Observation and Characterization of Convection Under Static & Rotating Conditions and the Possible Implications to the Jovian Atmosphere

Luke Siemens & Stephanie Monty

April 20, 2016

1 Introduction

Current models of Jupiter's atmosphere suggest the possibility of convection occurring deep within the planet's atmosphere. As early as 1977 Prinn & Barshay suggest that the presence of convection within the atmosphere at 1100K would explain the appearance of carbon monoxide within the upper atmosphere, as observed in 1977 (Beer & Taylor, 1978). More recently with observations from the Galileo mission to Jupiter, convection was inferred from the direct observation of long lived storms on the planet, appearing as "zonal jets" or "long-lived ovals" (Ingersoll et al., 2000). The appearance and longevity of the storms suggests an energy source within the planet and the transfer of energy from said source to the atmosphere via convection (Ingersoll et al., 2000).

Further support for the presence of convection within the planet can be found in measuring the effective and equilibrium temperature of Jupiter. The equilibrium temperature may be found from equating the incoming solar radiation to the outgoing reradiated solar energy, while the effective temperature may be found through fitting a black body spectrum to the integrated flux over all frequencies coming from the body (Lissauer & de Pater, 2013). The effective temperature for Jupiter was measured by the Voyager spacecraft to be $T_{eff} = 124.4$K (Hamel et al., 1981), while the equilibrium temperature can be calculated to be $T_{eq} = 110$K (Mihos, 2005). This discrepency in predicted and measured temperature alludes to the presence of an internal heat source within the Jupiter. The emission of this additional energy from the planet again supports the need for convection in order to transport the heat from within the planet to it's surface.

Of interest as well in the Jovian atmosphere is the effect of rotation on the theorized convection. Jupiter is a rapidly rotating planet with a rotation rate of approximately 9h 55m (Helled et al., 2009). This is inferred from the zonal winds in the upper atmosphere, which can reach ~100m/s (Helled et al., 2009). Rotation clearly then has an effect on the motion of Jupiter's atmosphere. The effect of the rotation could be the introduction of turbulence which may impact the formation of convection cells and efficiency of heat transfer via convection.

With convection being a probable process occurring within the planet, an attempt to study and become more familiar with the mechanisms associated with convection in an Earth-based laboratory could yield important results. An experiment was performed in order to observe and characterize convection cells on laboratory scales. This included a study into the generation and characterization of convection under both static and rotational conditions. The results of this "table top" convection experiment will be discussed in an attempt to relate any findings to the larger scale of the Jovian interior and atmosphere.

2 Theory

The most accessible form of convection to study, characterize and observe in a laboratory setting, is Rayleigh-Benard (RB) convection. RB convection is a buoyancy-driven convection which occurs when a fluid is heated from below and cooled from above. Henri Benard first observed the creation of hexagonal squares on the surface of a fluid heated from below in 1900, while Lord Rayleigh derived the theoretical requirement for the creation of convection within a layer of fluid "bounded" by two free surfaces later, in 1916 (Kundu et al.,2012). Rayleigh showed that a relationship existed between the*

As mentioned in the above section, a convective cell will be generated within a cylindrical vessel and observed through the aid of high speed cameras, rheoscopic fliuds various dyes. In order to determine the required temperature difference between the bottom of the vessel and the top to induce convection, a mathematical study will be made investigating the Rayleigh-Benard Instability. This will be done using the definition of the dimensionless Rayleigh Number, noting that the density profile of the vessel will be treated as a free variable. Working from the definition of the Rayleigh number.

$$R_a = \frac{g\beta}{\nu\alpha}(T_b - T_u)L^3$$

$$R_a = \frac{g\beta}{\mu k}c_p\rho^2(T_b - T_u)L^3$$

Noting above that all the variables excluding $T_b - T_u$ and ρ are fixed constants related to the thermal and viscous characteristics of the fluid to be used, while L is the height of the vessel and g is the acceleration due to gravity. The critical Rayleigh number, R_c, represents the value of the Rayleigh number, representing the ratio of gravitational to viscous forces, at which convection occurs in the system. For the solved case, in which the bottom boundary is rigid, while the top is free, the case of this experiment, the critical Rayleigh number is known. Thus, the temperature difference required to induce convection may then become only dependent on the density profile of the vessel. If the density if discrete, the temperate difference may be deduced for each layer of different density fluids, if the density profile is continuous the temperature difference required may be expressed in equation (2) where density becomes a function of height in the vessel. $\rho = \rho(L)$ (Bahrami, 2016).

