Re-introduction to Linear Algebra

One of the prerequisites for this course is *linear algebra*. This notebook is designed to "re-introduce" you to the topic. It guides you through some fundamental concepts using (Python) code and code-generated pictures. So, beyond reviewing math we hope it gives you yet-another opportunity to improve your Python code-reading skills.

This notebook contains a mix of pencil-and-paper exercises as well as coding "exercises." But for the coding exercises, we have provided solutions. You should read them to see that they make sense, and you might even try erasing them and seeing if you can generate the same (or similar) solutions on your own.

> *Aside.* You may be wondering why you need a linear algebra refresher at all. The answer is that linear algebra is, arguably, **the** mathematical and computational foundation for much of modern data analysis and machine learning. Modern computers are also very good at executing linear algebra operations quickly. Therefore, the more of a computation you can cast into a linear algebraic form, the easier it will be to speed up and scale up later on.
>
> In fact, there are many computations that do not even look like linear algebra at first, but can, in fact, be cast into "patterns" that do resemble it. These include database queries and searches in graphs and networks. Some of these ideas appear in Topic 2. Therefore, knowing linear algebra and knowing how it maps to code gives you a framework for doing fast computations more generally. So, even if the topic seems a bit dry at first glance, try to stick with it and you'll earn dividends in the long-run, well beyond this topic and course.

Additional resources...

If you need more reminders and examples of some of these ideas, we **highly recommend** the PDF notes from the edX course, Linear Algebra: Foundations to Frontiers ("LAFF"), which are available at http://ulaff.net/. It has many examples and explanations, which have borrowed liberally here.

Setup for plots (run but mostly ignore)

The visualizations in this notebook require a bit of setup code. You don't need to understand this code right now, but as your Python skills evolve, trying to understand how this code works will be good practice.

In [1]:
```
# Just for reference, this prints the currently version of Python
import sys
```

```
3.6.4 (default, Feb  9 2018, 18:50:54)
[GCC 5.4.1 20160904]
```

In [1]:
```
# Code for pretty-printing math notation
from IPython.display import display, Math, Latex, Markdown

def display_math(str_latex):
    display(Markdown('${}$'.format(str_latex)))

# Demo:
```

$x \in \square \implies y \in \square$ x∈S⟹y∈T

In [2]:
```
# Code for drawing diagrams involving vectors
import matplotlib.pyplot as plt
%matplotlib inline

DEF_FIGLEN = 4
DEF_FIGSIZE = (DEF_FIGLEN, DEF_FIGLEN)

def figure(figsize=DEF_FIGSIZE):
    return plt.figure(figsize=figsize)

def multiplot_figsize(plot_dims, base_figsize=DEF_FIGSIZE):
    return tuple([p*x for p, x in zip(plot_dims, base_figsize)])

def subplots(plot_dims, base_figsize=DEF_FIGSIZE, sharex='col', sharey='row', **kw_args):
    assert len(plot_dims) == 2, "Must define a 2-D plot grid."
    multiplot_size = multiplot_figsize(plot_dims, base_figsize)
    _, axes = plt.subplots(plot_dims[0], plot_dims[1],
                           figsize=multiplot_size[::-1],
                           sharex=sharex, sharey=sharey,
                           **kw_args)
    return axes

def new_blank_plot(ax=None, xlim=(-5, 5), ylim=(-5, 5), axis_color='gray', title=''):
    if ax is None:
        ax = plt.gca()
    else:
        plt.sca(ax)
    ax.axis('equal')
    if xlim is not None: ax.set_xlim(xlim[0], xlim[1])
    if ylim is not None: ax.set_ylim(ylim[0], ylim[1])
    if axis_color is not None:
        ax.axhline(color=axis_color)
        ax.axvline(color=axis_color)
    if title is not None:
        ax.set_title(title)
    return ax
```

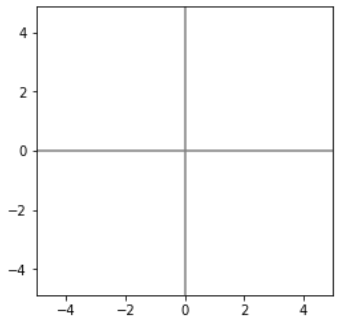
```
def draw_point2d(p, ax=None, marker='o', markersize=5, **kw_args):
    assert len(p) == 2, "Point must be 2-D."
    if ax is None: ax = plt.gca()
    ax.plot(p[0], p[1], marker=marker, markersize=markersize,
            **kw_args);

def draw_label2d(p, label, coords=False, ax=None, fontsize=14,
                 dp=(0.0, 0.1), horizontalalignment='center', verticalalignment='bottom',
                 **kw_args):
    assert len(p) == 2, "Position must be 2-D."
    if ax is None: ax = plt.gca()
    text = '{}'.format(label)
    if coords:
        text += ' = ({}, {})'.format(p[0], p[1])
    ax.text(p[0]+dp[0], p[1]+dp[1], text,
            fontsize=fontsize,
            horizontalalignment=horizontalalignment,
            verticalalignment=verticalalignment,
            **kw_args)

def draw_line2d(start, end, ax=None, width=1.0, color='black', alpha=1.0, **kw_args):
    assert len(start) == 2, "`start` must be a 2-D point."
    assert len(end) == 2, "`end` must be a 2-D point."
    if ax is None:
        ax = plt.gca()
    x = [start[0], end[0]]
    y = [start[1], end[1]]
    ax.plot(x, y, linewidth=width, color=color, alpha=alpha, **kw_args);

def draw_vector2d(v, ax=None, origin=(0, 0), width=0.15, color='black', alpha=1.0,
                  **kw_args):
    assert len(v) == 2, "Input vector must be two-dimensional."
    if ax is None:
        ax = plt.gca()
    ax.arrow(origin[0], origin[1], v[0], v[1],
             width=width,
             facecolor=color,
             edgecolor='white',
             alpha=alpha,
             length_includes_head=True,
             **kw_args);

def draw_vector2d_components(v, y_offset_sign=1, vis_offset=0.05, comp_width=1.5, **kw_args):
    assert len(v) == 2, "Vector `v` must be 2-D."
    y_offset = y_offset_sign * vis_offset
    draw_line2d((0, y_offset), (v[0], y_offset), width=comp_width, **kw_args)
    draw_line2d((v[0], y_offset), v, width=comp_width, **kw_args)

def draw_angle(theta_start, theta_end, radius=1, center=(0, 0), ax=None, **kw_args):
    from matplotlib.patches import Arc
    if ax is None: ax = plt.gca()
    arc = Arc(center, center[0]+2*radius, center[1]+2*radius,
              theta1=theta_start, theta2=theta_end,
              **kw_args)
    ax.add_patch(arc)

def draw_angle_label(theta_start, theta_end, label=None, radius=1, center=(0, 0), ax=None, **kw_args):
    from math import cos, sin, pi
    if ax is None: ax = plt.gca()
    if label is not None:
        theta_label = (theta_start + theta_end) / 2 / 360 * 2.0 * pi
        p = (center[0] + radius*cos(theta_label),
             center[1] + radius*sin(theta_label))
```

```
Ready!
```

Points in Euclidean geometry

First, recall the notion of dd-*dimensional space* that obeys Euclidean geometry. The space is an infinite set of points whose positions may be described in terms of dd coordinate axes, which are perpendicular to one another. Each axis is associated with real-values that range from $-\infty$ −∞ to ∞ ∞. Here is a snapshot of a $\mathrm{d} = 2$d=2-dimensional space with the usual x- and y-axes intersecting at the origin, $\mathrm{x} = 0, \mathrm{y} = 0$ x=0,y=0.

In [3]:
```
figure()
```



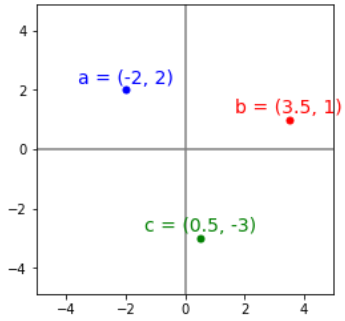
We will refer to these "standard" perpendicular axes as the *canonical axes* of a d-dimensional space.

The position of each point p in this space is a tuple of d real-valued coordinates, $p = (p_0, p_1, \ldots, p_{d-1})$. Each coordinate p_i is a real number, which in math terms we write by saying $p_i \in \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers. Each p_i measures the extent of p along the i-th axis. In 2-D, the x-coordinate of p is p_0 and the y-coordinate is p_1.

> *Note.* We are using a convention in which the axes and coordinates are numbered starting at 0, in part for consistency with how Python numbers the elements of its tuples, lists, and other collections.

Here is an example of three points, a, b, and c, in a 2-D Euclidean space. The code uses the natural data type for representing the points, namely, Python's built-in 2-tuple (i.e., pair) data type.

In [4]:
```
# Define three points
a = (-2, 2)
b = (3.5, 1)
c = (0.5, -3)

# Draw a figure containing these points
figure()
new_blank_plot()
draw_point2d(a, color='blue'); draw_label2d(a, 'a', color='blue', coords=True)
draw_point2d(b, color='red'); draw_label2d(b, 'b', color='red', coords=True)
```



Exercise. We will assume you are familiar with the basic geometry of Euclidean spaces. For example, suppose you connect the points into a triangle whose sides are $\overline{ab}$, $\overline{bc}$, and $\overline{ac}$. What are the lengths of the triangle's sides? What are its angles?

Vectors (vs. points)

In linear algebra, the first concept you need is that of a *vector*. A vector will look like a point but is, technically, a little bit different.

Definition: vectors. A d-dimensional vector is an "arrow" in d-dimensional space. It has a *length* and a *direction*. It does *not* have a position! Having said that, we will represent a vector by its length along each of the canonical axes, albeit using the following slightly different notation.

In particular, we will write a d-dimensional vector v as a *column vector*,

$$v \equiv \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-1} \end{pmatrix},$$

v≡[v0v1⋮vd−1],

where each entry v_i is the length of the vector with respect to the i-th axis. We will also refer to the entries as *elements* or *components* of v.

In our class, we are always interested in spaces in which the possible values of v_i are real numbers. Therefore, when we want to say a mathematical object v is a d-dimensional vector, we will sometimes write that using the shorthand, $v \in \mathbb{R}^d$, meaning v is an element of the set of all possible d-dimensional vectors with real-valued components.

> *Aside 1.* We usually use the term "coordinates" when referring to the components of a point. And while a vector does not have a position, making it not a point, we will nevertheless "abuse" terminology sometimes and refer to the "coordinates" of a vector when we mean

> "components" or "elements."
>
> *Aside 1.* The term "column" suggests there is a notion of a "row" vector. We'll discuss that later.

