LittleBookOfSAGE Documentation

Release 0.1

Avril Coghlan

January 29, 2017

Contents

By Avril Coghlan, University College Cork, Cork, Ireland. Email: a.coghlan@ucc.ie

This is a simple introduction to using the free and Open-Source SAGE mathematics software.

Contents:

CHAPTER 1

Sequences obtained by iteration of continuous real functions

1.1 Using SAGE for Iteration

This chapter tells you how to use the free and Open-Source SAGE mathematics software for studying iteration sequences generated by iteration of continuous real functions.

To use SAGE, you first need to start the SAGE program on your computer. You should have already installed SAGE on your computer (if not, for instructions on how to install SAGE, see the SAGE Installation Guide).

This booklet assumes that the reader has some basic knowledge of iteration, and the principal focus of the booklet is not to explain iteration, but rather how to study iteration sequences using SAGE.

If you are new to iteration, and want to learn more about any of the concepts presented here, I would highly recommend the Open University book "Iteration" (product code MS221 chapter B1), available second-hand from from the Open University Book Search.

1.2 Plotting iteration sequences of real functions

The iteration sequence:

(for n = 0, 1, 2, ...) is generated by the real function:

We can calculate the first 10 terms of the iteration sequence generated using function f(x), with initial term x = 0.5, by typing:

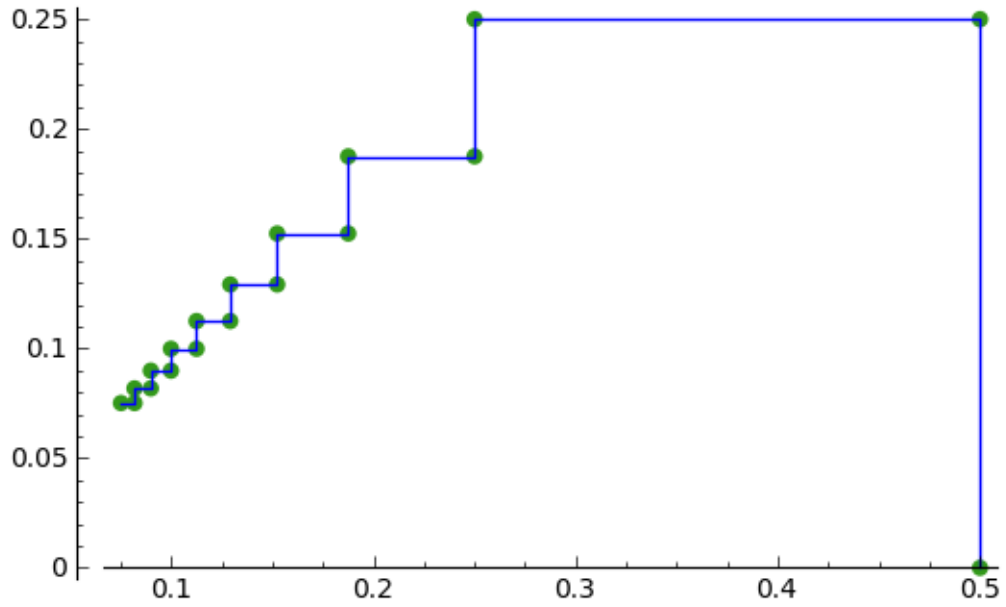
```
f(x) = x*(1-x)
p = 0.5
print(0,p)
for i in range(1,10):
   newp = f(p)
   print(i,newp)
   p = newp
# Output:
# (1, 0.250000000000000)
# (2, 0.187500000000000)
# (3, 0.152343750000000)
# (4, 0.129135131835938)
# (5, 0.112459249561653)
# (6, 0.0998121667496825)
# (7, 0.0898496981184161)
# (8, 0.0817767298664456)
# (9, 0.0750892963187959)
```

We can define a function to make a plot of successive points in an interation sequence defined by a real function f(x):

```
def plot_iter(f, n, x0):
    mypoints = []
    var('mypoint')
    mypoint = vector([x0,0])
    mypoints.append(mypoint)
    for i in range(1,n):
        newx = f(x0)
        mypoint = vector([x0,newx])
        mypoints.append(mypoint)
        mypoint = vector([newx,newx])
        mypoints.append(mypoint)
        x0 = newx
    myplot = points(mypoints, rgbcolor=(0.2,0.6, 0.1), pointsize=30) + line(mypoints)
    myplot.show()
```

Then, to make a plot of the first 10 successive points in the iteration sequence above, using an initial x value of x = 0.5, we can type:

```
f(x) = x*(1-x)
plot_iter(f, 10, 0.5)
```

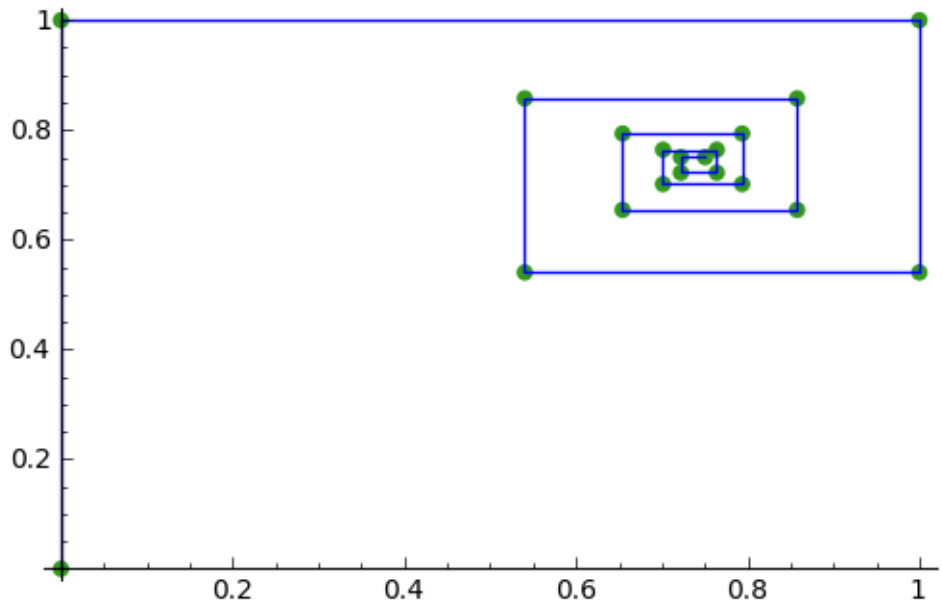


Another example is the iteration sequence:

(for n = 0, 1, 2, ...). This iteration sequence is generated by the real function:

To make a plot of the first 10 successive points in the iteration sequence above, with an initial x value of 0.0, we can type:

```
f(x) = cos(x)
plot_iter(f, 10, 0.0)
```



1.3 Finding fixed points of iteration sequences of real functions

A fixed point of an iteration sequence is a point for which the fixed point equation is true:

That is, if the iteration sequence is generated by real function f(x), the fixed point equation is:

The fixed point equation of the iteration sequence:

is:

This iteration sequence is generated by the real function:

To find the fixed point(s), we can solve for x in SAGE by typing:

```
solve( x*(1-x) == x, x)
# Output:
# [x == 0]
```

This tells us that the fixed point of iteration sequence:

is x = 0. In this case, it happens that this is an attracting fixed point, and the sequence converges to the fixed point x = 0 at the limit as n goes to Infinity.

Similarly, the fixed point equation of the iteration sequence:

is:

To find the fixed point, we can solve the fixed point equation in SAGE by typing:

```
solve( 0.5*(x + (3/x)) == x, x)
# Output:
# [x == -sqrt(3), x == sqrt(3)]
```

This tells us that the fixed points are x = -sqrt(3) and x = sqrt(3). Here it happens that x = sqrt(3) is an attracting fixed point, and the sequence converges to sqrt(3) in the limit as n goes to Infinity.