$$(T_b - T_u) = \left(\frac{\mu k}{g\beta}\right)\frac{R_c}{\rho^2 L^3} \tag{1}$$

$$(T_b - T_u) = \left(\frac{\mu k}{g\beta}\right)\frac{R_c}{\rho^2(L) L^3} \tag{2}$$

The density profile of the vessel will be approximated as being both linear and discrete as both continuous and discrete density profiles will be investigated. In reality the more easy of the two to generate and model will be used in the final experiment to determine the required temperature difference as given in equations (1) or (2).

A computational simulation will be created in order to investigate the feasibility and compatibility of both modeling convection in three dimensions and modeling convection in three dimensions via investigating convection in two dimensions. This will be done using COMSOL, Python and possible Fortran for any numerical analysis that may be necessary. The goal associated with the computational aspect of the project will be to successfully model convection in two dimensions with the possibility of extending to three.

Experimentally, the goal of the project will be to generate and observe convection cells. Once convection cells have been observed in a stationary reference frame, rotation will be introduced in an attempt to observe and record any perturbations to convection that might occur due to rotation. Any resultant turbulence that could could occur upon the introduction of rotation into a convective system will also be investigated. The effects of rotation will only be investigated upon the successful creation of convection within a stationary reference frame.

Utilizing the results from the mathematical, computational and experimental components of the project an overall result as to the possibility, or inference, of convection within Jupiter's atmosphere will be made. This will be done in part theoretically, through noting the temperature difference required in order to generate convection for fluids of a specific density, or density profile, and comparing this against the known density profile of the planets atmosphere. Experimentally, observing convection in the system could lead to inferring the existence of convection within Jupiter's atmosphere after considering comparable scale lengths and temperature differences between the two systems. If the appearance of convection is further supported through the appearance of convection computationally, via modeling, this will lead to further confidence in the final result.

3 Analysis

4 Conclusions

5 References

1. Bahrami, M. "Natural Convection." *Simon Fraser University*. Simon Fraser University, n.d. Web. 21 February 2016.

2. Beer, R. & Taylor, F. "The Abundance of Carbon Monoxide in Jupiter." *ApJ* 221 (1978): 1100-1109. Print.

3. Ingersoll, A.P. et al. "Moist convection as an energy source for the large-scale motions in Jupiter's atmosphere." *Nature* 403 (2000): 630-633. Print.

4. Prinn, R.G., Barshay, S.S. "Carbon Monoxide on Jupiter and Implications for Atmospheric Convection." *AAAS* 198.4321 (1977): 1031-1034. Print