Before discussing this representation of "length" and "direction" further, let's encode it in Python.

A code representation. As we did with points, let's again use tuples to represent the elements of a vector. Below, we define a Python function, `vector()`, whose arguments are, say, dd coordinates; it returns a tuple that holds these elements. In this $d = 2$d=2 example, suppose a vector vv has a length of $v_0 = 1.0$v0=1.0along the $0$0-th coordinate (e.g., x-axis) and $v_1 = 2.0$v1=2.0 in the $1$1st coordinate (e.g., y-axis):

In [5]:
```
def vector(*elems, dim=None):
    """
    Exercise: What does this function do?
    """
    if dim is not None:
        if len(elems) > 0:
            assert dim == len(elems), "Number of supplied elements differs from the requested dimension."
        else: # No supplied elements
            elems = [0.0] * dim
    return tuple(elems)

def dim(v):
    """Returns the dimensionality of the vector `v`"""
    return len(v)

v = vector(1.0, 2.0)
d = dim(v)
```

```
v = (1.0, 2.0)    <==  2-dimensional
```

In [6]:
```
# Another example: Creates a zero-vector of dimension 3
z3 = vector(dim=3)
```

```
z3 = (0.0, 0.0, 0.0)    <== 3-dimensional
```

Aside: Pretty-printing using LaTeX. Recall the abstract mathematical notation of a vector's elements as a vertical stack. Using the standard Python `print()`renders a vector as a row-oriented tuple. However, Jupyter notebooks also support LaTeX notation for rendering mathematical formulas in a "pretty" way. This feature means we can write Python code that generates LaTeX and renders it in the notebook!

You don't need to understand too much about how this process works. However, we mention it because you will see us define helper functions to help pretty-print math throughout this notebook.

In [7]:
```
def latex_vector(v, transpose=False):
    """Returns a LaTeX string representation of a vector"""
    s = r'''\left[ \begin{matrix} '''
    sep = r'''\\''' if not transpose else r''', &'''
    s += (r' {} ').format(sep).join([str(vi) for vi in v])
    s += r''' \end{matrix}\right]'''
    return s

# Demo: Pretty-print `v` from before
print("Standard Python output:", v)
print("\n'Mathy' output:")
v_latex = latex_vector(v)
```

```
Standard Python output: (1.0, 2.0)

'Mathy' output:
```

$$v \equiv \begin{matrix} 1.0 \\ [\, 2.0 \,] \end{matrix}$$ v≡[1.02.0]

Okay, back to math...

Definition: direction of the vector. To determine a vector's direction, start at the origin, and then take a step of size v_i vi along each axis ii. We say the vector points from the origin toward the ending point. That's its direction. We'll draw a picture momentarily to make this clearer.

Definition: length of a vector. The length of the vector is the straight-line (Euclidean) distance between the origin and the endpoint, if the vector is placed at the origin. With respect to the coordinates, this distance is given by the familiar formula,

$$\sqrt{v_0^2 + v_1^2 + \cdots + v_{d-1}^2},$$

v02+v12+⋯+vd−12,

that is, the square-root of the sum of squared lengths along each axis.

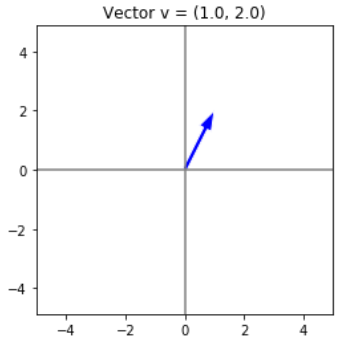
Here is a Python function to return this length.

In [8]:
```
def length(v):
    from math import sqrt
    return sqrt(sum([vi*vi for vi in v]))
```

```
The length of v = (1.0, 2.0) is about 2.23606797749979.
```

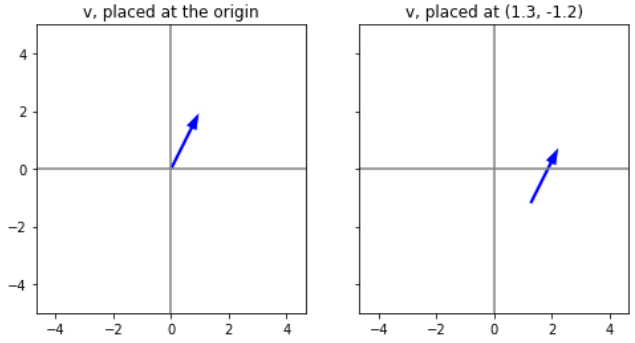
Pictures! Before things get too abstract, let's give the idea of a vector a visual footing.

In [9]:
```
figure()
new_blank_plot(title='Vector v = {}'.format(str(v)))
```



Remember: Vectors do not have a position! In the picture above, we drew the vector beginning at the origin. However, remember that a vector technically does not have a position. That is, if we do "move" it to some other point of the space, it is the *same* vector. So in the following picture, the blue vectors have the same representation, that is, identical components.

In [10]:
```
def random_vector(dim=2, v_min=-1, v_max=1):
    """Returns a random vector whose components lie in (`v_min`, `v_max`)."""
    from random import uniform
    v = vector(*[uniform(v_min, v_max) for _ in range(dim)])
    return v

def flip_signs_randomly(v):
    from random import choice
    return [choice([-1, 1])*vi for vi in v]

# Draw `v` at the origin
subfigs = subplots((1, 2))
new_blank_plot(subfigs[0], title='v, placed at the origin')
draw_vector2d(v, color='blue')

# Draw `v` somewhere else
dv = flip_signs_randomly(random_vector(dim=dim(v), v_min=1, v_max=3))
new_blank_plot(subfigs[1], title='v, placed at ({:.1f}, {:.1f})'.format(dv[0], dv[1]))
```



Definition: p**p-norms.** We defined the length using the familiar Euclidean notion of distance. However, there are actually many other kinds of distance. The pp-norm of a vector vv is one such generalized idea of a distance:

$$\|v\|_p \equiv \left(\sum_{i=0}^{d-1} |v_i|^p\right)^{\frac{1}{p}}.$$

‖v‖p≡(∑i=0d−1|vi|p)1p.

The usual Euclidean distance is the same as $p = 2$p=2, i.e., the "two-norm." There are some other commonly used norms.

- $p = 1$ p=1: The one-norm, which is the same as the "Manhattan distance." In machine learning applications, judicious use of this norm often leads to "sparse" models, that is, models where less important parameters are automatically driven to zero.
- $p = \infty$ p=∞: The infinity-norm, also known as the "max norm." It is the largest absolute entry, that is, $\|v\|_\infty = \max_i \|v_i\|$ ‖v‖∞=maxi‖vi‖.

Here is some code that implements the calculation of a norm.

In [11]:
```
def norm(v, p=2):
    assert p > 0
    from math import sqrt, inf, pow
    if p == 1: return sum([abs(vi) for vi in v])
    if p == 2: return sqrt(sum([vi*vi for vi in v]))
    if p == inf: return max([abs(vi) for vi in v])
    return pow(sum([pow(abs(vi), p) for vi in v]), 1.0/p)

def latex_norm(x, p=2):
```

```
    from math import inf
    if p == inf: p = r'\infty'
    s = r'\left\| '
    s += x
    s += r' \right\|_{}'.format(p)
    return s

import math
for p in [1, 2, math.inf]:
    v_pnorm_latex = latex_norm(v_latex, p)
```

$$\left\| \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix} \right\|_1 \approx 3.0$$ ‖[1.02.0]‖1≈3.0

$$\left\| \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix} \right\|_2 \approx 2.23606797749979$$ ‖[1.02.0]‖2≈2.23606797749979

$$\left\| \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix} \right\|_\infty \approx 2.0$$ ‖[1.02.0]‖∞≈2.0

Exercise. Convince yourself that the 1-norm, 2-norm, and ∞ ∞-norm satisfy the following properties.

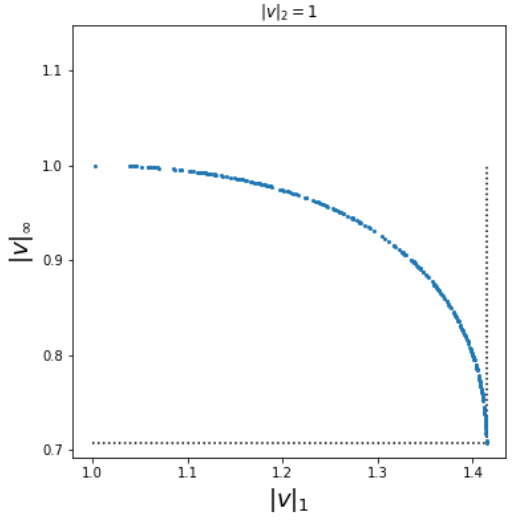
- *Triangle inequality.* $\|v + w\| \leq \|v\| + \|w\|$ ‖v+w‖≤‖v‖+‖w‖.
- *Absolute homogeneity.* Let σσ be a scalar value. Then $\|\sigma v\| = |\sigma| \cdot \|v\|$. ‖σv‖=|σ|⋅‖v‖.

Comparing norms. In the previous example, the one-norm is the largest value and the infinity-norm is the smallest. In fact, this holds in general and it is possible to show the following:

$$\|v\|_\infty \leq \|v\|_2 \leq \|v\|_1 \leq \sqrt{d}\|v\|_2 \leq d\|v\|_\infty .$$
‖v‖∞≤‖v‖2≤‖v‖1≤d‖v‖2≤d‖v‖∞.

Feel free either to prove it, or check it experimentally by running the following code.

In [12]:
```
from math import inf, sqrt

def normalize_vector(v, p=2):
    """Returns a rescaled version of the input vector `v`."""
    v_norm = norm(v, p=p)
    return vector(*[vi/v_norm for vi in v])

# Generate random points whose 2-norm equals 1. Then
# compute the 1-norm and inf-norm of these points.
norms_1 = [None] * 250
norms_inf = [None] * 250
for k in range(len(norms_1)):
    v = normalize_vector(random_vector())
    norms_1[k] = norm(v, p=1)
    norms_inf[k] = norm(v, p=inf)

figure(figsize=(6, 6))
new_blank_plot(xlim=None, ylim=None, axis_color=None, title='$\|v\|_2 = 1$')
plt.plot(norms_1, norms_inf, marker='o', markersize=2, linestyle='none')
plt.xlabel('$\|v\|_1$', fontsize=18);
plt.ylabel('$\|v\|_\infty$', fontsize=18);
plt.hlines(y=1/sqrt(2), xmin=1, xmax=sqrt(2), linestyle=':')
plt.vlines(x=sqrt(2), ymin=1/sqrt(2), ymax=1, linestyle=':')
```



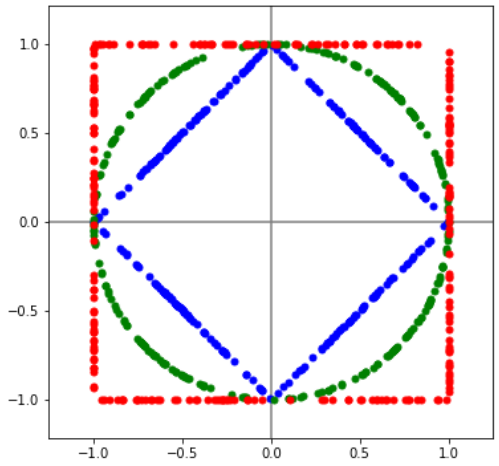
Exercise. Consider *all* the 2-D vectors whose pp-norm equals 1. Place all the vectors at the origin, and imagine their endpoints. What shapes do the endpoints sketch out, for $p = 1$ p=1, $p = 2$ p=2, and $p = \infty$ p=∞?

