24     radius = pyre.properties.float(default=1)
25     radius.doc = "the radius of the ball"
```

Ball - interface

- the implementation of `Ball.measure`

```
28      # interface
29      @pyre.export
30      def measure(self):
31          """
32          Compute my volume
33          """
34          # get functools and operator
35          import functools, operator
36          # get π
37          from math import pi as π
38          # compute the dimension of space
39          d = len(self.center)
40          # for even {d}
41          if d % 2 == 0:
42              # for even $d$
43              normalization = functools.reduce(operator.mul, range(1, d//2+1))
44              # compute the volume
45              return π**(d//2) * self.radius**d / normalization
46
47          # for odd {d}
48          normalization = functools.reduce(operator.mul, range(1, d+1, 2))
49          return 2**((d+1)//2) * π**((d-1)//2) / normalization
```

Ball - more interface

- the implementation of `Ball.contains`

```
52      @pyre.export
53      def contains(self, points):
54          """
55          Filter out the members of {points} that are exterior to this ball
56          """
57          # cache the center of the ball
58          center = self.center
59          # compute the radius squared
60          r2 = self.radius**2
61          # for each point
62          for point in points:
63              # compute the distance from the center
64              d2 = sum((p - r)**2 for p,r in zip(point, center))
65              # check whether this point is inside or outside
66              if r2 >= d2:
67                  yield point
68          # all done
69          return
```

Box

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

```
 9 # access the framework
10 import pyre
11 # my protocol
12 from .Shape import Shape
13
14
15 # declaration
16 class Box(pyre.component, family="gauss.shapes.box", implements=Shape):
17     """
18     A representation of the interior of a $d$-dimensional box
19     """
20
21     # public state
22     intervals = pyre.properties.array(default=((0,1),(0,1)))
23     intervals.doc = "the extent of the box along each axis"
```

Box - interface

- the implementation of `Box.measure`

```
26      # interface
27      @pyre.export
28      def measure(self):
29          """
30          Compute my volume
31          """
32          # get functools and operator
33          import functools, operator
34          # compute and return the volume
35          return functools.reduce(
36              operator.mul,
37              ((right-left) for left,right in self.intervals))
```

Box - more interface

- the implementation of `Box.contains`

```
40      @pyre.export
41      def contains(self, points):
42          """
43          Filter out the members of {points} that are exterior to this box
44          """
45          # cache my extent along each coördinate axis
46          intervals = self.intervals
47          # now, for each point
48          for point in points:
49              # for each coördinate
50              for p, (left,right) in zip(point, intervals):
51                  # if this point is outside the box
52                  if p < left or p > right:
53                      # move on to the next point
54                      break
55              # if we got here all tests passed, so
56              else:
57                  # this one is on the interior
58                  yield point
59          # all done
60          return
```

The meshes package

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

```
9  """
10 Package that contains the implemenations of point clouds
11 """
12
13 # the interfaces
14 from .PointCloud import PointCloud as cloud
15
16 # the components
17 from .Mersenne import Mersenne as mersenne
```

Point clouds

- the `PointCloud` protocol in `gauss/meshes/PointCloud.py`

```
 9 # access to the framework
10 import pyre
11
12 # declaration
13 class PointCloud(pyre.protocol, family="gauss.meshes"):
14     """
15     Protocol declarator for an unstructured collection of points
16     """
17
18     # my default implementation
19     @classmethod
20     def pyre_default(cls, **kwds):
21         """
22         The default {PointCloud} implementation
23         """
24         # use the built in random number generator
25         from .Mersenne import Mersenne
26         return Mersenne
27
28     # interface
29     @pyre.provides
30     def points(self, count, box):
31         """
32         Generate {count} random points on the interior of {box}
33         """
```

Generating points with the Mersenne Twister RNG

- in `gauss/meshes/Mersenne.py`

```
 9  # externals
10  import pyre
11  import random
12  import itertools
13  # my protocol
14  from .PointCloud import PointCloud
15
16  class Mersenne(pyre.component, family="gauss.meshes.mersenne", implements=PointCloud):
17      """
18      A point generator that uses the python builtin random number generator
19      """
20
21
22      # public state
23      seed = pyre.properties.int(default=None)
24      seed.doc = "initialization for the random number generator"
25
26
27      # interface
28      @pyre.export
29      def points(self, count, box):
30          """
31          Generate {count} random points chosen from the interior of {box}
32          """
33          # our random number generator
34          rng = self.rng.uniform
35          # get starmap from itertools
36          starmap = itertools.starmap
37          # get the extent of the box
38          intervals = box.intervals
39          # loop {count} times
40          while count > 0:
41              # decrement the counter
```

The `functors` package

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

