1 Linear Transformations

Lab Objective: *Linear transformations are the most basic and essential operators in vector space theory. In this lab we visually explore how linear transformations alter points in the Cartesian plane. We also empirically explore the computational cost of applying linear transformations via matrix multiplication.*

Linear Transformations

A *linear transformation* is a mapping between vector spaces that preserves addition and scalar multiplication. More precisely, let V and W be vector spaces over a common field $\mathbb{F}$. A map $L : V \to W$ is a linear transformation from V into W if

$$L(a\mathbf{x}_1 + b\mathbf{x}_2) = aL\mathbf{x}_1 + bL\mathbf{x}_2$$

for all vectors $\mathbf{x}_1,\ \mathbf{x}_2 \in V$ and scalars $a,\ b \in \mathbb{F}$.

Every linear transformation L from an m-dimensional vector space into an n-dimensional vector space can be represented by an $m \times n$ matrix A, called the *matrix representation* of L. To apply L to a vector $\mathbf{x}$, left multiply by its matrix representation. This results in a new vector $\mathbf{x}'$, where each component is some linear combination of the elements of $\mathbf{x}$. For linear transformations from $\mathbb{R}^2$ to $\mathbb{R}^2$, this process has the form

$$A\mathbf{x} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{x}'.$$

Linear transformations can be interpreted geometrically. To demonstrate this, consider the array of points H that collectively form a picture of a horse, stored in the file `horse.npy`. The coordinate pairs $\mathbf{x}_i$ are organized by column, so the array has two rows: one for x-coordinates, and one for y-coordinates. Matrix multiplication on the left transforms each coordinate pair, resulting in another matrix H' whose columns are the transformed coordinate pairs:

$$AH = A\begin{bmatrix} x_1 & x_2 & x_3 & \dots \\ y_1 & y_2 & y_3 & \dots \end{bmatrix} = A\left[\begin{array}{c|c|c|c} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \dots \end{array}\right] = \left[\begin{array}{c|c|c|c} A\mathbf{x}_1 & A\mathbf{x}_2 & A\mathbf{x}_3 & \dots \end{array}\right]$$
$$= \left[\begin{array}{c|c|c|c} \mathbf{x}_1' & \mathbf{x}_2' & \mathbf{x}_3' & \dots \end{array}\right] = \begin{bmatrix} x_1' & x_2' & x_3' & \dots \\ y_1' & y_2' & y_3' & \dots \end{bmatrix} = H'.$$

To begin, use `np.load()` to extract the array from the `npy` file, then plot the unaltered points as individual pixels. See Figure 1.1 for the result.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