Appendices

Appendix A Code and Data Processing Scripts

Script: processdata.py

```
from matplotlib import pyplot
from scipy.ndimage import imread
from scipy import fftpack
from radialfft import radial_fft
import numpy

r_clip = 0.85

image_flats = [("./data/CIMG2817.JPG", 1841, 1349, 1317),
("./data/exp2_p2/CIMG2927.JPG",1635, 1505, 1029),
("./data/exp2_p2/CIMG2928.JPG", 1677, 1393, 997)
#"./data/exp2_p2/CIMG2929.JPG",
#"./data/exp2_p2/CIMG2930.JPG",
#"./data/exp2_p2/CIMG2931.JPG",
#"./data/exp2_p2/CIMG2932.JPG",
#"./data/exp2_p2/CIMG2933.JPG",
#"./data/exp2_p2/CIMG2934.JPG"
]

image_data = [#("./data/CIMG2782.JPG", 1851, 1436, 1138,
    "r-", "heated water"),
                #("./data/CIMG2813.JPG", 1865, 1458, 881, "
                    r-", None),
                #("./data/CIMG2815.JPG", 1936, 1662, 1054,
                    "r-", None),
                ("./data/CIMG2816.JPG", 1793, 1444, 1271, "
                    b-", "Non-rotating system"),
                ("./data/CIMG2817.JPG", 1841, 1349, 1317, "
                    k-", "static fluid"),
                #("./data/CIMG2822.JPG", 1892, 1374, 1342,
                    "g-", "fluid with sugar layer"),
                #("./data/CIMG2826.JPG", 1783, 1586, 1050,
                    "g-", None),
                #("./data/CIMG2829.JPG", 1902, 1381, 1344,
                    "g-", None),
                #("./data/CIMG2830.JPG", 1912, 1319, 1131,
                    "g-", None),
                #("./data/CIMG2837.JPG", 1764, 1354, 1282,
                    "b-", "boiling sugar layer"),
```

```
          #("./data/CIMG2838.JPG", 1824, 1373, 1308,
               "b-", None),
          ("./data/exp2_p1/CIMG2919.JPG", 1961, 1326,
               785, "r-", "cooling rotating"),
          ("./data/exp2_p1/CIMG2920.JPG", 1889, 1391,
               937, "r-", None),
          ("./data/exp2_p1/CIMG2921.JPG", 1945, 1426,
               775, "r-", None),
          ("./data/exp2_p1/CIMG2922.JPG", 1847, 1528,
               849, "r-", None)]

#image_data = image_data[:3]
fig1, ax1 = pyplot.subplots()
fig2, ax2 = pyplot.subplots()
data_flats = [radial_fft(fname, x, y, r, r_clip=r_clip)
    for fname, x, y, r in image_flats]

data = [radial_fft(fname, x, y, r, r_clip=r_clip) for
    fname, x, y, r, style, label in image_data]

for i, d in enumerate(data[:2]):
    print("loaded " + str(i + 1) + " of " + str(len(
        image_data)) + ".")
    style, label = image_data[i][4], image_data[i][5]
#    d = radial_fft(fname, x, y, r, r_clip=r_clip)
    base = numpy.mean(numpy.array([data_flat.interpolate(
        d.x) for data_flat in data_flats]), axis=0)
    ax1.plot(d.x, d.data_powerspectrum, style, label=
        label)
    if label != "static fluid":
        ax2.plot(1.0/d.x, d.data_powerspectrum/base,
            style, label=label)
    else:
        ax2.plot(1.0/d.x, [1.0]*len(d.x), style, label=
            label)

for i, d in enumerate([data[3]]):
    print("loaded " + str(i + 1) + " of " + str(len(
        image_data)) + ".")
    style, label = image_data[i][4], image_data[i][5]
#    d = radial_fft(fname, x, y, r, r_clip=r_clip)
    base = numpy.mean(numpy.array([data_flat.interpolate(
        d.x) for data_flat in data_flats]), axis=0)
    val = numpy.mean(numpy.array([f.interpolate(d.x) for
        f in data[2:]]), axis=0)
    ax1.plot(d.x, d.data_powerspectrum, style, label=
```

```
        label)
    if label != "static fluid":
        ax2.plot(1.0/d.x, val/base, "r-", label="Rotating
             system")
    else:
        ax2.plot(1.0/d.x, [1.0]*len(d.x), style, label=
            label)
ax1.legend()
ax1.set_xlabel("$\\nu$")
ax2.legend()
ax2.set_xlabel("$\\frac{\\lambda}{r}$")
ax2.set_ylabel("Normalized FFT radial profile")
pyplot.show(fig1)
pyplot.show(fig2)
```