> *Hint.* Start by considering all the 2-D points whose two-norm, or Euclidean distance, equals 1. Convince yourself that their endpoints from the origin would all lie on a circle of radius 1. What shapes will $p = 1$ p=1 and $p = \infty$ p=∞ sketch out?

When you have an answer, check it by running the code below. It performs an experiment where, for each value of $p \in \{1, 2, \infty\}$ p∈{1,2,∞}, it generates a

In [13]:
```
from math import inf

figure(figsize=(6, 6))
new_blank_plot(xlim=(-1.25, 1.25), ylim=(-1.25, 1.25))

for p, color in zip([1, 2, inf], ['blue', 'green', 'red']):
    print("Points whose {}-norm equals 1 are shown in {}.".format(p, color))
    for _ in range(250):
        v = normalize_vector(random_vector(), p=p)
        # The `p`-norm of `v` is now equal to 1; plot `v`.
```

```
Points whose 1-norm equals 1 are shown in blue.
Points whose 2-norm equals 1 are shown in green.
Points whose inf-norm equals 1 are shown in red.
```



You should see that the norms are identical for $\begin{bmatrix} \pm 1.0 \\ 0 \end{bmatrix}$ [±1.00] and $\begin{bmatrix} 0 \\ \pm 1.0 \end{bmatrix}$ [0±1.0], and everywhere else, $\|v\|_1 < \|v\|_2 < \|v\|_\infty$ ‖v‖1<‖v‖2<‖v‖∞.

Basic operations: scaling, addition, and subtraction

The most elementary operations on vectors involve changing their lengths ("scaling" them), adding them, and subtracting them.

Let's start with scaling.

Operation: Scaling a vector. Given a vector vv, *scaling* it by a scalar value σσ simply multiplies every element of the vector by αα.

$$\sigma v = \sigma \cdot \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-1} \end{bmatrix} = \begin{bmatrix} \sigma v_0 \\ \sigma v_1 \\ \vdots \\ \sigma v_{d-1} \end{bmatrix}.$$

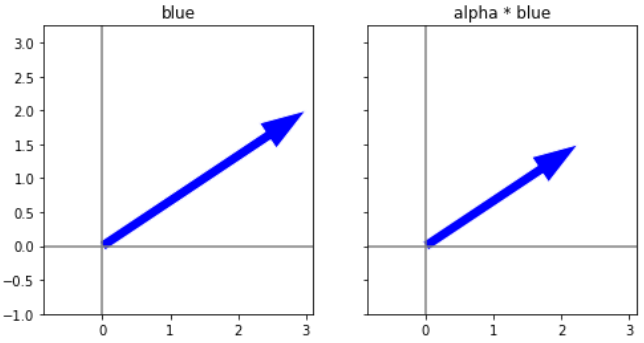
σv=σ⋅[v0v1⋮vd−1]=[σv0σv1⋮σvd−1].

Here is a picture of the scaling operation.

In [14]:
```
def scale(v, sigma):
    return tuple([sigma*vi for vi in v])

va = vector(3.0, 2.0)
sigma = 0.75
va_scaled = scale(va, sigma)

va_latex = latex_vector(va)
va_scaled_latex = latex_vector(va_scaled)
display_math(r'''(\sigma={}) {} = {}'''.format(sigma, va_latex, va_scaled_latex))
```

```
axes = subplots((1, 2))
new_blank_plot(axes[0], xlim=(-1, 3.25), ylim=(-1, 3.25), title='blue')
draw_vector2d(va, color='blue')

new_blank_plot(axes[1], xlim=(-1, 3.25), ylim=(-1, 3.25), title='alpha * blue')
```

$(\sigma = 0.75)\begin{bmatrix}3.0\\2.0\end{bmatrix} = \begin{bmatrix}2.25\\1.5\end{bmatrix}$ (σ=0.75)[3.02.0]=[2.251.5]



Operation: Vector addition. Adding two vectors v and w consists of matching and summing component-by-component, also referred to as *elementwise addition*:

$$v + w \equiv \begin{bmatrix}v_0\\\vdots\\v_{d-1}\end{bmatrix} + \begin{bmatrix}w_0\\\vdots\\w_{d-1}\end{bmatrix} = \begin{bmatrix}v_0 + w_0\\\vdots\\v_{d-1} + w_{d-1}\end{bmatrix}.$$

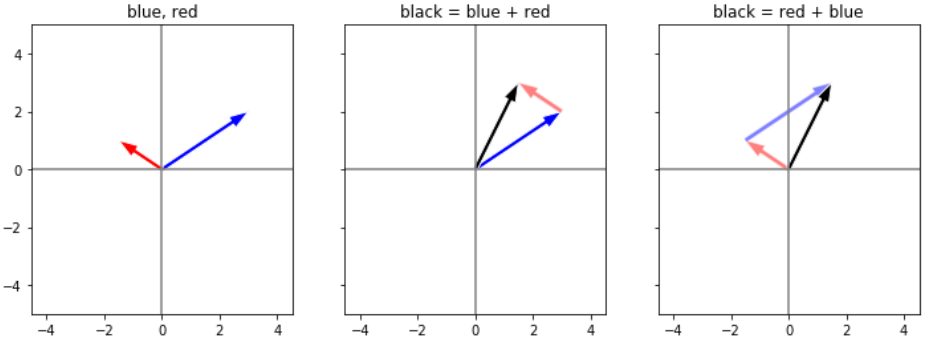
v+w≡[v0:vd−1]+[w0:wd−1]=[v0+w0:vd−1+wd−1].

Geometrically, the act of adding v and w is the same as connecting the end of v to the start of w, as illustrated by the following code and picture.

In [15]:
```
def add(v, w):
    assert len(v) == len(w), "Vectors must have the same length."
    return tuple([vi+wi for vi, wi in zip(v, w)])

vb = vector(-1.5, 1.0)
vc = add(va, vb)

vb_latex = latex_vector(vb)
vc_latex = latex_vector(vc)
```

$\begin{bmatrix}3.0\\2.0\end{bmatrix} + \begin{bmatrix}-1.5\\1.0\end{bmatrix} = \begin{bmatrix}1.5\\3.0\end{bmatrix}$ [3.02.0]+[−1.51.0]=[1.53.0]

In [16]:
```
axes = subplots((1, 3))
new_blank_plot(ax=axes[0], title='blue, red');
draw_vector2d(va, color='blue')
draw_vector2d(vb, color='red')

new_blank_plot(ax=axes[1], title='black = blue + red');
draw_vector2d(va, color='blue')
draw_vector2d(vb, origin=va, color='red', alpha=0.5)
draw_vector2d(vc)

new_blank_plot(ax=axes[2], title='black = red + blue');
draw_vector2d(vb, color='red', alpha=0.5)
draw_vector2d(va, origin=vb, color='blue', alpha=0.5)
```



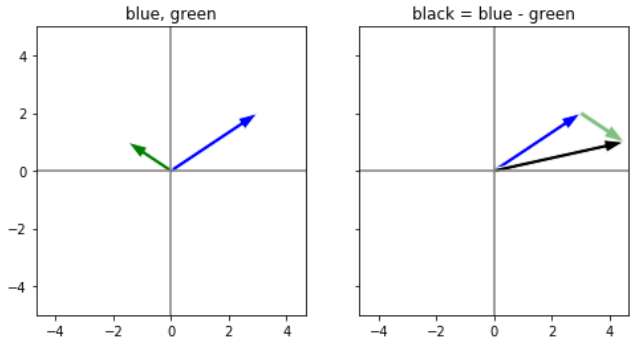
In the picture above, there are two vectors, "blue" and "red" (left subplot). Adding the red vector to the blue vector ("blue + red") is geometrically equivalent to attaching the start of the red vector to the end of the blue vector (middle subplot). Moreover, since scalar addition is symmetric ($a + b = b + a$ a+b=b+a), so, too, is vector addition (right subplot).

> *Aside.* Observe that our visualizations "exploit" the fact that vectors only have lengths and directions, not positions, so that vector addition becomes a symmetric operation.

Negation and subtraction. Subtracting two vectors is also done elementwise. Alternatively, one may view $v - w$v−w as $v + (-w)$v+(−w), that is, first scaling ww by -1 and then adding it to vv, which is what the code below implements.

In [17]:
```
def neg(v):
    return tuple([-vi for vi in v])

def sub(v, w):
    return add(v, neg(w))

vd = sub(va, vb)

vd_latex = latex_vector(vd)
display_math('{} + {} = {}'.format(va_latex, vb_latex, vd_latex))

axes = subplots((1, 2))
new_blank_plot(ax=axes[0], title='blue, green');
draw_vector2d(va, color='blue')
draw_vector2d(vb, color='green')

new_blank_plot(ax=axes[1], title='black = blue - green');
draw_vector2d(va, color='blue')
draw_vector2d(neg(vb), origin=va, color='green', alpha=0.5)
```

$$\begin{bmatrix} 3.0 \\ 2.0 \end{bmatrix} + \begin{bmatrix} -1.5 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 4.5 \\ 1.0 \end{bmatrix}$$ [3.02.0]+[−1.51.0]=[4.51.0]

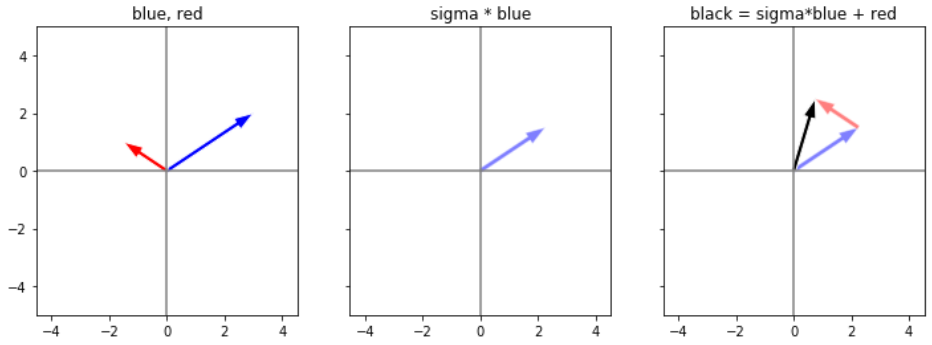


As the visualization indicates, scaling by -1 makes the vector point in the opposite direction.