# Load the array from the .npy file.
>>> data = np.load("horse.npy")

# Plot the x row against the y row with black pixels.
>>> plt.plot(data[0], data[1], 'k,')

# Set the window limits to [-1, 1] by [-1, 1] and make the window square.
>>> plt.axis([-1,1,-1,1])
>>> plt.gca().set_aspect("equal")
>>> plt.show()
```

Types of Linear Transformations

Linear transformations from $\mathbb{R}^2$ into $\mathbb{R}^2$ can be classified in a few ways.

- **Stretch**: Stretches or compresses the vector along each axis. The matrix representation is diagonal:

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}.$$

  If $a = b$, the transformation is called a *dilation*. The stretch in Figure 1.1 uses $a = \frac{1}{2}$ and $b = \frac{6}{5}$ to compress the x-axis and stretch the y-axis.

- **Shear**: Slants the vector by a scalar factor horizontally or vertically (or both simultaneously). The matrix representation is

$$\begin{bmatrix} 1 & a \\ b & 1 \end{bmatrix}.$$

  Pure horizontal shears ($b = 0$) skew the x-coordinate of the vector while pure vertical shears ($a = 0$) skew the y-coordinate. Figure 1.1 has a horizontal shear with $a = \frac{1}{2}, b = 0$.

- **Reflection**: Reflects the vector about a line that passes through the origin. The reflection about the line spanned by the vector $[a, b]^\mathsf{T}$ has the matrix representation

$$\frac{1}{a^2 + b^2}\begin{bmatrix} a^2 - b^2 & 2ab \\ 2ab & b^2 - a^2 \end{bmatrix}.$$

  The reflection in Figure 1.1 reflects the image about the y-axis ($a = 0$, $b = 1$).

- **Rotation**: Rotates the vector around the origin. A counterclockwise rotation of θ radians has the following matrix representation:

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

  A negative value of θ performs a clockwise rotation. Choosing $\theta = \frac{\pi}{2}$ produces the rotation in Figure 1.1.

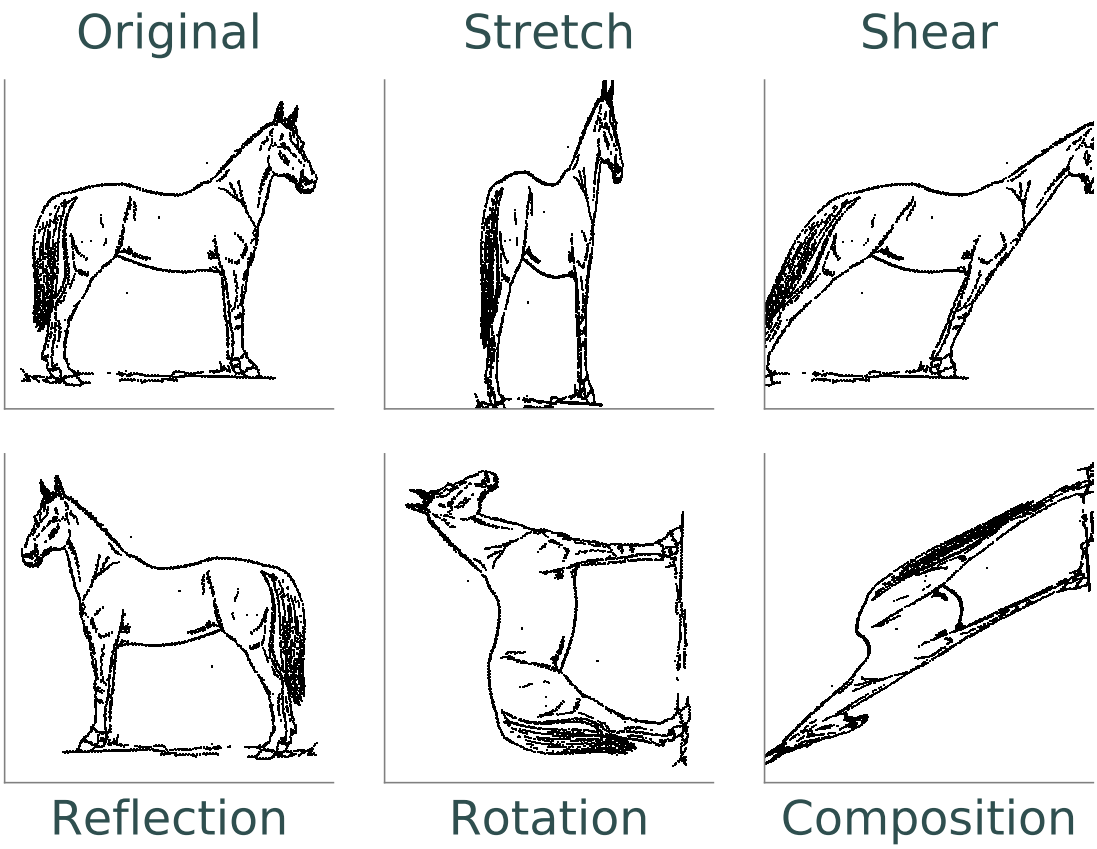


Figure 1.1: The points stored in `horse.npy` under various linear transformations.

Problem 1. Write a function for each type of linear transformation. Each function should accept an array to transform and the scalars that define the transformation (a and b for stretch, shear, and reflection, and θ for rotation). Construct the matrix representation, left multiply it with the input array, and return the transformed array.

To test these functions, write a function to plot the original points in `horse.npy` together with the transformed points in subplots for a side-by-side comparison. Compare your results to Figure 1.1.

Compositions of Linear Transformations

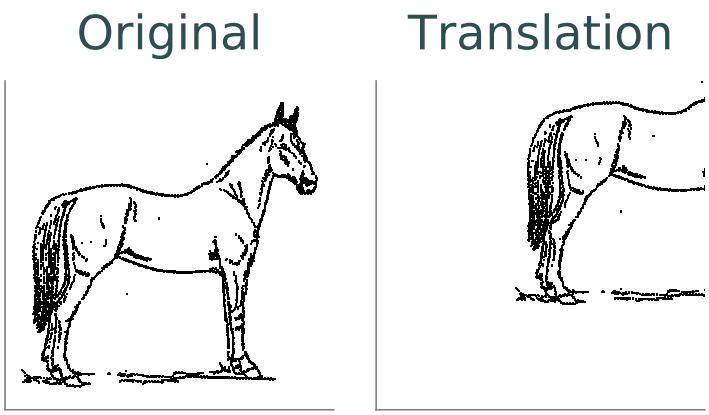
