# ME 701 – Development of Computer Applications in ME
# Homework 5 – Object-Oriented Python – Due 10/18/2017
**(Revised: October 15, 2017)**

---

**Instructions**. This homework focuses on the development of a simple 2-D constructive solid geometry module. I have provided you with templates for many of classes you need to define. The following problems provide software specifications for pieces of your module, starting from points, and building on up through a whole geometry made of regions.

To help you in your development, I've also provide a small unit test framework. The various **unit tests demonstrate how your classes and methods should work** for a few cases. The tests I provide may not provide complete *coverage*, i.e., they might not test every method for every possible use consistent with the specifications below. You should feel free to modify these tests and add your own. Such testing is extremely helpful for all but the smallest programming tasks, as it (1) formalizes the normal testing you would do in a (probably) much sloppier way and (2) lives on as a set of (hopefully) easily understood examples for how your code works. Invaluable for the coder with the memory of a (dying) goldfish.

### Problem 1 – On Point

We started off our discussion of object-oriented programming using the `Point` class as an example. In essence, a `Point` represents a point in 2-D space (or, equivalently, a 2-D vector starting from the origin). In some applications, it would be helpful to add two points (a translation), or to multiply a point by some constant (a scaling), e.g.,

```
>>> p1 = Point(1, 2)
>>> p2 = Point(2, 3)
>>> p3 = p1 + p3
>>> print p3
Point(x=3.000000, y=5.000000)
>>> p4 = p1 * 2
>>> print p4
Point(x=2.000000, y=4.000000)
```

**Deliverables**: modify the basic `Point` class to handle (1) addition of two points and (2) scaling of a point by a scalar value.

### Problem 2 – Nodes and Surfaces

The template code includes definitions for an abstract `Node` class, from which the `Primitive` and `Operator` classes are derived. Recall that an abstract class is one that defines a method signature but does not provide an implementation. In C-speak, it's like having a function declaration in a header without its definition in the source file. A *concrete* class inherits from

a *base* class and provides an implementation of the methods.

The `Primitive` class is a concrete class that represents terminal node, i.e., a surface. On the other hand, the `Operator` class represents a generic combination of two nodes. Like `Node`, `Operator` is an abstract class because its `contains` method is not (and should not) be implemented. Its other method `intersections` should be defined, however, and that's for you to do.

In addition, you should create the following two specializations of the `Operator` class:

- `class Union`, whose method `contains(p)` should return true if the point is in either its left or right nodes.
- `class Intersection`, whose `contains(p)` should return true if the point is in both its left and right nodes.

**Deliverables**:

1. Implement `Operator.intersections(r)`
2. Define `class Union` and implement `Union.contains(p)`
3. Define `class Intersection` and implement `Intersection.contains(p)`

### Problem 3 – Kids Love Shapes

The template code provides partial definitions for the abstract `Surface` and concrete `QuadraticSurface` classes. Your first task is to implement the `f` and `intersections` methods of `QuadraticSurface`.

Although you can define any 2-D quadratic surface with the approach outlined in the notes, it is easier to specialize that class further for common cases. Your job is to create the following classes that inherit from `QuadraticSurface`:

- `class PlaneV`; defines a vertical plane at $x = a$ via `v = PlaneV(a)`,
- `class PlaneH`; defines a horizontal plane at $y = b$ via `h = PlaneH(b)`
- `class Plane`; defines a plane of the form $y = mx + b$ via `p = Plane(m, b)`.
- `class Circle`; defines a circle of radius $r$ and center $(a, b)$ via `c = Circle(r, a, b)`.

You could also specialize the methods `intersections` and `f`, but your implementation of these methods in `QuadraticSurface` should work for *any* surface with the generic form given in the notes, i.e., $f(x, y) = Ax^2 + By^2 + Cxy + Dx + Ey + F$.

**Deliverables**:

- Implement `QuadraticSurface.f(p)`
- Implement `QuadraticSurface.intersections(r)`
- Implement `PlaneV`
- Implement `PlaneH`
- Implement `Plane`
- Implement `Circle`

### Problem 4 – Regions and Geometry

Now you have surfaces and nodes with which to implement a `Region` class. I've proposed the following definition:

```
class Region:

    def __init__(self) :
        self.node = None

    def append(self, node=None, surface=None, operation="U", sense=False) :
        assert((node and not surface) or (surface and not node))
        assert(operation in ["U", "I"])
        if isinstance(surface, Surface) :
            node = Primitive(surface, sense)
        if self.node is None :
            self.node = node
        else :
            O = Union if operation == "U" else Intersection
            self.node = O(self.node, node)

    def contains(self, p) :
        pass

    def intersections(self, r) :
        pass
```

Study the `append` method because it represents how we put all the surfaces and nodes together. The key point is that `Region` has an attribute `node` that represents at all times the top of the tree of nodes and surfaces. You should add an appropriate docstring to explain what the method is doing, what the assertions are doing, etc. In other words, make it obvious that you know what I'm doing. (This might seem like a silly exercise, but the ability to digest and then add to code that someone else wrote is a good way to improve your own programming.)