Sometimes, SAGE does not give us a solution to the fixed point equation. For example, for the iteration sequence:

(for n = 0, 1, 2, ...), the fixed point equation is:

If we try to solve this in SAGE, we don't get a useful answer:

```
solve (cos(x) == x, x)
# Output:
# [x == cos(x)]
```

In this case, we need to use the find_root() to solve the equation numerically. For example, to find a solution to the equation x = cos(x) in the range 0 to pi/2, we type:

```
find_root(cos(x) == x,0,pi/2)
# Output:
# 0.73908513321516067
```

This tells us that a fixed point of the iteration sequence is approximately x = 0.739. It happens that x = 0.739 is an attracting fixed point, and this iteration sequence will converge to x = 0.73908513321516067 in the limit as n goes to Infinity.

Similarly, the fixed point equation of the iteration sequence:

(where n = 0, 1, 2...) is:

To find the fixed points we solve the fixed point equation:

```
solve( (x*x) - 2.4 == x, x)
# Output:
# [x == -1/10*sqrt(265) + 1/2, x == 1/10*sqrt(265) + 1/2]
```

That is, the fixed points are x = -1/10*sqrt(265) + 1/2 and x = 1/10*sqrt(265) + 1/2. In this case, the fixed points happen to be repelling fixed points, and the iteration sequence tends to Infinity as n goes to Infinity.

1.4 Classifying fixed points of iteration sequences generated by real functions

The fixed point a of an iteration sequence generated by a function f(x) could be either attracting or repelling or indifferent.

The fixed point a is attracting if |f'(a)| < 1, is repelling if |f'(a)| > 1, and is indifferent if |f'(a)| = 1.

To classify a fixed point of an iteration sequence of a real function as attracting or repelling or indifferent, we can find the gradient of the function at the fixed point.

For example, for the iteration sequence:

the fixed point equation is:

and we can solve it by typing:

```
solve( ((1/8)*x*x) - x + 7 == x, x)
# Output:
# [x == -2*sqrt(2) + 8, x == 2*sqrt(2) + 8]
```

This tells us that there are two fixed points, x = -2*sqrt(2) + 8, and x = 2*sqrt(2) + 8.

To classify these two fixed points as attracting, repelling or indifferent, we need to find the gradient of the function:

at each fixed point.

The gradient of the function can be found by differentiating f(x), that is, finding f'(x):

```
f(x) = ((1/8)*x*x) - x + 7
diff(f(x))
# Output:
# 1/4*x - 1
```

Therefore, we can calculate the gradient at each fixed point by calculating the value of f'(x) at each fixed point:

```
f2(x) = diff(f(x))
f2(-2*sqrt(2) + 8)
# Output:
# -1/2*sqrt(2) + 1
f2(2*sqrt(2) + 8)
# Output:
# 1/2*sqrt(2) + 1
```

We probably would like to round these values to three decimal places, by typing:

```
round(-1/2*sqrt(2) + 1,3)
# Output:
# 0.293
round(1/2*sqrt(2) + 1,3)
# Output:
# 1.707
```

That is, the gradient of the function f(x) at the fixed point x = -2*sqrt(2) + 8 is about 0.293. The absolute value of 0.293 is less than 1, so x = -2*sqrt(2) + 8 is an attracting fixed point.

The gradient of f(x) at the other fixed point x = 2*sqrt(2) + 8 is about 1.707. The absolute value of 1.707 is greater than 1, so x = 2*sqrt(2) + 8 is a repelling fixed point; that is, no iteration sequence generated by f(x) converges to x = 2*sqrt(2) + 8 unless x_{n} = 2*sqrt(2) + 8 for some value of n.

1.5 Finding the interval of attraction of an attracting fixed point

If a is an attracting fixed point with an attracting fixed point a, then an "interval of attraction" I for fixed point a is an interval containing a, for which |f'(x)| < 1.

For example, the iteration sequence:

is generated by the real function:

We know from above that an attracting fixed point of this iteration sequence is x = -2*sqrt(2) + 8.

To find the interval of attraction for the attracting fixed point x = -2*sqrt(2) + 8, we need to find the interval for which |f'(x)| < 1, that is -1 < f'(x) < 1. We can do this in SAGE by typing:

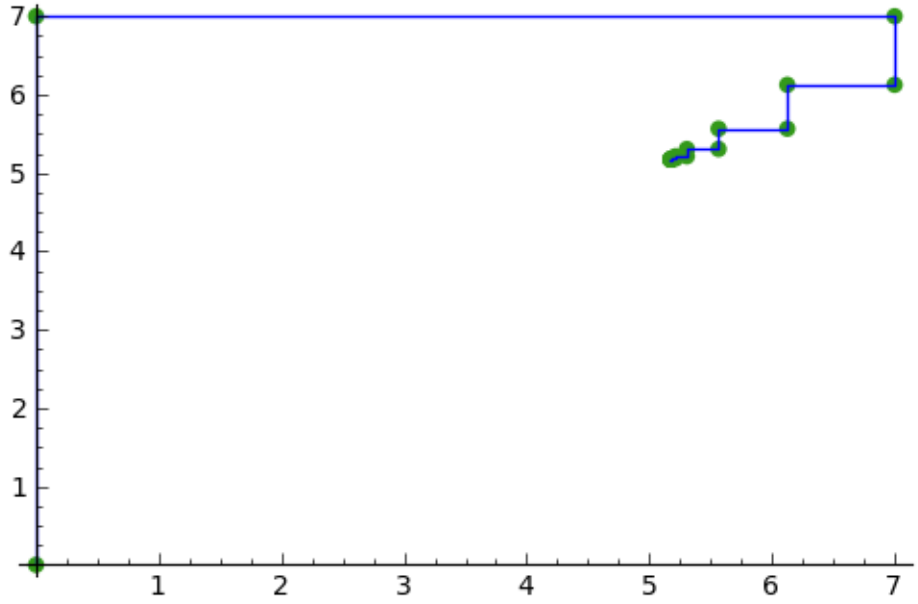
```
f(x) = ((1/8)*x*x) - x + 7
f2(x) = diff(f(x))
solve( -1 < f2(x), x)
# Output:
# [[x > 0]]
solve( f2(x) < 1, x)
# Output:
# [[x < 8]]
```

Thus, an interval of attraction for the attracting fixed point x = -2*sqrt(2) + 8 is (0, 8).