Module: radialfft.py

```
from matplotlib import pyplot
from scipy.ndimage import imread
from scipy import interpolate
from scipy import fftpack
import numpy

def radial_profile(data, center):
    y, x = numpy.indices((data.shape))
    r = numpy.sqrt((x - center[0])**2 + (y - center[1])
        **2)
    r = r.astype(numpy.int)
    value_bin = numpy.bincount(r.ravel(), data.ravel())
    num_bin = numpy.bincount(r.ravel())
    return value_bin/num_bin

class radial_fft:
    def __init__(self, fname, x, y, r, r_clip=1.0, r_ramp
        =5, r_size=1.0, max_input=255.0):
         self.x = x
         self.y = y
         self.r = r
         self.r_clip = r_clip #cliping radii
         self.r_ramp = r_ramp #ramp from background to
             mask
         self.r_size = r_size #radii size
         self.max_input = max_input
         if isinstance(fname, str):
```

```
            self.fname = fname
            self.raw_data = imread(fname)
        else:
            self.fname = None
            self.raw_data = fname
        self.data = None
        self._interpolate_fft = None

        self.f_sampling = None

        self._mask_data()
        self._fft()

    def _mask_data(self):
        #scale r (remove error on side)
        r = int(self.r_clip*self.r)
        X, Y = numpy.meshgrid(numpy.linspace(0, self.
            raw_data.shape[1], self.raw_data.shape[1]),
            numpy.linspace(0, self.raw_data.shape[0], self
            .raw_data.shape[0]))
        #define mask
        mask = 1.0 - 0.5*(numpy.tanh((numpy.sqrt((X -
            self.x)**2 + (Y - self.y)**2) - r)/self.r_ramp
            ) + 1.0)
        #find luminance
        if self.fname != None:
            luminance = numpy.sum((self.raw_data[:, :]/
                self.max_input)**2, axis=2)/3.0
        else:
            if len(self.raw_data.shape) > 2:
                luminance = numpy.sum((self.raw_data[:,
                    :]/self.max_input)**2, axis=2)/3.0
            else:
                luminance = numpy.abs(self.raw_data[:,
                    :]/self.max_input)
        #mask data
        luminance = luminance*mask
        #normalze
        mean = numpy.average(luminance, weights=mask)
        luminance = luminance/mean - mask
        luminance = luminance/numpy.max(numpy.abs(
            luminance))
        luminance = luminance[self.y-r-int(self.r_ramp):
            self.y+r+int(self.r_ramp), self.x-r-int(self.
            r_ramp):self.x+r+int(self.r_ramp)]
        self.data = luminance
```

```
    def _fft(self):
        r = int(self.r_clip*self.r)
        fft = fftpack.fft2(self.data)
        fft = fftpack.fftshift(fft)

        self.f_max = 2.0*self.r/(4.0*self.r_size)
        self.f_min = 1.0/(2.0*self.r_size)

        powerspec = numpy.abs(fft)**2
        self.data_powerspectrum = radial_profile(numpy.
            log10(powerspec), (powerspec.shape[0]/2,
            powerspec.shape[0]/2))

        self.x = numpy.linspace(self.f_min, len(self.
            data_powerspectrum)*self.f_min, len(self.
            data_powerspectrum))

        self._interpolate_fft = interpolate.
            UnivariateSpline(self.x, self.
            data_powerspectrum, s=0, k=2)

    def interpolate(self, x):
        return self._interpolate_fft(x)

"""
d = radial_fft("./data/CIMG2817.JPG", 1841, 1349, 1317)
import voronoi
import random
random.seed()
bw = voronoi.bowyer_watson(2.0)
for _ in range(10):
    bw.add_point(voronoi.vec(random.uniform(-1, 1),
        random.uniform(-1, 1)))
v = bw.get_voronoi(2000)
v.gaussian(0.04, 0.1)
data = 1.0 - 0.1*v.raster
d = radial_fft(data, 1000, 1000, 990, max_input=1.0)
"""
```