Let V, W, and Z be finite-dimensional vector spaces. If $L : V \to W$ and $K : W \to Z$ are linear transformations with matrix representations A and B, respectively, then the *composition* function $KL : V \to Z$ is also a linear transformation, and its matrix representation is the matrix product BA.

For example, if S is a matrix representing a shear and R is a matrix representing a rotation, then RS represents a shear followed by a rotation. In fact, any linear transformation $L : \mathbb{R}^2 \to \mathbb{R}^2$ is a composition of the four transformations discussed above. Figure 1.1 displays the composition of all four previous transformations, applied in order (stretch, shear, reflection, then rotation).

Affine Transformations

All linear transformations map the origin to itself. An *affine transformation* is a mapping between vector spaces that preserves the relationships between points and lines, but that may not preserve the origin. Every affine transformation T can be represented by a matrix A and a vector $\mathbf{b}$. To apply T to a vector x, calculate $A\mathbf{x}+\mathbf{b}$. If $\mathbf{b}=\mathbf{0}$ then the transformation is linear, and if $A=I$ but $\mathbf{b}\neq\mathbf{0}$ then it is called a *translation*.

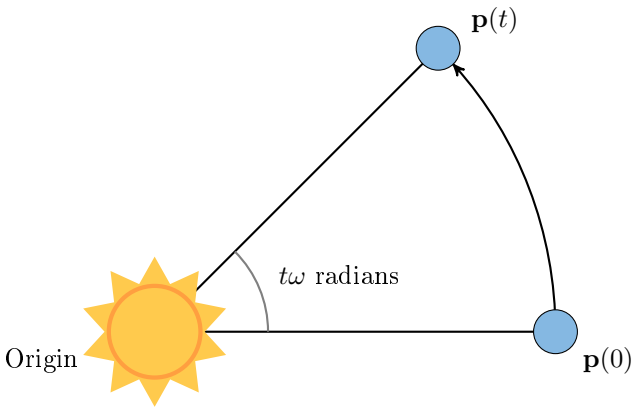
For example, if T is the translation with $\mathbf{b}=\left[\frac{3}{4},\frac{1}{2}\right]^\mathsf{T}$, then applying T to an image will shift it right by $\frac{3}{4}$ and up by $\frac{1}{2}$. This translation is illustrated below.



Affine transformations include all compositions of stretches, shears, rotations, reflections, and translations. For example, if S represents a shear and R a rotation, and if $\mathbf{b}$ is a vector, then $RS\mathbf{x}+\mathbf{b}$ shears, then rotates, then translates $\mathbf{x}$.

Modeling Motion with Affine Transformations

Affine transformations can be used to model particle motion, such as a planet rotating around the sun. Let the sun be the origin, the planet's location at time t be given by the vector $\mathbf{p}(t)$, and suppose the planet has angular velocity ω (a measure of how fast the planet goes around the sun). To find the planet's position at time t given the planet's initial position $\mathbf{p}(0)$, rotate the vector $\mathbf{p}(0)$ around the origin by $t\omega$ radians. Thus if $R(\theta)$ is the matrix representation of the linear transformation that rotates a vector around the origin by θ radians, then $\mathbf{p}(t)=R(t\omega)\mathbf{p}(0)$.



Composing the rotation with a translation shifts the center of rotation away from the origin, yielding more complicated motion.