This means that if we start of with an initial value of x that is within this interval of attraction, for example, x = 0, the iteration sequence:

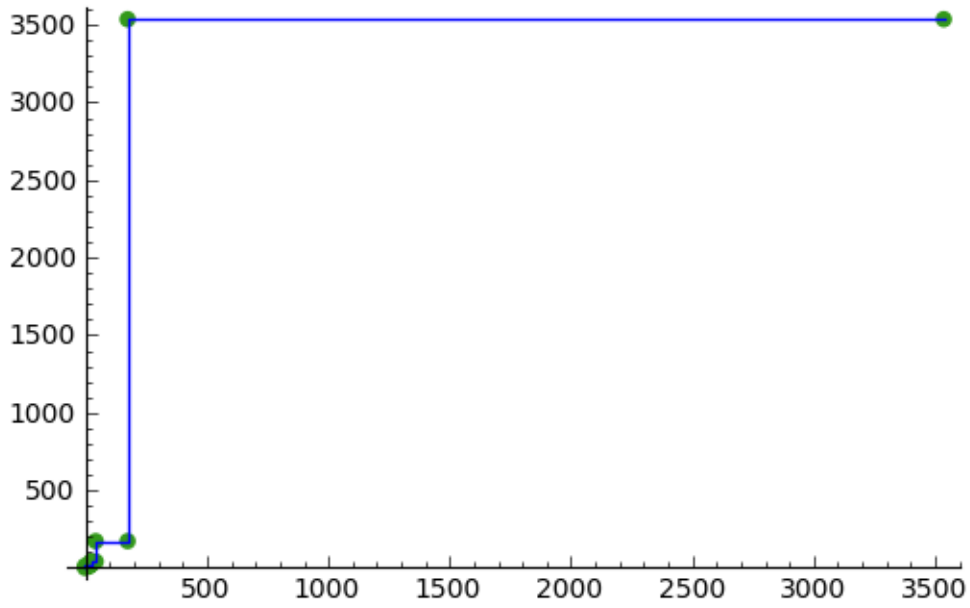
will converge to the attracting fixed point -2*sqrt(2) + 8 (about 5.172). We can show this by plotting it:

```
f(x) = ((1/8)*x*x) - x + 7
plot_iter(f, 10, 0.0)
```



On the other hand, if we start with an initial value of x that is outside of the interval of attraction (0, 8), for example, x = -3.5, then the iteration sequence will probably not converge. Let's make a plot to see:

```
f(x) = ((1/8)*x*x) - x + 7
plot_iter(f, 8, -3.5)
```



In this case, the picture shows that the iteration sequence seems to be increasing very rapidly, and probably will tend to Infinity, as n goes to Infinity.

1.6 Finding two-cycles of an interation sequence generated using a real function

The numbers a and b form a "two-cycle" of a real function f(x) if: f(a) = b, and f(b) = a, and a and b are distinct numbers.

For such a two-cycle, since f(b) = a, it is also true that f(f(a)) = a. Likewise, since f(a) = b, it is also true that f(f(b)) = b.

Therefore, to find the two-cycles of a real function f(x), we need to solve the two-cycle equation:

For example, to find the two-cycles of the function:

we can type in SAGE:

```
f(x) = -(x*x) + (2*x) + 1
solve( f(f(x)) == x, x)
# Output:
# [x == 1, x == 2, x == -1/2*sqrt(5) + 1/2, x == 1/2*sqrt(5) + 1/2]
round(-1/2*sqrt(5) + 1/2,3)
# Output:
# -0.618
round(1/2*sqrt(5) + 1/2,3)
# Output:
# 1.618
```

We can check the answers by seeing the values of f(x) for each of the solutions:

```
f(1)
# Output:
# 2
f(2)
# Output:
#  1
f(-1/2*sqrt(5) + 1/2)
# Output:
# -1/4*(sqrt(5) - 1)^2 - sqrt(5) + 2
round(-1/4*(sqrt(5) - 1)^2 - sqrt(5) + 2,3)
# Output:
# -0.618
f(1/2*sqrt(5) + 1/2)
# Output:
# -1/4*(sqrt(5) + 1)^2 + sqrt(5) + 2
round(-1/4*(sqrt(5) + 1)^2 + sqrt(5) + 2,3)
# Output:
# 1.618
```

This tells us that a = 2 and b = 1 is a two-cycle of the function:

In addition, x = -1/2*sqrt(5) + 1/2 is also a solution of the two-cycle equation:

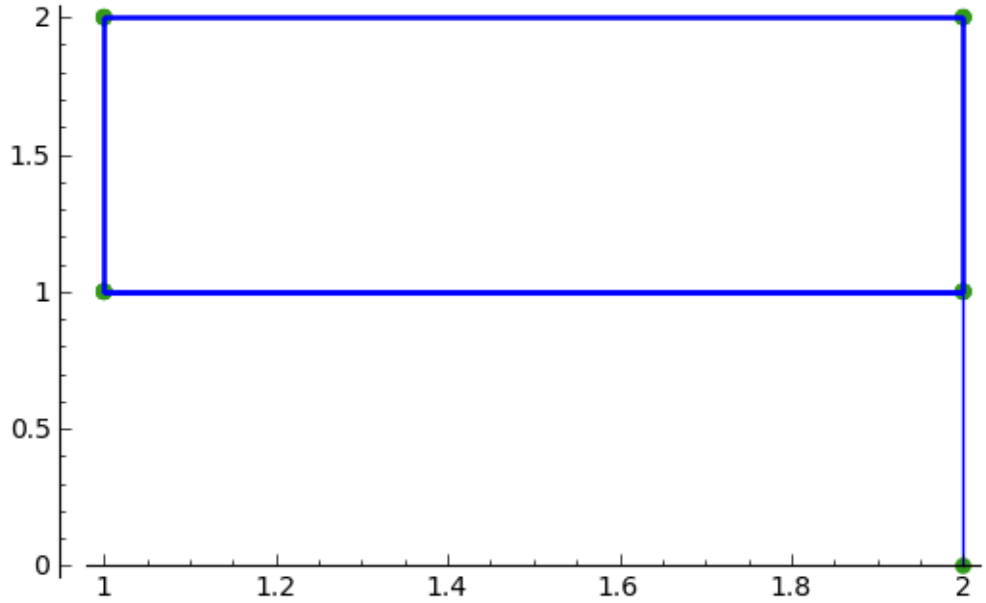
but is not a two-cycle, as f(x) = x, that is, the iteration sequence does not cycle between two distinct values. This is also true for x = 1/2*sqrt(5) + 1/2.

Thus, the only proper two-cycle of the function f(x) is 1, 2.

This means that the iteration sequence:

generated by function f(x), will alternate indefinitely between the values x = 2 and x = 1. We can investigate this by plotting the iteration sequence in SAGE:

```
f(x) = -(x*x) + (2*x) + 1
plot_iter(f, 10, 2.0)
```



In general, the solutions of the two-cycle equation f(f(x))=x are either fixed points of f(x) or members of two-cycles of f(x).

1.7 Classifying two-cycles of interation sequences of real functions

A two-cycle a, b of an iteration sequence could be either attracting, repelling or indifferent.

If |f'(a) * f'(b)| < 1, the two-cycle is attracting; if |f'(a) * f'(b)| > 1, it is repelling; if |f'(a) * f'(b)| = 1, it is indifferent; and if |f'(a) * f'(b)| = 0, it is super-attracting.

To find out, we find |f'(a) * f'(b)|. For example, to find out whether the two-cycle 1, 2 of

is attracting, repelling or indifferent, we type:

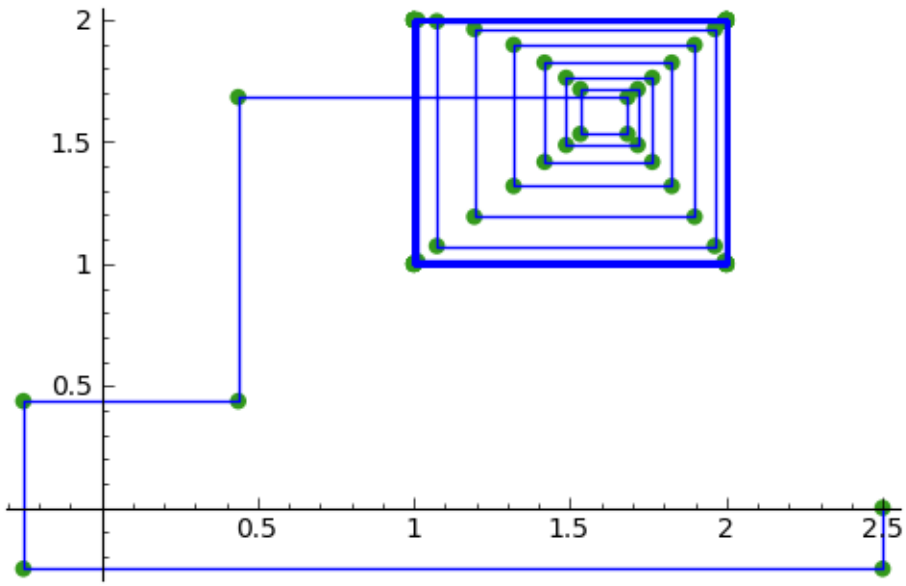
```
f(x) = -(x*x) + (2*x) + 1
f2(x) = diff(f(x))
f2(1) * f2(2)
# Output:
# 0
```

Here we find that |f'(a) * f'(b)| is 0, so the two-cycle 1, 2 is super-attracting.