Module: voronoi.py

```
import numpy
import scipy.ndimage
```

```
from matplotlib import pyplot

class vec:
    def __init__(self, x, y=None):
        if y == None:
            if len(x) == 2:
                self.coord = numpy.array(x)
        else:
            self.coord = numpy.array([x, y])

    def perp(self):
        return vec(-self.coord[1], self.coord[0])

    def dot(self, other):
        return numpy.sum(self.coord*other.coord)

    def project_on(self, other):
        return (self.dot(other)/other.dot(other))*other

    def length(self):
        return numpy.sqrt(self._length2())

    def _length2(self):
        return self.dot(self)

    def __eq__(self, other):
        return all(self.coord == other.coord)

    def __ne__(self, other):
        return any(self.coord != other.coord)

    def __lt__(self, other):
        return self._length2() < other._length2()

    def __gt__(self, other):
        return self._length2() > other._length2()

    def __le__(self, other):
        return self._length2() <= other._length2()

    def __ge__(self, other):
        return self._length2() >= other._length2()

    # math operations
    def __add__(self, other):
        return vec(self.coord + other.coord)
```

```
    def __radd__(self, other):
        return vec(self.coord + other.coord)

    def __sub__(self, other):
        return vec(self.coord - other.coord)

    def __rsub__(self, other):
        return vec(self.coord - other.coord)

    def __mul__(self, factor):
        return vec(self.coord*factor)

    def __rmul__(self, factor):
        return vec(self.coord*factor)

    def __truediv__(self, factor):
        return vec(self.coord/factor)

    def plot(self, axis=None):
        if axis == None:
            axis = pyplot
        axis.scatter([self.coord[0]], [self.coord[1]])

class ray:
    def __init__(self, origin, unitvec):
        self.origin = origin
        self.unitvec = unitvec/unitvec.length()

    def to_line(self, length):
        return line(self.origin, length*self.unitvec +
            self.origin)

    def intersect(self, other):
        diff = other.origin - self.origin
        y_0 = diff.dot(self.unitvec.perp())
        dy = other.unitvec.dot(self.unitvec.perp())
        t = -y_0/dy
        return other.origin + t*other.unitvec

    def plot(self, axis=None):
        if axis == None:
            axis = pyplot
        axis.scatter([self.origin.coord[0]], [self.origin
            .coord[1]])
        axis.plot([self.origin.coord[0], self.origin.
```

```
            coord[0] + self.unitvec.coord[0]], [self.
            origin.coord[1], self.origin.coord[1] + self.
            unitvec.coord[1]], linestyle="--")

class line:
    def __init__(self, A, B):
        if (A > B):
            A, B = B, A
        self.A = A
        self.B = B

    def midvec(self):
        return vec(0.5*(self.A.coord + self.B.coord))

    def bisector(self):
        direction = self.B.coord - self.A.coord
        return ray(self.midvec(), vec(direction[1], -
            direction[0]))

    def to_ray(self):
        return ray(self.A, self.B - self.A)

    def __eq__(self, other):
        return (self.A == other.A) and (self.B == other.B
            )

    def __ne__(self, other):
        return (self.A != other.A) or (self.B != other.B)

    def plot(self, axis=None):
        if axis == None:
            axis = pyplot
        self.A.plot(axis)
        self.B.plot(axis)
        axis.plot([self.A.coord[0], self.B.coord[0]], [
            self.A.coord[1], self.B.coord[1]], linestyle="
            -")

class triangle:
    def __init__(self, A, B, C):
        # reorder vecs
        if (A > B):
            A, B = B, A
        if (B > C):
            B, C = C, B
        if (A > B):
```

```
            A, B = B, A
        self.vertice = [A, B, C]
        self.lines = [line(A, B), line(B, C), line(C, A)]
        self._circumcircle()

    def _circumcircle(self):
        ray1 = self.lines[0].bisector()
        ray2 = self.lines[1].bisector()
        self.center = ray1.intersect(ray2)
        self.radius = (self.vertice[0] - self.center).
            length()

    def __eq__(self, other):
        return all(self.vertice[i] == other.vertice[i]
            for i in range(len(self.vertice)))

    def __ne__(self, other):
        return any(self.vertice[i] != other.vertice[i]
            for i in range(len(self.vertice)))

    def plot(self, axis=None):