Once you've done that, you need to implement the `contains` and `intersections` methods. In particular, the points returned by the latter should satisfy the following:

- They are unique (a ray might intersect more than one surface that defines a region, e.g, if the ray is along the $x$-axis and the region is bounded by the planes $y = \pm x$, then the point $(0, 0)$ could be included twice.)
- They include points only along forward of the ray origin. For example, if a region is bounded by the unit circle centered at the origin, then a ray centered at the origin in the positive $x$ direction would intersect the region at $(1, 0)$ but not $(-1, 0)$.
- They are ordered by increasing distance from the ray's origin, i.e., the first point is the first one encountered by the ray.

Finally, consider the following `Geometry` class:

```
class Geometry:
    noregion = -1

    def __init__(self,  xmin, xmax, ymin, ymax) :
        self.xmin, self.xmax = xmin, xmax
        self.ymin, self.ymax = ymin, ymax
        self.regions = []

    def add_region(self, r) :
        self.regions.append(r)

    def find_region(self, p) :
        """ Find the region that contains the point.  If none
        is found, return Geometry.noregion.

        Arguments:
            p : Point
        Returns:
            i : int
        """
        pass

    def plot(self, xmin, xmax, nx, ymin, ymax, ny) :
        # see the template file; discussion to follow
        pass
```

Its construction requires a bounding box defined by minimum and maximum values for $x$ and $y$, i.e., the square domain in which the geometry is confined. Regions are stored in an initially empty list. Regions are added by the `add_region` method. Your job is to implement the `find_region` method, which returns the index (in the list) of the region that contains the point. If no region contains the point, or if the point lives outside the bounding box, return `Geometry.noregion`.

**Deliverables**:

1. Document `Region.append`
2. Implement `Region.contains`
3. Implement `Region.intersections`
4. Implement `Geometry.find_region`

**Bonus Problem – Challenge!**

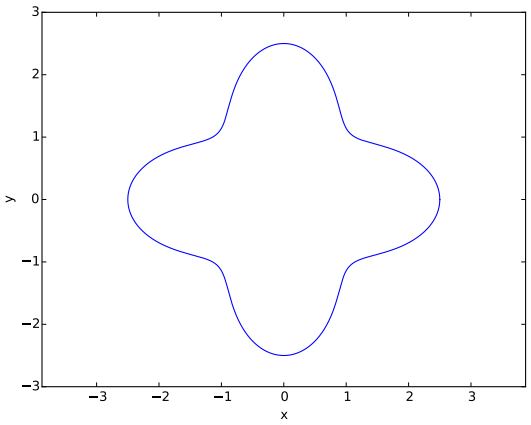
I'll give you 2 (actual) bonus points if you solve it (and 1 for a very valiant effort if not successful).

Consider a surface formed by wrapping a sine wave around a circle back onto itself. We can

do so using a parametric curve of the form

$$\begin{aligned} x &= [r + a\sin(n\phi + \phi_0)]\cos(\phi) + x_0 \\ y &= [r + a\sin(n\phi + \phi_0)]\sin(\phi) + y_0 \end{aligned} \tag{1}$$

where $\phi$ is the parameter, $r$ is the radius of the circle about which the wave is wrapped, $a$ is the magnitude of that wave, and $(x_0, y_0)$ is the center of the curve. The figure shows the resulting curve for $r = 2$, $n = 4$, $a = 0.5$, and $\phi_0 = \pi/2$.




You can convert this into implicit form by moving the $x_0$ and $y_0$ terms to the left side, squaring both equations, and adding the results, which all leads to

$$(x - x_0)^2 + (y - y_0)^2 = [r + a\sin(n\phi + \phi_0)]^2 \tag{2}$$

or

$$f(x, y) = (x - x_0)^2 + (y - y_0)^2 - [r + a\sin(n\phi + \phi_0)]^2 \,. \tag{3}$$

That pesky $\phi$ can be found from the original equations by noting

$$\phi = \arctan[(y - y_0)/(x - x_0)] \,. \tag{4}$$

**Deliverables**:

1. Explain how you would tackle the intersections problem for this surface.
2. Do it!