This means that the iteration sequence generated by function f(x), which starts at a nearby x-value, is likely to converge very quickly to the two-cycle 1, 2.

For example, if we start off with a nearby initial x-value of x = 2.5, let's see if the iteration sequence converges to the two-cycle 1, 2:

```
f(x) = -(x*x) + (2*x) + 1
plot_iter(f, 30, 2.5)
```



The picture shows that iteration sequence starting with x = 2.5 does indeed converge fairly quickly to the super-attracting two-cycle 1, 2.

1.8 Finding p-cycles of a real function

Some real functions have cycles that are longer than two, for example, the cycle

(n = 0, 1, 2...), where the initial value of x = 0, has a three-cycle, and so alternates between three numbers (approximately 1.3, 0.0, and -1.8).

A real function that cycles between p numbers is said to have a p-cycle.

To find the p-cycles of a real function in SAGE, we can define a SAGE function to find the p-cycles for us (thanks to D. S. McNeil and John Cremona of the SAGE support mailing list). for help with this):

```
def iter_apply(f0, n):
    fs = [f0()]
    for i in xrange(n-1):
        last = fs[-1]
        fs.append(f0(last))
    return fs
def find_pcycles(f0, n):
    fs = iter_apply(f0, n)
    req = fs[-1] == x # defining equation of the cycle
    roots = req.roots(ring=RR)
    for root, mult in roots:
        yield [fi(x=root) for fi in fs]
```

We can use the function find_pcycles() to find cycles of length 1 of function f(x):

```
f(x) = (x^2) - (176/100)
list(find_pcycles(f, 1))
# Output:
# [[-0.917744687875782],
#  [1.917744687787578]]
```

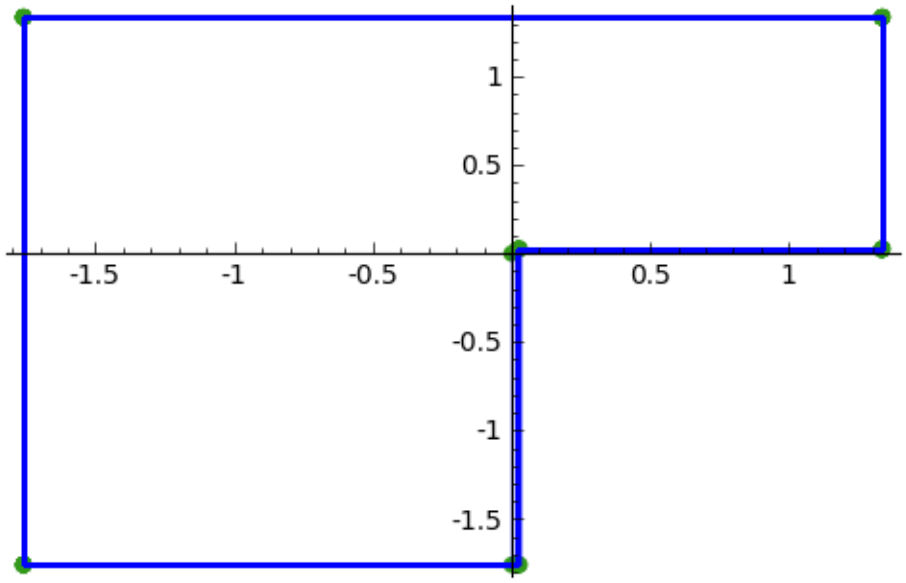
Similarly, we can find cycles of length 2 or 3:

```
list(find_pcycles(f, 2))
# Output:
# [[0.504987562112089, -1.50498756211209],
#  [-0.917744687875782,-0.917744687875783],
#  [-1.50498756211209, 0.504987562112089],
#  [1.91774468787578, 1.917744687875781]]
list(find_pcycles(f, 3))
# Output:
#  [[1.33560128916887, 0.0238308036295500, -1.75943209279837],
#   [1.27545967679485, -0.133202612870383, -1.74225706392450],
#   [-0.917744687875782, -0.917744687875783, -0.917744687875782],
#   [-1.74225706392451, 1.27545967679486, -0.133202612870348],
#   [-1.75943209279837, 1.33560128916886, 0.0238308036295145],
#   [-0.133202612870345, -1.74225706392451, 1.27545967679486],
#   [0.0238308036295096, -1.75943209279837, 1.33560128916886],
#   [1.91774468787578, 1.91774468787579, 1.917744687875791]]
```

One of the three-cycles found is 0.0238308036295096, -1.75943209279837, 1.33560128916886, which is approximately 0.0, -1.8 and 1.3, as mentioned above.

Let's plot the iteration sequence:

```
f(x) = (x^2) - (176/100)
plot_iter(f, 30, 0.0)
```



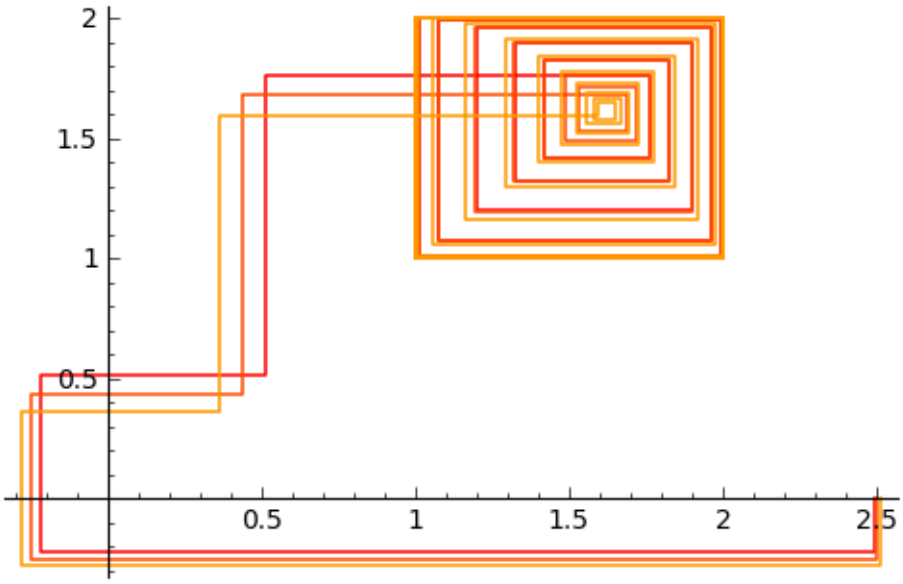
The picture shows that iteration sequence starting with an initial value of x of x = 0.0 does indeed iterate between approximately 0.0, -1.8 and 1.3, in a three-cycle.

1.9 And now for something pretty...

Let's make a nice cobweb pattern:

```
def plot_iter2(f, n, initialvalues):
    g = Graphics()
    for j, initialvalue in enumerate(initialvalues):
```

```
        x0 = initialvalue
        mypoints = []
        var('mypoint')
        mypoint = vector([x0,0])
        mypoints.append(mypoint)
        for i in range(1,n):
            newx = f(x0)
            mypoint = vector([x0,newx])
            mypoints.append(mypoint)
            mypoint = vector([newx,newx])
            mypoints.append(mypoint)
            x0 = newx
        myplot = line(mypoints,color=hue(sin(j/20)),thickness=1.1)
        g = g + myplot
    g.show()
f(x) = -(x*x) + (2*x) + 1
initialvalues = (2.49,2.50,2.51)
plot_iter2(f, 30, initialvalues)
```



I wouldn't mind a dress made out of that pattern!

1.10 Links and Further Reading

Some links are included here for further reading.

For background reading on iteration, I would recommend the Open University book "Iteration" (product code MS221 chapter B1), available second-hand from from the Open University Book Search.

For an in-depth introduction to SAGE, see the SAGE documentation website.

1.11 Acknowledgements

Thank you to Sphinx, http://sphinx.pocoo.org, used to create this document, and github, https://github.com/, used to store different versions of the document as I was writing it, and readthedocs, http://readthedocs.org/, used to build and distribute this document.

Many of the examples in this document have been inspired by examples in the excellent Open University book "Iteration" (product code MS221 chapter B1), available second-hand from from the Open University Book Search.

1.12 Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address a.coghlan@ucc.ie

1.13 License

CHAPTER 2

Carrying out geometric transformations using matrices

2.1 Using SAGE for Matrix Transformations

This chapter tells you how to use the free and Open-Source SAGE mathematics software for studying geometric transformations using matrices.

To use SAGE, you first need to start the SAGE program on your computer. You should have already installed SAGE on your computer (if not, for instructions on how to install SAGE, see the SAGE Installation Guide).

This booklet assumes that the reader has some basic knowledge of matrix transformations, and the principal focus of the booklet is not to explain matrix transformations, but rather how to study matrix transformations using SAGE.

If you are new to matrix transformation, and want to learn more about any of the concepts presented here, I would highly recommend the Open University book "Matrix transformations" (product code MS221 chapter B2), available second-hand from from the Open University Book Search.

2.2 Vectors in SAGE

We can enter vectors a and b into SAGE by typing:

```
a = vector([4,3])
print(a)
# Output:
# (4, 3)
b = vector([-5,2])
print(b)
# Output:
# (-5, 2)
```

We can then combine the vectors algebraically, for example, we can calculate 2a + (3/2)b by typing:

```
c = (2*a) + ((3/2)*b)
print(c)
# Output:
# (1/2, 9)
```

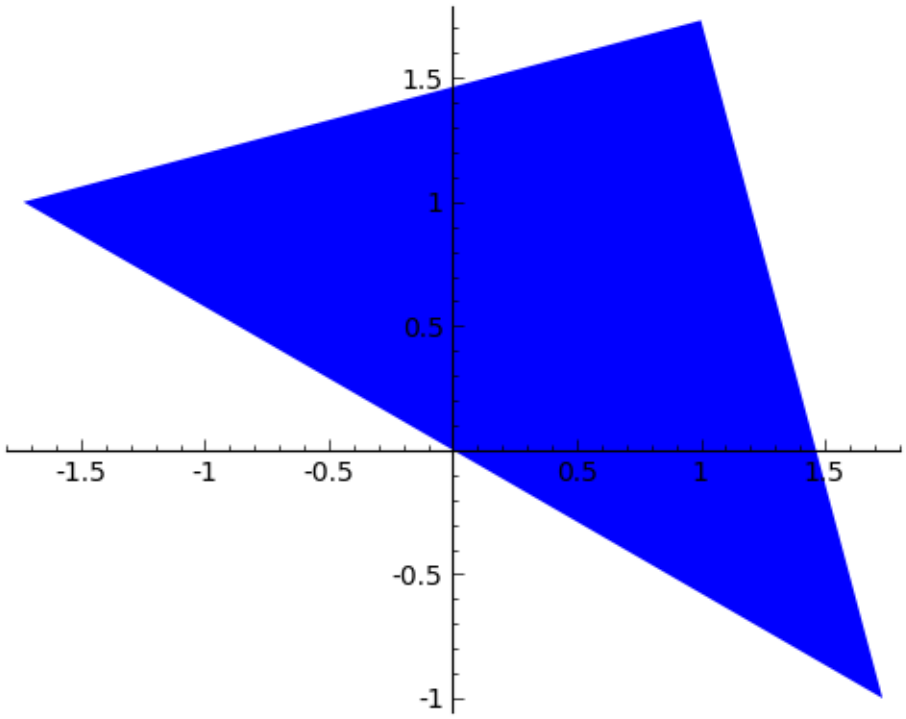
If P is a point in the plane, then the position vector p from the origin (0,0) to point P is called the position vector of P (with respect to the origin).

For example, if we have a triangle with vertices at points (1, sqrt(3)), (sqrt(3), -1), and (-sqrt(3),1), then we can represent the triangle by the three position vectors p1, p2 and p3 representing these three vectices:

```
p1 = vector([1, sqrt(3)])
p2 = vector([sqrt(3), -1])
p3 = vector([-sqrt(3), 1])
```

If we want to make a plot of a polygon (a triangle here) defined by certain position vectors, we can type in SAGE:

```
mypoints = list([p1,p2,p3])
P = polygon(mypoints)
P.set_aspect_ratio(1)
show(P)
```



Note that the set_aspect_ratio() command ensures that the scale on the x-axis and y-axis of the plot are the same.

2.3 Translation of a polygon

We can carry out a translation of a polygon, by addition of a vector a = (p, q). For example, the vector a = (2, 3) will move a shape two units to the right and three units up.

Let's write a function to apply a translation to a polygon:

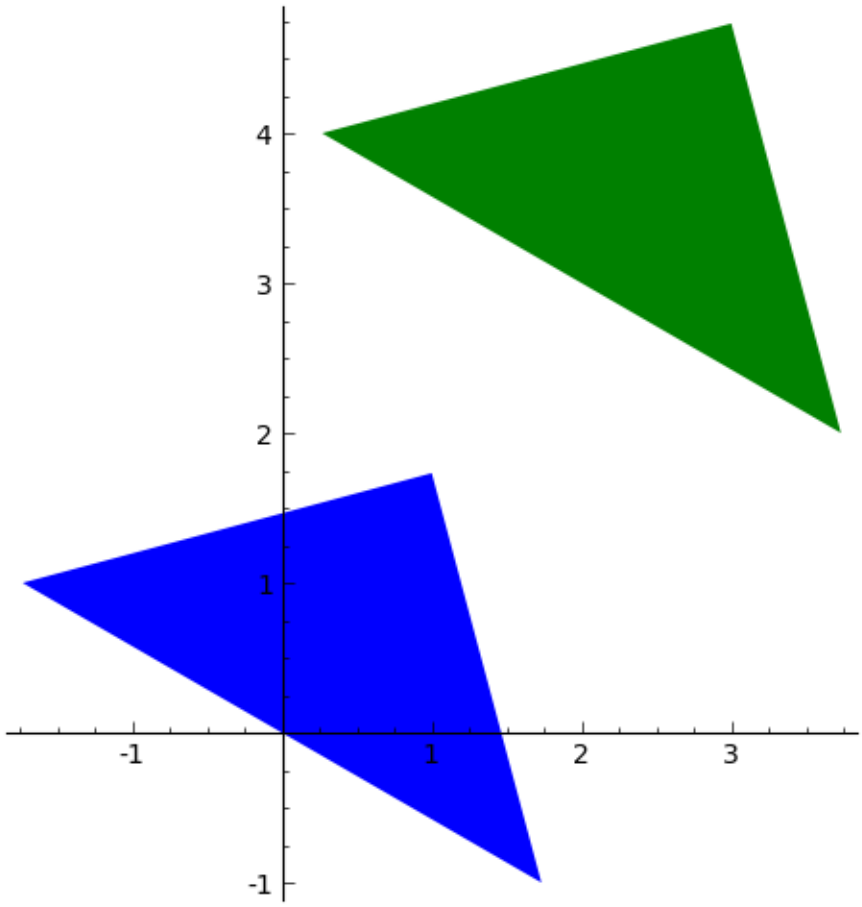
```
def translate_polygon(mypoints, a):
    mypoints2 = []
    for mypoint in mypoints:
        mypoint2 = mypoint + a
        mypoints2.append(mypoint2)
    return(mypoints2)
```

We can then move the triangle defined above two units to the right and three units upwards by applying the transformation described by vector a:

```
a = vector([2, 3])
mypoints2 = translate_polygon(mypoints, a)
```

Let's plot the original triangle (in blue) and the translated triangle (in green):

```
P = polygon(mypoints) + polygon(mypoints2,color="green")
P.set_aspect_ratio(1)
show(P)
```



The original triangle is shown in blue here, and the transformed one in green.

2.4 Matrices in SAGE

We can enter a matrix A into SAGE by typing:

```
A = matrix([[2,1],[3,2]])
print(A)
# Output:
# [2 1]
# [3 2]
```

We can then do nice things like multiplying a vector by a matrix:

```
a = vector([2, 3])
A = matrix([[2,1],[3,2]])
a * A
# (13, 8)
```

2.5 Rotation of a polygon

A rotation of a polygon, through an angle theta (radians) anticlockwise, can be obtained by multiplying the matrix:

$$\begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix}$$

by each point defining the polygon (for example, by each of the three points defining the vertices of a triangle).

This means that we can define a function to perform such a rotation of a polygon:

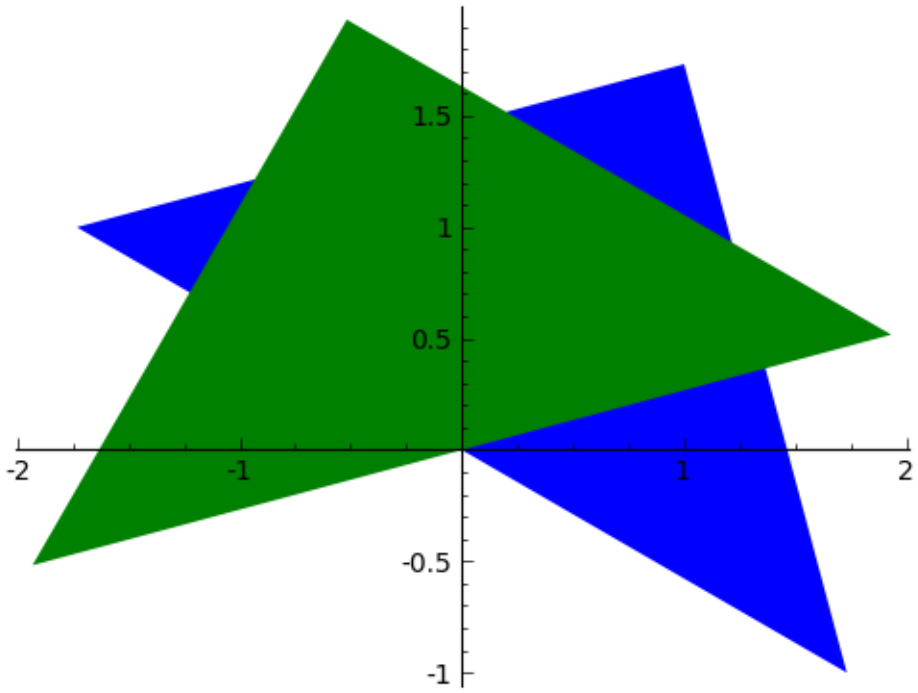
```
def rotate_polygon(mypoints, theta):
   A = matrix([[cos(theta),-(sin(theta))],[sin(theta),cos(theta)]])
   mypoints2 = []
   for mypoint in mypoints:
      mypoint2 = A* mypoint
      mypoints2.append(mypoint2)
   return(mypoints2)
```

Let's try rotating our triangle above by pi/4 radians (45 degrees) anticlockwise:

```
mypoints3 = rotate_polygon(mypoints, pi/4)
```

Let's plot the original triangle (in blue) and the rotated triangle (in green):

```
P = polygon(mypoints) + polygon(mypoints3,color="green")
P.set_aspect_ratio(1)
show(P)
```



2.6 Reflection of a polygon

A reflection of a polygon in a line through the origin that makes an angle theta measured anticlockwise from the positive x-axis, can be achieved by multiplying the matrix:

$$\begin{pmatrix} cos(2\theta) & sin(2\theta) \\ sin(2\theta) & -cos(2\theta) \end{pmatrix}$$

by each of the points that define the polygon (eg. by each of the three vertices of a triangle).

Aha! That means that we can define a function to carry out a reflection of a polygon:

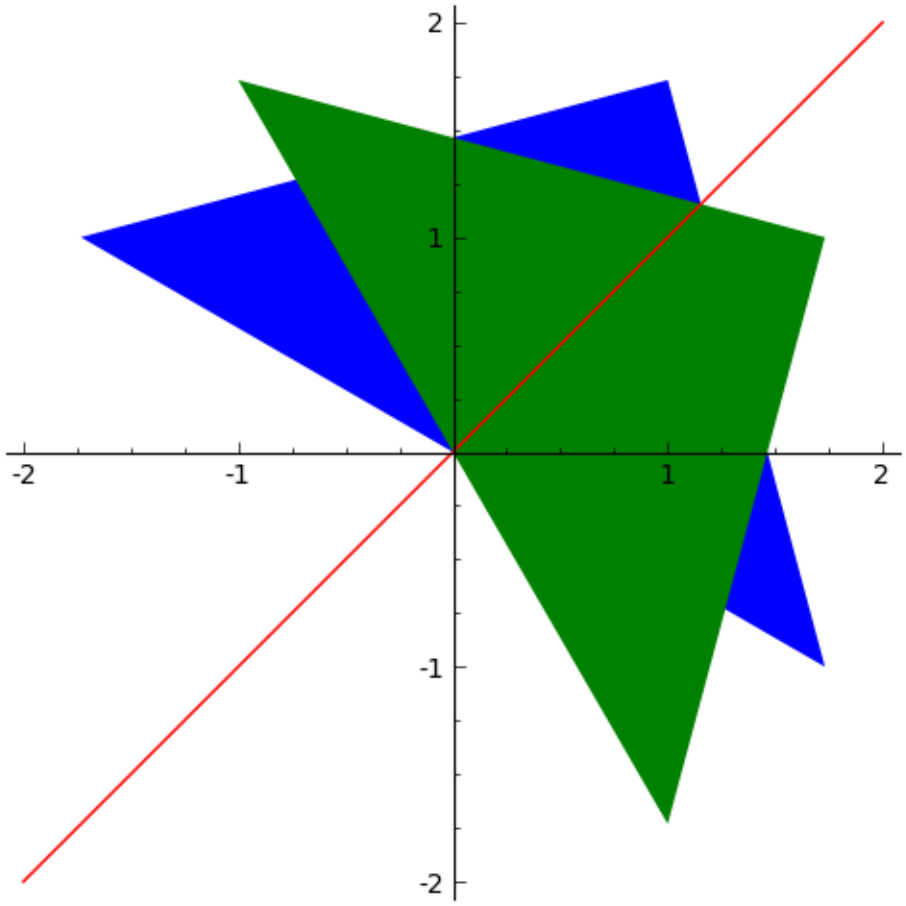
```
def reflect_polygon(mypoints, theta):
    A = matrix([[cos(2*theta),sin(2*theta)],[sin(2*theta),-(cos(2*theta))]])
    mypoints2 = []
    for mypoint in mypoints:
        mypoint2 = A* mypoint
        mypoints2.append(mypoint2)
    return(mypoints2)
```

For example, let's reflect the triangle with vertices at (1, sqrt(3)), (sqrt(3), -1), and (-sqrt(3),1), through a line that makes an angle of pi/4 radians (45 degrees) with respect to the positive x-axis:

```
mypoints4 = reflect_polygon(mypoints, pi/4)
```

Now let's plot the original triangle (in blue) and the transformed triangle (in green), with the line that the triangle was reflected through (in red):

```
P = polygon(mypoints) + polygon(mypoints4,color="green") + plot(x, (x, -2, 2), color="red")
P.set_aspect_ratio(1)
show(P)
```



2.7 Scaling of a polygon

A scaling of a polygon by factor a parallel to the x-axis, and by factor b parallel to the y-axis, can be achieved by multiplication of the matrix:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

by each of the points that define the polygon (for example, by each of the vertices of a triangle).

Let's define a SAGE function to scale a polygon like this:

```
def scale_polygon(mypoints, a, b):
    A = matrix([[a,0],[0,b]])
    mypoints2 = []
    for mypoint in mypoints:
        mypoint2 = A* mypoint
```

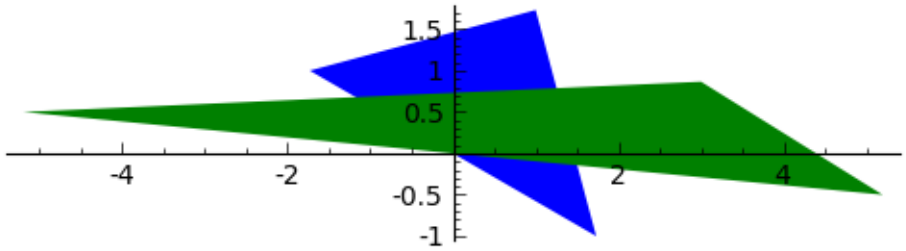
```
        mypoints2.append(mypoint2)
    return(mypoints2)
```

For example, we can scale our triangle with vertices at (1, sqrt(3)), (sqrt(3), -1), and (-sqrt(3),1), by a factor 3 parallel to the x-axis, and by a factor 0.5 parallel to the y-axis by typing:

```
mypoints5 = scale_polygon(mypoints, 3, 0.5)
```

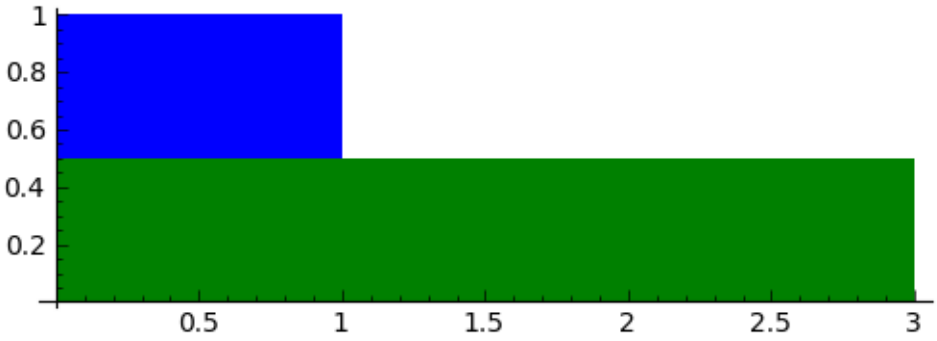
We can then plot the original triangle (in blue), and the scaled triangle (in green):

```
P = polygon(mypoints) + polygon(mypoints5,color="green")
P.set_aspect_ratio(1)
show(P)
```



Similarly, we can take the unit square, which has corners at (0,0), (1,0), (1,1) and (0,1), and apply the scaling by factor 3 parallel to the x-axis and factor 0.5 parallel to the y-axis:

```
p4 = vector([0,0])
p5 = vector([1,0])
p6 = vector([1,1])
p7 = vector([0,1])
mysquare = list([p4,p5,p6,p7])
mysquarescaled = scale_polygon(mysquare, 3, 0.5)
P = polygon(mysquare) + polygon(mysquarescaled,color="green")
P.set_aspect_ratio(1)
show(P)
```



We see that the transformed polygon has height 0.5, and width 3.0, as expected.

2.8 Shears parallel to a line

As well as translations, rotations, reflections and scalings, another type of geometric transformation is a shear parallel to a line. This is a transfomration that shifts each point parallel to a line, through a distance that is proportional to the perpendicular distance of the point from the line.

A shear parallel to the x-axis can be achieved by multiplying the matrix:

$$\begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}$$

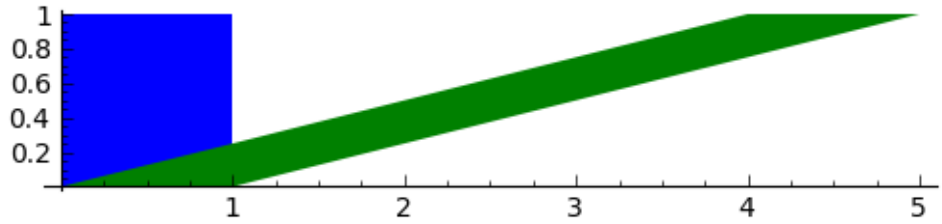
by each of the points of a polygon (eg. each of the vertices of a triangle). This is called an x-shear with factor a.

Let's define a function to carry out an x-shear with factor a:

```
def xshear_polygon(mypoints, a):
    A = matrix([[1,a],[0,1]])
    mypoints2 = []
    for mypoint in mypoints:
        mypoint2 = A* mypoint
        mypoints2.append(mypoint2)
    return(mypoints2)
```

Now let's see what happens when we apply an x-shear with factor 4 to the unit square:

```
mysquarexshear = xshear_polygon(mysquare, 4)
P = polygon(mysquare) + polygon(mysquarexshear,color="green")
P.set_aspect_ratio(1)
show(P)
```



Wow! Each point on the polygon has been moved to the right, parallel to the x-axis, through a distance that is proportional to the distance of the point from the x-axis. As a result, the two points (0,1) and (1,1) are moved far to the right, while the points (0,0) and (1,0) stay at the same place (since they are on the x-axis).

A y-shear with factor a is a similar idea: each point is moved parallel to the y-axis, by a distance that is proportion to the distance of the point from the y-axis. We achieve a y-shear with factor a by multiplying the matrix:

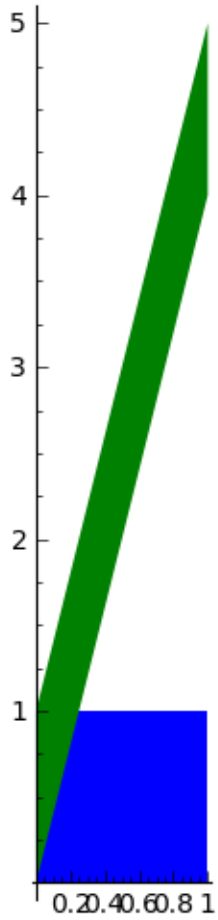
$$\begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix}$$

by each of the points of the polygon to which we want to apply the shear.

Let's define a function to apply a y-shear with factor a, and then apply a y-shear with factor 4 to the unit square:

```
def yshear_polygon(mypoints, a):
    A = matrix([[1,0],[a,1]])
    mypoints2 = []
    for mypoint in mypoints:
        mypoint2 = A* mypoint
        mypoints2.append(mypoint2)
    return(mypoints2)
```

```
mysquareyshear = yshear_polygon(mysquare, 4)
P = polygon(mysquare) + polygon(mysquareyshear,color="green")
P.set_aspect_ratio(1)
show(P)
```



2.9 Scaling of areas by geometric transformations

We saw above that a rotation of a polygon can be achieved by multiplying the matrix

$$\begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix}$$

by the points that define the polygon.

Similarly, a reflection of a polygon can be achieved by multiplying the matrix

$$\begin{pmatrix} cos(2\theta) & sin(2\theta) \\ sin(2\theta) & -cos(2\theta) \end{pmatrix}$$

by the points that define the polygon.

A scaling can be achieved by multiplying the matrix

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

by the points that define the polygon.

An x-shear and a y-shear are also achieved by multiplying certain matrices by the points that define the polygon.

Thus, all of these geometric transformations are achieved by multiplying some matrix:

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

by the points that define the polygon. Transformations that can be described by such a 2x2 matrix (reflections, rotations, scalings, shears) are called "linear transformations".

A useful thing to know is that the transformation will scale the area of the polygon by a factor equal to the determinant of matrix A. Furthermore, the orientation of the polygon will be preserved if the determinant of A is positive, and the orientation will be reversed if the determinant is negative.

This means that we can define a useful function that will take a matrix A representing a geometric transformation as its input, and will tell you: (i) what sort of transformation it is (reflection/rotation/scaling/x-shear/y-shear), (ii) by what factor the area of the polygon will be scaled, and (iii) whether the orientation will be preserved or not:

```
def classify_transformation(A):
    arccos00 = round(arccos(A[0,0]),6)
    arcsin00 = round(arcsin(A[1,0]),6)
    # check if it is an x-shear:
    if (A[0,0] == 1 and A[1,0] == 0 and A[1,1] == 1):
        a = A[0,1]
        print 'x-shear with factor', a
    # check if it is a y-shear:
    elif (A[0,0] == 1 and A[0,1] == 0 and A[1,1] == 1):
        a = A[1,0]
        print 'y-shear with factor', a
    # check if it is a scaling with factors a and b:
    elif (A[0,1] == 0 and A[1,0] == 0 and A[0,0] != A[1,1]):
        a = A[0,0]
        b = A[1,1]
        print 'scaling with factors', a, 'and', b
    # check if it is a rotation anticlockwise by theta radians:
    elif (A[0,0] == A[1,1] and A[0,1] == -A[1,0] and arccos00 == arcsin00):
        theta = arccos00
        fractionofpi = round(theta/pi,6)
```

```
        print 'anticlockwise rotation by angle', fractionofpi, '*pi radians'
    # check if it is a reflection through a line through the origin at theta radians
    # to the positive x-axis:
    elif (A[0,0] == -A[1,1] and A[0,1] == A[1,0] and arccos00 == arcsin00):
        theta = arccos00/2.0
        fractionofpi = round(theta/pi,6)
        print 'reflection through a line through the origin at', fractionofpi, '*pi radians'
    # find the determinant:
    detA = det(A)
    if (detA > 0):
        orientation = 'preserved'
    else:
        orientation = 'not preserved'
    print 'Areas scaled by', detA, ', orientation', orientation
```

Let's try it out:

```
A = matrix([[1,3],[0,1]])
classify_transformation(A)
# Output:
# x-shear with factor 3
# Areas scaled by 1 , orientation preserved
A = matrix([[0.5,-sqrt(3)/2],[sqrt(3)/2,0.5]])
classify_transformation(A)
# Output:
# anticlockwise rotation by angle 0.333333 *pi radians
# Areas scaled by 1.00000000000000 , orientation preserved
A = matrix([[0.5,sqrt(3)/2],[sqrt(3)/2,-0.5]])
classify_transformation(A)
# Output:
# reflection through a line through the origin at 0.166667 *pi radians
# Areas scaled by -1.00000000000000 , orientation not preserved
A = matrix([[3,0],[0,2]])
classify_transformation(A)
# Output:
# scaling with factors 3 and 2
# Areas scaled by 6 , orientation preserved
```

2.10 Links and Further Reading

Some links are included here for further reading.

For background reading on matrix transformations, I would recommend the Open University book "Matrix transformations" (product code MS221 chapter B2), available second-hand from from the Open University Book Search.

For an in-depth introduction to SAGE, see the SAGE documentation website.

2.11 Acknowledgements

Thank you to Sphinx, http://sphinx.pocoo.org, used to create this document, and github, https://github.com/, used to store different versions of the document as I was writing it, and readthedocs, http://readthedocs.org/, used to build and distribute this document.

Thank you to Roger Cortesi for Roger's Online Equation Editor, which I used for making some of the images of equations and matrices.

Many of the examples in this document have been inspired by examples in the excellent Open University book "Matrix transformations" (product code MS221 chapter B2), available second-hand from from the Open University Book Search.

2.12 Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address a.coghlan@ucc.ie

2.13 License

CHAPTER 3

Complex numbers

3.1 Using SAGE for complex numbers

This chapter tells you how to use the free and Open-Source SAGE mathematics software for studying complex numbers.

To use SAGE, you first need to start the SAGE program on your computer. You should have already installed SAGE on your computer (if not, for instructions on how to install SAGE, see the SAGE Installation Guide).

This booklet assumes that the reader has some basic knowledge of complex numbers, and the principal focus of the booklet is not to explain complex numbers, but rather how to study complex numbers using SAGE.

If you are new to complex numbers, and want to learn more about any of the concepts presented here, I would highly recommend the Open University book "Complex numbers" (product code MS221 chapter D1), available second-hand from from the Open University Book Search.

3.2 Adding, subtracting, multiplying and dividing complex numbers

Since Pythagoras, we know that .

In SAGE, we can use the symbol i in complex numbers, where:

and a typical complex number z has the form:

where a and b are real numbers.

3.3 Links and Further Reading

Some links are included here for further reading.

For background reading on complex numbers, I would recommend the Open University book "Complex numbers" (product code MS221 chapter D1), available second-hand from from the Open University Book Search.

For an in-depth introduction to SAGE, see the SAGE documentation website.

3.4 Acknowledgements

Thank you to Sphinx, http://sphinx.pocoo.org, used to create this document, and github, https://github.com/, used to store different versions of the document as I was writing it, and readthedocs, http://readthedocs.org/, used to build and distribute this document.

Many of the examples in this document have been inspired by examples in the excellent Open University book "Complex numbers" (product code MS221 chapter D1), available second-hand from from the Open University Book Search.

3.5 Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address a.coghlan@ucc.ie

3.6 License

CHAPTER 4

Acknowledgements

Thank you to Noel O'Boyle for helping in using Sphinx, http://sphinx.pocoo.org, to create this document, and github, https://github.com/, to store different versions of the document as I was writing it, and readthedocs, http://readthedocs.org/, to build and distribute this document.

CHAPTER 5

Contact

I will be grateful if you will send me (Avril Coghlan) corrections or suggestions for improvements to my email address a.coghlan@ucc.ie

CHAPTER 6

License