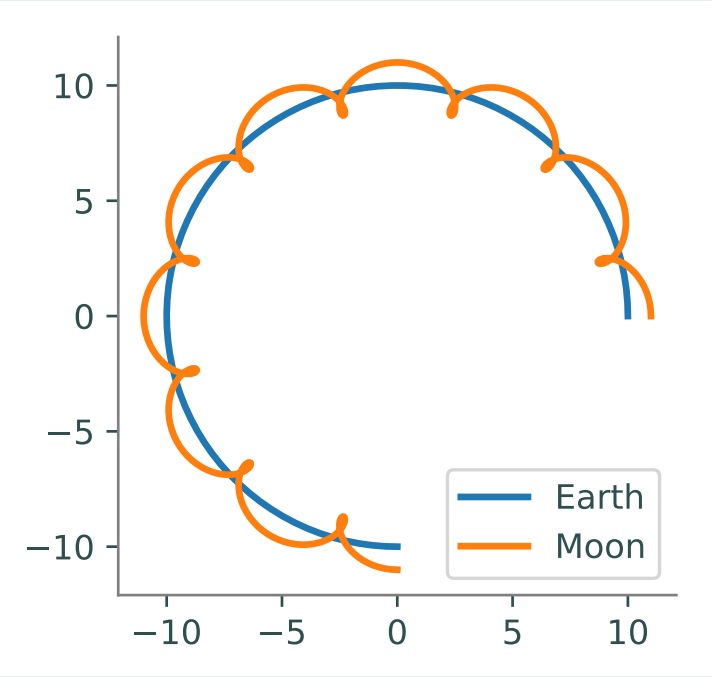
Problem 2. The moon orbits the earth while the earth orbits the sun. Assuming circular orbits, we can compute the trajectories of both the earth and the moon using only linear and affine transformations.

Assume an orientation where both the earth and moon travel counterclockwise, with the sun at the origin. Let $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ be the positions of the earth and the moon at time t, respectively, and let ω_e and ω_m be each celestial body's angular velocity. For a particular time t, we calculate $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ with the following steps.

1. Compute $\mathbf{p}_e(t)$ by rotating the initial vector $\mathbf{p}_e(0)$ counterclockwise about the origin by $t\omega_e$ radians.
2. Calculate the position of the moon relative to the earth at time t by rotating the vector $\mathbf{p}_m(0) - \mathbf{p}_e(0)$ counterclockwise about the origin by $t\omega_m$ radians.
3. To compute $\mathbf{p}_m(t)$, translate the vector resulting from the previous step by $\mathbf{p}_e(t)$.

Write a function that accepts a final time T, initial positions x_e and x_m, and the angular momenta ω_e and ω_m. Assuming initial positions $\mathbf{p}_e(0) = (x_e, 0)$ and $\mathbf{p}_m(0) = (x_m, 0)$, plot $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ over the time interval $t \in [0, T]$.

Setting $T = \frac{3\pi}{2}$, $x_e = 10$, $x_m = 11$, $\omega_e = 1$, and $\omega_m = 13$, your plot should resemble the following figure (fix the aspect ratio with `ax.set_aspect("equal")`). Note that a more celestially accurate figure would use $x_e = 400$, $x_m = 401$ (the interested reader should see `http://www.math.nus.edu.sg/aslaksen/teaching/convex.html`).



Timing Matrix Operations

Linear transformations are easy to perform via matrix multiplication. However, performing matrix multiplication with very large matrices can strain a machine's time and memory constraints. For the remainder of this lab we take an empirical approach in exploring how much time and memory different matrix operations require.

Timing Code

Recall that the `time` module's `time()` function measures the number of seconds since the Epoch. To measure how long it takes for code to run, record the time just before and just after the code in question, then subtract the first measurement from the second to get the number of seconds that have passed. Additionally, in IPython, the quick command `%timeit` uses the `timeit` module to quickly time a single line of code.

