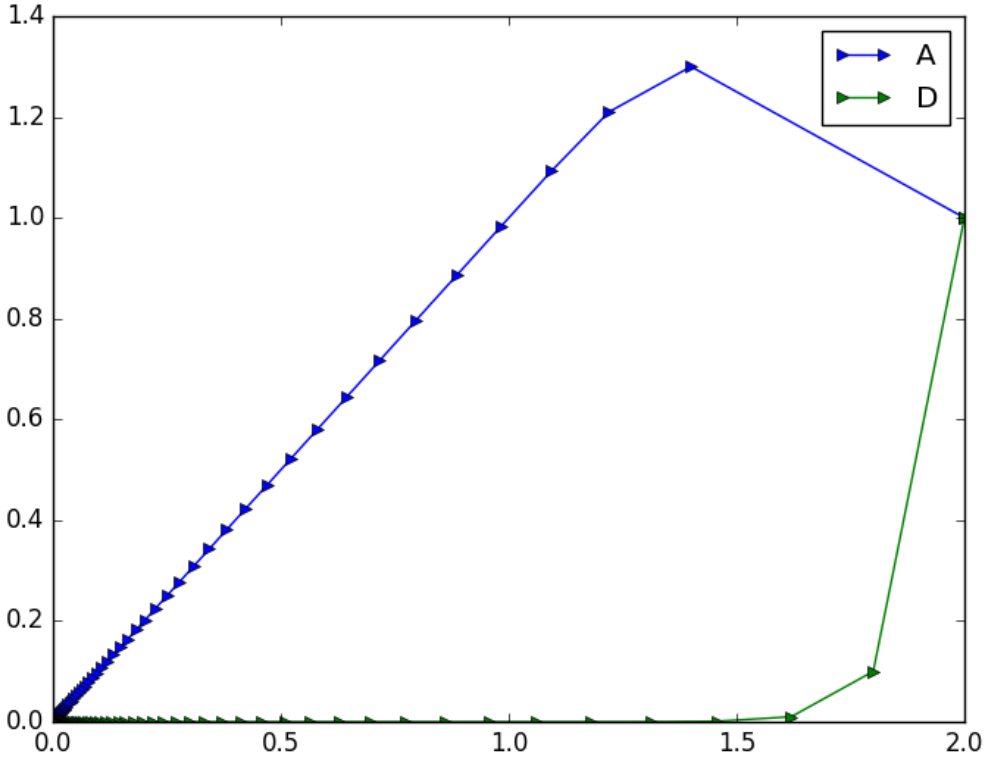
```
import numpy as np

Arr = lambda txt: np.array([[float(x.strip()) for x in y.split() if x.strip() !=
 ''] for y in txt.split(';') if y.split() != ''])

"""
For those of who you have never seen a list comprehension, let alone a nested list comprehension.

def Arr(txt):
   my_array = []
   for y in txt.split(';'):
      my_row = []
      for x in y.split():
         maybe_val = x.strip()
         if maybe_val != '':
            my_row.append(float(maybe_val))
      my_array.append(my_row)
   return numpy.array(my_array)
"""

"""

quick note on syntax:

F = lambda x: x**2

can be read pedantically as:

the variable F gets a function of one variable, returning the variable squared.

lambda syntax is a huge saver when working with simple mathematical functions, letting you trivially parameterise any thing as a function of one or more variables.

"""

# training wheels for everyone!

# could be written 'from np.linalg import det as determinant' but binding as a f
unction is a little more flexible and in my mind easier to parse
determinant = lambda A: np.linalg.det(A)

# returns a tuple containing a list of the scalar eigenvalues and then a list of
 the eigenvectors
eigen_vs = lambda A: np.linalg.eig(A)

# inverts an invertable array
inverse = lambda A: np.linalg.inv(A)

# solves a system of equations of the form Ax=b for unknowns x given a square no
n-singular matrix A and a vector of coefficients b
solve = lambda A, b: np.linalg.solve(A, b)

mul = lambda a, b: np.dot(a,b)

# rotation matrix we've used so often, defined for your future reference as 'R'
- no way we suffer a namespace collision.
R = lambda theta: np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.
cos(theta)]])
```

```
# import trainingwheels
from lin1 import *

# Question 1 Part A


V = Arr('2 1 4 0; 2 3 4 0; 2 3 4 1; 0 0 2 0; 0 0 0 1; 1 2 1 2')
# accessing "T" returns the transposed matrix
V.T
### array([[ 2.,  2.,  2.,  0.,  0.,  1.],
###        [ 1.,  3.,  3.,  0.,  0.,  2.],
###        [ 4.,  4.,  4.,  2.,  0.,  1.],
###        [ 0.,  0.,  1.,  0.,  1.,  2.]])

# solve our homogenous solution for the validated trivial case
solve(V[0:4].T, [0]*4)
### array([ 0.,  0.,  0.,  0.])
# oh look, it's all zeros

solve(V[0:4].T, [1]*4)
### array([ 0.25, -0.75,  1.  , -0.5 ])
# messing around, this one didn't work, for visually apparent reasons
solve(V[1:5].T, [1]*4)
### Traceback (most recent call last):
###   File "<input>", line 1, in <module>
###     solve(V[1:5].T, [1]*4)
###   File "lin1.py", line 46, in <lambda>
###     solve = lambda A, b: np.linalg.solve(A, b)
### LinAlgError: Singular matrix
solve(V[2:6].T, [1]*4)
### array([ 1., -1.,  2., -1.])
solve(Arr('2 1 4 0; 0 0 2 0; 0 0 0 1; 1 2 1 2').T, [1]*4)
### array([ 0.33333333, -0.33333333,  0.33333333,  0.33333333])

solve(Arr('2 1 4 0; 2 3 4 0; 0 0 2 0; 0 0 0 1').T, [0]*4)
### array([ 0.,  0.,  0.,  0.])

"""
ANSWERS

i) no valid bases specified as none constrain in the appropriate number of dimensions (4)
ii) Only the first combination, as discussed below, is a valid span
iii)

so, looking for compatible combinations of vectors V1..6, I visually identified V1..4, V3..6, and a m

I skipped the step of solving for the homogenous solution here, using my formulated matrix and an alg
   provide the directly extracted the particular solutions Cp for all relevant As

besides the provided [1/4, −3/4, 1, −1/2], we also found [1, −1, 2, −1] and [1/3, −1/3, 1/3, 1/3]

iv) we're looking for the set of points where the system of equations specified by V1, V2, V3, and V4 intersect − the co
mbination of values for a vector 'a b c d' that's dotted with our matrix A to satisfy the conditions A*v=0 − the only set
of values the solver found for v were [0, 0, 0, 0]
"""
```


A
D
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.5
1.0
1.5
2.0

```
from lin1 import *
D = Arr('0.9 0 ; 0 0.1')

ANSWERS_2a = """
```

operates by shrinking by 9/10ths and 1/10th at right angles to eachother, as the non−zero values lie along the diagonal

there're two axes at play, so two linearly independent eigenvectors, each with their unique eigenvalues.

as we have zeros where our values don't lie along the identity matrix, (basis = I?), it's pretty easy to read the values dir ectly off the matrix

A*v=l*v
A*v−l*v=0
A*v−l*I*v=0
(A−l*I)*v=0

only sensible when l != 0

scalar multiplication of l across the identity matrix

elementwise subtraction of l*I from the original matrix

for a matrix 'a b; c d', it's pretty easy to see how

'a−l b; c d−l'

becomes:

l**2−(a+d)l+(ad−bc)=0

for our specific case:

'0.9−l 0; 0 0.1−l' = '0 0; 0 0'

(just noticed why we multiply by I here)

and the only values for our eigenvalues here are the values along our diagonal!

we can check our math with the expansion provided

(l**2−(0.9+1.0)*l+(0.9*0.1−0*0) = 0

x**2 − x + 0.09 = 0

gives x=0.9,0.1

makes perfect sense!

```
"""


eigen_vs(D)
### (array([ 0.9,  0.1]), array([[ 1.,  0.],
###         [ 0.,  1.]]))
###


# Our answers were correct!
```

```
from lin1 import *
import itertools
A = Arr('0.5 0.4; 0.4 0.5')
mul(A,Arr('2;1'))
### array([[ 1.4],
###        [ 1.3]])

def apply_matrix(initial_conditions, matrix):
    val = initial_conditions
    while True:
        yield val
        val = mul(matrix, val)

def get_first_N(N, initial_conditions, matrix):
    return [x[1] for x in itertools.takewhile(lambda x: x[0] < N,
        enumerate(apply_matrix(initial_conditions, matrix)))]


x0 = Arr('2;1')

import matplotlib
from matplotlib import pyplot
xs = get_first_N(100, x0, A)
plot_A = pyplot.plot([x[0] for x in xs], [x[1] for x in xs], '-*', label='A')
xs = get_first_N(100, x0, Arr('0.9 0; 0 0.1'))
plot_D = pyplot.plot([x[0] for x in xs], [x[1] for x in xs], '-o', label='D')
pyplot.legend()
pyplot.show()

ANSWERS_2b == """

let's call lambda l for awhile...

i) (l**2-(0.5+0.5)*l+(0.25-0.16)) = 0

- this formulation looks quite familiar... it reduces to the expansion from part A!

spiffy! looks familiar! same eigenvalues!

and looking at these eigenvalues, no matter what our starting point, we'll to trend to zero...

so our final solution is (0,0)

"""

# closed form for the difference equation
b2 = lambda n: (A**n)*[2, 1]

# as N->oo, X->0
```

D shrinks along both axes, along one much faster than the other (9x?)

R(pi/4) applies this manipulation at a 45 degree angle to the axes, so that Yn will trend towards zero via a different set of axes

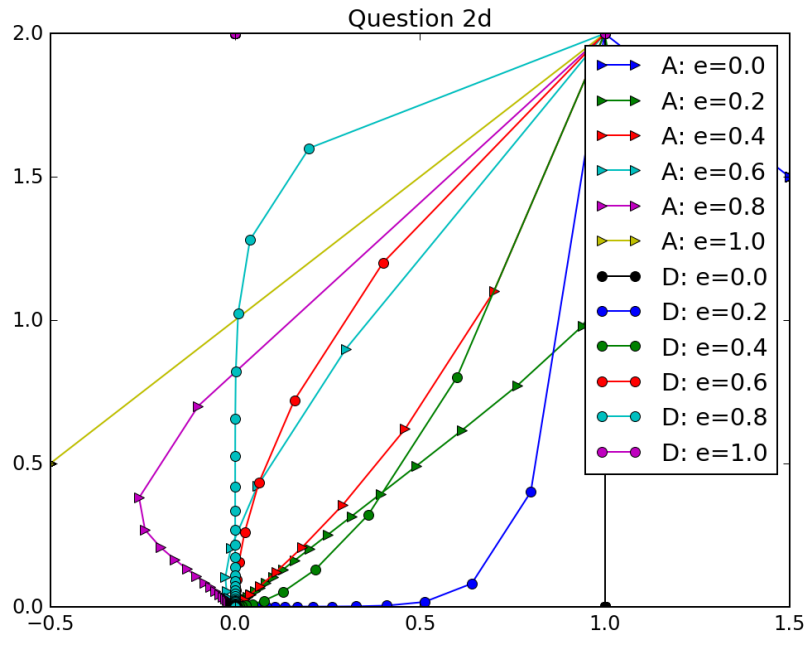
ii) fact, the difference between the axes diminishes as they approach zero

iii) fact, verified earlier

iv) fact, definitional?

v) they’re similar as they have the same eigenvalues, and thus the same general behavior, even if they differ in application of axes

vi) the transformations are the same, albiet applied across different axes – as theta approaches zero, the two parallel transformations between A and D become identical


Question 2d
A: e=0.0
A: e=0.2
A: e=0.4
A: e=0.6
A: e=0.8
A: e=1.0
D: e=0.0
D: e=0.2
D: e=0.4
D: e=0.6
D: e=0.8
D: e=1.0
2.0
1.5
1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5

```
from lin1 import *
import itertools

A = lambda e: np.array([[0.5, 0.5-e], [0.5-e, 0.5]])
D = lambda e: np.array([[1.0-e, 0], [0, e]])
def apply_matrix(initial_conditions, matrix):
    val = initial_conditions
    while True:
        yield val
        val = mul(matrix, val)

def get_first_N(N, initial_conditions, matrix):
    return [x[1] for x in itertools.takewhile(lambda x: x[0] < N, enumerate(appl
y_matrix(initial_conditions, matrix)))]


x0 = Arr('1;2')

import matplotlib
from matplotlib import pyplot

for e in np.linspace(0, 1, 6):
    xs = get_first_N(100, x0, A(e))
    pyplot.plot([x[0] for x in xs], [x[1] for x in xs], '->', label='A: e='+str(e
))

for e in np.linspace(0, 1, 6):
    xs = get_first_N(100, x0, D(e))
    pyplot.plot([x[0] for x in xs], [x[1] for x in xs], '-o', label='D: e='+str(e
))
pyplot.title("Question 2d")
pyplot.legend()
pyplot.savefig("Question2d.png", bbox_inches='tight', figsize=(9,10), dpi=144)
```

l is an eigenvalue of a n by n matrix A

i) fact – columns are linearly dependent
ii) fact – these would be the eigenvectors?
iii) fact – definitional, but to expand if you have a matrix in reduced echelon form, and you broadcast your eigenvalues across the identity matrix and subtract, you'll an un-invertable matrix with a linearly dependent row of all zeros

There’s one solution if the determinant is a real number – looking at the formulation of the determinant as the product of the diagonal in row echelon form, it’s pretty easy to see that as long as you have an upper triangular matrix, you’ll have a non-zero determinant.
If the determinant is zero, there’re either infinitely many, or zero solutions.

```
from lin1 import *
B = Arr('1 1; 2 2')
determinant(B)
### 0.0
C = Arr('1 1 1; 0 1 1; 1 0 1')
determinant(C)
### 1.0
D = Arr('2 1 1; 1 -1 1; 3 0 2')
determinant(D)
### 3.3306690738754642e-16
###

def fake_rre(b):
    assert b.shape == (3,4)
    b[2] -= b[0]*b[-1][0]/b[0][0]
    b[1] -= b[0]*b[1][0]/b[0][0]
    b[2] -= b[1]*b[2][1]/b[1][1]
    b[1] -= b[2]*b[2][2]/b[1][2]
    b[0] -= b[1]*b[0][1]/b[1][1]
    b[0] -= b[2]*b[2][2]/b[0][2]
    for i in range(b.shape[0]):
        b[i] *= {True: -1, False: 1}[b[i][i] < 0]
    return b

fake_rre(Arr('2 1 1 0; 1 -1 1 0; 3 0 2 0'))
### array([[ 2.        ,  0.        ,  1.33333333,  0.        ],
###        [-0.        ,  1.5       , -0.5       , -0.        ],
###        [ 0.        ,  0.        ,  0.        ,  0.        ]])
###

# determinant is actually zero, not foo e-16
```

i) det(R(theta)) == 1 - correct! the rotation matrix is scale-invariant, doesn't change "the area of the parallelogram" that the determinant can be thought of as representing for 2-space
ii) correct! see above! - the rotation matrix doesn't scale, so the scale factor is only determined by D
iii) correct! I think this is just a generalisation of what we've already gotten from above
iv) correct - product of the diagonal minus the product of... well, another, smaller, diagonal. the 3x3 matrixes-side-by-side trick works nicely here

If this question didn’t ask us for "general" 2x2 matricies, I’d have been able to keep using my previous tact - numpy is pretty nifty, but doesn’t provide generalisable tools for exploring symbolic math. As such, I went down a novel, spectacular rabbit hole, and brushed up on the SymPy library, a pure-Python computer algebra system that I’ve leveraged in the past for controls homework and various laplace domain circuit simulations.

So, for the purpose of illustrating these principals, we’ll use a combination of simple reason and algebra for the 2x2 and sympy for 3x3s, etc.

The identity matrix specifies a square from the axes to (1,1) - the determinant of the identity matrix is 1, and the identity matrix applies a trivial / tautological transformation, resulting in the original matrix. If the identity matrix is shifted to another quadrant, say bounding from the axes to (-1,-1), it still defines a 1x1 square, still has a determinant of one, and so on, so forth. Any variant on the theme, such as a matrix with the rows ’1 0’ and ’0 -1’ still has an area of one, but has a determinant of -1. We can use this trivial 2d example to think through det(AB) == det(A)det(B), which I work through in code for additional info.

A thought experiment with a modified 2x2 identity matrix of the rows ’1 0’ and ’0 0’ the trivial non-invertable transformation, as it serves to isolate what’s canonically considered to be the ’x’ component, turning ’y’ to 0. No coming back from that. The determinant of the matrix (tautologically) the product of the diagonal, is zero.

Does area get expanded to four or five space? Is it still called area?

The identity matrix is a pretty useful mental model for generalising this for N-space, it’s always going to have a determinant of 1, and you can play with arbitrarily large identity matrixes without any messy algebra.

```
### current bpython session - make changes and save to reevaluate session.
### lines beginning with ### will be ignored.
### To return to bpython without reevaluating make no changes to this file
### or save an empty file.
import sympy as s
import string

# l is lambda, or our eigenvalues
l = s.var('l')
def get_unique(N_vars, sets = string.lowercase):
    symbol_names = []
    while len(symbol_names) < N_vars:
        for letter in sets:
            if letter not in globals().keys() and letter not in symbol_names:
                symbol_names += [letter]
                break
        if letter == sets[-1]:
            break
    return s.var(' '.join(symbol_names))

A = s.Matrix(2,2,get_unique(4))
B = s.Matrix(2,2,get_unique(4))
A
### Matrix([
### [a, b],
### [c, d]])
B
### Matrix([
### [e, f],
### [g, h]])
(A*B).det()
### a*d*e*h - a*d*f*g - b*c*e*h + b*c*f*g
A.det()
### a*d - b*c
B.det()
### e*h - f*g
(A*B).det() == s.expand(A.det()*B.det())
### True
diagonal_test = s.diag(*get_unique(3))
n2 = s.Matrix(2,2,(a, b, c, d))
n2
### Matrix([
### [a, b],
### [c, d]])
s.solveset(n2.charpoly(l).as_expr(), l)
### {a/2 + d/2 - sqrt(a**2 - 2*a*d + 4*b*c + d**2)/2, a/2 + d/2 + sqrt(a**2 - 2*
a*d + 4*b*c + d**2)/2}
n2.eigenvals().keys()
### [a/2 + d/2 - sqrt(a**2 - 2*a*d + 4*b*c + d**2)/2, a/2 + d/2 + sqrt(a**2 - 2*
a*d + 4*b*c + d**2)/2]
s.solveset(l**2-(a+d)*l+(a*d-b*c), l)
### {a/2 + d/2 - sqrt(a**2 - 2*a*d + 4*b*c + d**2)/2, a/2 + d/2 + sqrt(a**2 - 2*
a*d + 4*b*c + d**2)/2}

ANSWER_4a = """
```

above, we can see that the eigenvalues are equivalent to the roots of the characteristic polynomial, which are equal to th e roots of the provided formulation for lambda

```
"""


n3 = s.diag(a,d,e)
n3[3] = c
n3[1] = b
n3
### Matrix([
### [a, b, 0],
### [c, d, 0],
### [0, 0, e]])
n3.charpoly(l).as_expr()
### -a*d*e + b*c*e + l**3 + l**2*(-a - d - e) + l*(a*d + a*e - b*c + d*e)
n2.charpoly(l).as_expr()
### a*d - b*c + l**2 + l*(-a - d)
c4 = s.Matrix(3, 3, (a, b, c, 0, 1, 0, 0, 0, 1))
c4.charpoly(l).as_expr(l)
### -a + l**3 + l**2*(-a - 2) + l*(2*a + 1)

ANSWER_4bc = """
```

see above

```
"""
```

a) linear combination is pretty easy to make out as:

X(n) = (1/2)**n*5*U[0]+(1/2)**(n-4)*U[4]

where U[n] is the step function at a specified index

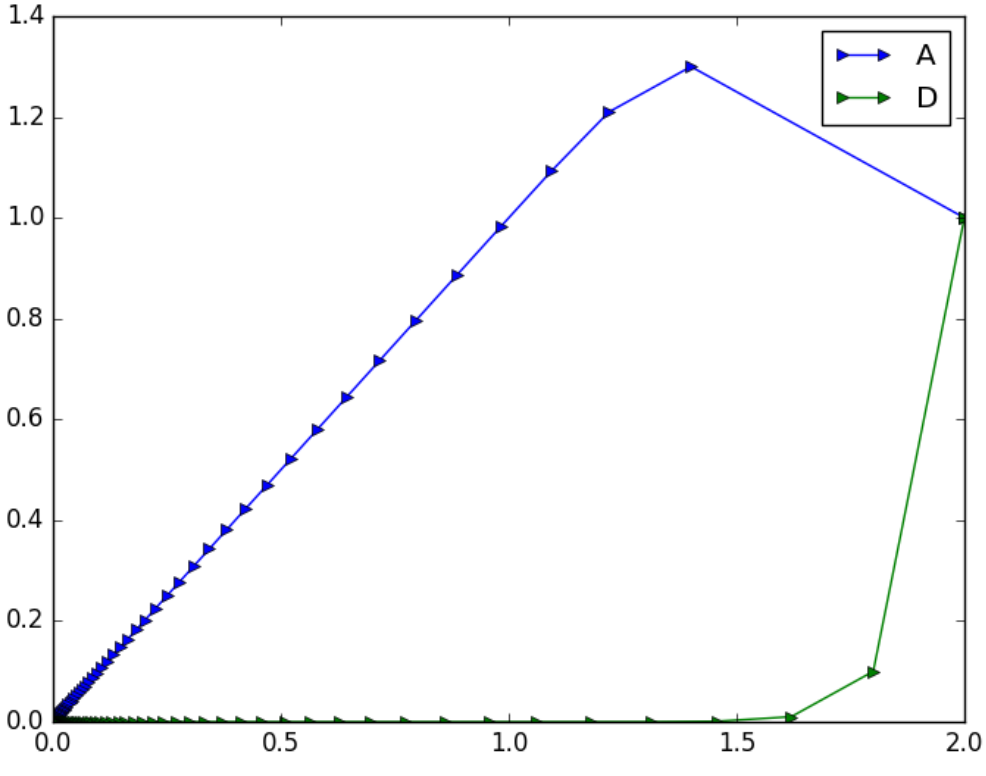
b) This one is odd. I'd like to here how Professor Hoffman would solve this...

X(n) = (1/2)**n+a**n*n doesn't look right

X(n) = (1/2+log(a))**n... isolate the exponent... idk.

```
import numpy as np

Arr = lambda txt: np.array([[float(x.strip()) for x in y.split() if x.strip() !=
 ''] for y in txt.split(';') if y.split() != '']) 

"""
For those of who you have never seen a list comprehension, let alone a nested list comprehension.

def Arr(txt):
   my_array = []
   for y in txt.split(';'):
      my_row = []
      for x in y.split():
         maybe_val = x.strip()
         if maybe_val != '':
            my_row.append(float(maybe_val))
      my_array.append(my_row)
   return numpy.array(my_array)
"""

"""

quick note on syntax:

F = lambda x: x**2

can be read pedantically as:

the variable F gets a function of one variable, returning the variable squared.

lambda syntax is a huge saver when working with simple mathematical functions, letting you trivially parameterise any thing as a function of one or more variables.

"""

# training wheels for everyone!

# could be written 'from np.linalg import det as determinant' but binding as a f
unction is a little more flexible and in my mind easier to parse
determinant = lambda A: np.linalg.det(A)

# returns a tuple containing a list of the scalar eigenvalues and then a list of
 the eigenvectors
eigen_vs = lambda A: np.linalg.eig(A)

# inverts an invertable array
inverse = lambda A: np.linalg.inv(A)

# solves a system of equations of the form Ax=b for unknowns x given a square no
n-singular matrix A and a vector of coefficients b
solve = lambda A, b: np.linalg.solve(A, b)

mul = lambda a, b: np.dot(a,b)

# rotation matrix we've used so often, defined for your future reference as 'R'
- no way we suffer a namespace collision.
R = lambda theta: np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.
cos(theta)]])
```

```
# import trainingwheels
from lin1 import *

# Question 1 Part A


V = Arr('2 1 4 0; 2 3 4 0; 2 3 4 1; 0 0 2 0; 0 0 0 1; 1 2 1 2')
# accessing "T" returns the transposed matrix
V.T
### array([[ 2.,  2.,  2.,  0.,  0.,  1.],
###        [ 1.,  3.,  3.,  0.,  0.,  2.],
###        [ 4.,  4.,  4.,  2.,  0.,  1.],
###        [ 0.,  0.,  1.,  0.,  1.,  2.]])

# solve our homogenous solution for the validated trivial case
solve(V[0:4].T, [0]*4)
### array([ 0.,  0.,  0.,  0.])
# oh look, it's all zeros

solve(V[0:4].T, [1]*4)
### array([ 0.25, -0.75,  1.  , -0.5 ])
# messing around, this one didn't work, for visually apparent reasons
solve(V[1:5].T, [1]*4)
### Traceback (most recent call last):
###   File "<input>", line 1, in <module>
###     solve(V[1:5].T, [1]*4)
###   File "lin1.py", line 46, in <lambda>
###     solve = lambda A, b: np.linalg.solve(A, b)
### LinAlgError: Singular matrix
solve(V[2:6].T, [1]*4)
### array([ 1., -1.,  2., -1.])
solve(Arr('2 1 4 0; 0 0 2 0; 0 0 0 1; 1 2 1 2').T, [1]*4)
### array([ 0.33333333, -0.33333333,  0.33333333,  0.33333333])

solve(Arr('2 1 4 0; 2 3 4 0; 0 0 2 0; 0 0 0 1').T, [0]*4)
### array([ 0.,  0.,  0.,  0.])

"""
ANSWERS

i) no valid bases specified as none constrain in the appropriate number of dimensions (4)
ii) Only the first combination, as discussed below, is a valid span
iii)

so, looking for compatible combinations of vectors V1..6, I visually identified V1..4, V3..6, and a m

I skipped the step of solving for the homogenous solution here, using my formulated matrix and an alg
   provide the directly extracted the particular solutions Cp for all relevant As

besides the provided [1/4, −3/4, 1, −1/2], we also found [1, −1, 2, −1] and [1/3, −1/3, 1/3, 1/3]

iv) we're looking for the set of points where the system of equations specified by V1, V2, V3, and V4 intersect − the co
mbination of values for a vector 'a b c d' that's dotted with our matrix A to satisfy the conditions A*v=0 − the only set
of values the solver found for v were [0, 0, 0, 0]
"""
```


A
D
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.5
1.0
1.5
2.0

```
from lin1 import *
D = Arr('0.9 0 ; 0 0.1')

ANSWERS_2a = """
```

operates by shrinking by 9/10ths and 1/10th at right angles to eachother, as the non−zero values lie along the diagonal

there're two axes at play, so two linearly independent eigenvectors, each with their unique eigenvalues.

as we have zeros where our values don't lie along the identity matrix, (basis = I?), it's pretty easy to read the values directly off the matrix

A*v=Î»*v
A*v−Î»*v=0
A*v−Î»*I*v=0
(A−Î»*I)*v=0

only sensible when Î» != 0

scalar multiplication of Î» across the identity matrix

elementwise subtraction of Î»*I from the original matrix

for a matrix 'a b; c d', it's pretty easy to see how

'a−Î» b; c d−Î»'

becomes:

Î»**2−(a+d)Î»+(ad−bc)=0

for our specific case:

'0.9−Î» 0; 0 0.1−Î»' = '0 0; 0 0'

(just noticed why we multiply by I here)

and the only values for our eigenvalues here are the values along our diagonal!

we can check our math with the expansion provided

(Î»**2−(0.9+1.0)*Î»+(0.9*0.1−0*0) = 0

x**2 − x + 0.09 = 0

gives x=0.9,0.1

makes perfect sense!

```
"""


eigen_vs(D)
### (array([ 0.9,  0.1]), array([[ 1.,  0.],
###         [ 0.,  1.]]))
###


# Our answers were correct!
```

```
from lin1 import *
import itertools
A = Arr('0.5 0.4; 0.4 0.5')
mul(A,Arr('2;1'))
### array([[ 1.4],
###        [ 1.3]])

def apply_matrix(initial_conditions, matrix):
    val = initial_conditions
    while True:
        yield val
        val = mul(matrix, val)

def get_first_N(N, initial_conditions, matrix):
    return [x[1] for x in itertools.takewhile(lambda x: x[0] < N,
        enumerate(apply_matrix(initial_conditions, matrix)))]


x0 = Arr('2;1')

import matplotlib
from matplotlib import pyplot
xs = get_first_N(100, x0, A)
plot_A = pyplot.plot([x[0] for x in xs], [x[1] for x in xs], '-*', label='A')
xs = get_first_N(100, x0, Arr('0.9 0; 0 0.1'))
plot_D = pyplot.plot([x[0] for x in xs], [x[1] for x in xs], '-o', label='D')
pyplot.legend()
pyplot.show()

ANSWERS_2b == """

let's call Î» l for awhile...

i) (l**2-(0.5+0.5)*l+(0.25-0.16)) = 0

- this formulation looks quite familiar... it reduces to the expansion from part A!

spiffy! looks familiar! same eigenvalues!

and looking at these eigenvalues, no matter what our starting point, we'll to trend to zero...

so our final solution is (0,0)

"""

# closed form for the difference equation
b2 = lambda n: (A**n)*[2, 1]

# as N->oo, X->0
```

D shrinks along both axes, along one much faster than the other (9x?)

R(pi/4) applies this manipulation at a 45 degree angle to the axes, so that Yn will trend towards zero via a different set of axes

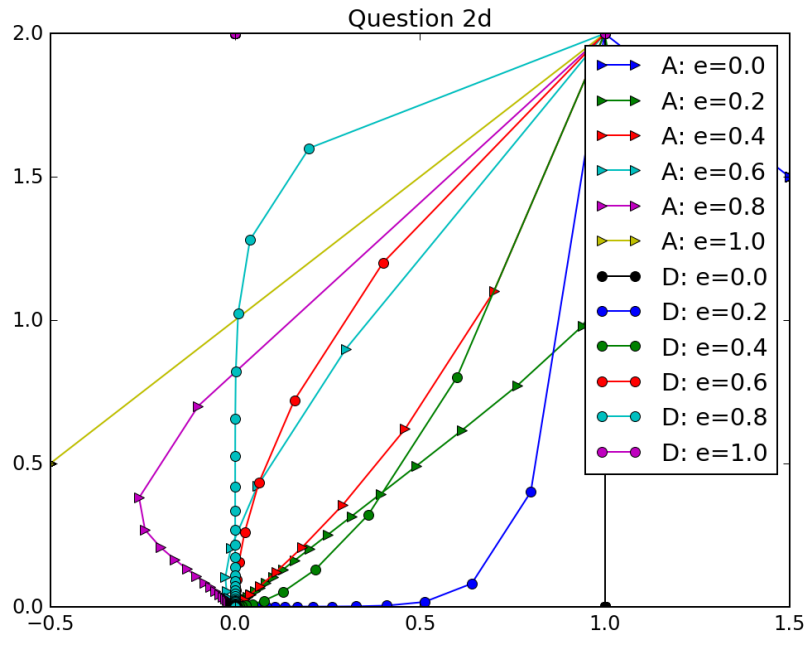
ii) fact, the difference between the axes diminishes as they approach zero

iii) fact, verified earlier

iv) fact, definitional?

v) they’re similar as they have the same eigenvalues, and thus the same general behavior, even if they differ in application of axes

vi) the transformations are the same, albiet applied across different axes – as theta approaches zero, the two parallel transformations between A and D become identical


Question 2d
A: e=0.0
A: e=0.2
A: e=0.4
A: e=0.6
A: e=0.8
A: e=1.0
D: e=0.0
D: e=0.2
D: e=0.4
D: e=0.6
D: e=0.8
D: e=1.0
2.0
1.5
1.0
0.5
0.0
−0.5
0.0
0.5
1.0
1.5

```
from lin1 import *
import itertools

A = lambda e: np.array([[0.5, 0.5-e], [0.5-e, 0.5]])
D = lambda e: np.array([[1.0-e, 0], [0, e]])
def apply_matrix(initial_conditions, matrix):
    val = initial_conditions
    while True:
        yield val
        val = mul(matrix, val)

def get_first_N(N, initial_conditions, matrix):
    return [x[1] for x in itertools.takewhile(lambda x: x[0] < N, enumerate(appl
y_matrix(initial_conditions, matrix)))]


x0 = Arr('1;2')

import matplotlib
from matplotlib import pyplot

for e in np.linspace(0, 1, 6):
    xs = get_first_N(100, x0, A(e))
    pyplot.plot([x[0] for x in xs], [x[1] for x in xs], '->', label='A: e='+str(e
))

for e in np.linspace(0, 1, 6):
    xs = get_first_N(100, x0, D(e))
    pyplot.plot([x[0] for x in xs], [x[1] for x in xs], '-o', label='D: e='+str(e
))
pyplot.title("Question 2d")
pyplot.legend()
pyplot.savefig("Question2d.png", bbox_inches='tight', figsize=(9,10), dpi=144)
```

l is an eigenvalue of a n by n matrix A

i) fact – columns are linearly dependent
ii) fact – these would be the eigenvectors?
iii) fact – definitional, but to expand if you have a matrix in reduced echelon form, and you broadcast your eigenvalues across the identity matrix and subtract, you'll an un-invertable matrix with a linearly dependent row of all zeros

There’s one solution if the determinant is a real number - looking at the formulation of the determinant as the product of the diagonal in row echelon form, it’s pretty easy to see that as long as you have an upper triangular matrix, you’ll have a non-zero determinant.
If the determinant is zero, there’re either infinitely many, or zero solutions.

```
from lin1 import *
B = Arr('1 1;2 2')
determinant(B)
### 0.0
C = Arr('1 1 1;0 1 1;1 0 1')
determinant(C)
### 1.0
D = Arr('2 1 1;1 -1 1;3 0 2')
determinant(D)
### 3.3306690738754642e-16
###

def fake_rre(b):
    assert b.shape == (3,4)
    b[2] -= b[0]*b[-1][0]/b[0][0]
    b[1] -= b[0]*b[1][0]/b[0][0]
    b[2] -= b[1]*b[2][1]/b[1][1]
    b[1] -= b[2]*b[2][2]/b[1][2]
    b[0] -= b[1]*b[0][1]/b[1][1]
    b[0] -= b[2]*b[2][2]/b[0][2]
    for i in range(b.shape[0]):
        b[i] *= {True: -1, False: 1}[b[i][i] < 0]
    return b

fake_rre(Arr('2 1 1 0;1 -1 1 0;3 0 2 0'))
### array([[ 2.        ,  0.        ,  1.33333333,  0.        ],
###        [-0.        ,  1.5       , -0.5       , -0.        ],
###        [ 0.        ,  0.        ,  0.        ,  0.        ]])
###

# determinant is actually zero, not foo e-16
```

i) det(R(theta)) == 1 - correct! the rotation matrix is scale-invariant, doesn't change "the area of the parallelogram" that the determinant can be thought of as representing for 2-space
ii) correct! see above! - the rotation matrix doesn't scale, so the scale factor is only determined by D
iii) correct! I think this is just a generalisation of what we've already gotten from above
iv) correct - product of the diagonal minus the product of... well, another, smaller, diagonal. the 3x3 matrixes-side-by-side trick works nicely here

If this question didn’t ask us for "general" 2x2 matricies, I’d have been able to keep using my previous tact - numpy is pretty nifty, but doesn’t provide generalisable tools for exploring symbolic math. As such, I went down a novel, spectacular rabbit hole, and brushed up on the SymPy library, a pure-Python computer algebra system that I’ve leveraged in the past for controls homework and various laplace domain circuit simulations.

So, for the purpose of illustrating these principals, we’ll use a combination of simple reason and algebra for the 2x2 and sympy for 3x3s, etc.

The identity matrix specifies a square from the axes to (1,1) - the determinant of the identity matrix is 1, and the identity matrix applies a trivial / tautological transformation, resulting in the original matrix. If the identity matrix is shifted to another quadrant, say bounding from the axes to (-1,-1), it still defines a 1x1 square, still has a determinant of one, and so on, so forth. Any variant on the theme, such as a matrix with the rows ’1 0’ and ’0 -1’ still has an area of one, but has a determinant of -1. We can use this trivial 2d example to think through det(AB) == det(A)det(B), which I work through in code for additional info.

A thought experiment with a modified 2x2 identity matrix of the rows ’1 0’ and ’0 0’ the trivial non-invertable transformation, as it serves to isolate what’s canonically considered to be the ’x’ component, turning ’y’ to 0. No coming back from that. The determinant of the matrix (tautologically) the product of the diagonal, is zero.

Does area get expanded to four or five space? Is it still called area?

The identity matrix is a pretty useful mental model for generalising this for N-space, it’s always going to have a determinant of 1, and you can play with arbitrarily large identity matrixes without any messy algebra.

```
### current bpython session - make changes and save to reevaluate session.
### lines beginning with ### will be ignored.
### To return to bpython without reevaluating make no changes to this file
### or save an empty file.
import sympy as s
import string

# l is lambda, or our eigenvalues
l = s.var('l')
def get_unique(N_vars, sets = string.lowercase):
    symbol_names = []
    while len(symbol_names) < N_vars:
        for letter in sets:
            if letter not in globals().keys() and letter not in symbol_names:
                symbol_names += [letter]
                break
        if letter == sets[-1]:
            break
    return s.var(' '.join(symbol_names))

A = s.Matrix(2,2,get_unique(4))
B = s.Matrix(2,2,get_unique(4))
A
### Matrix([
### [a, b],
### [c, d]])
B
### Matrix([
### [e, f],
### [g, h]])
(A*B).det()
### a*d*e*h - a*d*f*g - b*c*e*h + b*c*f*g
A.det()
### a*d - b*c
B.det()
### e*h - f*g
(A*B).det() == s.expand(A.det()*B.det())
### True
diagonal_test = s.diag(*get_unique(3))
n2 = s.Matrix(2,2,(a, b, c, d))
n2
### Matrix([
### [a, b],
### [c, d]])
s.solveset(n2.charpoly(l).as_expr(), l)
### {a/2 + d/2 - sqrt(a**2 - 2*a*d + 4*b*c + d**2)/2, a/2 + d/2 + sqrt(a**2 - 2*
a*d + 4*b*c + d**2)/2}
n2.eigenvals().keys()
### [a/2 + d/2 - sqrt(a**2 - 2*a*d + 4*b*c + d**2)/2, a/2 + d/2 + sqrt(a**2 - 2*
a*d + 4*b*c + d**2)/2]
s.solveset(l**2-(a+d)*l+(a*d-b*c), l)
### {a/2 + d/2 - sqrt(a**2 - 2*a*d + 4*b*c + d**2)/2, a/2 + d/2 + sqrt(a**2 - 2*
a*d + 4*b*c + d**2)/2}

ANSWER_4a = """
```

above, we can see that the eigenvalues are equivalent to the roots of the characteristic polynomial, which are equal to th e roots of the provided formulation for lambda

```
"""


n3 = s.diag(a,d,e)
n3[3] = c
n3[1] = b
n3
### Matrix([
### [a, b, 0],
### [c, d, 0],
### [0, 0, e]])
n3.charpoly(l).as_expr()
### -a*d*e + b*c*e + l**3 + l**2*(-a - d - e) + l*(a*d + a*e - b*c + d*e)
n2.charpoly(l).as_expr()
### a*d - b*c + l**2 + l*(-a - d)
c4 = s.Matrix(3, 3, (a, b, c, 0, 1, 0, 0, 0, 1))
c4.charpoly(l).as_expr(l)
### -a + l**3 + l**2*(-a - 2) + l*(2*a + 1)

ANSWER_4bc = """
```

see above

```
"""
```

a) linear combination is pretty easy to make out as:

X(n) = (1/2)**n*5*U[0]+(1/2)**(n-4)*U[4]

where U[n] is the step function at a specified index

b) This one is odd. I'd like to here how Professor Hoffman would solve this...

X(n) = (1/2)**n+a**n*n doesn't look right

X(n) = (1/2+log(a))**n... isolate the exponent... idk.