Lastly, observe that scaling and addition—e.g., $\sigma v + w$σv+w—combine as expected.

In [18]:
```
ve = add(va_scaled, vb)

ve_latex = latex_vector(ve)
display_math(r'''{} {} + {} = {}'''.format(sigma, va_latex, vb_latex, ve_latex))

axes = subplots((1, 3))
new_blank_plot(axes[0], title='blue, red')
draw_vector2d(va, color='blue')
draw_vector2d(vb, color='red')

new_blank_plot(axes[1], title='sigma * blue')
draw_vector2d(va_scaled, color='blue', alpha=0.5)

new_blank_plot(axes[2], title='black = sigma*blue + red')
draw_vector2d(va_scaled, color='blue', alpha=0.5)
draw_vector2d(vb, origin=va_scaled, color='red', alpha=0.5)
```

$$0.75 \begin{bmatrix} 3.0 \\ 2.0 \end{bmatrix} + \begin{bmatrix} -1.5 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0.75 \\ 2.5 \end{bmatrix}$$ 0.75[3.02.0]+[−1.51.0]=[0.752.5]



Dot (or "inner") products

Another critically important operation on vectors is the *dot product*.

Definition. The *dot product* (or *inner product*) between two d-dimensional vectors, u and w, will be denoted by $\langle u, w \rangle$ and defined as follows:

$$\langle u, w \rangle \equiv u_0 w_0 + \cdots + u_{d-1} w_{d-1} = \sum_{i=0}^{d-1} u_i w_i .$$

That is, take u and w, compute their elementwise products, and then sum these products.

Observation. The result of a dot product is a single number, i.e., a scalar.

Here is a Python implementation, followed by an example.

In [19]:
```
def dot(u, w):
```

In [20]:
```
u = (1, 2.5)
w = (3.25, 1.75)

display_math('u = ' + latex_vector(u))
display_math('w = ' + latex_vector(w))
u_dot_w_sum_long_latex = '+'.join([r'({}\cdot{})'.format(ui, wi) for ui, wi in zip(u, w)])
```

$u = \begin{bmatrix} 1 \\ 2.5 \end{bmatrix}$ u=[12.5]

$w = \begin{bmatrix} 3.25 \\ 1.75 \end{bmatrix}$ w=[3.251.75]

$\langle u, w \rangle = (1 \cdot 3.25) + (2.5 \cdot 1.75) = 7.625$ ⟨u,w⟩=(1·3.25)+(2.5·1.75)=7.625

There is another commonly used notation for the dot product, which we will use extensively when working with matrices. It requires the concept of a *row vector*.

Definition: row vectors and (vector) transposes. Recall that we used the term *column* with vectors and drew a vector as a vertical stack. As the very term "column" suggests, there is also a concept of a *row vector*. It will become important to distinguish between row and column vectors when we discuss matrices.

In this class, the convention we will try to use is that a vector is a column vector unless otherwise specified; and when we need a "row" version of v, we will use the operation called the *transpose* to get it from the (column) version, denoted as

$$v^T \equiv [\, v_0, v_1, \ldots, v_{d-1} \,] .$$

vT≡[v0,v1,…,vd−1].

In [21]:
```
```

$v^T = [\, 0.7170893139959232, \quad 1.0 \,]$ vT=[0.7170893139959232,1.0]

Notation: vector transpose form of the dot product. Armed with the notions of both row and column vectors, here is an alternative way we will define a dot product:

$$\langle u, v \rangle \equiv u^T w \equiv [u_0, \ldots, u_{d-1}] \cdot \begin{bmatrix} w_0 \\ \vdots \\ w_{d-1} \end{bmatrix}.$$

⟨u,v⟩≡uTw ≡ [u0,…,ud−1]·[w0:wd−1].

That is, given two (column) vectors u and w, the dot product is the sum of the elementwise products between the transpose of u and v. We read "$u^T w$uTw" as "uu-transpose times v."

In [22]:
```
u_dot_w_vec_latex = latex_vector(u, transpose=True) + r' \cdot ' + latex_vector(w)
```

$\langle u, w \rangle = u^T w = [\, 1, \quad 2.5 \,] \cdot \begin{bmatrix} 3.25 \\ 1.75 \end{bmatrix} = (1 \cdot 3.25) + (2.5 \cdot 1.75) = 7.625$ ⟨u,w⟩=uTw=[1,2.5]·[3.251.75]=(1·3.25)+(2.5·1.75)=7.625

Exercise. Write some code to verify, using some examples, that $\langle u, u \rangle = u^T u = \|u\|_2^2$ ⟨u,u⟩=uTu=‖u‖22. In other words, the dot product of a vector with itself is the two-norm of that vector, squared.

In [23]:

```
```

A geometric interpretation of the dot product

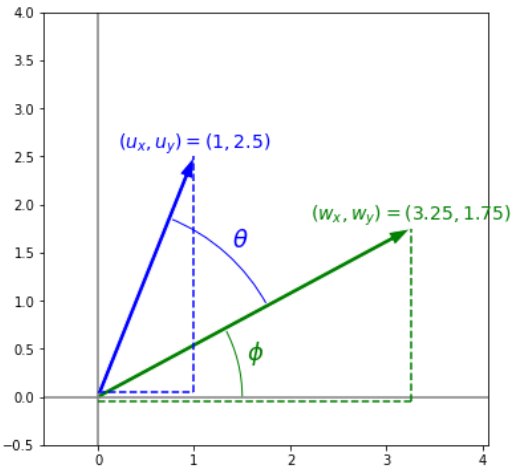
Here is another important fact about the dot product that, later on, will help us interpret it.

Fact. $u^T w = \|u\|_2 \|w\|_2 \cos\theta$ uTw=‖u‖2‖w‖2cosθ, where θθ is the angle between uu and ww.

To see this fact, consider the following diagram.

In [24]:

```
def radians_to_degrees(radians):
    from math import pi
    return radians * (180.0 / pi)

def get_angle_degrees(point):
    assert len(point) == 2, "Point must be 2-D."
    from math import pi, atan2
    return radians_to_degrees(atan2(point[1], point[0]))

figure((6, 6))
new_blank_plot(xlim=(-0.5, 4), ylim=(-0.5, 4));

draw_vector2d(u, color='blue', width=0.05)
plt.text(u[0], u[1], '$(u_x, u_y) = ({}, {})$'.format(u[0], u[1]), color='blue',
         horizontalalignment='center', verticalalignment='bottom', fontsize=14)
draw_vector2d(w, color='green', width=0.05)
plt.text(w[0], w[1], '$(w_x, w_y) = ({}, {})$'.format(w[0], w[1]), color='green',
         horizontalalignment='center', verticalalignment='bottom', fontsize=14)

draw_vector2d_components(u, y_offset_sign=1, color='blue', linestyle='dashed')
draw_vector2d_components(w, y_offset_sign=-1, color='green', linestyle='dashed')

phi_degrees = get_angle_degrees(w)
theta_degrees = get_angle_degrees(u) - phi_degrees
draw_angle(0, phi_degrees, radius=1.5, color='green')
draw_angle_label(0, phi_degrees, r'$\phi$', radius=1.6, color='green', fontsize=18)
draw_angle(phi_degrees, phi_degrees+theta_degrees, radius=2, color='blue')
```



Exercise. Let $u = \begin{bmatrix} u_x \\ u_y \end{bmatrix}$ u=[uxuy] and $w = \begin{bmatrix} w_x \\ w_y \end{bmatrix}$ w=[wxwy]. The vector uu is shown as the blue line and ww as the green line in the figure above. Let ϕϕ be the angle that ww makes with the x-axis, and let θθ be the angle between uu and ww. Using trigonometric identities from elementary geometry, prove that $u^T w = \|u\|_2 \|w\|_2 \cos\theta$ uTw=‖u‖2‖w‖2cosθ.

> *Hint.* Here is one way to start: observe that, for instance, $w_y = \|w\|_2 \sin\phi$ wy=‖w‖2sinϕ and $u_x = \|u\|_2 \cos(\theta + \phi)$ ux=‖u‖2cos(θ+ϕ), and then apply one or more trigonometric identities as needed.

Interpretation. So what does the dot product mean? One can interpret it as a "strength of association" between the two vectors, similar to statistical correlation.

To see why, observe that the dot product accounts for both the lengths of the vectors and their relative orientation.

The vector lengths are captured by the product of their lengths ($\|u\|_2\|w\|_2$ ||u||2||w||2). The longer the vectors are, the larger the product of their lengths, $\|u\|_2\|w\|_2$ ||u||2||w||2. If you know only the lengths of the vectors, their dot product can never be larger than this product.

The relative orientation is captured by $\cos\theta$cosθ. That factor moderates the maximum possible value. In particular, if the two vectors point in exactly the same direction, meaning $\theta = 0$ θ=0, then $\cos\theta = 1$ cosθ=1 and the dot product is exactly the maximum, $\|u\|_2\|w\|_2$ ||u||2||w||2. If instead the vectors point in opposite directions, meaning $\theta = \pi$ radians $= 180^\circ$ θ=π radians=180∘, then $\cos\theta = -1$ cosθ=−1 and the dot product is $-\|u\|_2\|w\|_2$ −||u||2||w||2. For any other values of θθ between $0$0 and 2π radians2π radians (or 360°360∘), $|\cos\theta| < 1$ |cosθ|<1 so that $|u^T w| < \|u\|_2\|w\|_2$ |uTw|<||u||2||w||2.

In the context of data analysis, the analogous measurement to $\cos\theta$cosθ is the Pearson correlation coefficient. Each vector would represent a regression line that goes through some sample of points, with $\cos\theta$cosθ measuing the angle between the two regression lines. The diagram above gives you a geometric way to think about such correlations.

Linear transformations

The basic operations we considered above take one or more vectors as input and *transform* them in some way. In this part of the notebook, we examine what is arguably the most important general class of transformations, which are known as *linear transformations*. Before doing so, let's start with some auxiliary concepts.

Definition: vector-valued functions (also, *vector functions*). Let $f(v)$f(v) be a function that takes as input any vector vv and returns another vector. Because $f$freturns a vector, we will sometimes refer to it as a *vector-valued function* or just *vector function* for short.