```
In [1]: import time

In [2]: def for_loop():
   ...:     """Go through ten million iterations of nothing."""
   ...:     for _ in range(int(1e7)):
   ...:         pass

In [3]: def time_for_loop():
   ...:     """Time for_loop() with time.time()."""
   ...:     start = time.time()             # Clock the starting time.
   ...:     for_loop()
   ...:     return time.time() - start      # Return the elapsed time.

In [4]: time_for_loop()
0.24458789825439453

In [5]: %timeit for_loop()
248 ms +- 5.35 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

Timing an Algorithm

Most algorithms have at least one input that dictates the size of the problem to be solved. For example, the following functions take in a single integer n and produce a random vector of length n as a list or a random $n \times n$ matrix as a list of lists.

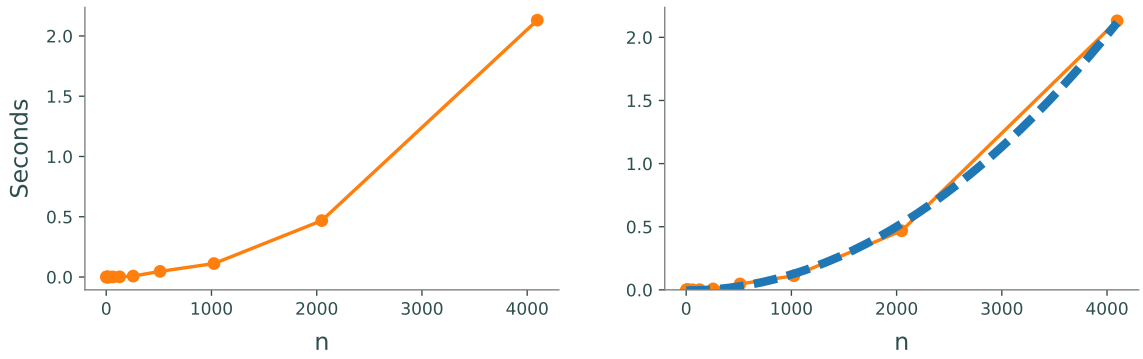
```
from random import random
def random_vector(n):       # Equivalent to np.random.random(n).tolist()
    """Generate a random vector of length n as a list."""
    return [random() for i in range(n)]

def random_matrix(n):       # Equivalent to np.random.random((n,n)).tolist()
    """Generate a random nxn matrix as a list of lists."""
    return [[random() for j in range(n)] for i in range(n)]
```

Executing `random_vector(n)` calls `random()` n times, so doubling n should about double the amount of time `random_vector(n)` takes to execute. By contrast, executing `random_matrix(n)` calls `random()` n^2 times (n times per row with n rows). Therefore doubling n will likely more than double the amount of time `random_matrix(n)` takes to execute, especially if n is large.