```
9  """
10 Package with functor definitions
11 """
12
13 # the interface
14 from .Functor import Functor as functor
15
16 # the components
17 from .Constant import Constant as constant
18 from .Gaussian import Gaussian as gaussian
19 from .One import One as one
```

Functors

- the `Functor` protocol in `gauss/functors/Functor.py`

```
 9 # access to the framework
10 import pyre
11
12 # declaration
13 class Functor(pyre.protocol, family="gauss.functors"):
14     """
15     Protocol declarator for function objects
16     """
17
18     # the suggested default implementation
19     @classmethod
20     def pyre_default(cls, **kwds):
21         """
22         The default implementation of the {Functor} protocol
23         """
24         from .Constant import Constant
25         return Constant
26
27     # interface
28     @pyre.provides
29     def eval(self, points):
30         """
31         Evaluate the function at the supplied points
32         """
```

The `Constant` functor

- in `gauss/functors/Constant.py`

```
 9 # access the framework
10 import pyre
11 # my protocol
12 from .Functor import Functor
13
14
15 class Constant(pyre.component, family="gauss.functors.constant", implements=Functor):
16     """
17     A constant function
18     """
19
20     # public state
21     value = pyre.properties.float(default=1)
22     value.doc = "the value of the constant functor"""
23
24     # interface
25     @pyre.export
26     def eval(self, points):
27         """
28         Compute the value of the function on the supplied points
29         """
30         # local cache of the constant
31         value = self.value
32         # loop over the points and return my value regardless
33         for point in points: yield value
34         # all done
35         return
```

A non-trivial functor

- the declaration of the traits

```
 9 # access the framework
10 import pyre
11 # my protocol
12 from .Functor import Functor
13
14
15 class Gaussian(pyre.component, family="gauss.functors.gaussian", implements=Functor):
16     r"""
17     Component that implements the normal distribution with mean μand variance σ^2
18
19         g(x; μ,σ) = \frac{1}{\sqrt{2π} σ} e^{-\frac{|x-μ|^2}{2σ^2}}
20
21     μand σare implemented as component properties so that Gaussian can conform to the
22     functor interface. See gauss.interfaces.functor for more details.
23     """
24
25     # public state
26     mean = pyre.properties.array(default=[0])
27     mean.doc = "the mean of the gaussian distribution"
28     mean.aliases.add("μ")
29
30     spread = pyre.properties.float(default=1)
31     spread.doc = "the variance of the gaussian distribution"
32     spread.aliases.add("σ")
```

A non-trivial functor – continued

- the implementation of `eval`

```
35      # interface
36      @pyre.export
37      def eval(self, points):
38          """
39          Compute the value of the gaussian
40          """
41          # access the math symbols
42          from math import exp, sqrt, pi as π
43          # cache the inventory items
44          μ= self.μ
45          σ= self.σ
46          # precompute the normalization factor
47          normalization = 1 / sqrt(2*π) / σ
48          # and the scaling of the exposnential
49          scaling = 2 * σ**2
50          # loop over points and yield the computed value
51          for x in points:
52              # compute |x - μ|^2
53              # this works as long as x and μhave the same length
54              r2 = sum((x_i - μ_i)**2 for x_i, μ_i in zip(x, μ))
55              # yield the value at the current x
56              yield normalization * exp(- r2 / scaling)
57          # all done
58          return
```

The `integrators` package

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

```
9  """
10 Package with integrator implementations
11 """
12
13 # the interface
14 from .Integrator import Integrator as integrator
15
16 # the component
17 from .MonteCarlo import MonteCarlo as montecarlo
```

Integrators

- in `gauss/integrators/Integrator.py`

```
9  # access to the framework
10 import pyre
11
12 # declaration
13 class Integrator(pyre.protocol, family="gauss.integrators"):
14     """
15     Protocol declarator for integrators
16     """
17
18     # access to the required protocols
19     from ..shapes.Shape import Shape as shape
20     from ..functors.Functor import Functor as functor
21
22     # public state
23     region = shape()
24     region.doc = "the region of integration"
25
26     integrand = functor()
27     integrand.doc = "the functor to integrate"
28
29     # my preferred implementation
30     @classmethod
31     def pyre_default(cls, **kwds):
32         # use {MonteCarlo} by default
33         from .MonteCarlo import MonteCarlo
34         return MonteCarlo
35
36     # interface
37     @pyre.provides
38     def integrate(self):
39         """
40         Compute the integral of {integrand} over {region}
41         """
```

The Monte Carlo integrator

- in `gauss/integrators/MonteCarlo.py`