Note that the input and output vectors of ff need **not** have the same lengths!

Example: $\text{scale}_\alpha(v)$scaleα(v)**.** Scaling is a simple example of a vector function. If we name this function scalescale and parameterize it by the scaling coefficient αα, then we might write it down mathematically as

$$\text{scale}_\alpha(v) \equiv \alpha v.$$
scaleα(v)≡αv.

So,

$$\text{scale}_{1.25}(v) = 1.25 \begin{bmatrix} v_0 \\ \vdots \\ v_{d-1} \end{bmatrix} = \begin{bmatrix} 1.25v_0 \\ \vdots \\ 1.25v_{d-1} \end{bmatrix}.$$
scale1.25(v)=1.25[v0:vd−1]=[1.25v0:1.25vd−1].

The code implementation would look identical to the Python `scale()` we defined previously.

Example: $\text{avgpairs}(v)$avgpairs(v)**.** Let vv be a vector whose length, dd, is even. Here is a vector function that returns a new vector of half the number of components, where elements of the new vector are the averages of adjacent pairs of vv:

$$\text{avgpairs}(v) \equiv \begin{bmatrix} \frac{1}{2}(v_0 + v_1) \\ \frac{1}{2}(v_2 + v_3) \\ \vdots \\ \frac{1}{2}(v_{d-2} + v_{d-1}) \end{bmatrix}.$$
avgpairs(v)≡[12(v0+v1)12(v2+v3)⋮12(vd−2+vd−1)].

Exercise. Write a Python function that implements $\text{avgpairs}(v)$avgpairs(v).

In [25]:
```
# Sample solution; how would you have done it?
def avgpairs(v):
    assert dim(v) % 2 == 0, "Input vector `v` must be of even dimension."
    v_pairs = zip(v[:-1:2], v[1::2])
    v_avg = [0.5*(ve + vo) for ve, vo in v_pairs]
    return vector(*v_avg)

v_pairs = vector(1, 2, 3, 4, 5, 6, 7, 8)
```
```
(1, 2, 3, 4, 5, 6, 7, 8) => (1.5, 3.5, 5.5, 7.5)
```

Definition: linear functions (or linear transformations). A function $f(v)$f(v) is a *linear transformation* if it satisfies the following two properties:

1. $f(\sigma v) = \sigma f(v)$f(σv)=σf(v), where σσ is a scalar value.
2. $f(v + w) = f(v) + f(w)$f(v+w)=f(v)+f(w).

The first property says that ff applied to a scaled vector is the same as first applying ff to the vector and scaling the result. The second property says that $f$ffapplied to the sum of two vectors is the same as first applying ff to the individual vectors and then adding the result.

When combined, these properties are equivalent to the more concise statement that $f(\sigma v + w) = \sigma f(v) + f(w)$f(σv+w)=σf(v)+f(w).

Exercise. The function $\text{scale}_\alpha(v)$scaleα(v) is a linear transformation—true or false?

> *Answer.* This statement is true:
>
> - Property 1: $\text{scale}_\alpha(\sigma v) = \alpha(\sigma v) = \sigma(\alpha v) = \sigma\text{scale}_\alpha(v)$. scaleα(σv)=α(σv)=σ(αv)=σscaleα(v).
> - Property 2: $\text{scale}_\alpha(v + w) = \alpha(v + w) = \alpha v + \alpha w = \text{scale}_\alpha(v) + \text{scale}_\alpha(w)$. scaleα(v+w)=α(v+w)=αv+αw=scaleα(v)+scaleα(w).

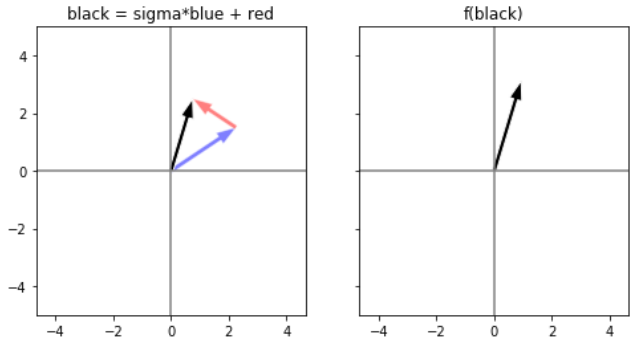
To see this fact in action, run the following experiment.

In [26]:
```
DEFAULT_ALPHA = 1.25
def scale_alpha(v, alpha=DEFAULT_ALPHA):
    return scale(v, alpha)

def latex_scale_alpha(x, alpha=DEFAULT_ALPHA):
    return r'\mathrm{{scale}}_{{{}}}\left( {} \right)'.format(alpha, x)
```

$\text{scale}_{1.25}\,(\text{x}) \equiv 1.25 \cdot \text{x}$scale1.25(x)≡1.25·x

In [27]:
```
# f(ve) where ve = sigma*va + vb
u0 = scale_alpha(ve)

u0_latex = latex_vector(u0)
arg_str = r'{} {} + {}'.format(sigma, va_latex, vb_latex)
lhs_str = latex_scale_alpha(arg_str)
arg2_str = ve_latex
mid_str = latex_scale_alpha(arg2_str)
rhs_str = u0_latex
display_math(r'u_0 \equiv {} = {} = {}'.format(lhs_str, mid_str, rhs_str))

axes = subplots((1, 2))
new_blank_plot(axes[0], title='black = sigma*blue + red')
draw_vector2d(va_scaled, color='blue', alpha=0.5)
draw_vector2d(vb, origin=va_scaled, color='red', alpha=0.5)
draw_vector2d(ve)

new_blank_plot(axes[1], title='f(black)')
```

$$u_0 \equiv \text{scale}_{1.25}\left(0.75 \begin{bmatrix} 3.0 \\ 2.0 \end{bmatrix} + \begin{bmatrix} -1.5 \\ 1.0 \end{bmatrix}\right) = \text{scale}_{1.25}\left(\begin{bmatrix} 0.75 \\ 2.5 \end{bmatrix}\right) = \begin{bmatrix} 0.9375 \\ 3.125 \end{bmatrix}$$ u0≡scale1.25(0.75[3.02.0]+[−1.51.0])=scale1.25([0.752.5])=[0.93753.125]

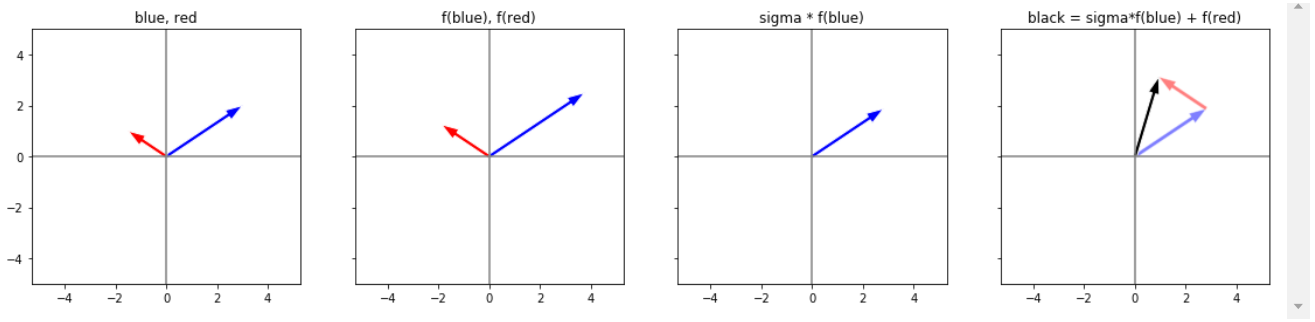


In [28]:
```
display_math(r'''\frac{{{}}}{{{}}} = {}'''.format(latex_norm(rhs_str),
                                                  latex_norm(arg_str),
```

$$\frac{\left\| \begin{bmatrix} 0.9375 \\ 3.125 \end{bmatrix} \right\|_2}{\left\| 0.75 \begin{bmatrix} 3.0 \\ 2.0 \end{bmatrix} + \begin{bmatrix} -1.5 \\ 1.0 \end{bmatrix} \right\|_2} = 1.2499999999999998$$ ‖[0.93753.125]‖2‖0.75[3.02.0]+[−1.51.0]‖2=1.2499999999999998

In [29]:
```
# sigma*f(va) + f(vb)
u1_a = scale_alpha(va)
u1_aa = scale(u1_a, sigma)
u1_b = scale_alpha(vb)
u1 = add(u1_aa, u1_b)

display_math(r'''u_1 \equiv {} \cdot {} + {} = {}'''.format(sigma,
                                                            latex_scale_alpha(va_latex),
                                                            latex_scale_alpha(vb_latex),
                                                            latex_vector(u1)))

_, axes = plt.subplots(1, 4, figsize=(19, 4), sharey='row')

new_blank_plot(axes[0], title='blue, red')
draw_vector2d(va, color='blue')
draw_vector2d(vb, color='red')

new_blank_plot(axes[1], title='f(blue), f(red)')
draw_vector2d(u1_a, color='blue')
draw_vector2d(u1_b, color='red')

new_blank_plot(axes[2], title='sigma * f(blue)')
draw_vector2d(u1_aa, color='blue')

new_blank_plot(axes[3], title='black = sigma*f(blue) + f(red)')
draw_vector2d(u1_aa, color='blue', alpha=0.5)
draw_vector2d(u1_b, origin=u1_aa, color='red', alpha=0.5)
```

$$u_1 \equiv 0.75 \cdot \text{scale}_{1.25}\begin{bmatrix} 3.0 \\ 2.0 \end{bmatrix} + \text{scale}_{1.25}\begin{bmatrix} -1.5 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0.9375 \\ 3.125 \end{bmatrix}$$ u1≡0.75·scale1.25([3.02.0])+scale1.25([−1.51.0])=[0.93753.125]



Exercise. The function $\mathrm{avgpairs}(v)$avgpairs(v) is a linear transformation—true or false?

Exercise. Let $\mathrm{norm}(v) \equiv \|v\|_1$ norm(v)≡‖v‖1. Is $\mathrm{norm}(v)$norm(v) a linear transformation?

Exercise. Let $\mathrm{offset}_\delta(v) \equiv v + \delta$offsetδ(v)≡v+δ, where δδ is some scalar value (i.e., δδ is a single number, not a vector). That is,

$$\mathrm{offset}_\delta(v) \equiv \begin{bmatrix} v_0 + \delta \\ v_1 + \delta \\ \vdots \\ v_{d-1} + \delta \end{bmatrix}.$$

offsetδ(v)≡[v0+δv1+δ:vd−1+δ].

Is $\mathrm{offset}_\delta(v)$offsetδ(v) a linear transformation?