To visualize this phenomenon, we time `random_matrix()` for $n = 2^1,\ 2^2,\ \ldots,\ 2^{12}$ and plot n against the execution time. The result is displayed below on the left.

```
>>> domain = 2**np.arange(1,13)
>>> times = []
>>> for n in domain:
...     start = time.time()
...     random_matrix(n)
...     times.append(time.time() - start)
...
>>> plt.plot(domain, times, 'g.-', linewidth=2, markersize=15)
>>> plt.xlabel("n", fontsize=14)
>>> plt.ylabel("Seconds", fontsize=14)
>>> plt.show()
```



The figure on the left shows that the execution time for `random_matrix(n)` increases quadratically in n. In fact, the blue dotted line in the figure on the right is the parabola $y = an^2$, which fits nicely over the timed observations. Here a is a small constant, but it is much less significant than the exponent on the n. To represent this algorithm's growth, we ignore a altogether and write `random_matrix(n)` $\sim n^2$.

Note

An algorithm like `random_matrix(n)` whose execution time increases quadratically with n is called $O(n^2)$, notated by `random_matrix(n)` $\in O(n^2)$. Big-oh notation is common for indicating both the *temporal complexity* of an algorithm (how the execution time grows with n) and the *spatial complexity* (how the memory usage grows with n).

Problem 3. Let A be an $m \times n$ matrix with entries a_{ij}, $\mathbf{x}$ be an $n \times 1$ vector with entries x_k, and B be an $n \times p$ matrix with entries b_{ij}. The matrix-vector product $A\mathbf{x} = \mathbf{y}$ is a new $m \times 1$ vector and the matrix-matrix product $AB = C$ is a new $m \times p$ matrix. The entries y_i of $\mathbf{y}$ and c_{ij} of C are determined by the following formulas:

$$y_i = \sum_{k=1}^{n} a_{ik}x_k \qquad\qquad c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

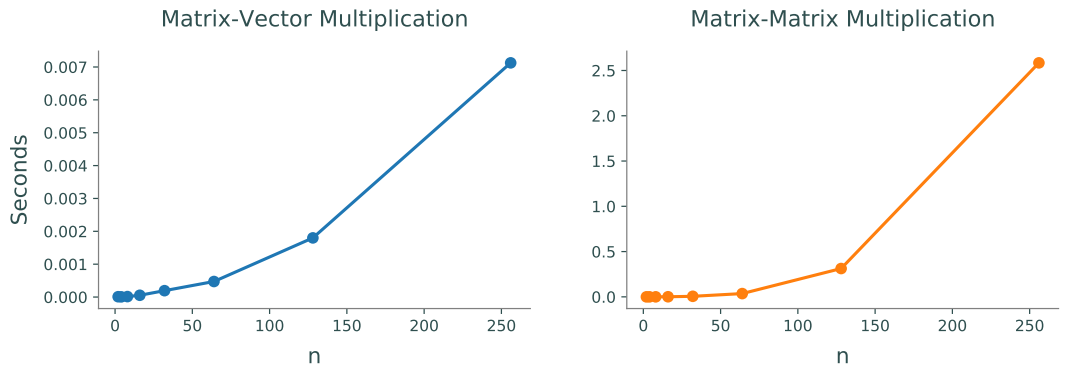
These formulas are implemented below **without** using NumPy arrays or operations.

```
def matrix_vector_product(A, x):    # Equivalent to np.dot(A,x).tolist()
    """Compute the matrix-vector product Ax as a list."""
    m, n = len(A), len(x)
    return [sum([A[i][k] * x[k] for k in range(n)]) for i in range(m)]

def matrix_matrix_product(A, B):    # Equivalent to np.dot(A,B).tolist()
    """Compute the matrix-matrix product AB as a list of lists."""
    m, n, p = len(A), len(B), len(B[0])
    return [[sum([A[i][k] * B[k][j] for k in range(n)])
                                    for j in range(p) ]
                                    for i in range(m) ]
```

Time each of these functions with increasingly large inputs. Generate the inputs A, $\mathbf{x}$, and B with `random_matrix()` and `random_vector()` (so each input will be $n \times n$ or $n \times 1$). Only time the multiplication functions, not the generating functions.