```
 9  # access to the framework
10  import pyre
11
12  # protocols
13  from .Integrator import Integrator
14  from ..functors import functor
15  from ..meshes import cloud
16  from ..shapes import shape, box, ball
17
18
19  # declaration
20  class MonteCarlo(pyre.component, family="gauss.integrators.montecarlo", implements=Integrator):
21      """
22      A Monte Carlo integrator
23      """
24
25      # public state
26      samples = pyre.properties.int(default=10**5)
27      samples.doc = "the number of integrand evaluations"
28
29      box = shape(default=box)
30      box.doc = "the bounding box for my mesh"
31
32      mesh = cloud()
33      mesh.doc = "the generator of points at which to evaluate the integrand"
34
35      region = shape(default=ball)
36      region.doc = "the shape that defines the region of integration"
37
38      integrand = functor()
39      integrand.doc = "the functor to integrate"
```

The Monte Carlo integrator – continued

- the implementation of `integrate`

```
42      # interface
43      @pyre.export
44      def integrate(self):
45          """
46          Compute the integral as specified by my public state
47          """
48          # compute the normalization
49          normalization = self.box.measure()/self.samples
50          # get the set of points
51          points = self.mesh.points(count=self.samples, box=self.box)
52          # narrow the set down to the ones interior to the region of integration
53          interior = self.region.contains(points)
54          # sum up and scale the integrand contributions
55          integral = normalization * sum(self.integrand.eval(interior))
56          # and return the value
57          return integral
```

Top level – the `gauss` package

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

```
9  """
10 This is the implementation for the interfaces and components of gauss, a sample pyre
11 application.
12
13 See gauss.license() for terms of use
14 """
15
16 # access to the package contents
17 from . import functors, integrators, meshes, shapes, util
18
19 # misc
20 def copyright():
21     """
22     Return the package copyright note
23     """
24     return _gauss_copyright
25
26
27 def license():
28     """
29     Return the package license
30     """
31     return _gauss_license
32
33
34 def version():
35     """
36     Return the package version
37     """
38     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:~/tmp>python3.3
Python 3.3.0 (default, Sep 29 2012, 08:16:08)
[GCC 4.2.1 Compatible Apple Clang 3.1 (tags/Apple/clang-318.0.58)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
enabling readline
>>> import gauss
>>> mc = gauss.integrators.montecarlo()
>>> mc.samples
100000
>>> mc.box.intervals
((0, 1), (0, 1))
>>> mc.region
<gauss.shapes.Ball.Ball object at 0x1068d0610>
>>> mc.region.radius
1.0
>>> mc.region.center
(0, 0)
>>> mc.integrand
<gauss.functors.Constant.Constant object at 0x10634b5d0>
>>> mc.integrand.value
1.0
>>> 4 * mc.integrate()
3.13464
```

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

```
1  one = 1
2
3  [ mc ] ; configure our Monte Carlo integrator instance
4  samples = 10**6
5  region = ball#frisbee ; equivalent to import:gauss.shapes.ball#frisbee
6  integrand = constant ; equivalent to import:gauss.functors.constant
7
8  [ gauss.functors.constant # mc.integrand ] ; if mc.integrand is a constant
9  value = {one}
10
11 [ gauss.functors.gaussian # mc.integrand ] ; if mc.integrand is a gaussian
12 mean = (0, 0)
13 spread = {one}/3
```

Creating an application

- applications are the top level component managers