        if axis == None:
            axis = pyplot
        for line in self.lines:
            line.plot(axis)
        circle = pyplot.Circle(self.center.coord, self.
            radius, fill=False)

#           if axis == pyplot:
#               axis.gca().add_artist(circle)
#           else:
#               axis.add_artist(circle)

class bowyer_watson:
    def __init__(self, size):
        self.size = size
        A = vec(-numpy.sqrt(6)*self.size/2, -self.size/
            numpy.sqrt(2))
        B = vec(numpy.sqrt(6)*self.size/2, -self.size/
            numpy.sqrt(2))
        C = vec(0, 2*self.size/numpy.sqrt(2))
        self.border = [line(A, B), line(B, C), line(C, A)
            ]
        self.triangles = [triangle(A, B, C)]

    def add_point(self, point):
```

```
        if (abs(point.coord[0]) > self.size/2) or (abs(
            point.coord[1]) > self.size/2):
            raise ValueError("point: " + str(point) + "
                out size of bounds for square with length:
                 " + str(self.size) + " centered at the
                origin.")

        bad_triangles = [triangle for triangle in self.
            triangles if ((point - triangle.center).
            _length2() < triangle.radius**2)]
        self.triangles = [triangle for triangle in self.
            triangles if not (triangle in bad_triangles)]

        polygon=[]
        for bad_triangle in bad_triangles:
            for line in bad_triangle.lines:
                line_in_polygon = False
                if line in polygon:
                    i = polygon.index(line)
                    del polygon[i]
                else:
                    polygon = polygon + [line]

        for line in polygon:
            self.triangles = self.triangles + [triangle(
                line.A, line.B, point)]

    def get_voronoi(self, resolution = 100):
        triangles = self.triangles

        outside=[]
        triangle_outside = []
        triangle_inside = []
        for triangle in triangles:
            for tline in triangle.lines:
                line_in_outside = False
                if tline in outside:
                    i = outside.index(tline)
                    triangle_inside = triangle_inside +
                        [(triangle, triangle_outside[i])]
                    del outside[i]
                    del triangle_outside[i]
                else:
                    outside = outside + [tline]
                    triangle_outside = triangle_outside +
                        [triangle]
```

```
        lines = []
        for triangle_pair in triangle_inside:
            A, B = triangle_pair[0].center, triangle_pair
                [1].center
            lines = lines + [line(A, B)]
        return voronoi(lines, [-self.size/2, self.size
            /2], resolution)

    def plot(self, axis=None, setup_only=False):
        if axis == None:
            axis = pyplot
        pyplot.plot([-self.size/2, -self.size/2, self.
            size/2, self.size/2, -self.size/2], [-self.
            size/2, self.size/2, self.size/2, -self.size
            /2, -self.size/2], linestyle="--")
        circle = pyplot.Circle((0., 0.), numpy.sqrt(2)*
            self.size/2, fill=False)

        if axis == pyplot:
            axis.gca().add_artist(circle)
        else:
            axis.add_artist(circle)

        for line in self.border:
            line.plot(axis)

        if not setup_only:
            for triangle in self.triangles:
                triangle.plot(axis)

class voronoi:
    def __init__(self, lines, limit=[-1, 1], resolution
        =100):
        self.lines = lines
        self.limit = limit
        self.resolution = resolution
        self.raster = None
        self.regions = None
        self.neighbors = {}
        self.max_id = 0
        self.filter_max = 0.2

    def rasterize(self):
        self.raster = numpy.zeros((self.resolution, self.
            resolution))
        self.regions = numpy.zeros((self.resolution, self
```

```
            .resolution), dtype=int)
        self.neighbors = {}
        self.max_id = 0

        for line in self.lines:
            self.bresenham_line(self.map_line(line))

    def find_regions(self):
        region_id = 1
        boundry = [(0, 0)]
        while len(boundry) != 0:
            x, y = boundry[0]
            boundry = boundry[1:]
            if self.canvas(x, y + 1):
                if self.regions[x, y + 1] == 0:
                    boundry, region_id = self.find_region
                        (region_id, x, y + 1, boundry)
            if self.canvas(x + 1, y):
                if self.regions[x + 1, y] == 0:
                    boundry, region_id = self.find_region
                        (region_id, x + 1, y, boundry)

    def find_region(self, region_id, x_0, y_0, boundry
        =[]):
        if self.raster[x_0, y_0]:
            return [(x_0, y_0)] + boundry, region_id

        temp = [(x_0, y_0)]
        while len(temp) != 0:
            x, y = temp[0]
            temp = temp[1:]
            self.regions[x, y] = region_id
            if self.is_clear(x, y + 1) and (not (x, y +
                1) in temp):
                temp += [(x, y + 1)]
            elif not self.is_clear(x, y + 1, False):
                if not (x, y + 1) in boundry:
                    boundry += [(x, y + 1)]
                if self.canvas(x, y + 2):
                    other_id = self.regions[x, y + 2]
                    self._add_neighbor(region_id,
                        other_id)

            if self.is_clear(x, y - 1) and (not (x, y -
                1) in temp):
                temp += [(x, y - 1)]
```

```
        elif not self.is_clear(x, y - 1, False):
            if not (x, y - 1) in boundry:
                boundry += [(x, y - 1)]
            if self.canvas(x, y - 2):
                other_id = self.regions[x, y - 2]
                self._add_neighbor(region_id,
                    other_id)

        if self.is_clear(x + 1, y) and (not (x + 1, y
            ) in temp):
            temp += [(x + 1, y)]
        elif not self.is_clear(x + 1, y, False):
            if not (x + 1, y) in boundry:
                boundry += [(x + 1, y)]
            if self.canvas(x + 2, y):
                other_id = self.regions[x + 2, y]
                self._add_neighbor(region_id,
                    other_id)

        if self.is_clear(x - 1, y) and (not (x - 1, y
            ) in temp):
            temp += [(x - 1, y)]
        elif not self.is_clear(x - 1, y, False):
            if not (x - 1, y) in boundry:
                boundry += [(x - 1, y)]
            if self.canvas(x - 2, y):
                other_id = self.regions[x - 2, y]
                self._add_neighbor(region_id,
                    other_id)

    self.max_id = region_id
    return boundry, region_id + 1

def mean_distance(self, resolution=None):
    if resolution != None:
        self.resolution = resolution
    self.rasterize()
    self.find_regions()

    centroids = []
    for i in range(1, self.max_id + 1):
        temp = numpy.zeros((self.resolution, self.
            resolution), dtype=int)
        temp[self.regions == i] = 1
        total = numpy.sum(temp)
        y = numpy.sum(temp*numpy.linspace(0, self.
```

```
            resolution - 1, self.resolution)[:, numpy.
            newaxis])/total
        x = numpy.sum(temp*numpy.linspace(self.
            resolution - 1, 0, self.resolution)[numpy.
            newaxis, :])/total
        centroids += [(y, x)]

    evaluated = []
    distances = []
    for i in range(1, self.max_id + 1):
        for j in self.neighbors[i]:
            a, b = i - 1, j - 1
            if a > b:
                a, b = b, a
            if not (a, b) in evaluated:
                evaluated += [(a, b)]
                dx = centroids[a][0] - centroids[b
                    ][0]
                dy = centroids[a][1] - centroids[b
                    ][1]
                distances += [numpy.sqrt(dx**2 + dy
                    **2)]
    distances = numpy.array(distances)
    return self.inv_map(numpy.mean(distances)), self.
        inv_map(numpy.std(distances)/numpy.sqrt(len(
        distances)))

def bresenham_line(self, line):
    dx = numpy.abs(line.A.coord[0] - line.B.coord[0])
    dy = numpy.abs(line.A.coord[1] - line.B.coord[1])
    x, y = line.A.coord[0], line.A.coord[1]
    sx = -1 if line.A.coord[0] > line.B.coord[0] else
         1
    sy = -1 if line.A.coord[1] > line.B.coord[1] else
         1

    if dx > dy:
        err = dx/2.
        while int(numpy.floor(x)) != int(numpy.floor(
            line.B.coord[0])):
            if self.canvas(numpy.floor(x), numpy.
                floor(y)):
                self.raster[int(numpy.floor(x)), int(
                    numpy.floor(y))] = 1.0
            err -= dy
            if err < 0:
```

```
                y += sy