Report your findings in a single figure with two subplots: one with matrix-vector times, and one with matrix-matrix times. Choose a domain for n so that your figure accurately describes the growth, but avoid values of n that lead to execution times of more than 1 minute. Your figure should resemble the following plots.



Logarithmic Plots

Though the two plots from Problem 3 look similar, the scales on the y-axes show that the actual execution times differ greatly. To be compared correctly, the results need to be viewed differently.

A *logarithmic plot* uses a logarithmic scale—with values that increase exponentially, such as 10^1, 10^2, 10^3, ...—on one or both of its axes. The three kinds of log plots are listed below.

- **log-lin**: the x-axis uses a logarithmic scale but the y-axis uses a linear scale.
  Use `plt.semilogx()` instead of `plt.plot()`.
- **lin-log**: the x-axis is uses a linear scale but the y-axis uses a log scale.
  Use `plt.semilogy()` instead of `plt.plot()`.
- **log-log**: both the x and y-axis use a logarithmic scale.
  Use `plt.loglog()` instead of `plt.plot()`.

Since the domain $n = 2^1, 2^2, \ldots$ is a logarithmic scale and the execution times increase quadratically, we visualize the results of the previous problem with a log-log plot. The default base for the logarithmic scales on logarithmic plots in Matplotlib is 10. To change the base to 2 on each axis, specify the keyword arguments `basex=2` and `basey=2`.

Suppose the domain of n values are stored in `domain` and the corresponding execution times for `matrix_vector_product()` and `matrix_matrix_product()` are stored in `vector_times` and `matrix_times`, respectively. Then the following code produces Figure 1.5.

```
>>> ax1 = plt.subplot(121)  # Plot both curves on a regular lin-lin plot.
>>> ax1.plot(domain, vector_times, 'b.-', lw=2, ms=15, label="Matrix-Vector")
>>> ax1.plot(domain, matrix_times, 'g.-', lw=2, ms=15, label="Matrix-Matrix")
>>> ax1.legend(loc="upper left")

>>> ax2 = plot.subplot(122) # Plot both curves on a base 2 log-log plot.
>>> ax2.loglog(domain, vector_times, 'b.-', basex=2, basey=2, lw=2)
>>> ax2.loglog(domain, matrix_times, 'g.-', basex=2, basey=2, lw=2)

>>> plt.show()
```

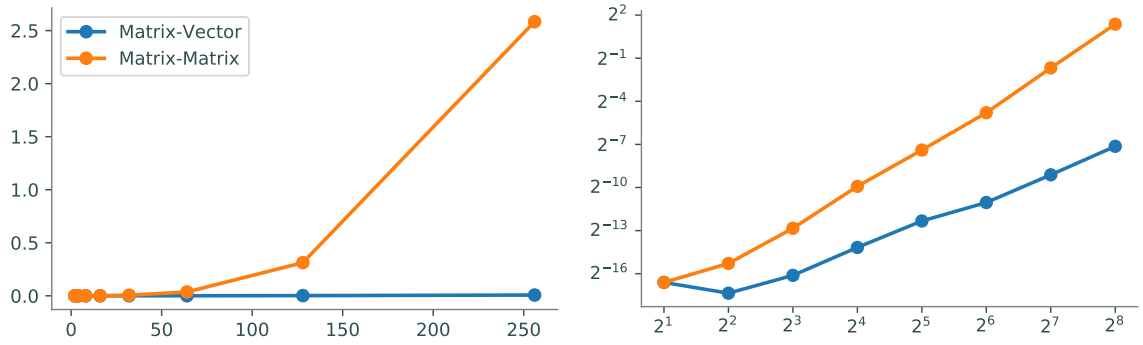


Figure 1.5

In the log-log plot, the slope of the `matrix_matrix_product()` line is about 3 and the slope of the `matrix_vector_product()` line is about 2. This reflects the fact that matrix-matrix multiplication (which uses 3 loops) is $O(n^3)$, while matrix-vector multiplication (which only has 2 loops) is only $O(n^2)$.

