Code for exploring ATFs

In Section 2.4, we gave a detailed introduction to ATFs. In this online supplement, we will continue the topic to discuss the Python realization of mutations.

To start, we use `Decimal` data type for the calculation. As seen in the mutation sequences providing a full filling at the accumulation point (Proposition 3.2.10), the affine lengths of edges can get extremely small after only a few steps of mutation. This goes beyond the limit of any type of traditional floating data type and could lead to errors and breakdowns of the program. With `Decimal`, however, one can set however many digits needed with exact precision. This helps greatly when looking for the inner corners near the accumulation point. Further, with enough digits, one can compute the continued fractions of the ratios $|OY|/|OX|$ and, after the periodic pattern is clear, reverse engineer the precise quadratic irrational accumulation points. Here is the code for our setup, with 29 digits:

```
import decimal
from decimal import Decimal as D
# number of digits calculated:
decimal.getcontext().prec = 10000
# number of digits printed:
N = 29
```

We construct the `node` class to integrate the vertex, the nodal ray at the vertex, and the edge departing clockwise from that vertex.

```
class node (object):
    def __init__ (self, vertex, nodal_ray, edge,
                                          affine_length):
        self.vertex = [D(vertex[0]), D(vertex[1])]
        self.nodal_ray = [D(nodal_ray[0]), D(nodal_ray[1])]
        self.edge = [D(edge[0]), D(edge[1])]
        self.affine_length = D(affine_length)
```

The next definition, `init_polydisk(b)`, initializes the polydisk $P(1, \beta)$:

```
def init_polydisk (b):
    global n
    global nodes
    n = 4
    nodes = [None] * 4
    nodes[0] = node([0,0], [1,1], [0,1], 1.)
    nodes[1] = node([0,1], [1,-1], [1,0], b)
    nodes[2] = node([b,1], [-1,-1], [0,-1], 1)
    nodes[3] = node([b,0], [-1,1], [-1,0], b)
```

The following two functions, `dist` and `dot`, are hand-written helper functions to facilitate the usage of `Decimal`. We then compute the mutation matrix M.

```
def dist (x,y):
    # distance between x and y
    return ( (x[0]-y[0])**2 + (x[1]-y[1])**2 ).sqrt()
def dot (mat, vec):
    # multiplication of 2*2mat and 2*1vec
    return [ mat[0][0]*vec[0]+mat[0][1]*vec[1],
             mat[1][0]*vec[0]+mat[1][1]*vec[1] ]

def solve_matrix (v1, v2, w1, w2):
    # solve the matrix M such that M(v1)=v2, M(w1)=w2
    mat = [ [w1[1], -v1[1]],
            [-w1[0], v1[0]] ]

    res = [ dot(mat, [v2[0],w2[0]]),
            dot(mat, [v2[1],w2[1]]) ]

    res[0][0] = res[0][0] / (v1[0]*w1[1]-v1[1]*w1[0])
    res[0][1] = res[0][1] / (v1[0]*w1[1]-v1[1]*w1[0])
    res[1][0] = res[1][0] / (v1[0]*w1[1]-v1[1]*w1[0])
    res[1][1] = res[1][1] / (v1[0]*w1[1]-v1[1]*w1[0])

    return res
```

In the program we label the vertices clockwise using the numbers $0-3$, starting from the origin as 0. Then `intersect_one(i,j)` solves for the intersection point between the lines of the i-th nodal ray and the j-th edge. The function will return the intersection point if it lies on the edge segment and $[-1,-1]$ otherwise.

```
def intersect_one (i,j):
    # solve the intersection between i-th nodal ray
    # and j-th edge
    global n
    global nodes

    # copy as local variables
    n1 = nodes[i].vertex
    n2 = nodes[j].vertex
    n3 = nodes[(j+1)%n].vertex
    v1 = nodes[i].nodal_ray
    v2 = nodes[j].edge

    # solve for the intersection point
    vec = [ v1[1]*n1[0]-v1[0]*n1[1],
                v2[1]*n2[0]-v2[0]*n2[1] ]
```

```
    mat = [[ -v2[0], v1[0] ],
           [ -v2[1], v1[1] ]]

    itx = dot(mat, vec)
    itx[0] = itx[0] / (v1[0]*v2[1] - v1[1]*v2[0])
    itx[1] = itx[1] / (v1[0]*v2[1] - v1[1]*v2[0])

    # check if the intersection is on the edge
    if abs(n2[0] - n3[0]) == 0:
        lmbda = (itx[1]-n3[1]) / (n2[1]-n3[1])
    else:
        lmbda = (itx[0]-n3[0]) / (n2[0]-n3[0])
    if (lmbda<0 or lmbda>1):
        return [-1,-1]

    return itx
```

The next function, `intersect_all(x)`, solves for the edge that the x-th nodal ray intersects with. This is achieved by finding the intersections of the x-th nodal ray with all other edges, throwing away invalid intersections, and keeping the one with the shortest distance to the x-th vertex.

```
def intersect_all (x):
    # solve the intersecting edge for the x-th nodal ray
    global n
    global nodes

    # the variables for the intersecting edge
    min_edge = x
    min_itx = []
    min_dis = math.inf

    for i in range(n):
        # i is adjacent to x
        if (i==x or i==(x-1)%n):
        continue

    # the intersection of x-th nodal ray
    # and i-th edge is invalid
    itx = intersect_one(x,i)
    if (itx == [-1,-1]):
        continue

    # maintain the closest intersection
```

```
        dis = dist(nodes[x].vertex, itx)
        if (dis < min_dis):
            min_edge = i
            min_itx = itx
            min_dis = dis

        return (min_edge, min_itx)
```

With the above foundations, the function `mutate(x)` calculates the polygon after mutating the x-th nodal ray. It has two secondary helper functions `mutate_counterclockwise` and `mutate_clockwise`, depending on whether the intersecting edge is to the left or right of the the nodal ray. Here we demonstrate the code for the former as the two are extremely similar.

```
    def mutate_counterclockwise (head, tail, itx):
        # mutate with nodal_ray < intersecting edge
        global n
        global nodes

        mat = solve_matrix( nodes[head].nodal_ray,
                            nodes[head].nodal_ray,
                            nodes[head].edge,
                            nodes[(head-1)%n].edge )

        # construct the new node
        new_length = nodes[tail].affine_length
                     * dist(itx,
                     nodes[(tail+1)%n].vertex)
                     / dist(nodes[tail].vertex,
                     nodes[(tail+1)%n].vertex)
        new = node(itx, [-nodes[head].nodal_ray[0],
                         -nodes[head].nodal_ray[1]],
                         nodes[tail].edge, new_length)
        nodes = np.insert(nodes, tail+1, new)

        # adjust the head and tail node
      nodes[tail].affine_length -= new_length
      nodes[head-1].affine_length += nodes[head].affine_length
      nodes = np.delete(nodes, head)

        # update remaining nodes
        for i in range(head, tail):
            pre = nodes[(i-1)%n]
            nodes[i].vertex[0] = pre.vertex[0]
```

```
                                + pre.affine_length*pre.edge[0]
            nodes[i].vertex[1] = pre.vertex[1]
                                + pre.affine_length*pre.edge[1]
            nodes[i].nodal_ray = dot(mat, nodes[i].nodal_ray)
            nodes[i].edge = dot(mat, nodes[i].edge)

    def mutate (x):
        # mutate once by x-th nodal_ray
        global n
        global nodes

        # y is the intersecting edge
        # itx is the intersection point
        (y, itx) = intersect_all(x)

        if (x<y):
            mutate_counterclockwise(x,y,itx)
            return y
        else:
            mutate_clockwise(y,x,itx)
            return y+1
```

Finally, we have two interface functions `plot_nodes` and `print_embd` that output direct information for use. The first, `plot_nodes`, plots the polygon with respect to edge length ratio; `print_embd` prints the staircase coordinate (z, λ) such that $E(1, z) \overset{s}{\hookrightarrow} P(\lambda, \lambda b)$. The embedding is constructed by fitting the right triangle ΔOXY into the polygon. Below is an example that gets the v^2y^2 example above.

```
b = (6 + 5 * D(30).sqrt()) / 12
init_polydisk(b)
mutate(2)
mutate(2)
mutate(1)
mutate(1)
plot_nodes()
print_embd()
```

It should be easy to generalize the initialization functions for other types of polygons beyond rectangles, such as triangles and trapezoids. For a complete file of the code, see the Github Repository.