/Group « /S /Transparency /I true /CS /DeviceRGB»

Arnold_cat_map

Unknown Author

July 25, 2013

Part I

Calculating and visualising the Arnold cat map

In [61]:
```
%install_ext https://raw.github.com/minrk/ipython_extensions/master/nbtoc.
```

```
Installed nbtoc.py. To use it, type:
  %load_ext nbtoc
```

In [63]:
```
%reload_ext nbtoc
```

In [64]:
```
%nbtoc
```

The Arnold cat map is a canonical example of a (uniformly) *hyperbolic* (“*chaotic*”) dynamical system. In this notebook, we will use computational geometry to construct figures showing the action of the cat map.

In [57]:
```
%nbtoc
```

In [8]:
```
import numpy as np
```

In [9]:
```
def norm(v):
    return np.sqrt(np.dot(v, v))
```

In [10]:
```
%load_ext nbtoc
```

```
The nbtoc extension is already loaded. To reload it, use:
  %reload_ext nbtoc
```

In [11]:
```
%nbtoc
```

1 Simple computational geometry

It is first necessary to do define some simple concepts and functions from computational geometry.

In [13]:
```python
def find_intersection_segments(segment1, segment2):
    """Find intersection of two line segments

    Returns the value of t corresponding to the parametrization of segment
    or None if they do not intersect within the segment ends

    Use representation with normal of ls2 for simplicity

    Parametrize segment1  as  p1 + t.v
    Use representation of segment2 as {x:(x-c).n = 0}
    """

    n = segment2.n
    c = segment2.start

    x = segment1.start
    v = segment1.v


    threshold = 1.e-15

    if abs(np.dot(v,n)) < threshold:  # 0: lines are parallel
        return None


    t_star = np.dot(c - x, n) / np.dot(v, n)  # intersection time

    if 0.0 <= t_star <= 1.0:
        return (t_star, x + t_star * v)

    else:
        return None
```

In [14]:
```python
segment = DirectedLineSegment(np.r_[0, 0], np.r_[2, 2])
```

In [15]:
```python
segment
```

Out [15]: DirectedLineSegment([0, 0], [2, 2])

In [16]:
```python
segment.v
```

Out [16]: array([-2., -2.])

In [17]:
```python
segment1 = DirectedLineSegment([0, 0], [1, 1])
segment2 = DirectedLineSegment([3, 3], [4, 4])

print find_intersection_segments(segment1, segment2)
```

None

In [18]:
```python
segment1 = DirectedLineSegment([0, 0], [1, 1])
segment2 = DirectedLineSegment([1, 0], [0, 2])

print find_intersection_segments(segment1, segment2)
```

None

In [19]:
```python
def test_find_intersection_segments():

    segment1 = DirectedLineSegment([0, 0], [1, 1])
    segment2 = DirectedLineSegment([0, 1], [1, 0])

    t_star, intersection_point = find_intersection_segments(segment1, segm
    assert t_star == 0.5
    assert all(intersection_point == np.array([0.5, 0.5]))


    # parallel lines:
    segment1 = DirectedLineSegment([0, 0], [1, 1])
    segment2 = DirectedLineSegment([3, 3], [4, 4])

    assert find_intersection_segments(segment1, segment2) == None


    segment1 = DirectedLineSegment([0, 0], [1, 1])
    segment2 = DirectedLineSegment([1, 1], [1, 2])

    t_star, intersection_point = find_intersection_segments(segment1, segm
    assert t_star == 1.0
    assert all(intersection_point == np.array([1.0, 1.0]))
```

1.2 Polygons

Next we join line segments up into directed polygons. Here we are assuming that the polygons are closed loops, i.e. that there is a link from the last vertex back to the first.

In [20]:
```python
class DirectedPolygon:
    """A directed polygon

    Parameters
    ==========
    vertices:
        a list of vertices
    """


    def __init__(self, vertices):
        self.vertices = np.asarray(vertices)

        self.segments = []

        for i in range(len(self.vertices)-1):
            self.segments.append(DirectedLineSegment(self.vertices[i], ver

        self.segments.append(DirectedLineSegment(vertices[-1], vertices[0]

    def __repr__(self):
        return self.segments.__repr__()

    def draw(self):
        plt.fill(self.vertices[:,0], self.vertices[:,1], alpha=0.5)
```

In [21]:
```python
unit_square = DirectedPolygon([(0.,0.), (1.,0.), (1.,1.), (0.,1.)])
unit_square
```

Out [21]: [DirectedLineSegment([0, 0], [1, 0]), DirectedLineSegment([1, 0], [1, 1]), DirectedLineSegment([1, 1], [0, 1]), DirectedLineSegment([0, 1], [0, 0])]

In [22]:
```
%matplotlib inline
%config InlineBackend.figure_format = "svg"
from matplotlib import pyplot as plt
from matplotlib import rcParams

from matplotlib.ticker import MultipleLocator

from IPython.config import Config
settings = Config(rcParams)

from matplotlib import style
style.use("ggplot")

settings["font.family"] = "Calluna"
settings["font.sans-serif"] = "Calluna"
```

```
/Users/dsanders/development/matplotlib/lib/matplotlib/__init__.py:934:
UserWarning: Bad val "%(backend)s" on line #32
        "backend      : %(backend)s
"
        in file "/Users/dsanders/Dropbox/ipynb/matplotlibrc"
        Unrecognized backend string "%(backend)s": valid strings are
['pdf', 'pgf', 'Qt4Agg', 'GTK', 'GTKAgg', 'ps', 'agg', 'cairo',
'MacOSX', 'GTKCairo', 'WXAgg', 'template', 'TkAgg', 'GTK3Cairo',
'GTK3Agg', 'svg', 'WebAgg', 'CocoaAgg', 'emf', 'gdk', 'WX']
  (val, error_details, msg))
```

In [23]:
```
settings
```

Out [23]:
```
'agg.path.chunksize': 0,
 'animation.avconv_args': '',
 'animation.avconv_path': 'avconv',
 'animation.bitrate': -1,
 'animation.codec': 'mpeg4',
 'animation.convert_args': '',
 'animation.convert_path': 'convert',
 'animation.ffmpeg_args': '',
 'animation.ffmpeg_path': 'ffmpeg',
 'animation.frame_format': 'png',
 'animation.mencoder_args': '',
 'animation.mencoder_path': 'mencoder',
 'animation.writer': 'ffmpeg',
 'axes.axisbelow': False,
 'axes.color_cycle': ['b', 'g', 'r', 'c', 'm', 'y', 'k'],
 'axes.edgecolor': 'k',
 'axes.facecolor': 'w',
 'axes.formatter.limits': [-7, 7],
 'axes.formatter.use_locale': False,
 'axes.formatter.use_mathtext': False,
 'axes.grid': False,
 'axes.hold': True,
```

```
 'axes.labelcolor': 'k',
 'axes.labelsize': 'medium',
 'axes.labelweight': 'normal',
 'axes.linewidth': 1.0,
 'axes.titlesize': 'large',
 'axes.unicode_minus': True,
 'axes.xmargin': 0,
 'axes.ymargin': 0,
 'axes3d.grid': True,
 'backend': 'module://IPython.kernel.zmq.pylab.backend_inline',
 'backend.qt4': 'PyQt4',
 'backend_fallback': True,
 'contour.negative_linestyle': 'dashed',
 'datapath': '/Users/dsanders/development/matplotlib/lib/matplotlib
/mpl-data',
 'docstring.hardcopy': False,
 'examples.directory': '',
 'figure.autolayout': False,
 'figure.dpi': 80,
 'figure.edgecolor': 'white',
 'figure.facecolor': 'white',
 'figure.figsize': (6.0, 4.0),
 'figure.frameon': True,
 'figure.max_open_warning': 20,
 'figure.subplot.bottom': 0.125,
 'figure.subplot.hspace': 0.2,
 'figure.subplot.left': 0.125,
 'figure.subplot.right': 0.9,
 'figure.subplot.top': 0.9,
 'figure.subplot.wspace': 0.2,
 'font.cursive': ['Apple Chancery',
  'Textile',
  'Zapf Chancery',
  'Sand',
  'cursive'],
 'font.family': 'Calluna',
 'font.fantasy': ['Comic Sans MS',
  'Chicago',
  'Charcoal',
  'ImpactWestern',
  'fantasy'],
 'font.monospace': ['Bitstream Vera Sans Mono',
  'DejaVu Sans Mono',
  'Andale Mono',
  'Nimbus Mono L',
  'Courier New',
  'Courier',
  'Fixed',
  'Terminal',
  'monospace'],
 'font.sans-serif': 'Calluna',
 'font.serif': ['Bitstream Vera Serif',
  'DejaVu Serif',
  'New Century Schoolbook',
```