Problem 4. NumPy is built specifically for fast numerical computations. Repeat the experiment of Problem 3, timing the following operations:

- matrix-vector multiplication with `matrix_vector_product()`.
- matrix-matrix multiplication with `matrix_matrix_product()`.
- matrix-vector multiplication with `np.dot()` or `@`.
- matrix-matrix multiplication with `np.dot()` or `@`.

Create a single figure with two subplots: one with all four sets of execution times on a regular linear scale, and one with all four sets of execution times on a log-log scale. Compare your results to Figure 1.5.

Note

Problem 4 shows that **matrix operations are significantly faster in NumPy than in plain Python**. Matrix-matrix multiplication grows cubically regardless of the implementation; however, with lists the times grows at a rate of an^3 while with NumPy the times grow at a rate of bn^3, where a is much larger than b. NumPy is more efficient for several reasons:

1. Iterating through loops is very expensive. Many of NumPy's operations are implemented in C, which are much faster than Python loops.
2. Arrays are designed specifically for matrix operations, while Python lists are general purpose.
3. NumPy carefully takes advantage of computer hardware, efficiently using different levels of computer memory.

However, in Problem 4, the execution times for matrix multiplication with NumPy seem to increase somewhat inconsistently. This is because the fastest layer of computer memory can only handle so much information before the computer has to begin using a larger, slower layer of memory.

Additional Material

Image Transformation as a Class

Consider organizing the functions from Problem 1 into a class. The constructor might accept an array or the name of a file containing an array. This structure would makes it easy to do several linear or affine transformations in sequence.

```
>>> horse = ImageTransformer("horse.npy")
>>> horse.stretch(.5, 1.2)
>>> horse.shear(.5, 0)
>>> horse.relect(0, 1)
>>> horse.rotate(np.pi/2.)
>>> horse.translate(.75, .5)
>>> horse.display()
```

Animating Parametrizations

The plot in Problem 2 fails to fully convey the system's evolution over time because time itself is not part of the plot. The following function creates an animation for the earth and moon trajectories.

```
from matplotlib.animation import FuncAnimation

def solar_system_animation(earth, moon):
    """Animate the moon orbiting the earth and the earth orbiting the sun.
    Parameters:
        earth ((2,N) ndarray): The earth's postion with x-coordinates on the
            first row and y coordinates on the second row.
        moon ((2,N) ndarray): The moon's postion with x-coordinates on the
            first row and y coordinates on the second row.
    """
    fig, ax = plt.subplots(1,1)                     # Make a figure explicitly.
    plt.axis([-15,15,-15,15])                       # Set the window limits.
    ax.set_aspect("equal")                          # Make the window square.
    earth_dot,  = ax.plot([],[], 'C0o', ms=10)      # Blue dot for the earth.
    earth_path, = ax.plot([],[], 'C0-')             # Blue line for the earth.
    moon_dot,   = ax.plot([],[], 'C2o', ms=5)       # Green dot for the moon.
    moon_path,  = ax.plot([],[], 'C2-')             # Green line for the moon.
    ax.plot([0],[0],'y*', ms=20)                    # Yellow star for the sun.

    def animate(index):
        earth_dot.set_data(earth[0,index], earth[1,index])
        earth_path.set_data(earth[0,:index], earth[1,:index])
        moon_dot.set_data(moon[0,index], moon[1,index])
        moon_path.set_data(moon[0,:index], moon[1,:index])
        return earth_dot, earth_path, moon_dot, moon_path,
    a = FuncAnimation(fig, animate, frames=earth.shape[1], interval=25)
    plt.show()
```