```
10 # externals
11 import pyre
12 import gauss
13
14 # the application
15 class Quad(pyre.application):
16     """
17     A harness for {gauss} integrators
18     """
19
20     # public state
21     samples = pyre.properties.int(default=10**5)
22     integrator = pyre.facility(interface=gauss.integrator)
23
24     # required interface
25     @pyre.export
26     def main(self, *args, **kwds):
27         # pass the requested number of samples to the integrator
28         self.integrator.samples = self.samples
29         # get it to integrate
30         integral = self.integrator.integrate()
31         # print the answer
32         print("integral = {}".format(integral))
33         # return success
34         return 0
```

Auto-launching

- instantiating and launching the application

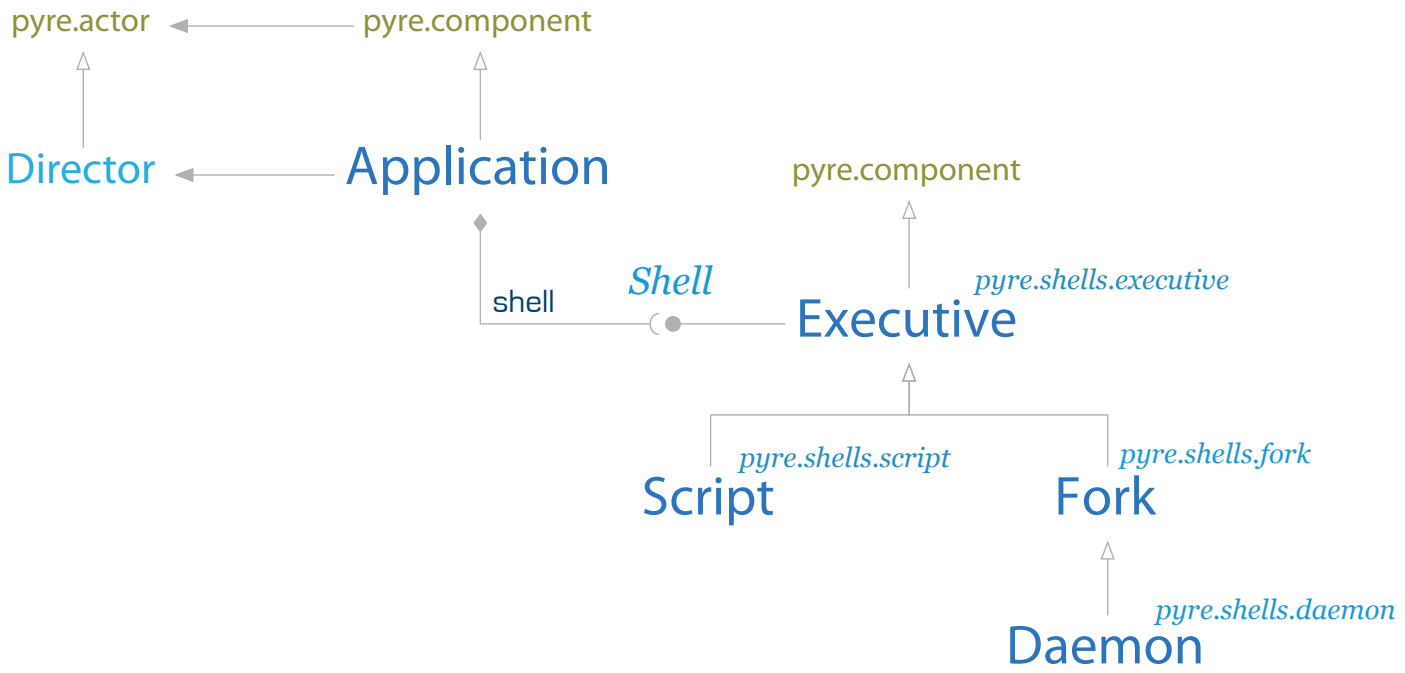
```
54 # main
55 if __name__ == "__main__":
56     # externals
57     import sys
58     # instantiate the application
59     q = Quad(name='quad')
60     # run it and return its exit code to the os
```

- a sample configuration file

```
8  ; application settings
9  [ quad ]
10 samples = 10**6
11
12 ; cconfiguration for the integrator
13 [ gauss.integrators.montecarlo # quad.integrator ]
14 box.intervals = ((-1,1), (-1,1))
15 region = ball
16 integrand = constant
17
18 ; when the integrand is the constant functor
19 [ gauss.functors.constant # quad.integrator.integrand ]
20 value = 1
```

The application component

- the shell hierarchy in pyre



- our `Quad` derives from `Application`, so it has a `shell`

Parallel integration

- the `mpi` entry point

```
36      @pyre.export
37      def main_mpi(self, *args, **kwds):
38          # access the mpi package
39          import mpi
40          # find out how many tasks were launched
41          size = mpi.world.size
42          # find out my rank
43          rank = mpi.world.rank
44          # figure out how many samples to do and pass that on to my integrator
45          self.integrator.samples = self.samples / size
46          # integrate: average the estimates produced by each task
47          integral = mpi.sum(self.integrator.integrate())/size
48          # node 0: print the answer
49          if rank == 0: print("integral = {}".format(integral))
50          # all done
51          return 0
```

- the `mpi` package is part of the pyre distribution
  - handles initialization and finalization of `MPI`
  - simplifies most of the “overhead” activities
  - provides an OO veneer

Running in parallel

- minor modifications to the configuration file...

```
8  ; application settings
9  [ quad ]
10 samples = 10**6
11
12 ; cconfiguration for the integrator
13 [ gauss.integrators.montecarlo # quad.integrator ]
14 box.intervals = ((-1,1), (-1,1))
15 region = ball
16 integrand = constant
17
18 ; when the integrand is the constant functor
19 [ gauss.functors.constant # quad.integrator.integrand ]
20 value = 1
21
22 ; for MPI
23 [ quad ]
24 shell = mpi
25
26 [ mpi.shells.mpirun # quad.shell ]
27 tasks = 8
28 launcher = openmpirun
```

References I

[1] J. M. Hammersley. Monte Carlo methods for solving multivariable problems. *Ann. New York Acad. Sci.*, 86:844–874, 1960.

[2] C. W. Ueberhuber. *Numerical Computation 2: Methods, Software, and Analysis*, chapter on Monte Carlo Techniques, pages 124–125 and 132–138. Springer-Verlag, 1997.

[3] S. Weinzierl. Introduction to Monte Carlo methods. 2000. URL http://arxiv.org/abs/hep-ph/0006269.