Exercise. Suppose we are operating in a two-dimensional space. Let

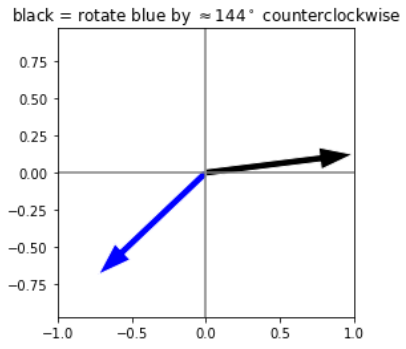
$$\mathrm{rotate}_\theta(v) \equiv \begin{bmatrix} v_0 \cos\theta - v_1 \sin\theta \\ v_0 \sin\theta + v_1 \cos\theta \end{bmatrix}.$$

rotateθ(v)≡[v0cosθ−v1sinθv0sinθ+v1cosθ].

Is $\mathrm{rotate}_\theta(v)$rotateθ(v) a linear transformation?

While, after, or instead of pondering the answer for rotate_θrotateθ, here is some code to visualize its effects. (This code selects a rotation angle at random, so you run it repeatedly to see the effects under different angles.)

In [30]:
```
def random_angle(min_angle=0, max_angle=None):
    """
    Returns a random angle in the specified interval
    or (0, pi) if no interval is given.
    """
    from math import pi
    from random import uniform
    if max_angle is None: max_angle = pi
    return uniform(0, max_angle)

def rotate(v, theta=0):
    from math import cos, sin
    assert dim(v) == 2, "Input vector must be 2-D."
    return vector(v[0]*cos(theta) - v[1]*sin(theta),
                  v[0]*sin(theta) + v[1]*cos(theta))

v_rand = normalize_vector(random_vector())
theta_rand = random_angle()
w_rand = rotate(v_rand, theta_rand)

figure()
new_blank_plot(xlim=(-1, 1), ylim=(-1, 1),
               title=r'black = rotate blue by $\approx${:.0f}$^\circ$ counterclockwise'.format(radians_to_degrees(theta_rand)))
draw_vector2d(v_rand, color='blue', width=0.05)
```



Linear transformations using matrices

If f is a linear transformation, then $f(\alpha v + \beta w) = \alpha f(v) + \beta f(w)$f(αv+βw)=αf(v)+βf(w). In fact, the reverse is *also* true: if $f(\alpha v + \beta w) = \alpha f(v) + \beta f(w)$f(αv+βw)=αf(v)+βf(w), then f must be a linear transformation. Refer to the LAFF notes for a formal proof of this fact.

Here is one immediate consequence of this fact. Suppose you have n vectors, named $v_0, v_1, \ldots, v_{n-1}$v0,v1,…,vn−1. Here, v_i vi is the name of a vector, rather than an element of a vector; the components of v_i vi would be $v_{0,i}, v_{1,i}, \ldots$v0,i,v1,i,…. Let's also suppose you have n scalars, named $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$α0,α1,…,αn−1. Then,

$$f(\alpha_0 v_0 + \alpha_1 v_1 + \cdots + \alpha_{n-1} v_{n-1}) = \alpha_0 f(v_0) + \alpha_1 f(v_1) + \cdots + \alpha_{n-1} f(v_{n-1}).$$
f(α0v0+α1v1+⋯+αn−1vn−1)=α0f(v0)+α1f(v1)+⋯+αn−1f(vn−1).

This fact makes f very special because it allows you to figure out the effect of f if you are allowed to "sample" it on particular values, the $\{v_i\}$. For example, suppose $f(v)$ operates on two-dimensional vectors and returns two-dimensional vectors as a result.

Exercise. Let $f(v)$ be a linear transformation on two-dimensional vectors v. Suppose you are told that

$$f\left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$
$$f\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} -1 \\ 2 \end{bmatrix}.$$
f([10])=[35]f([01])=[−12].

Determine $f\left(\begin{bmatrix} 2 \\ -4 \end{bmatrix}\right)$ f([2−4]).

Answer. Start by observing that

$$\begin{bmatrix} 2 \\ -4 \end{bmatrix} = 2\begin{bmatrix} 1 \\ 0 \end{bmatrix} - 4\begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$
[2−4]=2[10]−4[01].

Since f is a linear transformation,

$$f\left(\begin{bmatrix} 2 \\ -4 \end{bmatrix}\right) = f\left(2\begin{bmatrix} 1 \\ 0 \end{bmatrix} - 4\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = 2f\left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) - 4f\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = 2\begin{bmatrix} 3 \\ 5 \end{bmatrix} - 4\begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 3 - 4 \cdot (-1) \\ 2 \cdot 5 - 4 \cdot 2 \end{bmatrix} = \begin{bmatrix} 10 \\ 2 \end{bmatrix}.$$
f([2−4])=f(2[10]−4[01])=2f([10])−4f([01])=2[35]−4[−12]=[2⋅3−4⋅(−1)2⋅5−4⋅2]=[102].

Canonical axis vectors. In the preceding example, you were given samples of f on two very special vectors, namely, a distinct set of perpendicular unit vectors having a "1" in just one component of the vector. Let's denote these special vectors by e_j ej where, in a dd-dimensional space,

$$e_j \equiv \begin{bmatrix} 0 & \leftarrow \text{0-th component} \\ \vdots & \\ 0 & \\ 1 & \leftarrow \text{j-th component} \\ 0 & \\ \vdots & \\ 0 & \leftarrow \text{(d − 1)-th component} \end{bmatrix}$$
ej≡[0←0-th component:01←j-th component0:0←(d−1)-th component]

We'll refer to these as the *canonical axis vectors*.

> In the LAFF notes, these are called the *unit basis vectors*, which is more standard terminology in the linear algebra literature.

Matrix representations. Any vector may be written in terms of these canonical axis vectors.

$$v = \begin{bmatrix} v_0 \\ \vdots \\ v_{d-1} \end{bmatrix} = v_0 \begin{bmatrix} 1 \\ \vdots \\ 0 \end{bmatrix} + \cdots + v_{d-1} \begin{bmatrix} 0 \\ \vdots \\ 1 \end{bmatrix} = v_0 e_0 + \cdots + v_{d-1} e_{d-1} = \sum_{j=0}^{d-1} v_j e_j.$$
v=[v0:vd−1]=v0[1:0]+⋯+vd−1[0:1]=v0e0+⋯+vd−1ed−1=∑j=0d−1vjej.

Therefore, consider any linear transformation, f. If you are given samples of f along all axes, $\{x_j \equiv f(e_j)\}$ {xj≡f(ej)}, then that is enough to calculate f for any vector v:

$$f(v) = f\left(\sum_{j=0}^{d-1} v_j e_j\right) = \sum_{j=0}^{d-1} v_j f(e_j) = \sum_{j=0}^{d-1} v_j x_j.$$
f(v)=f(∑j=0d−1vjej)=∑j=0d−1vjf(ej)=∑j=0d−1vjxj.

Let's suppose v is n-dimensional. Therefore, there are n canonical axis vectors. If the result of $f(v)$ is m-dimensional, then each of the x_j xj vectors is also $m$mdimensional. Therefore, we can write each x_j xj as

$$x_j = \begin{bmatrix} x_{0,j} \\ \vdots \\ x_{m-1,j} \end{bmatrix}.$$

A convenient way to represent the full collection of all n of the x_j vectors is in the form of a *matrix*, where each column corresponds to one of these x_j vectors:

$$X = \begin{bmatrix} x_0 & \cdots & x_{n-1} \end{bmatrix} = \begin{bmatrix} x_{0,0} & \cdots & x_{0,n-1} \\ \vdots & \ddots & \vdots \\ x_{m-1,0} & \cdots & x_{m-1,n-1} \end{bmatrix}.$$

Aside: Specifying the dimensions of a matrix. Similar to the way we "declare" a vector v to have dimension d by writing $v \in \mathbb{R}^d$, we have a similar notation to specify that a matrix has dimensions of, say, m-by-n: we write $X \in \mathbb{R}^{m \times n}$.

Example. Recall the previous example where

$$f\left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 5 \end{bmatrix} \quad \text{and} \quad f\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} -1 \\ 2 \end{bmatrix}.$$

What is the matrix X that represents f?

Answer. We were given f at the canonical axis vectors, which then become the columns of the corresponding matrix.

$$X = \begin{bmatrix} 3 & -1 \\ 5 & 2 \end{bmatrix}.$$

Exercise. What is the matrix X that represents $\text{rotate}_\theta(v)$?

A code representation. In this course, you will see many ways of representing matrices and vectors. For the purpose of this notebook, let's store a matrix as a tuple of (column) vectors. Here is a Python function, `matrix(x0, x1, ...)`, where each argument is a column vector, that stores the columns in a tuple.

In [31]:
```
def matrix(*cols):
    if len(cols) > 2:
        a_cols = cols[:-1]
        b_cols = cols[1:]
        for k, (a, b) in enumerate(zip(a_cols, b_cols)):
            assert dim(a) == dim(b), \
                   "Columns {} and {} have different lengths ({} vs. {})".format(k, k+1, dim(a), dim(b))
    return tuple(cols)

def num_cols(X):
    return len(X)

def num_rows(X):
    return dim(X[0]) if num_cols(X) >= 1 else 0

# Demo
X = matrix(vector(3, 5), vector(-1, 2))
```

```
X = ((3, 5), (-1, 2))
```

Here is a snippet of code that will be useful for pretty-printing matrices.

In [32]:
```
def matelem(X, row, col):
    assert col < dim(X), "Column {} is invalid (matrix only has {} columns).".format(col, dim(X))
    x_j = X[col]
    assert row < dim(x_j), "Row {} is invalid (matrix only has {} rows).".format(row, dim(x_j))
    return x_j[row]

def latex_matrix(X):
    m, n = num_rows(X), num_cols(X)
    s = r'\left[\begin{matrix}'
    for i in range(m):
        if i > 0: s += r' \\' # New row
        for j in range(n):
            if j > 0: s += ' & '
            s += str(matelem(X, i, j))
    s += r' \end{matrix}\right]'
    return s

X_latex = latex_matrix(X)
```

$$X = \begin{bmatrix} 3 & -1 \\ 5 & 2 \end{bmatrix}$$ X=[3−152]

Definition: matrix-vector product (matrix-vector multiply). Let's define the *matrix-vector product* (or *matrix-vector multiply*) as follows.

Given an mm-by-nn matrix XX and a vector vv of length nn, the *matrix-vector product* is given by

$$Xv = \begin{bmatrix} x_0 & \cdots & x_{n-1} \end{bmatrix} \begin{bmatrix} v_0 \\ \vdots \\ v_{n-1} \end{bmatrix} = x_0 v_0 + \cdots + x_{n-1} v_{n-1}.$$

Xv=[x0⋯xn−1][v0⋮vn−1]=x0v0+⋯+xn−1vn−1.