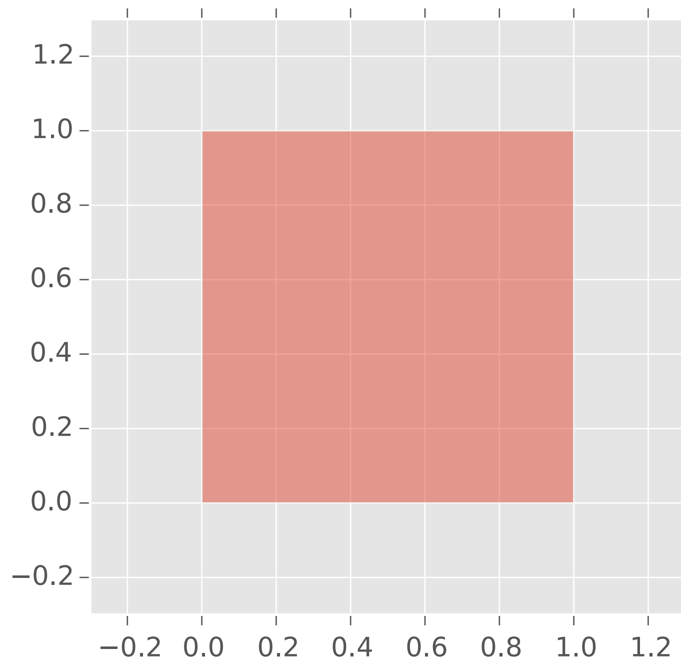
```
 'Century Schoolbook L',
 'Utopia',
 'ITC Bookman',
 'Bookman',
 'Nimbus Roman No9 L',
 'Times New Roman',
 'Times',
 'Palatino',
 'Charter',
 'serif'],
'font.size': 10,
'font.stretch': 'normal',
'font.style': 'normal',
'font.variant': 'normal',
'font.weight': 'normal',
'grid.alpha': 1.0,
'grid.color': 'k',
'grid.linestyle': ':',
'grid.linewidth': 0.5,
'image.aspect': 'equal',
'image.cmap': 'jet',
'image.interpolation': 'bilinear',
'image.lut': 256,
'image.origin': 'upper',
'image.resample': False,
'interactive': True,
'keymap.all_axes': 'a',
'keymap.back': ['left', 'c', 'backspace'],
'keymap.forward': ['right', 'v'],
'keymap.fullscreen': ('f', 'ctrl+f'),
'keymap.grid': 'g',
'keymap.home': ['h', 'r', 'home'],
'keymap.pan': 'p',
'keymap.quit': ('ctrl+w', 'cmd+w'),
'keymap.save': ('s', 'ctrl+s'),
'keymap.xscale': ['k', 'L'],
'keymap.yscale': 'l',
'keymap.zoom': 'o',
'legend.borderaxespad': 0.5,
'legend.borderpad': 0.4,
'legend.columnspacing': 2.0,
'legend.fancybox': False,
'legend.fontsize': 'large',
'legend.frameon': True,
'legend.handleheight': 0.7,
'legend.handlelength': 2.0,
'legend.handletextpad': 0.8,
'legend.isaxes': True,
'legend.labelspacing': 0.5,
'legend.loc': 'upper right',
'legend.markerscale': 1.0,
'legend.numpoints': 2,
'legend.scatterpoints': 3,
'legend.shadow': False,
```

```
 'lines.antialiased': True,
 'lines.color': 'b',
 'lines.dash_capstyle': 'butt',
 'lines.dash_joinstyle': 'round',
 'lines.linestyle': '-',
 'lines.linewidth': 1.0,
 'lines.marker': 'None',
 'lines.markeredgewidth': 0.5,
 'lines.markersize': 6,
 'lines.solid_capstyle': 'projecting',
 'lines.solid_joinstyle': 'round',
 'mathtext.bf': 'serif:bold',
 'mathtext.cal': 'cursive',
 'mathtext.default': 'it',
 'mathtext.fallback_to_cm': True,
 'mathtext.fontset': 'cm',
 'mathtext.it': 'serif:italic',
 'mathtext.rm': 'serif',
 'mathtext.sf': 'sans
-serif',
 'mathtext.tt': 'monospace',
 'patch.antialiased': True,
 'patch.edgecolor': 'k',
 'patch.facecolor': 'b',
 'patch.linewidth': 1.0,
 'path.effects': [],
 'path.simplify': True,
 'path.simplify_threshold': 0.1111111111111111,
 'path.sketch': None,
 'path.snap': True,
 'pdf.compression': 6,
 'pdf.fonttype': 3,
 'pdf.inheritcolor': False,
 'pdf.use14corefonts': False,
 'pgf.debug': False,
 'pgf.preamble': [''],
 'pgf.rcfonts': True,
 'pgf.texsystem': 'xelatex',
 'plugins.directory': '.matplotlib_plugins',
 'polaraxes.grid': True,
 'ps.distiller.res': 6000,
 'ps.fonttype': 3,
 'ps.papersize': 'letter',
 'ps.useafm': False,
 'ps.usedistiller': False,
 'savefig.bbox': None,
 'savefig.directory': '~',
 'savefig.dpi': 72,
 'savefig.edgecolor': 'w',
 'savefig.extension': 'png',
 'savefig.facecolor': 'w',
 'savefig.format': 'png',
 'savefig.frameon': True,
 'savefig.jpeg_quality': 95,
```

```
 'savefig.orientation': 'portrait',
 'savefig.pad_inches': 0.1,
 'svg.embed_char_paths': True,
 'svg.fonttype': 'path',
 'svg.image_inline': True,
 'svg.image_noscale': False,
 'text.antialiased': True,
 'text.color': 'k',
 'text.dvipnghack': None,
 'text.hinting': True,
 'text.hinting_factor': 8,
 'text.latex.preamble': [''],
 'text.latex.preview': False,
 'text.latex.unicode': False,
 'text.usetex': False,
 'timezone': 'UTC',
 'tk.pythoninspect': False,
 'tk.window_focus': False,
 'toolbar': 'toolbar2',
 'verbose.fileo': 'sys.stdout',
 'verbose.level': 'silent',
 'webagg.open_in_browser': True,
 'webagg.port': 8988,
 'webagg.port_retries': 50,
 'xtick.color': 'k',
 'xtick.direction': 'in',
 'xtick.labelsize': 'medium',
 'xtick.major.pad': 4,
 'xtick.major.size': 4,
 'xtick.major.width': 0.5,
 'xtick.minor.pad': 4,
 'xtick.minor.size': 2,
 'xtick.minor.width': 0.5,
 'ytick.color': 'k',
 'ytick.direction': 'in',
 'ytick.labelsize': 'medium',
 'ytick.major.pad': 4,
 'ytick.major.size': 4,
 'ytick.major.width': 0.5,
 'ytick.minor.pad': 4,
 'ytick.minor.size': 2,
 'ytick.minor.width': 0.5
```

In [24]:

```
unit_square.draw()
plt.axis('scaled')

plt.xlim(-0.3, 1.3)
plt.ylim(-0.3, 1.3)

plt.show()
```



In [25]:

```
#theta = np.pi / 4.
#R = np.array([np.cos(theta), np.sin(theta), -np.sin(theta), np.cos(theta),

#rotated_square_vertices = [np.dot(R, vertex) for vertex in unit_square.ve
#rotated_square = DirectedPolygon(rotated_square_vertices)

#rotated_square.draw()
```

In [26]:

```
def find_starting_point(polygon1, polygon2):
    """Find the starting point of the intersection of two polygons
    Returns None if they do not intersect"""

    for i in range(len(polygon1.segments)):
        for j in range(len(polygon2.segments)):

            segment1 = polygon1.segments[i]
            segment2 = polygon2.segments[i]

            intersection = find_intersection_segments(segment1, segment2)

            if intersection:
                return (intersection[1], i, j)  # don't need the intersecti

    return None



def intersection_polygons(polygon1, polygon2):
    """Calculate intersection of two (convex) polygons
    First find an intersection point.
    Then follow this around by taking consecutive pieces of each polygon?
    """

    starting_point, i, j = find_starting_point(polygon1, polygon2)
```

```
    # i is the number of the segment of the first polygon, j of the second

    if starting_point is None:
        return None

    intersection_path = [starting_point]

    current_segment = polygon2.segments[j]
    remainder = DirectedLineSegment(starting_point, current_segment.end)

#    find_intersection_segments(remainder
```

In [27]:
```
def test_find_starting_point():

    unit_square = DirectedPolygon([(0.,0.), (1.,0.), (1.,1.), (0.,1.)])

    theta = np.pi / 4.
    R = np.array([np.cos(theta), np.sin(theta), -np.sin(theta), np.cos(the

    rotated_square_vertices = [np.dot(R, vertex) for vertex in unit_square
    rotated_square = DirectedPolygon(rotated_square_vertices)


    starting_point = find_starting_point(unit_square, rotated_square)

    assert starting_point[0] == 0.0
    assert all(starting_point[1] == np.array([0.0, 0.0]))


    new_square = [vertex + np.array([10., 0.]) for vertex in unit_square.v

    starting_point = find_starting_point(unit_square, new_square)

    assert starting_point == None
```

In [28]:
```
new_square_vertices = [vertex + np.array([10., 0.]) for vertex in unit_squ
new_square = DirectedPolygon(new_square_vertices)
find_starting_point(unit_square, new_square)
```

In [29]:
```
new_square
```

Out [29]:
```
[DirectedLineSegment([10, 0], [11, 0]), DirectedLineSegment([11, 0],
[11, 1]), DirectedLineSegment([11, 1], [10, 1]),
DirectedLineSegment([10, 1], [10, 0])]
```

In [30]:
```
unit_square
```

Out [30]:
```
[DirectedLineSegment([0, 0], [1, 0]), DirectedLineSegment([1, 0], [1,
1]), DirectedLineSegment([1, 1], [0, 1]), DirectedLineSegment([0, 1],
[0, 0])]
```

2 Mapping vertices and polygons

A *map* is a function $f : \mathbb{R}^2 \to \mathbb{R}^2$. Our first example is the map $\mathbf{x} \overset{f}{\mapsto} \mathrm{M} \cdot \mathbf{x}$, where

$$M := \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}:$$

In [31]:
```
M = np.array([[2, 1], [1, 1]])
print M
```

```
[[2 1]
 [1 1]]
```

In [32]:
```
def f(x):
    return np.dot(M, x)
```

`f` applies the map to a single vertex. Now we need something to apply it to all the vertices of a polygon, creating a new polygon:

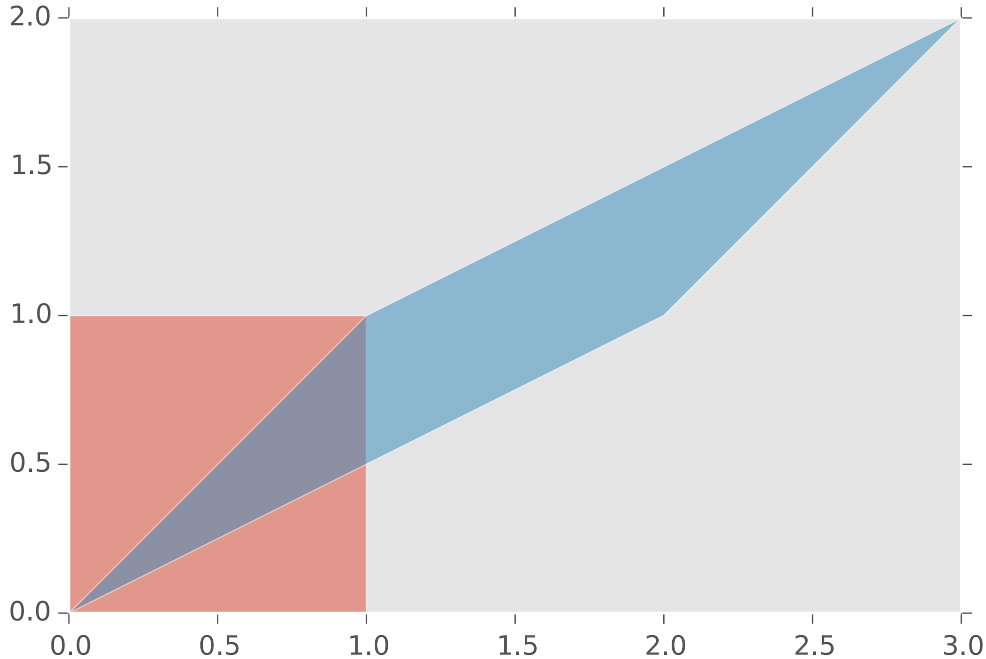
In [33]:
```
def map_poly(f, poly):
    """Apply the map f to each vertex of the Poygon poly, returning a new

    vertices = poly.vertices
    new_vertices = [f(vertex) for vertex in vertices]

    return DirectedPolygon(new_vertices)
```

Let's apply our map `f` to the unit square, and then apply it again to get the second iterate $f^{(2)} := f \circ f$:

In [34]:
```
unit_square = DirectedPolygon([(0.,0.), (1.,0.), (1.,1.), (0.,1.)])

iterates = [unit_square]
iterates.append(map_poly(f, iterates[-1]))
```

Let us draw the original unit square and its first iterate:

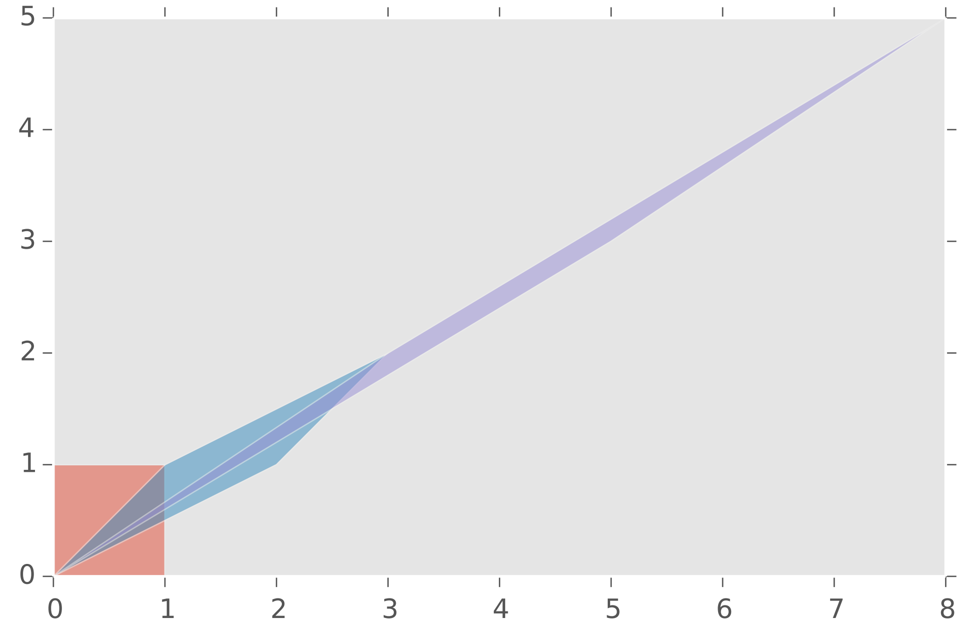
In [35]:
```
for i in iterates[0:2]:
    i.draw()

plt.grid()

plt.axis('scaled')

plt.show()
```



In [36]:
```
def draw_iterates(n):
    """Draw iterates up to and including the nth"""

    while len(iterates) < n+1:
        iterates.append(map_poly(f, iterates[-1]))

    for i in iterates[0:n+1]:
        i.draw()

    plt.grid()

    plt.axis('scaled')

    plt.show()
```

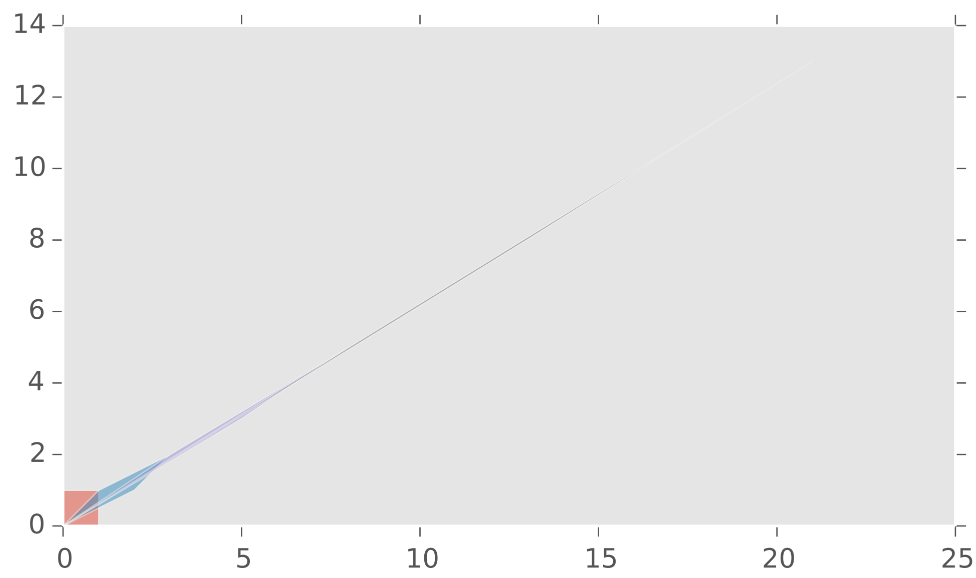
Let's make a function to draw higher iterates:We can add the second iterate to the mix:

In [37]:
```
draw_iterates(2)
```



And why not add the next as well, just for fun:

In [38]:
```
draw_iterates(3)
```



It is clear that the resulting parallelogram is aligning itself along a line. This line is the *unstable manifold* of the origin. Since the map is *linear*, the unstable manifold is equal to the unstable *subspace* of the linearization (which is just the map itself, since it is linear).

The unstable subspace is given by the eigenvector whose eigenvalue is larger than 1. Let's calculate these:

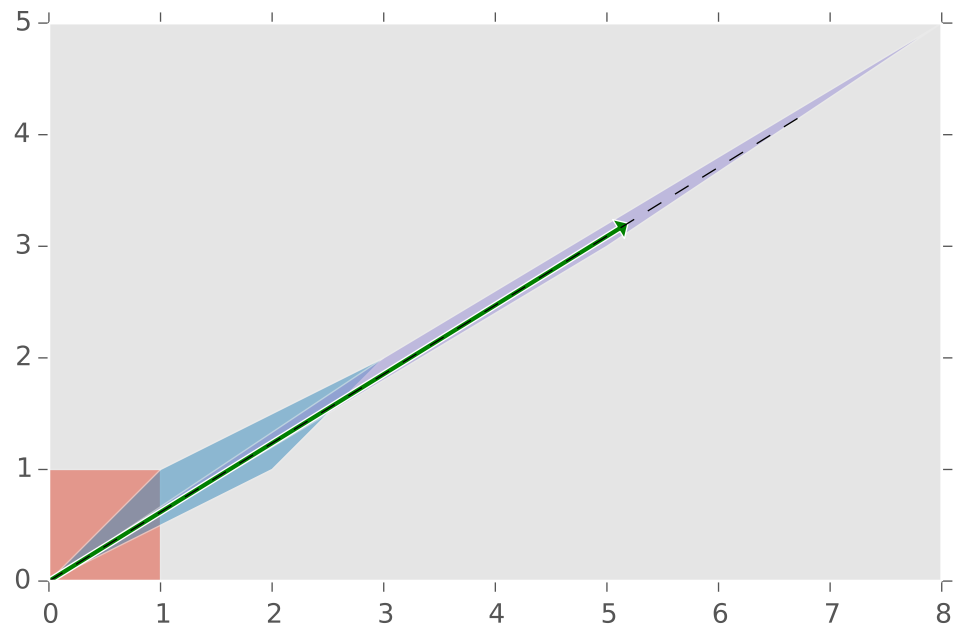
In [39]:
```
from numpy import linalg
```

In [40]:
```
lamb, v = linalg.eig(M)
lamb, v
```

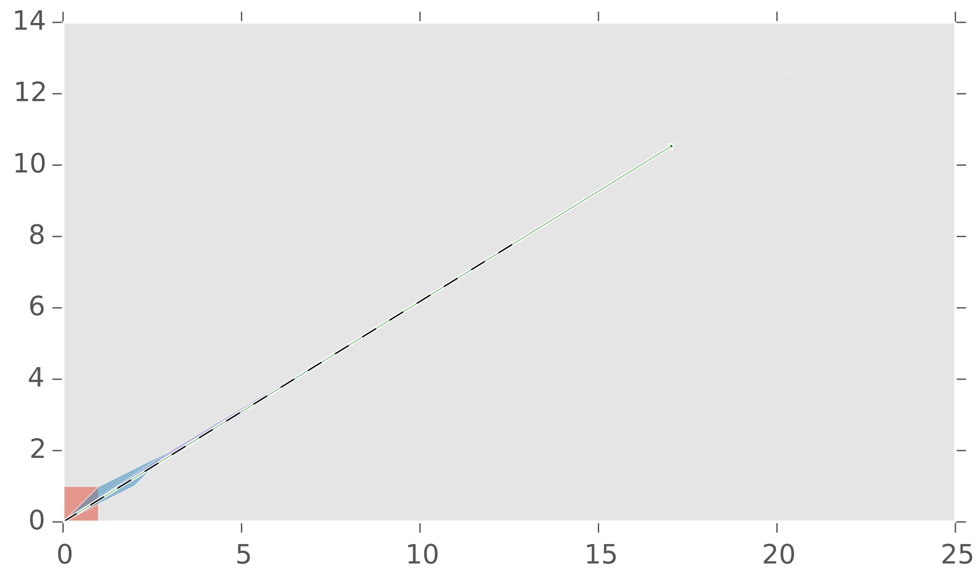
Out [40]:
```
(array([ 2.61803399,  0.38196601]),
 array([[ 0.85065081, -0.52573111],
        [ 0.52573111,  0.85065081]]))
```

Note that the eigenvectores are returned in the *columns* of the matrix. We can now add the eigenvector to our plot:

In [41]:
```
fig, (ax1) = plt.subplots()

plt.plot([0, v[0,0]*8], [0, v[1,0]*8], 'k--', lw=0.5)
plt.arrow(0, 0, v[0,0]*6, v[1, 0]*6, axes=ax1, head_width=0.2, head_length

draw_iterates(2)
```



In [42]:
```
plt.plot([0, v[0,0]*15], [0, v[1,0]*15], 'k--', lw=0.5)
plt.arrow(0, 0, v[0,0]*20, v[1, 0]*20, axes=ax1, head_width=0.2, head_leng

draw_iterates(3)
```



In [43]:
```
ax.
```

```
  File "<ipython-input-43-8245e8db10a3>", line 1
    ax.
       ^
SyntaxError: invalid syntax
```

2.1 Interpolating the Arnold cat map

To visualise a bit better the stretching action of the map, we could think about interpolating between the identity and the cat map. To look at that, we would like to do some symbolic computations, so we import the `sympy` package:

In [ ]:
```
import sympy
```

In [ ]:
```
from sympy import init_printing
init_printing()  # turn on pretty printing in the notebook
```

Let's define the cat map and the identity matrix:

In [ ]:
```
Identity = sympy.Matrix([[1, 0], [0, 1]])
CatMap = sympy.Matrix([[2, 1], [1, 1]])

Identity, CatMap
```

(Strictly speaking, this is not the cat map, since we have not included the modulo-1 operation. However, we can think of this as a lift of the cat map.)Let's try a linear interpolation with a parameter α:

```
In []: alpha = sympy.symbols('alpha')
       alpha
```

Let's define the interpolated matrix $\mathsf{M}_\alpha := \alpha\mathsf{I} + (1-\alpha)\mathsf{M}$:

```
In []: M_interp = lambda alpha:  alpha*Identity + (1-alpha)*CatMap
       M_interp(alpha)
```

It's determinant is:

```
In [44]: det = M_interp(alpha).det()
         det
```

```
        ---------------------------------------------------------------
-------------

    NameError                                 Traceback (most recent
call last)

        <ipython-input-44-88b5ddc9158f> in <module>()
    ----> 1 det = M_interp(alpha).det()
          2 det


        NameError: name 'M_interp' is not defined
```

To have an area-preserving map, we need the determinant to be 1, so we solve the equation $\det(\mathsf{M}_\alpha) = 1$ for α:

```
In []: sympy.solve(det - 1, alpha)
```

The only solutions are 0 and 1, which correspond to the identity and the cat map, respectively. So we see that we *cannot* interpolate linearly between these two matrices in a way that preserves area.This is because the area-preserving condition is very strong. Let's try a different approach: we fix the origin and linearly interpolate only *one* of the vertices, for example the furthest one, $(1,1)$. Then the interpolation gives

$$\mathbf{x}_\alpha = \mathsf{M}_\alpha \left[\begin{pmatrix} 1 \\ 1 \end{pmatrix} \right] :$$

```
In [45]: x = lambda alpha: M_interp(alpha) * sympy.Matrix([1,1])
         x(alpha)
```

```
        ---------------------------------------------------------------
-------------

    NameError                                 Traceback (most recent
call last)

        <ipython-input-45-42d68db75eae> in <module>()
          1 x = lambda alpha: M_interp(alpha) * sympy.Matrix([1,1])
    ----> 2 x(alpha)


        NameError: name 'alpha' is not defined
```

Let's choose one other point to be along the line joining $(1,0)$ to its image. We have already seen that we can't use the same interpolation parameter, so we choose a different one, β:

In [46]:
```
beta = sympy.symbols("beta")
beta
```

```
    ---------------------------------------------------------------

-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-46-af354a7b0673> in <module>()
    ----> 1 beta = sympy.symbols("beta")
          2 beta


        NameError: name 'sympy' is not defined
```

In [47]:
```
y = lambda beta: M_interp(beta) * sympy.Matrix([1,0])
z = lambda beta: M_interp(beta) * sympy.Matrix([0,1])

y(beta), z(beta)
```

```
    ---------------------------------------------------------------

-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-47-81d3eb615122> in <module>()
          2 z = lambda beta: M_interp(beta) * sympy.Matrix([0,1])
          3
    ----> 4 y(beta), z(beta)


        NameError: name 'beta' is not defined
```

Let's calculate the area of the resulting shape; note that the shape is *not* a parallelogram, but rather an arbitrary quadrilateral. This is given in terms of the vectors describing the diagonals (lines joining opposite vertices) of the qualidrateral, $\mathbf{d}_1$ and $\mathbf{d}_2$, by

$$A = \frac{1}{2}\left|\mathbf{d}_1 \times \mathbf{d}_2\right|;$$

see this Wikipedia page.

In [48]:
```
diag1 = lambda alpha: x(alpha)
diag2 = lambda beta: y(beta) - z(beta)

diag1(alpha), diag2(beta)
```

```
    ---------------------------------------------------------------

-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-48-9e69d7a00b91> in <module>()
          2 diag2 = lambda beta: y(beta) - z(beta)
          3
    ----> 4 diag1(alpha), diag2(beta)
```

```
NameError: name 'alpha' is not defined
```

In [49]:
```
def cross(v1, v2):
    return v1[0]*v2[1] - v1[1]*v2[0]
```

In [50]:
```
A = lambda alpha, beta: cross(diag1(alpha), diag2(beta)) / 2
A(alpha, beta)
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-50-e26e34161d62> in <module>()
          1 A = lambda alpha, beta: cross(diag1(alpha), diag2(beta)) /
2
    ----> 2 A(alpha, beta)


        NameError: name 'alpha' is not defined
```

We wish this to be 1 to have an area-preserving (*nonlinear*!) map, so we solve for β to satisfy this constraint:

In [51]:
```
beta = sympy.solve(A(alpha, beta)-1, beta)[0]
beta
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-51-bdcf35f33283> in <module>()
    ----> 1 beta = sympy.solve(A(alpha, beta)-1, beta)[0]
          2 beta


        NameError: name 'sympy' is not defined
```

In [52]:
```
beta.subs({alpha:0.5})
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-52-8ae3983a34f6> in <module>()
    ----> 1 beta.subs(alpha:0.5)


        NameError: name 'beta' is not defined
```

Since this is out of the allowed range $[0, 1]$, there is no way to obtain an area-preserving map apparently.So the new interpolating *nonlinear* map would be as follows:We now have the coordinates of the 4 vertices of the new quadrilateral, given by $(0,0)$, $\mathbf{y}(\beta)$, $\mathbf{z}(\beta)$ and $\mathbf{x}(\alpha)$.

In [53]:
```
y(beta).subs({alpha:0.5})
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-53-c073fa6f6b8b> in <module>()
    ----> 1 y(beta).subs(alpha:0.5)


        NameError: name 'beta' is not defined
```

In [54]:
```
origin = sympy.Matrix([0, 0])
origin
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-54-5717539c7a6c> in <module>()
    ----> 1 origin = sympy.Matrix([0, 0])
          2 origin


        NameError: name 'sympy' is not defined
```

In [55]:
```
new_vertices = lambda alpha, beta: [origin, y(beta), x(alpha), z(beta)]
```

In [56]:
```
new_vertices_float = np.array([vertex.subs({alpha:0.5}).evalf() for verte
new_vertices_float
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-56-fa3e62f52b2c> in <module>()
    ----> 1 new_vertices_float =
np.array([vertex.subs(alpha:0.5).evalf() for vertex in
new_vertices(alpha, beta)])
          2 new_vertices_float


        NameError: name 'alpha' is not defined
```

2.2 Correct attempt

Rather, we must recognise that if we stretch out the farthest corner, the other corners must be pulled *in* towards $y = x$ in order to maintain the area-preserving property. To obtain a parallelogram, we impose that $\mathbf{yy}_\alpha$ and $\mathbf{zz}_\alpha$ are reflection-symmetric around $y = x$.

In [56]:

```
```