                err += dx
            x += sx
    else:
        err = dy/2.
        while int(numpy.floor(y)) != int(numpy.floor(
            line.B.coord[1])):
            if self.canvas(numpy.floor(x), numpy.
                floor(y)):
                self.raster[int(numpy.floor(x)), int(
                    numpy.floor(y))] = 1.0
            err -= dx
            if err < 0:
                x += sx
                err += dy
            y += sy
    if self.canvas(numpy.floor(x), numpy.floor(y)):
        self.raster[int(numpy.floor(x)), int(numpy.
            floor(y))] = 1.0

def map_point(self, point):
    return vec(self.resolution*(point.coord - numpy.
        array([self.limit[0], self.limit[0]]))/numpy.
        array([self.limit[1] - self.limit[0], self.
        limit[1] - self.limit[0]]))

def inv_map(self, x):
    return x*(self.limit[1] - self.limit[0])/self.
        resolution

def map_line(self, unmapped):
    return line(self.map_point(unmapped.A), self.
        map_point(unmapped.B))

def canvas(self, x, y):
    return (0 <= x < self.resolution) and (0 <= y <
        self.resolution)

def _add_neighbor(self, region_id, other_id):
    if (region_id != 0) and (other_id != 0):
        if region_id in self.neighbors:
            if not other_id in self.neighbors[
                region_id]:
                self.neighbors[region_id] += [
                    other_id]
        else:
```

```
            self.neighbors[region_id] = [other_id]

        if other_id in self.neighbors:
            if not region_id in self.neighbors[
                other_id]:
                self.neighbors[other_id] += [
                    region_id]
        else:
            self.neighbors[other_id] = [region_id]

    def is_clear(self, x, y, strict=True):
        if strict:
            if self.canvas(x, y):
                if (self.raster[x, y] == 0) and (self.
                    regions[x, y] == 0):
                    return True
            return False
        else:
            if self.canvas(x, y):
                return self.raster[x, y] == 0
            return True

    def gaussian(self, width, bounds, resolution=None):
        if resolution != None:
            self.resolution = resolution
        self.rasterize()

        efall_px = (width/2.)*self.resolution/(self.limit
            [1] - self.limit[0])
        range_px = int((bounds/2.)*self.resolution/(self.
            limit[1] - self.limit[0]))
        if range_px < 1:
            raise ValueError("range is too small, filter
                must be larger than 2x2")
        if efall_px > range_px:
            raise ValueError("gausian width larger than
                filter bounds")
        if range_px > self.resolution*self.filter_max/2.:
            raise ValueError("filter bounds larger than "
                + str(self.filter_max) + " * resolution."
            )

        X, Y = numpy.meshgrid(numpy.linspace(-range_px,
            range_px, 2*range_px + 1), numpy.linspace(-
            range_px, range_px, 2*range_px + 1))
        filter = numpy.exp(-(X**2 + Y**2)/(2.0*efall_px
```

```
            **2))
        filter = filter/numpy.sum(filter)
        self.raster = scipy.ndimage.filters.
            gaussian_filter(self.raster, efall_px,
            truncate=range_px/efall_px)
        self.raster[self.raster > numpy.sum(filter[int(
            len(filter)/2)])] = numpy.sum(filter[int(len(
            filter)/2)])
        self.raster = self.raster/numpy.max(self.raster)

    def plot_raster(self, axis=None):
        if axis == None:
            axis = pyplot
        axis.imshow(numpy.transpose(self.raster, axes=(1,
            0))[::-1], interpolation="none")

    def plot_region(self, axis=None):
        if axis == None:
            axis = pyplot
        axis.imshow(numpy.transpose(self.regions, axes
            =(1, 0))[::-1], interpolation="none")

    def plot_vector(self, axis=None):
        if axis == None:
            axis = pyplot
        for line in self.lines:
            line.plot(axis)

"""
import random
random.seed(0)
r = 2.0
bw=bowyer_watson(r)
for _ in range(10):
    bw.add_point(vec(random.uniform(-r/2, r/2), random.
        uniform(-r/2, r/2)))

#bw.plot()
#pyplot.show()
v = bw.get_voronoi(2000)
#bw.plot(setup_only = True)
#v.plot_vector()
#pyplot.show()
v.gaussian(0.04, 0.10)
#v.plot_raster()
#pyplot.show()
```

```
print(v.mean_distance(400))
"""
```