With this notation, a linear transformation ff represented by XX may be written as $f(v) = Xv$f(v)=Xv.

> We will sometimes write the matrix vector product with an explicit "dot" operator, $X \cdot v$X·v. This usage is arbitrary and we will use it for aesthetic reasons only.

Linear combination. The action of a matrix-vector product is to use the entries of vv to scale the corresponding columns of XX followed by a sum of the resulting scaled vectors. We say that XvXv is a *linear combination* of the columns of XX. The "coefficients" or "weights" of this linear combination are the elements of vv.

Example. Continuing the example above, calculate $f([2, -4]^T)$f([2,−4]T) using XX.

Answer. We can apply the matrix-vector product:

$$\begin{bmatrix} 3 & -1 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ -4 \end{bmatrix} = 2 \begin{bmatrix} 3 \\ 5 \end{bmatrix} - 4 \begin{bmatrix} -1 \\ 2 \end{bmatrix},$$

[3−152][2−4]=2[35]−4[−12],

which is the same result as before.

Here is a Python code implementation of the matrix-vector product. Take a moment to verify that you understand how it works.

In [33]:
```
def matvec(X, v):
    assert dim(X) == dim(v), "Matrix and vector have mismatching shapes."
    w = [0] * num_rows(X)
    for x_j, v_j in zip(X, v):
        w = add(w, scale(x_j, v_j))
    return w

v = vector(2, -4)
w = matvec(X, v)

v_latex = latex_vector(v)
w_latex = latex_vector(w)
```

$$Xv = \begin{bmatrix} 3 & -1 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ -4 \end{bmatrix} = \begin{bmatrix} 10 \\ 2 \end{bmatrix}$$ Xv=[3−152][2−4]=[102]

Exercise. Another way to think about the action of a matrix-vector product is to imagine computing one entry of the output at a time. That is, let $w = Xv$w=Xv. Write down a formula for $w_i$wi.

Answer. We want the ii-th entry of $w = Xv$w=Xv. Let's denote that by $w_i = (Xv)_i$wi=(Xv)i. From the definition of matrix-vector product given above,

$$w_i = (Xv)_i = (x_0 v_0 + x_1 v_1 + \cdots + x_{n-1} v_{n-1})_i,$$

wi=(Xv)i=(x0v0+x1v1+⋯+xn−1vn−1)i,

that is, the ii-th entry of the right-hand side. So, we need to gather the ii-th entry of every term on the right-hand side, e.g.,

$$w_i = (Xv)_i = (x_0 v_0 + x_1 v_1 + \cdots + x_{n-1} v_{n-1})_i = x_{i,0} v_0 + x_{i,1} v_1 + \cdots + x_{i,n-1} v_{n-1}, = \sum_{j=0}^{n-1} x_{i,j} v_j.$$

wi=(Xv)i=(x0v0+x1v1+⋯+xn−1vn−1)i=xi,0v0+xi,1v1+⋯+xi,n−1vn−1,=∑j=0n−1xi,jvj.

In other words, the ii-th entry of the output ww is the dot product between the ii-th row of XX and vv.

Example: linear systems. Matrix notation is also a convenient way to write down systems of linear equations. One setting in which we will see such systems is in linear regression.

Suppose you want to predict college GPA given a student's high school GPA and SAT score given observed data. Let ii be a student with college GPA $c_i$ci, high school GPA $h_i$hi, and SAT score $s_i$si, and suppose there are mm students in total, numbered from 0 to $m-1$m−1. As an initial guess, you might hypothesize a linear relationship among these variables, e.g., $c_i \approx \theta_0 h_i + \theta_1 s_i + \theta_2$ ci≈θ0hi+θ1si+θ2, where θ_0θ0, θ_1θ1, and θ_2θ2 are unknowns. That is, considering all of the data, you believe that

$$\begin{aligned} c_0 &\approx \theta_0 h_0 + \theta_1 s_0 + \theta_2 \\ &\vdots \\ c_i &\approx \theta_0 h_i + \theta_1 s_i + \theta_2 \\ &\vdots \\ c_{m-1} &\approx \theta_0 h_{m-1} + \theta_1 s_{m-1} + \theta_2. \end{aligned}$$

c0≈θ0h0+θ1s0+θ2⋮ci≈θ0hi+θ1si+θ2⋮cm−1≈θ0hm−1+θ1sm−1+θ2.

Letting

$$c \equiv \begin{bmatrix} c_0 \\ \vdots \\ c_{m-1} \end{bmatrix}, \quad X \equiv \begin{bmatrix} h_0 & s_0 & 1 \\ \vdots & & \\ h_{m-1} & s_{m-1} & 1 \end{bmatrix}, \quad \text{and} \quad \theta \equiv \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix},$$

c≡[c0…cm−1],X≡[h0s01:hm−1sm−11],andθ≡[θ0θ1θ2],

Definition: matrix transpose. In the same way that we can transpose a row or column vector to turn it into a column or row vector, we can also transpose all rows or columns of a matrix. This operation is the *matrix transpose*, denoted by X^TXT. That is, if XX is an mm-by-nn matrix, then the transpose operation $Y = X^T$Y=XTproduces an nn-by-mm matrix YY such that $y_{i,j} = x_{j,i}$ yi,j=xj,i. For example, if

$$X = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix},$$

X=[012345],

then

$$Y = X^T = \begin{bmatrix} 0 & 3 \\ 1 & 4 \\ 2 & 5 \end{bmatrix}.$$

Y=XT=[031425].

Example: Markov chains

Let's connect linear transformations and matrices to a concept in data analysis, namely, that of a *Markov chain*.

> The idea for this example comes from the LAFF notes.

Mobile messaging. Alice, Bob, and Carol are friends. Suppose we have some information on how likely each person is to send a text message to the other. They live in a closed universe in which they only text one another, and no one else.

The data we have appears in the table below, where the rows correspond to the sender and the columns correspond to the receiver. Each (i, j)(i,j) entry of the table indicates the probability that person ii (row ii) sends a text to person jj (column jj). The rows sum to 1.

Send / Recv	Alice	Bob	Carol
Alice	0.0	0.4	0.6
Bob	0.5	0.1	0.4
Carol	0.3	0.7	0.0

For example, the probability that Bob will send a message to Carol is 0.4 and the probability that Carol will send a message to Alice is 0.3. Let's further suppose that a person only sends a text when he or she has received a text. The likelihood of being a recipient is given by these probabilities.

> Bob sometimes sends messages to himself! It's not because he's crazy; rather, he uses texts as a way to send reminders to himself. Also, let's not worry about the existential question of who sends the first message if they are always sending only after receiving.

An analysis question. Suppose Bob receives a message. Who is most likely to get the next message? (Answer: Alice.) Now suppose 100 more messages are transmitted after the first one that Bob sent. Who is the most likely recipient?

We can answer these questions using matrix-vector products.

Let XX be the matrix representing the table above:

$$X = \begin{bmatrix} 0 & 0.4 & 0.6 \\ 0.5 & 0.1 & 0.4 \\ 0.3 & 0.7 & 0.0 \end{bmatrix}.$$

X=[00.40.60.50.10.40.30.70.0].

Next, let $r(k)$r(k) be a vector of dimension 3 whose elements represent the probabilities that Alice, Bob, and Carol are the recipient of the kk-th message. Since Bob receives the first message,

$$r(0) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}. \quad \leftarrow \text{Bob is the first recipient}$$

r(0)=[010].←Bob is the first recipient

We'd like to compute $r(1)$r(1) from $r(0)$r(0) and XX. Essentially, we want it to "pick out" Bob's row of the table. Recalling that a matrix-vector product produces linear combinations of the columns, the way to pick out this row is to transpose the matrix and then multiply it by $r(0)$r(0), e.g.,

$$r(1) = X^T r(0) = \begin{bmatrix} 0 & 0.5 & 0.3 \\ 0.4 & 0.1 & 0.7 \\ 0.6 & 0.4 & 0.0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 0 \cdot \begin{bmatrix} 0 \\ 0.4 \\ 0.6 \end{bmatrix} + 1 \cdot \begin{bmatrix} 0.5 \\ 0.1 \\ 0.4 \end{bmatrix} + 0 \cdot \begin{bmatrix} 0.6 \\ 0.4 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.1 \\ 0.4 \end{bmatrix}.$$

r(1)=XTr(0)=[00.50.30.40.10.70.60.40.0][010]=0·[00.40.6]+1·[0.50.10.4]+0·[0.60.40]=[0.50.10.4].

In [34]:
```
XT = matrix(vector(0, 0.4, 0.6),
```

```
            vector(0.5, 0.1, 0.4),
            vector(0.3, 0.7, 0))
r = [vector(0, 1, 0)]
r.append(matvec(XT, r[-1]))

XT_latex = latex_matrix(XT)
r_latex = [latex_vector(ri) for ri in r]
```

$$X^T r(0) = \begin{bmatrix} 0 & 0.5 & 0.3 \\ 0.4 & 0.1 & 0.7 \\ 0.6 & 0.4 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.1 \\ 0.4 \end{bmatrix} = r(1).$$ XTr(0)=[00.50.30.40.10.70.60.40][010]=[0.50.10.4]=r(1).

What about $r(2)$r(2)? If we interpret $r(k)$r(k) as a vector of probabilities of who was the last person to have received a message, then we can keep multiplying, i.e., $r(2) = X^T r(1)$ r(2)=XTr(1). Doing so assumes that the next recipient depends only on the most recent recipient, which is an assumption known as the *Markov property*. We will formalize Markov chains in a different topic, but for now, hopefully you see that this analysis can compute $r(k)$r(k) by $r(k) = X^T r(k-1) = X^T \cdot X^T \cdot r(k-2) = (X^T)^k r(0)$ r(k)=XTr(k−1)=XT·XT·r(k−2)=(XT)kr(0).

In [35]:
```
r = [vector(0, 1, 0)] # Start over
for _ in range(10):
    r.append(matvec(XT, r[-1]))

# Display the last few iterations:
k_left = min(3, len(r))
for k in range(len(r)-k_left, len(r)):
```

$$r(8) = \begin{bmatrix} 0.29082461 \\ 0.38113284999999997 \\ 0.32804254 \end{bmatrix}$$ r(8)=[0.290824610.381132849999999970.32804254]

$$r(9) = \begin{bmatrix} 0.28897918699999997 \\ 0.38407290699999996 \\ 0.326947906 \end{bmatrix}$$ r(9)=[0.288979186999999970.384072906999999960.326947906]

$$r(10) = \begin{bmatrix} 0.2901208253 \\ 0.38286249969999997 \\ 0.327016675 \end{bmatrix}$$ r(10)=[0.290120825300.382862499699999970.327016675]

You should observe that the values appear to be converging and that Bob is a little bit more likely to be the recipient at any given future moment in time.

Aside: visualizing a linear transformation. Here is a visual experiment that you can play with to get a rough idea of the effect of a linear transformation.

Let ZZ be a matrix. The following code generates fifty random vectors of unit length (in the two-norm). For each vector $v_k$vk, it applies the linear transformation represented by ZZ, resulting in a new vector $w_k = Z v_k$ wk=Zvk. It then draws an arrow corresponding to the *change* $d_k = w_k - v_k$ dk=wk−vk, placing it at the endpoint of $v_k$vk if $v_k$vkis placed at the origin. In other words, the arrow shows how different points on the unit circle "move" under the transformation ZZ.

Try changing ZZ to see what happens under different linear transformations. (You can uncomment the line that generates a random ZZ.) What do you observe?

In [36]:
```
# Here is the matrix from above:
Z = matrix(vector(3, 5), vector(-1, 2))

# Or, uncomment this to try a random matrix:
#Z = matrix(random_vector(v_min=-5, v_max=5), random_vector(v_min=-5, v_max=5))

display_math('Z = ' + latex_matrix(Z))

figure(figsize=(6, 6))
new_blank_plot(xlim=(-6, 6), ylim=(-6, 6))
for _ in range(50):
    vk = normalize_vector(random_vector())
    wk = matvec(Z, vk)
    dk = sub(wk, vk)
```

$$Z = \begin{bmatrix} 3 & -1 \\ 5 & 2 \end{bmatrix}$$ Z=[3−152]

Matrix-matrix products

This part of the notebook illustrates what is arguably the most important primitive operation in linear algebra, the *matrix-matrix product*. Let's get there by an example.

Example: Combined rotations. Recall the linear transformation corresponding to rotation,

$$\text{rotate}_\theta(v) \equiv \begin{bmatrix} v_0 \cos\theta - v_1 \sin\theta \\ v_0 \sin\theta + v_1 \cos\theta \end{bmatrix}.$$

rotateθ(v)≡[v0cosθ−v1sinθv0sinθ+v1cosθ].

It can be written as a matrix-vector product,

$$\text{rotate}_\theta(v) = \underbrace{\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}}_{\equiv R(\theta)} \begin{bmatrix} v_0 \\ v_1 \end{bmatrix}.$$

rotateθ(v)=[cosθ−sinθsinθcosθ]≡R(θ)[v0v1].

Here, we've given the name of the matrix of coefficients $R(\theta)$R(θ).

Now, suppose you rotate a vector first by θθ and then by a different angle ϕϕ. Applying these rotations "inside-out," that is, first by θθ and then by ϕϕ results in the following sequence of products:

$$\begin{aligned} \text{rotate}_\phi(\text{rotate}_\theta(v)) &= R(\phi) \cdot (R(\theta) \cdot v) \\ &= R(\phi) \cdot \left(\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} \right) \\ &= R(\phi) \cdot \begin{bmatrix} v_0 \cos\theta - v_1 \sin\theta \\ v_0 \sin\theta + v_1 \cos\theta \end{bmatrix} \\ &= \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \cdot \begin{bmatrix} v_0 \cos\theta - v_1 \sin\theta \\ v_0 \sin\theta + v_1 \cos\theta \end{bmatrix} \\ &= \begin{bmatrix} \cos\phi \cdot (v_0 \cos\theta - v_1 \sin\theta) - \sin\phi \cdot (v_0 \sin\theta + v_1 \cos\theta) \\ \sin\phi \cdot (v_0 \cos\theta - v_1 \sin\theta) + \cos\phi \cdot (v_0 \sin\theta + v_1 \cos\theta) \end{bmatrix}. \end{aligned}$$

rotateϕ(rotateθ(v))=R(ϕ)·(R(θ)·v)=R(ϕ)·([cosθ−sinθsinθcosθ][v0v1])=R(ϕ)·[v0cosθ−v1sinθv0sinθ+v1cosθ]=[cosϕ−sinϕsinϕcosϕ]·[v0cosθ−v1sinθv0sinθ+v1cosθ]=[c

Rearranging terms,

$$\begin{bmatrix} (\cos\phi\cos\theta - \sin\phi\sin\theta)v_0 + (-\cos\phi\sin\theta - \sin\phi\cos\theta)v_1 \\ (\sin\phi\cos\theta + \cos\phi\sin\theta)v_0 + (\cos\phi\cos\theta - \sin\phi\sin\theta)v_1 \end{bmatrix} = \begin{bmatrix} \cos\phi\cos\theta - \sin\phi\sin\theta & -\cos\phi\sin\theta - \sin\phi\cos\theta \\ \sin\phi\cos\theta + \cos\phi\sin\theta & \cos\phi\cos\theta - \sin\phi\sin\theta \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \end{bmatrix},$$

[(cosϕcosθ−sinϕsinθ)v0+(−cosϕsinθ−sinϕcosθ)v1(sinϕcosθ+cosϕsinθ)v0+(cosϕcosθ−sinϕsinθ)v1]=[cosϕcosθ−sinϕsinθ−cosϕsinθ−sinϕcosθsinϕcosθ+cosϕsinθcos

in other words, a matrix-vector product where the two rotation matrices, $R(\phi)$R(ϕ) and $R(\theta)$R(θ) have been combined.

Look carefully at this combined matrix. The first column looks like it could have been constructed from the matrix-vector product,

$$\begin{bmatrix} \cos\phi\cos\theta - \sin\phi\sin\theta \\ \sin\phi\cos\theta + \cos\phi\sin\theta \end{bmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix},$$

[cosϕcosθ−sinϕsinθsinϕcosθ+cosϕsinθ]=[cosϕ−sinϕsinϕcosϕ][cosθsinθ],

where the vector is just the first column of $R(\theta)$R(θ).

Similarly, the second column could have been constructed from

$$\begin{bmatrix} -\cos\phi\sin\theta - \sin\phi\cos\theta \\ \cos\phi\cos\theta - \sin\phi\sin\theta \end{bmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix},$$

[−cosϕsinθ−sinϕcosθcosϕcosθ−sinϕsinθ]=[cosϕ−sinϕsinϕcosϕ][−sinθcosθ],

where the vector is just the second column of $R(\theta)$R(θ). In other words, the combination of $R(\phi)$R(ϕ) and $R(\theta)$R(θ) is obtained by a sequence of matrix-vector products, where $R(\phi)$R(ϕ) is multiplied by each column of $R(\theta)$R(θ) to get each corresponding column of their combination. These observations motivate the definition of a *matrix-matrix product*.

Definition: matrix-matrix product, or simply, *matrix product* or *matrix multiply*. Let AA be an mm-by-kk matrix and let BB be a kk-by-nn matrix. View BB by its columns,

$$B = \begin{bmatrix} b_0 & b_1 & \cdots & b_{n-1} \end{bmatrix}.$$

B=[b0b1⋯bn−1].

Then the *matrix-matrix product*, ABAB (also denoted as $A \cdot B$A·B) is defined as

$$AB \equiv A \begin{bmatrix} b_0 & b_1 & \cdots & b_{n-1} \end{bmatrix} = \begin{bmatrix} Ab_0 & Ab_1 & \cdots & Ab_{n-1} \end{bmatrix}.$$

AB≡A[b0b1⋯bn−1]=[Ab0Ab1⋯Abn−1].

Note that a matrix-vector product is just the special case in which $n = 1$ n=1, i.e., BB has just one column.

Exercise. Implement a Python function, `matmat(A, B)`, that performs a matrix-matrix product according to the preceding definition.

In [37]:
```
def matmat(A, B):
```

```
    m, k_A = num_rows(A), num_cols(A)
    k_B, n = num_rows(B), num_cols(B)
    assert k_A == k_B, "Inner-dimensions of `A` and `B` do not match."

    C_cols = []
    for bi in B:
        C_cols.append(matvec(A, bi))
    C = matrix(*C_cols)
```

In [38]:
```
A = matrix(vector(1, 2, 3), vector(4, 5, 6), vector(7, 8, 9))
B = matrix(vector(-1, -1, -1), vector(1, 1, 1), vector(0.5, 0.25, 0.125))
C = matmat(A, B)

A_latex = latex_matrix(A)
B_latex = latex_matrix(B)
C_latex = latex_matrix(C)
```

$$\begin{bmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{bmatrix}\cdot\begin{bmatrix}-1 & 1 & 0.5\\ -1 & 1 & 0.25\\ -1 & 1 & 0.125\end{bmatrix}=\begin{bmatrix}-12 & 12 & 2.375\\ -15 & 15 & 3.25\\ -18 & 18 & 4.125\end{bmatrix}$$
[147258369]·[−110.5−110.25−110.125]=[−12122.375−15153.25−18184.125]

Exercise. Let $C = AB$C=AB, where $A \in \mathbb{R}^{m\times k}$A∈Rm×k and $B \in \mathbb{R}^{k\times n}$B∈Rk×n, so that $C \in \mathbb{R}^{m\times n}$C∈Rm×n. Show that every output element, $c_{i,j}$ ci,j, may be computed by the scalar formula,

$$c_{i,j} = \sum_{s=0}^{k-1} a_{i,s} \cdot b_{s,j},$$
ci,j=∑s=0k−1ai,s·bs,j,

that is, as the dot product between row ii of AA and column jj of BB. If you learned how to multiply matrices in high school or an introductory college class, you most likely learned this formula. We will sometimes refer to this way of computing a matrix multiply as the *dot product method*.

Exercise. Let $C = AB^T$ C=ABT, where $A \in \mathbb{R}^{m\times k}$A∈Rm×k and $B \in \mathbb{R}^{n\times k}$B∈Rn×k, so that $B^T \in \mathbb{R}^{k\times n}$BT∈Rk×n and $C \in \mathbb{R}^{m\times n}$C∈Rm×n. Show that,

$$C = \sum_{s=0}^{k-1} a_s \cdot b_s^T,$$
C=∑s=0k−1as·bsT,

where $a_s$as and $b_s$bs denote the ss-th columns of AA and BB, respectively.

> **Important note!** The previous two exercises are two important facts about matrix products, namely, that there are different ways to think about how to compute it. We will frequently go back-and-forth between different methods, so it is best if you convince yourself that these formulas are right (*or find the bugs in them if they are not right*) and memorize the (*correct*) relations.

Fin! This is the end of this notebook. At this point, you should read the Da Kuang's notes on linear algebra (see link on edX under "Topic 3.")

In [ ]:

In [ ]: