UNIVERSITY OF CALIFORNIA, BERKELEY
DEPARTMENT OF STATISTICS

Homework #4

Student name: *Shizhe Zhang*

Course: *EECS 227AT* – Professor: *Gireeja Ranade*
Date: *February 19, 2026*

Problem 1 Gradients, Jacobians, and Hessians

The gradient of a scalar-valued function $g : \mathbb{R}^n \to \mathbb{R}$ is the column vector of length n, denoted as ∇g, containing the derivatives of components of g with respect to the input variables:

$$(\nabla g(\vec{x}))_i = \frac{\partial g}{\partial x_i}(\vec{x}), \quad i = 1, \ldots, n. \tag{1}$$

The Hessian of a scalar-valued function $g : \mathbb{R}^n \to \mathbb{R}$ is the $n \times n$ matrix, denoted as $\nabla^2 g$, containing the second derivatives of components of g with respect to the input variables:

$$(\nabla^2 g(\vec{x}))_{ij} = \frac{\partial^2 g}{\partial x_i \partial x_j}(\vec{x}), \quad i = 1, \ldots, n,\ j = 1, \ldots, n. \tag{2}$$

The Jacobian of a vector-valued function $\vec{g} : \mathbb{R}^n \to \mathbb{R}^m$ is the $m \times n$ matrix, denoted as $D\vec{g}$, containing the derivatives of components of $\vec{g}$ with respect to the input variables:

$$(D\vec{g}(\vec{x}))_{ij} = \frac{\partial g_i}{\partial x_j}(\vec{x}), \quad i = 1, \ldots, m,\ j = 1, \ldots, n. \tag{3}$$

For the remainder of the class, we will repeatedly have to take gradients, Hessians and Jacobians of functions we are trying to optimize. This exercise serves as a warm up for future problems.

For the first two parts, suppose $A \in \mathbb{R}^{n \times n}$ is a square matrix whose entries are denoted a_{ij} and whose rows are denoted $\vec{a}_1^\top, \ldots, \vec{a}_n^\top$, and $\vec{b} \in \mathbb{R}^n$ is a vector whose entries are denoted b_i.

(a) Compute the Jacobians for the following functions.

i. $\vec{g}(\vec{x}) = A\vec{x}$.

ii. $\vec{g}(\vec{x}) = f(\vec{x})\vec{x}$ where $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable.

iii. $\vec{g}(\vec{x}) = f(A\vec{x} + \vec{b})\vec{x}$ where $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable.

(b) Compute the gradients and Hessians for the following functions.

i. $g_1(\vec{x}) = \vec{x}^\top A\vec{x}$.

ii. $g_2(\vec{x}) = \|\vec{x}\|_2^2$.

iii. $g_3(\vec{x}) = g_2(A\vec{x} - \vec{b}) = \|A\vec{x} - \vec{b}\|_2^2$. (Use the chain rule and the Jacobians computed in part 1(a).)

iv. $g_4(\vec{x}) = \log(\sum_{i=1}^{n} e^{x_i})$.

v. $g_5(\vec{x}) = g_4(A\vec{x} - \vec{b}) = \log\left(\sum_{i=1}^{n} e^{\vec{a}_i^\top \vec{x} - b_i}\right)$. (Use the chain rule and the Jacobians computed in part 1(a); you can use the gradient ∇g_4 and Hessian $\nabla^2 g_4$ in your answer without having to rewrite it.)

vi. $g_6(\vec{x}) = e^{\|\vec{x}\|_2^2} = e^{g_2(\vec{x})}$. (Use the chain rule and the Jacobians computed in part 1(a).)

vii. $g_7(\vec{x}) = e^{\|A\vec{x} - \vec{b}\|_2^2} = g_6(A\vec{x} - \vec{b})$. (Use the chain rule and the Jacobians computed in part 1(a); you can use the gradient ∇g_6 and Hessian $\nabla^2 g_6$ in your answer without having to rewrite it.)

Consider the case now where all vectors and matrices above are scalar; do your answers above make sense? (No need to answer this in your submission.)

(c) Plot/hand-draw the level sets of the following functions:

i. $g(x_1, x_2) = \frac{x_1^2}{4} + \frac{x_2^2}{9}$

ii. $g(x_1, x_2) = x_1 x_2$

Also point out the gradient directions in the level-set diagram. Additionally, compute the first and second order Taylor series approximation around the point $(1, 1)$ for each function and comment on how accurately they approximate the true function.

Answer.

(a)

(i) Consider $A = \begin{bmatrix} a_1 \\ \vdots \\ a_i \\ \vdots \\ a_n \end{bmatrix}$ where a_i is the i-th row of A. Then

$$g_i(\vec{x}) = a_i^\top \vec{x}$$

$$\frac{\partial g_i}{\partial x_j}(\vec{x}) = a_{ij}$$

Thus,

$$D\vec{g}(\vec{x}) = A.$$

(ii)

$$g_i(\vec{x}) = f(\vec{x})\, x_i,$$

For $i = j$,

$$\frac{\partial g_i}{\partial x_j}(\vec{x}) = \frac{\partial f}{\partial x_j}(\vec{x})\, x_i + f(\vec{x}),$$

And for $i \neq j$,

$$\frac{\partial g_i}{\partial x_j}(\vec{x}) = \frac{\partial f}{\partial x_j}(\vec{x})\, x_i$$

Thus,

$$D\vec{g}(\vec{x}) = f(\vec{x})I + \vec{x}\,\nabla f(\vec{x})^\top.$$

(iii) Let $h(\vec{x}) := f(A\vec{x} + \vec{b})$. Then $\vec{g}(\vec{x}) = h(\vec{x})\vec{x}$.
From problem(ii), we have,

$$D\vec{g}(\vec{x}) = h(\vec{x})I + \vec{x}\,\nabla h(\vec{x})^\top,$$

By the chain rule,

$$\nabla h(\vec{x}) = A^\top \nabla f(A\vec{x} + \vec{b}),$$

so

$$D\vec{g}(\vec{x}) = f(A\vec{x} + \vec{b})I + \vec{x}\left(A^\top \nabla f(A\vec{x} + \vec{b})\right)^\top.$$

(b) Fact,

$$f(\vec{x}) = \beta^\top \vec{x} \implies \nabla f(\vec{x}) = \beta$$

(i) $g_1(\vec{x}) = \vec{x}^\top A\vec{x}$:

$$\nabla g_1(\vec{x}) = (A + A^\top)\vec{x}, \qquad \nabla^2 g_1(\vec{x}) = A + A^\top.$$

(ii) $g_2(\vec{x}) = \|\vec{x}\|_2^2 = \vec{x}^\top \vec{x}$:

$$\nabla g_2(\vec{x}) = 2\vec{x}, \qquad \nabla^2 g_2(\vec{x}) = 2I.$$

(iii) $g_3(\vec{x}) = \|A\vec{x} - \vec{b}\|_2^2$:

$$\nabla g_3(\vec{x}) = 2A^\top(A\vec{x} - \vec{b}), \qquad \nabla^2 g_3(\vec{x}) = 2A^\top A.$$

(iv) $g_4(\vec{x}) = \log\Big(\sum_{i=1}^n e^{x_i}\Big)$.

Let $s(\vec{x}) = \sum_{i=1}^n e^{x_i}$ and $p_i(\vec{x}) = \frac{e^{x_i}}{s(\vec{x})}$.

Then

$$\nabla g_4(\vec{x}) = \vec{p}(\vec{x}),$$

For the Hessian, if $i = j$,

$$(\nabla^2 g_4(\vec{x}))_{ij} = p_i(\vec{x})(1 - p_i(\vec{x})),$$

and if $i \neq j$,

$$(\nabla^2 g_4(\vec{x}))_{ij} = -p_i(\vec{x})p_j(\vec{x}).$$

Thus,

$$\nabla^2 g_4(\vec{x}) = \mathrm{diag}(\vec{p}(\vec{x})) - \vec{p}(\vec{x})\vec{p}(\vec{x})^\top.$$

(v) $g_5(\vec{x}) = g_4(A\vec{x} - \vec{b})$:

$$\nabla g_5(\vec{x}) = A^\top \nabla g_4(A\vec{x} - \vec{b}) = A^\top \vec{p}(A\vec{x} - \vec{b}),$$

$$\nabla^2 g_5(\vec{x}) = A^\top \nabla^2 g_4(A\vec{x} - \vec{b})\, A.$$

(vi) $g_6(\vec{x}) = e^{\|\vec{x}\|_2^2}$. Let $f(\vec{x}) = \|\vec{x}\|_2^2 = \vec{x}^\top \vec{x}$. Thus, $\nabla f = 2\vec{x}$ and $\nabla^2 f = 2I$,

$$\nabla g_6(\vec{x}) = e^{f(\vec{x})} \nabla f(\vec{x}) = 2e^{\|\vec{x}\|_2^2} \vec{x},$$

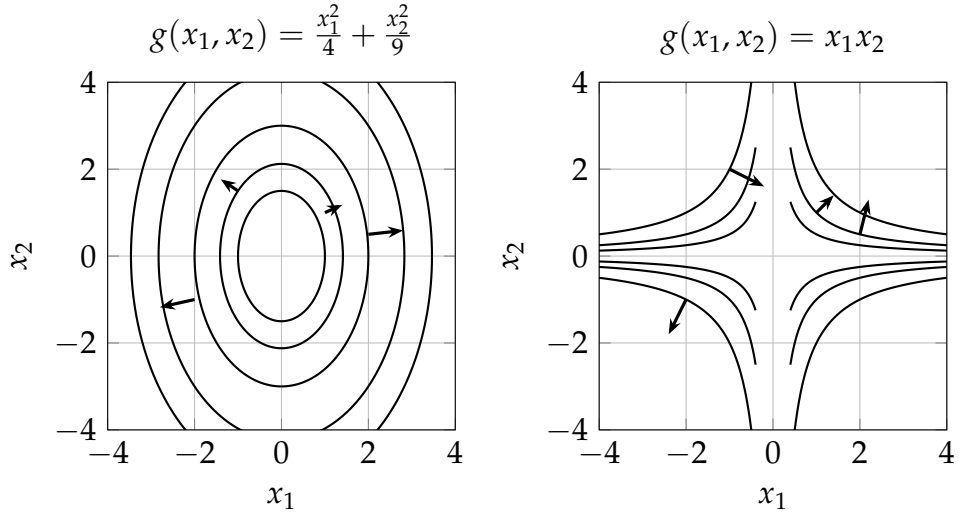
$$\nabla^2 g_6(\vec{x}) = e^{f(\vec{x})} \big(\nabla f(\vec{x}) \nabla f(\vec{x})^\top + \nabla^2 f(\vec{x})\big) = e^{\|\vec{x}\|_2^2} \big(4\vec{x}\vec{x}^\top + 2I\big).$$

(vii) $g_7(\vec{x}) = g_6(A\vec{x} - \vec{b})$.

$$\nabla g_7(\vec{x}) = A^\top \nabla g_6(A\vec{x} - \vec{b})$$

Applying the result in 1(a),

$$\nabla^2 g_7(\vec{x}) = A^\top \nabla^2 g_6(A\vec{x} - \vec{b})\, A$$



(c)

(i) $g(x_1, x_2) = \frac{x_1^2}{4} + \frac{x_2^2}{9}$. Level sets $\{(x_1, x_2) : \frac{x_1^2}{4} + \frac{x_2^2}{9} = c\}$ are ellipses centered at $(0, 0)$.

$$\nabla g(x_1, x_2) = \begin{bmatrix} \frac{x_1}{2} \\ \frac{2x_2}{9} \end{bmatrix}, \qquad \nabla^2 g(x_1, x_2) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{2}{9} \end{bmatrix}.$$

At $(1, 1)$:

$$g(1, 1) = \frac{1}{4} + \frac{1}{9} = \frac{13}{36}, \qquad \nabla g(1, 1) = \begin{bmatrix} \frac{1}{2} \\ \frac{2}{9} \end{bmatrix}.$$

First-order Taylor at $(1, 1)$:

$$T_1(x_1, x_2) = \frac{13}{36} + \frac{1}{2}(x_1 - 1) + \frac{2}{9}(x_2 - 1) = -\frac{13}{36} + \frac{1}{2}x_1 + \frac{2}{9}x_2.$$

Second-order Taylor at $(1,1)$:

$$T_2(x_1,x_2) = T_1(x_1,x_2) + \frac{1}{2}\begin{bmatrix} x_1-1 & x_2-1 \end{bmatrix}\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{2}{9} \end{bmatrix}\begin{bmatrix} x_1-1 \\ x_2-1 \end{bmatrix}.$$

Since g is quadratic, $T_2 = g$; T_1 is a local linear approximation.

(ii) $g(x_1,x_2) = x_1x_2$. Level sets $\{(x_1,x_2) : x_1x_2 = c\}$ are rectangular hyperbolas.

$$\nabla g(x_1,x_2) = \begin{bmatrix} x_2 \\ x_1 \end{bmatrix}, \qquad \nabla^2 g(x_1,x_2) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

At $(1,1)$:

$$g(1,1) = 1, \qquad \nabla g(1,1) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

First-order Taylor at $(1,1)$:

$$T_1(x_1,x_2) = 1 + (x_1-1) + (x_2-1) = x_1 + x_2 - 1.$$

Second-order Taylor at $(1,1)$:

$$\begin{aligned} T_2(x_1,x_2) &= T_1(x_1,x_2) + \frac{1}{2}\cdot 2(x_1-1)(x_2-1) \\ &= 1 + (x_1-1) + (x_2-1) + (x_1-1)(x_2-1) \\ &= x_1x_2. \end{aligned}$$

Again, T_2 is exact because g is a degree-2 polynomial; T_1 is only locally accurate.

Problem 2 Neural Networks and Backpropagation

Neural networks are parametric functions that have been widely used to fit complex patterns in vision and natural languages. Given some training data of the form $(\vec{x}_i, y_i)$, a neural network N is trained to minimize a loss function on the data. This is often done using gradient descent, an optimization method we will cover later in this class. Gradient descent requires us to compute the gradients of the loss function with respect to the parameters of the neural network. In practice, computational frameworks for neural networks compute the gradients automatically and efficiently via back-propagation, which uses the chain rule to recursively compute the gradients of the loss function. In this problem, we study a toy neural network trained on a single data point $(\vec{x}, y)$.

In particular, consider the following simplified three-layer neural network N, representing a map from $\mathbb{R}^d$ to $\mathbb{R}$ whose parameters are $(\vec{w}_1, w_2, w_3) \in \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}$:

$$p_1 = \vec{w}_1^\top \vec{x} \tag{4}$$
$$h_1 = \sigma(p_1) \tag{5}$$
$$p_2 = w_2 h_1 \tag{6}$$
$$h_2 = \sigma(p_2) \tag{7}$$
$$z = w_3 h_2 \tag{8}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is a nonlinear function (also called the activation function), whose derivative is denoted by $\sigma' : \mathbb{R} \to \mathbb{R}$.

We want the output of the network $z = N(\vec{x})$ to match the true label y. A natural choice of the loss function encouraging this behavior is the squared loss:

$$L(y, z) \stackrel{\text{def}}{=} \frac{1}{2}(y - z)^2. \tag{9}$$

In the parts that follow, we will compute the derivative of L with respect to the parameters $\vec{w}_1, w_2, w_3$.

(a) Compute the following gradients and partial derivatives sequentially from left-to-right:

$$\frac{\partial L}{\partial z}, \frac{\partial L}{\partial w_3}, \frac{\partial L}{\partial h_2}, \frac{\partial L}{\partial p_2}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial h_1}, \frac{\partial L}{\partial p_1}, \nabla_{\vec{w}_1} L, \nabla_{\vec{x}} L. \tag{10}$$

Here $\nabla_{\vec{x}} L$ is the gradient whose entries are the derivatives of L with respect to the entries of $\vec{x}$, etc. We compute the first 4 derivatives for you.

$$\frac{\partial L}{\partial z} = z - y, \frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial z} h_2, \frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial z} w_3, \frac{\partial L}{\partial p_2} = \frac{\partial L}{\partial h_2} \sigma'(p_2). \tag{11}$$

Note how $\frac{\partial L}{\partial w_3}$ can be calculated using $\frac{\partial L}{\partial z}$ and $\frac{\partial L}{\partial p_2}$ can be calculated using $\frac{\partial L}{\partial h_2}$. In numerical computation, the result of $\frac{\partial L}{\partial z}$ and $\frac{\partial L}{\partial h_2}$ can thus be reused. This technique of saving computations for calculating the derivatives of a neural network is called back-propagation.

Use the chain rule to calculate the 5 remaining derivatives $\frac{\partial L}{\partial w_2}$, $\frac{\partial L}{\partial h_1}$, $\frac{\partial L}{\partial p_1}$, $\nabla_{\vec{w}_1} L$, and $\nabla_{\vec{x}} L$.

(b) Now suppose that Equation (8) is written as

$$z' = w_3 h_2 + h_1. \tag{12}$$

That is, we define a new neural network N' with parameters $(\vec{w}_1, w_2, w_3) \in \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}$ as follows:

$$p_1 = \vec{w}_1^\top \vec{x} \tag{13}$$
$$h_1 = \sigma(p_1) \tag{14}$$
$$p_2 = w_2 h_1 \tag{15}$$
$$h_2 = \sigma(p_2) \tag{16}$$
$$z' = w_3 h_2 + h_1. \tag{17}$$

This introduces a change in the network architecture, called the skip connection.

Again, compute the following gradients and partial derivatives with respect to the loss function

$$L(y, z') = \frac{1}{2}(y - z')^2 :$$

$$\frac{\partial L}{\partial z'}, \frac{\partial L}{\partial w_3}, \frac{\partial L}{\partial h_2}, \frac{\partial L}{\partial p_2}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial h_1}, \frac{\partial L}{\partial p_1}, \nabla_{\vec{w}_1} L, \nabla_{\vec{x}} L. \tag{18}$$

We compute the first 4 derivatives for you.

$$\frac{\partial L}{\partial z'} = z' - y, \tag{1}$$
$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial z'} h_2, \tag{2}$$
$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial z'} w_3, \tag{3}$$
$$\frac{\partial L}{\partial p_2} = \frac{\partial L}{\partial h_2} \sigma'(p_2). \tag{19}$$

Use the chain rule to calculate the 5 remaining derivatives $\frac{\partial L}{\partial w_2}$, $\frac{\partial L}{\partial h_1}$, $\frac{\partial L}{\partial p_1}$, $\nabla_{\vec{w}_1} L$, and $\nabla_{\vec{x}} L$.

(c) In optimizing a neural network using gradient descent, we need the gradient of the loss function with respect to the parameters of the network. Please express $\frac{\partial L}{\partial w_3}$, $\frac{\partial L}{\partial w_2}$, and $\nabla_{\vec{w}_1} L$ for N and N' respectively with no dependence on partial derivatives of other variables. We compute $\frac{\partial L}{\partial w_3}$ for you, as follows.

- For N, we have $\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial z} h_2 = (z - y) h_2$.
- For N', we have $\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial z'} h_2 = (z' - y) h_2$.

Express the remaining derivatives $\frac{\partial L}{\partial w_2}$ and $\nabla_{\vec{w}_1} L$ within N and N'.

(d) Many activation functions σ have the property that $\sigma' \leq 1$. For example, the sigmoid function

$$\sigma(p) = \frac{1}{1+e^{-p}}$$

is sometimes used as an activation function. Its derivative,

$$\sigma'(p) = \frac{e^{-p}}{(1+e^{-p})^2}$$

has the range $(0, 1/4]$. That is, $\sigma' < 1$.

Consider the case when σ' is much smaller than 1, such that $\sigma'(p)\sigma'(q) \approx 0$ for any p, q, but $\sigma'(p) \not\approx 0$ for any p. Consider the derivatives $\frac{\partial L}{\partial w_3}$, $\frac{\partial L}{\partial w_2}$, and $\nabla_{\vec{w}_1} L$ within the neural network N; with the above approximations, which of them will approximately be zero? Also answer this question for the neural network N'.

NOTE: Some of the above gradients will indeed be approximately zero, and this is called the vanishing gradient problem in deep learning.

Answer.

(a)

$$\begin{aligned} p_1 &= \vec{w}_1^\top \vec{x}, \\ h_1 &= \sigma(p_1), \\ p_2 &= w_2 h_1, \\ h_2 &= \sigma(p_2), \\ z &= w_3 h_2, \end{aligned}$$

$$L(y, z) = \frac{1}{2}(y-z)^2.$$

$$\begin{aligned} \frac{\partial L}{\partial z} &= z - y, \\ \frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_3} = (z-y)\, h_2, \\ \frac{\partial L}{\partial h_2} &= \frac{\partial L}{\partial z}\frac{\partial z}{\partial h_2} = (z-y)\, w_3, \\ \frac{\partial L}{\partial p_2} &= \frac{\partial L}{\partial h_2}\frac{\partial h_2}{\partial p_2} = (z-y)\, w_3\, \sigma'(p_2). \end{aligned}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial p_2}\frac{\partial p_2}{\partial w_2} = \frac{\partial L}{\partial p_2}\, h_1 = (z-y)\, w_3\, \sigma'(p_2)\, h_1.$$

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial p_2}\frac{\partial p_2}{\partial h_1} = \frac{\partial L}{\partial p_2}\, w_2 = (z-y)\, w_3\, \sigma'(p_2)\, w_2.$$

$$\frac{\partial L}{\partial p_1} = \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial p_1} = \frac{\partial L}{\partial h_1}\,\sigma'(p_1) = (z-y)\,w_3\,\sigma'(p_2)\,w_2\,\sigma'(p_1).$$

$\nabla_{\vec{w}_1} L$. Since $p_1 = \vec{w}_1^{\top}\vec{x}$, we have $\nabla_{\vec{w}_1} p_1 = \vec{x}$, hence

$$\nabla_{\vec{w}_1} L = \frac{\partial L}{\partial p_1}\,\nabla_{\vec{w}_1} p_1 = \frac{\partial L}{\partial p_1}\,\vec{x} = (z-y)\,w_3\,\sigma'(p_2)\,w_2\,\sigma'(p_1)\,\vec{x}.$$

$\nabla_{\vec{x}} L$. Since $\nabla_{\vec{x}} p_1 = \vec{w}_1$,

$$\nabla_{\vec{x}} L = \frac{\partial L}{\partial p_1}\,\nabla_{\vec{x}} p_1 = \frac{\partial L}{\partial p_1}\,\vec{w}_1 = (z-y)\,w_3\,\sigma'(p_2)\,w_2\,\sigma'(p_1)\,\vec{w}_1.$$

(b)

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial p_2}\frac{\partial p_2}{\partial w_2} = \frac{\partial L}{\partial p_2}\,h_1 = (z'-y)\,w_3\,\sigma'(p_2)\,h_1.$$

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial z'}\frac{\partial z'}{\partial h_1} + \frac{\partial L}{\partial p_2}\frac{\partial p_2}{\partial h_1}.$$

Since $\frac{\partial z'}{\partial h_1} = 1$ and $\frac{\partial p_2}{\partial h_1} = w_2$, we get

$$\frac{\partial L}{\partial h_1} = (z'-y)\cdot 1 + \big((z'-y)\,w_3\,\sigma'(p_2)\big)\,w_2 = (z'-y)\Big(1 + w_2 w_3 \sigma'(p_2)\Big).$$

$$\frac{\partial L}{\partial p_1} = \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial p_1} = \frac{\partial L}{\partial h_1}\,\sigma'(p_1) = (z'-y)\Big(1 + w_2 w_3 \sigma'(p_2)\Big)\sigma'(p_1).$$

$$\nabla_{\vec{w}_1} L = \frac{\partial L}{\partial p_1}\,\nabla_{\vec{w}_1} p_1 = \frac{\partial L}{\partial p_1}\,\vec{x} = (z'-y)\Big(1 + w_2 w_3 \sigma'(p_2)\Big)\sigma'(p_1)\,\vec{x}.$$

$$\nabla_{\vec{x}} L = \frac{\partial L}{\partial p_1}\,\nabla_{\vec{x}} p_1 = \frac{\partial L}{\partial p_1}\,\vec{w}_1 = (z'-y)\Big(1 + w_2 w_3 \sigma'(p_2)\Big)\sigma'(p_1)\,\vec{w}_1.$$

(c)

Network N.

$$\frac{\partial L}{\partial w_2} = (z-y)\,w_3\,\sigma'(p_2)\,h_1, \qquad \nabla_{\vec{w}_1} L = (z-y)\,w_3\,\sigma'(p_2)\,w_2\,\sigma'(p_1)\,\vec{x}.$$

Network N'.

$$\frac{\partial L}{\partial w_2} = (z'-y)\,w_3\,\sigma'(p_2)\,h_1, \qquad \nabla_{\vec{w}_1} L = (z'-y)\Big(1 + w_2 w_3 \sigma'(p_2)\Big)\sigma'(p_1)\,\vec{x}.$$

(d)

Network N. From part (c),

$$\frac{\partial L}{\partial w_3} = (z-y)h_2, \qquad \frac{\partial L}{\partial w_2} = (z-y)\, w_3\, \sigma'(p_2)\, h_1, \qquad \nabla_{\vec{w}_1} L = (z-y)\, w_3\, \sigma'(p_2)\, w_2\, \sigma'(p_1)\, \vec{x}.$$

Under the approximation $\sigma'(p_2)\sigma'(p_1) \approx 0$, we have

$$\nabla_{\vec{w}_1} L \approx \vec{0}.$$

$$\frac{\partial L}{\partial w_2} \not\approx 0, \qquad \frac{\partial L}{\partial w_3} \not\approx 0$$

Network N'. From part (c),

$$\frac{\partial L}{\partial w_3} = (z'-y)h_2, \qquad \frac{\partial L}{\partial w_2} = (z'-y)\, w_3\, \sigma'(p_2)\, h_1,$$

$$\nabla_{\vec{w}_1} L = (z'-y)\Big(1 + w_2 w_3 \sigma'(p_2)\Big)\sigma'(p_1)\, \vec{x}.$$

$$\nabla_{\vec{w}_1} L = (z'-y)\sigma'(p_1)\vec{x} \;+\; (z'-y)\, w_2 w_3\, \sigma'(p_2)\sigma'(p_1)\, \vec{x}.$$

$$\nabla_{\vec{w}_1} L \approx (z'-y)\sigma'(p_1)\vec{x},$$

which is *not* approximately zero in general (since $\sigma'(p_1) \not\approx 0$). As in N, $\frac{\partial L}{\partial w_2}$ has only one σ' factor and $\frac{\partial L}{\partial w_3}$ has none, so none of them will be approximately zero in general.

Problem 3 PCA and Senate Voting Data

In this problem, we consider a matrix of senate voting data, which we manipulate in Python. The data is contained in a $n \times d$ data matrix X, where each row corresponds to a senator and each column to a bill. Each entry of X is either 1, -1 or 0, depending on whether the senator voted for the bill, against the bill, or abstained, respectively.

(a) Suppose we want to assign a score to each senator based on their voting pattern, and then observe the empirical variance of these scores. To describe this, let us choose a $\vec{a} \in \mathbb{R}^d$ and a scalar $b \in \mathbb{R}$. We define the score for senator i as:

$$f(\vec{x}_i, \vec{a}, b) = \vec{x}_i^\top \vec{a} + b, \quad i = 1, 2, \ldots, n. \tag{20}$$

Note that $\vec{x}_i^\top$ denotes the ith row of X and is a row vector of length d, as in the previous problem.

Let us denote by $\vec{z} = f(X, \vec{a}, b)$ the column vector of length n obtained by stacking the scores for each senator. Then

$$\vec{z} = f(X, \vec{a}, b) = X\vec{a} + b\vec{1} \in \mathbb{R}^n \tag{21}$$

where $\vec{1}$ is a vector with all entries equal to 1. Let us denote the mean value of $\vec{z}$ by $\mu(\vec{z}) = \frac{1}{n}\vec{1}^\top \vec{z}$. Let $\vec{\mu}(X) \in \mathbb{R}^d$ denote the vector containing the mean of each column of X. Then

$$\begin{aligned}
\mu(\vec{z}) &= \frac{1}{n}\sum_{i=1}^{n} z_i && (22)\\
&= \frac{1}{n}\sum_{i=1}^{n} (\vec{a}^\top \vec{x}_i + b) && (23)\\
&= \vec{a}^\top \left(\frac{1}{n}\sum_{i=1}^{n} \vec{x}_i\right) + b && (24)\\
&= \vec{a}^\top \vec{\mu}(X) + b. && (25)
\end{aligned}$$

The empirical variance of the scores can then be obtained as

$$\sigma^2(\vec{z}) = \frac{1}{n}(\vec{z} - \mu(\vec{z})\vec{1})^\top(\vec{z} - \mu(\vec{z})\vec{1}) \tag{26}$$

$$= \frac{1}{n}((X\vec{a} + b\vec{1}) - (\vec{a}^\top\vec{\mu}(X) + b)\vec{1})^\top((X\vec{a} + b\vec{1}) - (\vec{a}^\top\vec{\mu}(X) + b)\vec{1}) \tag{27}$$

$$= \frac{1}{n}(X\vec{a} + b\vec{1} - (\vec{a}^\top\vec{\mu}(X))\vec{1} - b\vec{1})^\top(X\vec{a} + b\vec{1} - (\vec{a}^\top\vec{\mu}(X))\vec{1} - b\vec{1}) \tag{28}$$

$$= \frac{1}{n}(X\vec{a} - (\vec{a}^\top\vec{\mu}(X))\vec{1})^\top(X\vec{a} - (\vec{a}^\top\vec{\mu}(X))\vec{1}) \tag{29}$$

$$= \frac{1}{n}(X\vec{a} - \vec{1}\vec{\mu}(X)^\top\vec{a})^\top(X\vec{a} - \vec{1}\vec{\mu}(X)^\top\vec{a}) \tag{30}$$

$$= \frac{1}{n}((X - \vec{1}\vec{\mu}(X)^\top)\vec{a})^\top((X - \vec{1}\vec{\mu}(X)^\top)\vec{a}) \tag{31}$$

$$= \frac{1}{n}\vec{a}^\top(X - \vec{1}\vec{\mu}(X)^\top)^\top(X - \vec{1}\vec{\mu}(X)^\top)\vec{a}. \tag{32}$$

Note that this variance is therefore a function of the centered data matrix $X - \vec{1}\vec{\mu}(X)^\top$ in which the mean of each column is zero. It also does not depend on b.

For the remainder of this problem, we assume that the data has been pre-centered (i.e., $\vec{\mu}(X) = \vec{0}$); note that this has been pre-computed for you in the code notebook. Assume also that $b = 0$, so that $\mu(\vec{z}) = 0$. Defining $f(X, \vec{a}) \stackrel{\text{def}}{=} f(X, \vec{a}, 0)$ and replacing $\vec{z}$ with $f(X, \vec{a})$, we can then write the simpler empirical variance formula

$$\sigma^2(f(X, \vec{a})) = \frac{1}{n}\vec{a}^\top X^\top X\vec{a}. \tag{33}$$

Suppose we restrict $\vec{a}$ to have unit-norm. In the provided code, find the maximum empirical variance $\sigma^2(f(X, \vec{a}))$ over all unit-norm $\vec{a}$, and find the $\vec{a}$ that maximizes it.

(b) We next consider party affiliation as a predictor for how a senator will vote. Follow the instructions in the notebook to compute the mean voting vector for each party and relate it to the direction of maximum variance.

(c) Recall from problem 1 that given a vector $\vec{z} = X\vec{u}$ (i.e., the vector of scalar projections of each row of X along $\vec{u}$), we can compute its empirical variance as

$$\sigma^2(\vec{z}) = \vec{u}^\top\Sigma\vec{u}, \tag{34}$$

where $\Sigma(X) = \frac{1}{n}X^\top X$ is the empirical covariance matrix of X. We will show in a future homework problem that the variance along each principal component $\vec{a}_i$ is precisely its corresponding eigenvalue of $\Sigma(X)$, i.e., $\lambda_i\{\Sigma(X)\}$. (For now, just note that this fact should make intuitive sense, since PCA is searching for directions of maximum variance of the data, and these occur along the covariance matrixs eigenvectors.)

In the Notebook, compute the sum of the variance along $\vec{a}_1$ and $\vec{a}_2$ and plot the data projected on the $\vec{a}_1\vec{a}_2$ plane.

(d) Suppose we want to find the bills that are most and least contentious i.e., those that have high variability in senators votes, and those for which voting was almost unanimous. Follow the instructions in the notebook to compute a measure of contentiousness for each bill, plot the vote counts for exemplar bills, and comment on the voting trends.

(e) Suppose we want to infer the political affiliations of two senators whose voting records are known to us. Follow the instructions in the notebook to infer the political affiliation of the Green and Grey colored senators using PCA.

(f) Finally, we can use the defined score $f(X, \vec{a}, b)$, computed along the first principal component $\vec{a}_1$ to classify the most and least extreme senators based on their voting record. Follow the instructions in the Notebook to compute these scores and comment on their relationship to partisan affiliation.

Answer.

(a)

$$\sigma^2(\vec{z}) = \frac{1}{n}\vec{a}^\top X^\top X\vec{a} = \vec{a}^\top \Sigma(X)\vec{a}, \qquad \Sigma(X) \stackrel{\text{def}}{=} \frac{1}{n}X^\top X.$$

Maximizing variance over $\|\vec{a}\|_2 = 1$ is the Rayleigh quotient problem

$$\max_{\|\vec{a}\|_2=1} \vec{a}^\top \Sigma(X)\vec{a}.$$

Let $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$ be the eigenvalues of $\Sigma(X)$ with orthonormal eigenvectors $\{\vec{a}_1, \ldots, \vec{a}_d\}$.

$$\max_{\|\vec{a}\|_2=1} \vec{a}^\top \Sigma(X)\vec{a} = \lambda_1, \quad \text{achieved by } \vec{a} = \pm\vec{a}_1.$$

(b)-(f) c.f. notebook.

Problem 4 SVD Transformation

In this problem we will interpret the linear map defined by matrix $A \in \mathbb{R}^{n\times n}$ by looking at its singular value decomposition, $A = UDV^\top$. Recall that here $U, D, V \in \mathbb{R}^{n\times n}$ and U, V are orthonormal matrices while D is a diagonal matrix. We will first look at how $V^\top$, D and U each separately transform the unit circle

$$C = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1\}$$

and then look at their effect as a whole.

(a) Let $\vec{x} \in \mathbb{R}^n$ and $\vec{z} = V^\top \vec{x}$. Show that

$$\vec{x} = \sum_{i=1}^{n} z_i \vec{v}_i,$$

where $\vec{z} = [z_1 \ \cdots \ z_n]^\top$ and $V = [\vec{v}_1 \ \cdots \ \vec{v}_n]$. This shows that $\vec{z}$ is the vector of coordinates that represents $\vec{x}$ in the basis defined by the columns of V.

For the rest of the problem we restrict ourselves to the case where $A \in \mathbb{R}^{2\times 2}$ and move to the notebook.

Answer. Since V is an orthonormal matrix, we have $V^\top V = I$. Therefore,

$$\vec{z} = V^\top \vec{x} \implies V\vec{z} = VV^\top \vec{x} = \vec{x}.$$

where $V\vec{z} = z_1\vec{v}_1 + \cdots + z_n\vec{v}_n$ is the linear combination of the columns of V with coefficients given by the entries of $\vec{z}$.

Problem 5 Homework Process

With whom did you work on this homework? List the names and SIDs of your group members.
NOTE: If you didnt work with anyone, you can put "none" as your answer.

Answer. none

senator_pca

February 19, 2026

1 PCA and senate voting data

In this problem, we are given the $n \times d$ data matrix X with entries in $\{-1, 0, 1\}$, where each row corresponds to a senator and each column to a bill. We first import this data, print some relevant values, and normalize it as necessary to ready it for further computation.

1.0.1 Places you will need to modify this code are enclosed in a `#TODO` block. You should not need to modify code outside these blocks to complete the problems. Questions that you are expected to answer in text are marked in red. For solution files, solutions will be presented in blue.

[1]:
```
# import the necessary packages for data manipulation, computation and PCA
import pandas as pd
import numpy as np
import scipy as sp
from numpy import linalg as LA
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
%matplotlib inline

np.random.seed(7)
```

[2]:
```
# import the data matrix
!gdown 1De3PwqrmFsMyKOVeejMs4PSoRiJIS8D1 -O senator_pca_data_matrix.csv
!gdown 1uYVNV1ZV7wx_4KiSFxyRZPbG2P5r4bMB -O senator_pca_politician_labels.txt
senator_df =  pd.read_csv('senator_pca_data_matrix.csv')
affiliation_file = open('senator_pca_politician_labels.txt', 'r')
affiliations = [line.split('\n')[0].split(' ')[1] for line in affiliation_file.
 ↪readlines()]
X = np.array(senator_df.values[:, 3:].T, dtype='float64') # transpose to get␣
 ↪senators as rows
print('X.shape: ', X.shape)
n = X.shape[0] # number of senators
d = X.shape[1] # number of bills

# this is just used for plotting, feel free to ignore
assert set(affiliations) == {"Red", "Blue", "Yellow"}
# assign a marker and hatch to each affiliation
```

```
markers = [("Red", "o", "/"), ("Blue", "^", "-"), ("Yellow", "D", "+")]
```

```
Downloading…
From: https://drive.google.com/uc?id=1De3PwqrmFsMyKOVeejMs4PSoRiJIS8D1
To: /content/senator_pca_data_matrix.csv
100% 270k/270k [00:00<00:00, 30.2MB/s]
Downloading…
From: https://drive.google.com/uc?id=1uYVNV1ZV7wx_4KiSFxyRZPbG2P5r4bMB
To: /content/senator_pca_politician_labels.txt
100% 1.18k/1.18k [00:00<00:00, 4.19MB/s]
X.shape:  (100, 542)
```

We observe that the number of rows, n, is the number of senators and is equal to 100. The number of columns, d, is the number of bills and is equal to 542.

[3]:
```
# print an example row of the data matrix
typical_row = X[0]
print(typical_row.shape)
print(typical_row)
```

```
(542,)
[ 1.  1.  1. -1. -1.  1.  1.  1.  1. -1.  1. -1. -1.  1.  1. -1.  1.  1.
  1.  1.  1. -1.  1.  1.  1. -1.  1. -1.  1.  1.  1.  1.  1. -1.  1. -1.
 -1. -1. -1.  1.  1. -1. -1. -1. -1.  1.  1.  1. -1.  1.  1. -1.  1.  1.
 -1.  1.  1.  1.  1. -1.  1. -1. -1. -1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1. -1.  0. -1.  1.  1.  1. -1. -1.  1.  1. -1. -1.  1.  1.  1. -1.
  1. -1.  1. -1.  1.  1. -1. -1. -1.  1.  1.  1. -1. -1. -1. -1. -1. -1.
  1. -1.  1.  1. -1. -1. -1.  1. -1.  1. -1.  1.  0.  0.  1.  1. -1.  1.
  1. -1.  1.  1. -1.  1. -1. -1.  1.  1.  1.  1.  0. -1. -1.  1.  1. -1.
  1.  1.  1.  1.  1.  0.  1.  0.  1.  1.  1.  1.  1.  1.  1. -1.  1.  1.
 -1.  1. -1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. -1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1. -1. -1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1. -1.  1.  1.  1.  1.  1.  1.  1.  1. -1.
  1.  1.  0.  1.  0. -1.  1.  1.  1.  1.  1.  1. -1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1. -1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. -1.  1.  1.  1. -1.
  1.  1.  1.  1.  1.  1. -1. -1. -1.  1.  1. -1.  1. -1. -1.  1.  1.  1.
 -1.  1.  1.  1. -1.  1. -1.  1. -1. -1.  1. -1. -1.  1.  1.  1. -1.  1.
  1.  1.  1.  1. -1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. -1. -1.
  1. -1.  1. -1. -1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1. -1.  1.  1.  1.  1.  1. -1.  1. -1.
  1.  1.  1.  1.  1.  1. -1.  1. -1. -1. -1. -1.  1.  1.  1.  1.  1.  1.
  1.  1.  1. -1.  1.  1.  1.  1.  1.  1.  1. -1. -1.  1. -1.  1.  1.  1.
  1.  1. -1.  1. -1.  1. -1.  1.  1. -1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  0.  1. -1.  1.  1.  1.  1.  1. -1. -1. -1.  1.  1.  0.  1.  1.  1.
  1.  1.  1.  1. -1. -1.  0.  0.  0.  0.  0.  0.  0.  1.  1. -1. -1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1. -1.  1. -1.  1.  1.  1.  1. -1. -1.
  1.  1.  1.  1. -1. -1.  1.  1. -1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. -1.  1.  1.  1.
```

```
 -1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1. -1. -1. -1.  1.  1.  1.  1. -1. -1.  1.  1. -1.  1.  1.  1.  1.
  1.  1.]
```

A row of X consists of 542 entries -1 (senator voted against), 1 (senator voted for), or 0 (senator abstained), one for each bill.

[4]:
```
# print an example column of the data matrix
typical_column = X[:,0]
print(typical_column.shape)
print(typical_column)
```

```
(100,)
[ 1.  1.  1.  1.  1.  1.  1. -1.  1. -1.  1. -1.  1. -1. -1. -1.  1.  1.
 -1.  1.  1. -1.  1. -1.  1.  1.  1. -1. -1.  1.  1.  1. -1.  1.  1.  1.
 -1. -1. -1. -1.  1. -1. -1.  1.  1. -1. -1. -1. -1. -1.  1.  1. -1. -1.
  1.  1. -1. -1. -1. -1. -1.  1.  1.  1.  1.  1. -1. -1. -1.  1. -1. -1.
  1. -1. -1.  1.  1.  1. -1. -1. -1.  1.  1. -1.  1. -1.  1.  1.  1. -1.
 -1. -1. -1. -1.  1.  1.  1. -1. -1. -1.]
```

A column of X consists of 100 entries in {-1, 0, 1}, one for each senator that voted on the bill.

[ ]:
```
# compute the mean vote on each bill
X_mean = np.mean(X, axis = 0)
plt.plot(X_mean)
plt.title('means of each column of X')
plt.xlabel('column/bill')
plt.ylabel('mean vote')
plt.show()
```

We observe that the mean of the columns is not zero, so we center the data by subtracting the mean of each bill's vote from its respective column.

[10]:
```
# center the data matrix
X_original = X.copy() # save a copy for part (d) and (e)
X = X - np.mean(X, axis = 0)
```

```
107.5
```

1.1 a) Maximizing $\sigma^2(f(X, \vec{a}))$

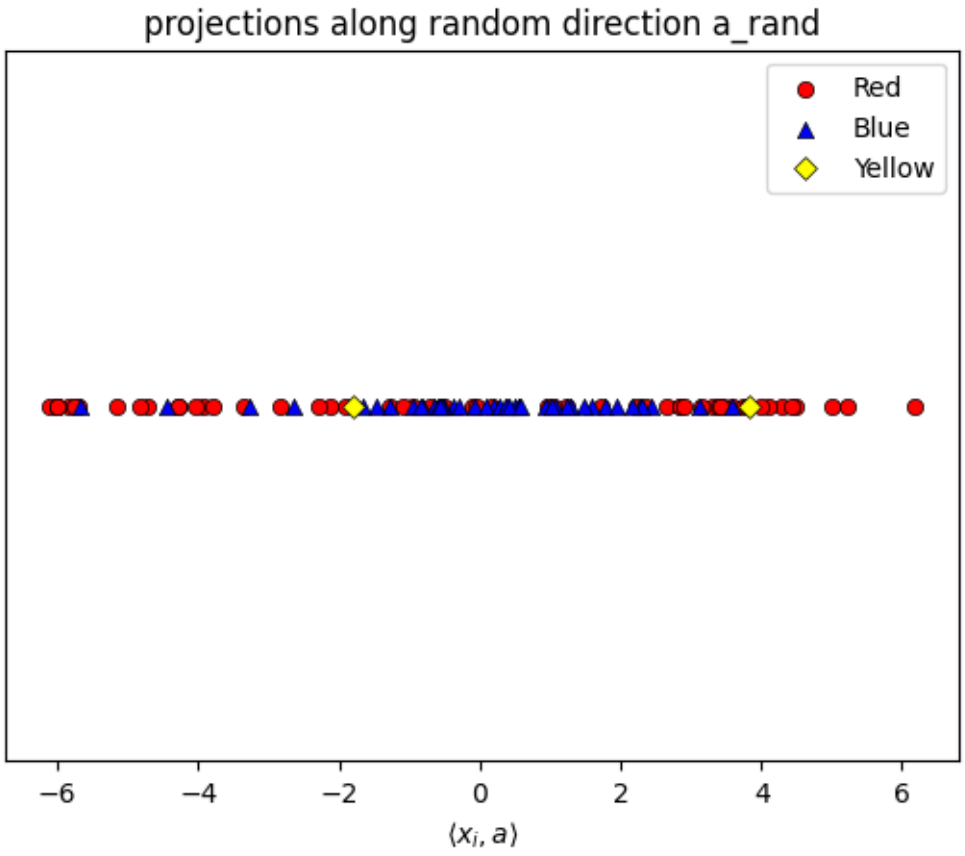
In this problem, you are asked to find a unit-norm vector $\vec{a} \in \mathbb{R}^d$ maximizing the empirical variance $\sigma^2(f(X, \vec{a}))$.

We first provide a function to calculate the scores, $f(X, \vec{a})$.

[22]:
```
# define score function
def f(X, a):
    return X @ a
```

Before we calculate the $\vec{a}$ that maximizes variance, let's observe what the scalar projections on a random direction $\vec{a}$ look like.

[12]:
```
# generate a random direction and normalize the vector
a_rand = np.random.rand(d)
a_rand = a_rand/np.linalg.norm(a_rand)

# compute associated scores along a_rand
scores_rand = f(X, a_rand)

# visualize the scores along a_rand, coloring them by party affiliation
for aff, marker, _ in markers:
    plt.scatter(
        scores_rand[np.array(affiliations) == aff],
        np.zeros_like(scores_rand[np.array(affiliations) == aff]),
        c=aff, marker=marker, edgecolors="black", linewidth=0.5, label=aff
    )
plt.legend()
plt.title('projections along random direction a_rand')
plt.xlabel('$\\langle x_i, a\\rangle$')
cur_axes = plt.gca()
cur_axes.axes.get_yaxis().set_visible(False)
plt.show()

print('variance along random direction a_rand: ', scores_rand.var())
```

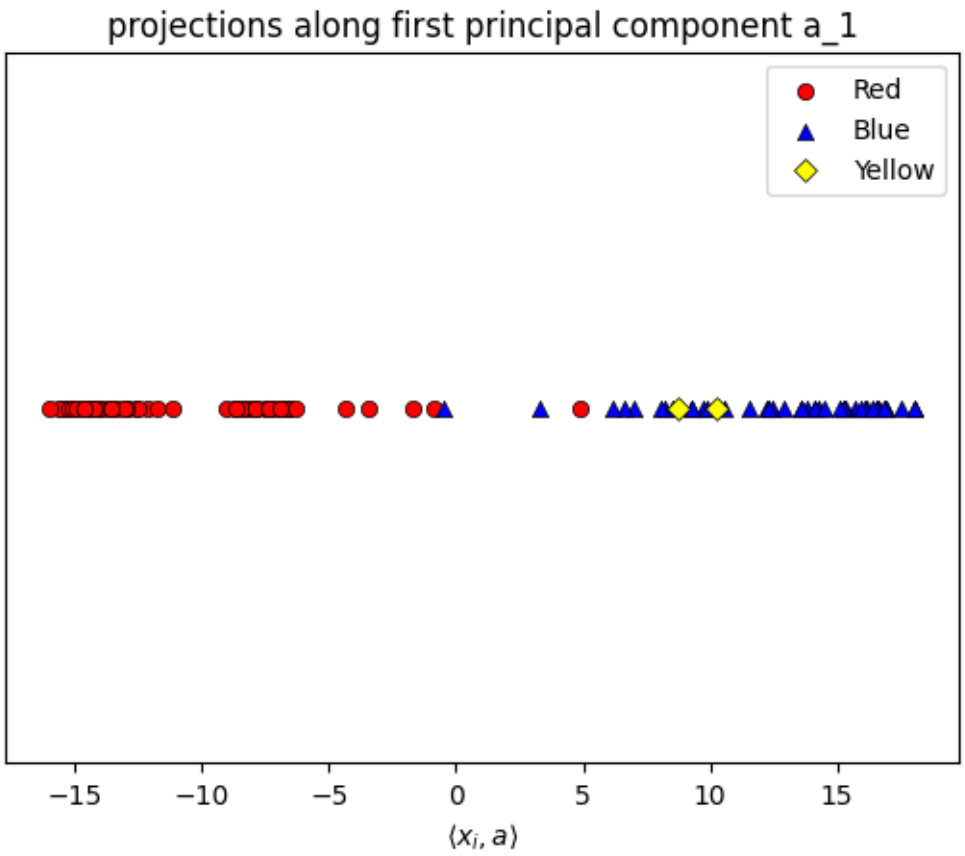


```
variance along random direction a_rand:  9.26745439089334
```

Note here that projecting along the random vector `a_rand` does not explain much variance at all — data points are clustered together and intermixed across parties. It is clear that this direction does not give us any information about the senators' affiliations.

[21]:
```
###############################################################################
### TODO: Calculate a_1, the first principal component of X.
# Hint: The PCA package imported from sklearn.decomposition will be useful here,
# in particular the function pca.fit(). What should the dimensions of a_1 be?
pca = PCA(n_components=1)
pca.fit(X)
a_1 = pca.components_[0]
assert a_1.shape == (d,)

### end TODO
###############################################################################

a_1 = a_1/np.linalg.norm(a_1)
# compute and visualize the scores along a_1
scores_a_1 = f(X, a_1)

for aff, marker, _ in markers:
    plt.scatter(
        scores_a_1[np.array(affiliations) == aff],
        np.zeros_like(scores_a_1[np.array(affiliations) == aff]),
        c=aff, marker=marker, edgecolors="black", linewidth=0.5, label=aff
    )
plt.legend()
plt.title('projections along first principal component a_1')
plt.xlabel('$\\langle x_i, a \\rangle$')
cur_axes = plt.gca()
cur_axes.axes.get_yaxis().set_visible(False)
plt.show()

print('variance along first principal component: ', scores_a_1.var())
```

```
(542,)
```




```
variance along first principal component:  149.74896507620733
```

If you computed `a_1` correctly, you should observe that the variance is much higher than the `a_rand` projection, and that blue and red dots are now spread in two clusters. This makes sense: the first principal component is the direction along which data varies most, and that is often along party lines. You just found a mathematical model for partisanship!

1.2 b) Comparison to party averages

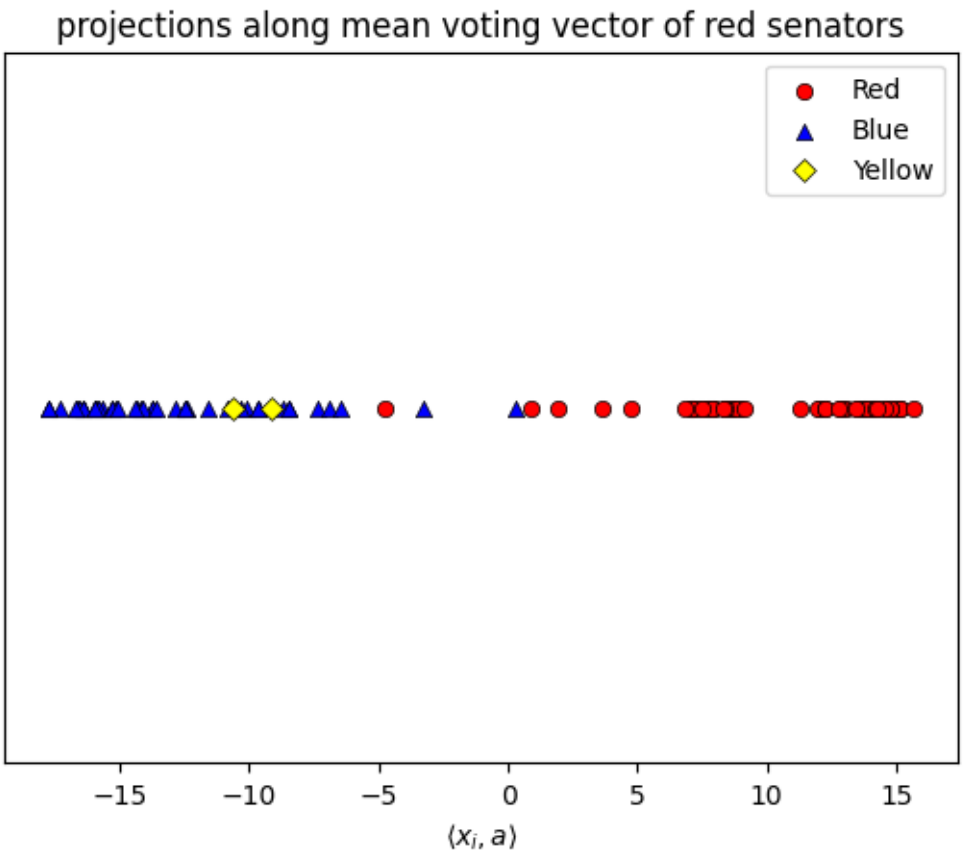
We observed above that the direction of maximum variance appears to be determined by party alignment; we now want to quantify how true that is by computing variance along vectors that describe the average position of each party. Specifically, we will compute variance along the following two vectors:

- `a_mean_red`: unit vector along the mean of rows of `X` corresponding to 'Red' senators
- `a_mean_blue`: unit vector along the mean of rows of `X` corresponding to 'Blue' senators

Fill in the code as indicated below to calculate these values and compute their relationships to `a_1` and each other, then answer the interpretation question that follows.

[30]:
```
###############################################################################
### TODO: Calculate mu_red, the array of dimension (542, ) whose values
```

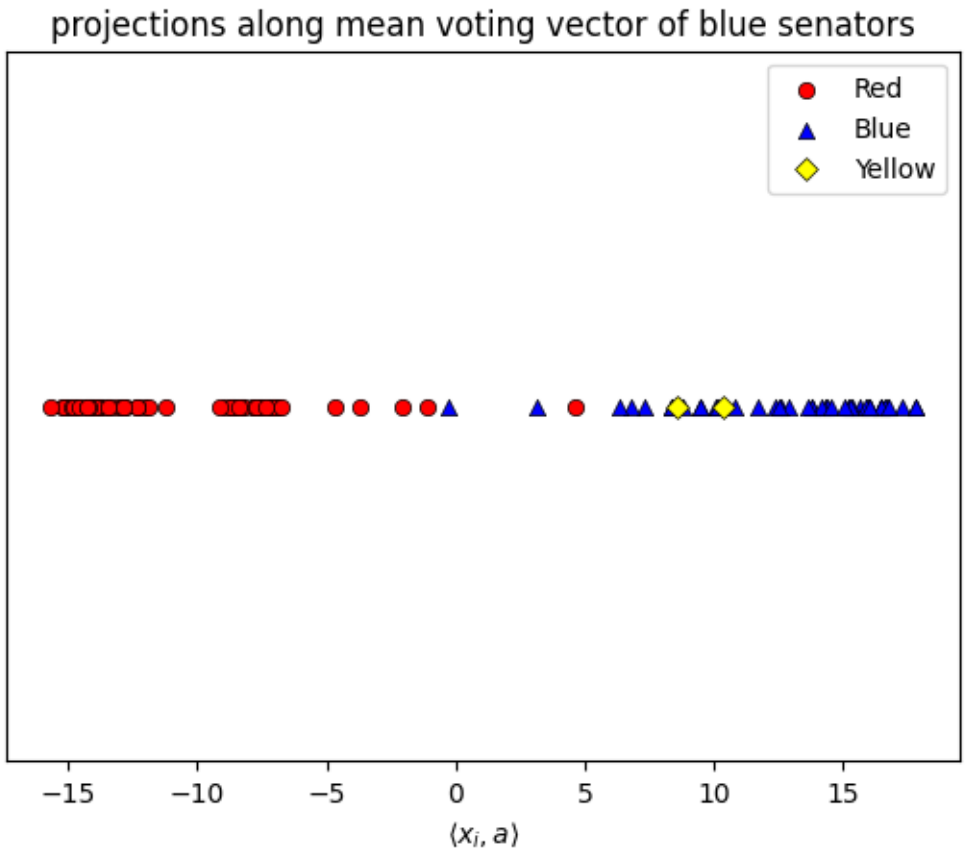
```
### are the mean across rows of X corresponding to 'Red' senators only.
# Hint: Print out the 'affiliations' variable and observe its contents.
mask = np.array(affiliations) == 'Red'

mu_red = np.mean(X[mask], axis = 0)

### end TODO
##############################################################################

# normalize the vector to generate unit a_mean_red
a_mean_red = mu_red/np.linalg.norm(mu_red)

# compute and visualize the scores along a_mean_red
scores_mean_red = f(X, a_mean_red)

for aff, marker, _ in markers:
    plt.scatter(
        scores_mean_red[np.array(affiliations) == aff],
        np.zeros_like(scores_mean_red[np.array(affiliations) == aff]),
        c=aff, marker=marker, edgecolors="black", linewidth=0.5, label=aff
    )
plt.legend()
plt.title('projections along mean voting vector of red senators')
plt.xlabel('$\\langle x_i, a \\rangle$')
cur_axes = plt.gca()
cur_axes.axes.get_yaxis().set_visible(False)
plt.show()

print('variance along mean voting vector of red senators: ', scores_mean_red.
 ↪var())
```




```
variance along mean voting vector of red senators:  148.80699963205723
```

[31]:
```
##############################################################################
### TODO: Calculate mu_blue, the array of dimension (542, ) whose values
### are the mean across rows of X corresponding to 'RBlue' senators only.
# Hint: Print out the 'affiliations' variable and observe its contents.
# print(len(affiliations))
# print(affiliations)

mask = np.array(affiliations) == 'Blue'

mu_blue = np.mean(X[mask], axis = 0)
### end TODO
##############################################################################

# normalize the vector to generate unit a_mean_blue
a_mean_blue = mu_blue/np.linalg.norm(mu_blue)

# compute and visualize the scores along a_mean_blue
scores_mean_blue = f(X, a_mean_blue)
```

```
for aff, marker, _ in markers:
    plt.scatter(
        scores_mean_blue[np.array(affiliations) == aff],
        np.zeros_like(scores_mean_blue[np.array(affiliations) == aff]),
        c=aff, marker=marker, edgecolors="black", linewidth=0.5, label=aff
    )
plt.legend()
plt.title('projections along mean voting vector of blue senators')
plt.xlabel('$\\langle x_i, a \\rangle$')
cur_axes = plt.gca()
cur_axes.axes.get_yaxis().set_visible(False)
plt.show()

print('variance along mean voting vector of blue senators: ', scores_mean_blue.
 ↪var())
```




```
variance along mean voting vector of blue senators:  148.9088414400461
```

[32]:
```
# compute dot product of and angle between a_mean_red and a_mean_blue:
dot_product_blue_red = a_mean_blue.T @ a_mean_red
angle_blue_red = np.arccos(dot_product_blue_red) * 180/np.pi

print('dot product of a_mean_blue and a_mean_red: ', dot_product_blue_red)
print('angle between a_mean_blue and a_mean_red (degrees): ', angle_blue_red)
```

```
dot product of a_mean_blue and a_mean_red:  -0.9992350984093116
angle between a_mean_blue and a_mean_red (degrees):  177.75886458298177
```

1.2.1 TODO: Interpretation

Comment on the relationships between `a_mean_red` and `a_mean_blue` above based on their dot products and relative angles.

TODO: The two vectors are almost on the opposite direction to each other, i.e. $\mu_{blue} \approx -\mu_{red}$.

Next, we will see how aligned the mean voting vectors are with the first principal component of the data.

[33]:
```
# check angle between mean voting vector of red senators and the first␣
↪principal component as well as that of blue senators and the first principal␣
↪component
dot_product_red_a1 = a_mean_red.T @ a_1
angle_red_a1 = np.arccos(dot_product_red_a1) * 180/np.pi

print('dot product of a_mean_red and a_1: ', dot_product_red_a1)
print('angle between a_mean_red and a_1 (degrees): ', angle_red_a1)

dot_product_blue_a1 = a_mean_blue.T @ a_1
angle_blue_a1 = np.arccos(dot_product_blue_a1) * 180/np.pi

print('dot product of a_mean_blue and a_1: ', dot_product_blue_a1)
print('angle between a_mean_blue and a_1 (degrees): ', angle_blue_a1)
```

```
dot product of a_mean_red and a_1:  -0.9965356912813021
angle between a_mean_red and a_1 (degrees):  175.22941782780575
dot product of a_mean_blue and a_1:  0.9969831227823048
angle between a_mean_blue and a_1 (degrees):  4.451697983372387
```

1.2.2 TODO: Interpretation

Comment on the relationships between Red and Blue senators to partisanship based on the two products of `a_mean_blue` and `a_mean_red` with `a_1`, the top principal component of the covariance, i.e., the maximum variance direction.

TODO: The blue vector is almost in the same direction with the first principle components, which indicates that the first principle component $\approx -\mu_{red}$.

1.3 c) Computing total variance

We now wish to observe the variance of the data along the first two principal component axes.

Fill in the code below to calculate the total variance of the data along the first two principal components a_1 and a_2 and to plot the data on the corresponding axes.

[41]:
```
################################################################################
### TODO: Calculate the Sigma matrix (defined in LaTeX problem) and the total␣
 ↪variance across a_1 and a_2.
# Hint: The latter value is equal to the sum of the two largest eigenvalues of␣
 ↪Sigma. You can use either the PCA library or the numpy.linalg library that␣
 ↪were already imported at the start of this notebook.
Sigma = X.T @ X / n
pca = PCA(n_components=2)
pca.fit(X)
total_variance = np.sum(pca.explained_variance_) * (n-1) / n
### end TODO
################################################################################
print(Sigma)
print('total variance explained by first two principal components: ',␣
 ↪total_variance)
```

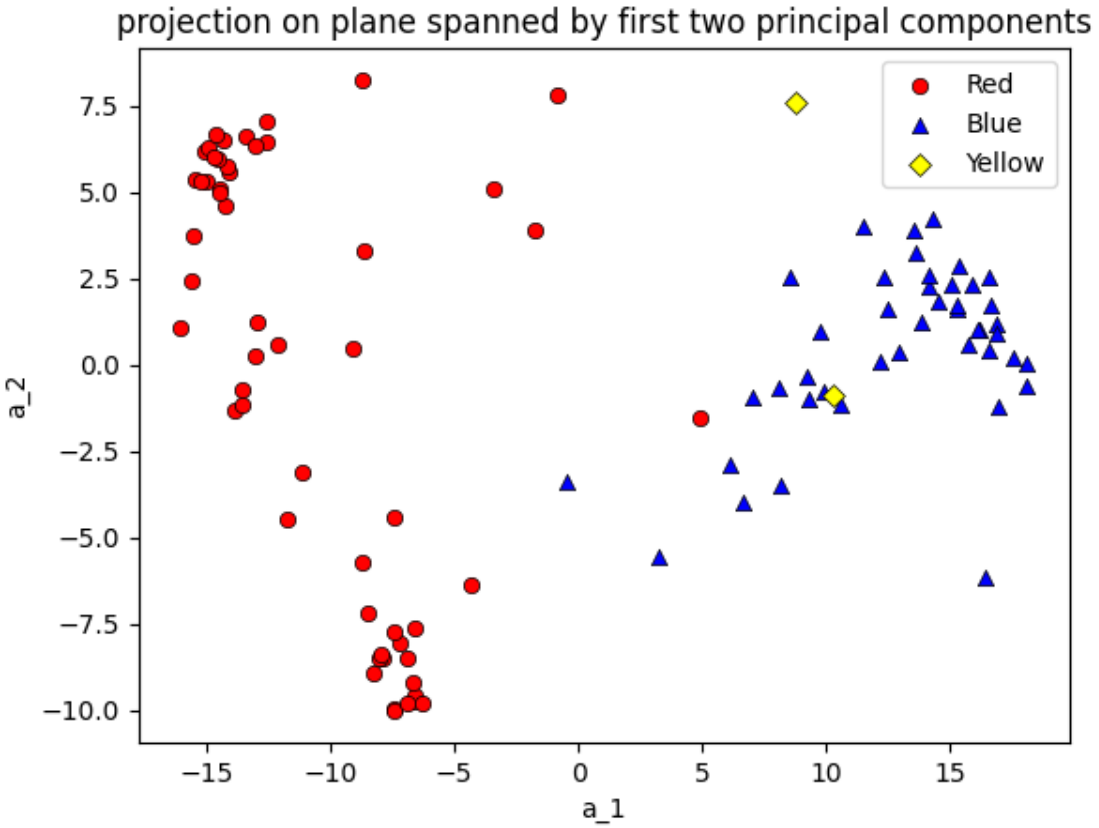
```
[[ 1.     -0.78   -0.8    …  0.14   -0.24   -0.23  ]
 [-0.78    0.9996  0.8592 … -0.0892  0.298   0.2886]
 [-0.8     0.8592  0.9984 … -0.0984  0.296   0.2872]
 …
 [ 0.14   -0.0892 -0.0984 …  0.2684  0.024   0.0178]
 [-0.24    0.298   0.296  …  0.024   0.47    0.173 ]
 [-0.23    0.2886  0.2872 …  0.0178  0.173   0.4651]]
total variance explained by first two principal components:  175.17116160523148
```

Next, we calculate and plot the projection onto the plane spanned by the first two principal components.

[42]:
```
# calculate projected data matrix and observe its shape
pca = PCA(n_components=2)
projected = pca.fit_transform(X)
print(projected.shape)

# plot projected data matrix
for aff, marker, _ in markers:
    plt.scatter(
        projected[np.array(affiliations) == aff, 0],
        projected[np.array(affiliations) == aff, 1],
        c=aff, marker=marker, edgecolors="black", linewidth=0.5, label=aff
    )
plt.legend()
plt.xlabel('a_1')
```

```
plt.ylabel('a_2')
plt.title('projection on plane spanned by first two principal components')
plt.show()
```

```
(100, 2)
```



1.4 d) Finding bills that are the most/least contentious

We now wish to observe which bills are the most and least contentious — i.e., those for which most senators voted unanimously, and those for which support was most varied. We consider one possible way of quantifying this relationship mathematically.

We can compute the variance of each column of X — each of which corresponds to a particular bill — and use this variance as a measure of "contentiousness" (i.e., the more contentious a bill, the higher its variance in terms of senator vote count). Note that the variance of a particular bill in column j can be viewed as the variance of scores along $\vec{e}_j$, where $\vec{e}_j$ is a basis vector whose j^{th} entry is 1 and all others 0.

Fill in the code below to calculate the variance of X, extract the most and least contentious bills, and plot their vote counts, commenting on your results where indicated.

[46]:
```
# calculate the variance of each column
list_variances = X.var(axis=0)
bills = senator_df['bill_type bill_name bill_ID'].values

###############################################################################
### TODO: Compute sorted_idx_variances, an np.array of shape (542,) containing
### integer entries that are the indices of variance scores in list_variances in
### decreasing order of variance. For example, if list_variances = [1, 3, 2, 4],
### then sorted_idx_variances = np.array([3,1,2,0]).
# Hint: Use np.argsort().
sorted_idx_variances = np.argsort(list_variances)[::-1]

### end TODO
###############################################################################

print(sorted_idx_variances.shape)
```

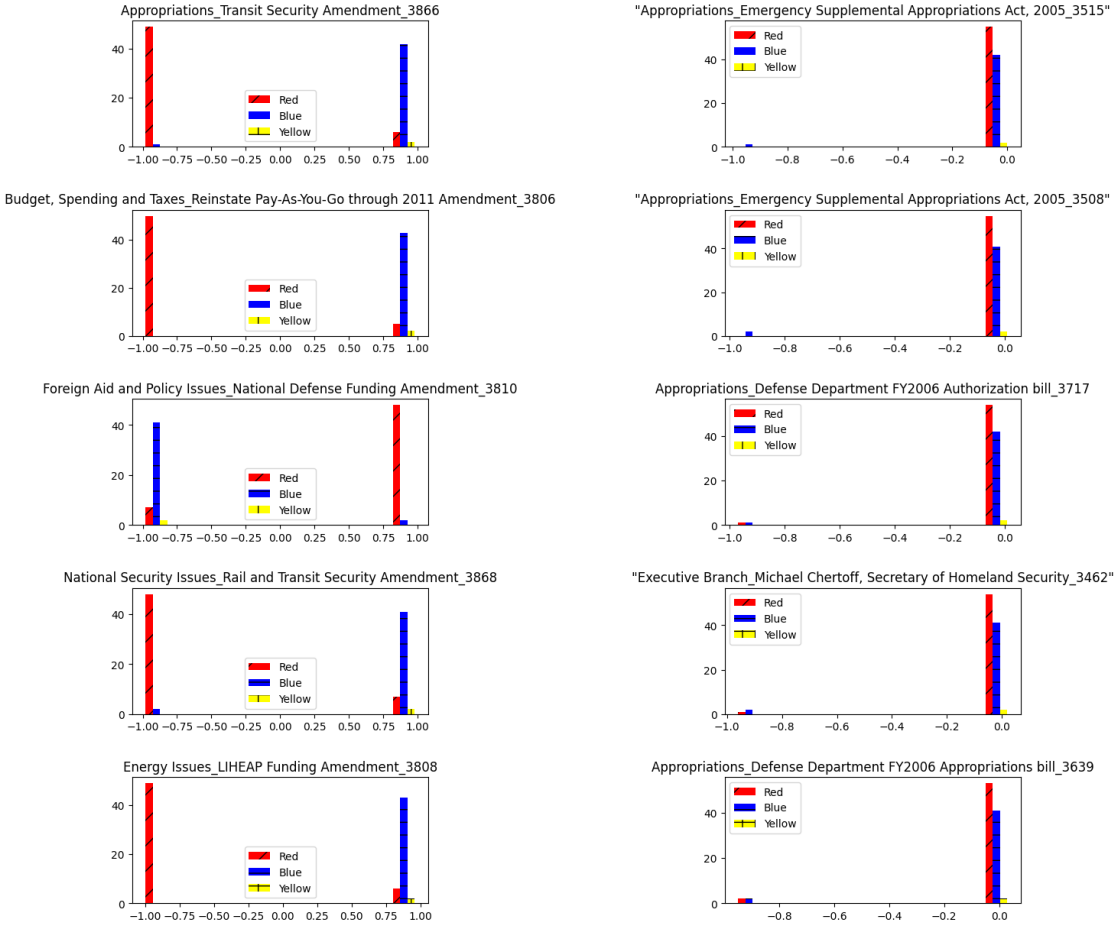
```
(542,)
```

Using this sorted index, we can now plot the vote counts for the top 5 highest and lowest variance bills.

[47]:
```
# retrieve the bills with the 5 highest and lowest variances
top_5 = [bills[sorted_idx_variances[i]] for i in range(5)]
bot_5 = [bills[sorted_idx_variances[-1-i]] for i in range(5)]

# set up figure with all desired subplots
fig, axes = plt.subplots(5,2, figsize=(15,15))

# plot highest variance bills
for i in range(5):
    idx = sorted_idx_variances[i]

    # retrieve vote counts from original uncentered data matrix
    Xs = []
    colors = []
    labels = []
    hatches = []
    for color, _, hatch in markers:
        Xs.append(X_original[np.array(affiliations) == color, idx])
        colors.append(color)
        labels.append(color)
        hatches.append(hatch)

    _, _, patches = axes[i, 0].hist(
        Xs,
        color=colors,
        label=labels,
```

```
    )
    for patch_set, hatch in zip(patches, hatches):
        for patch in patch_set.patches:
            patch.set_hatch(hatch)

    axes[i,0].legend()
    axes[i,0].set_title(bills[idx])

# plot lowest variance bills
for i in range(1,6):
    idx2 = sorted_idx_variances[-i]

    # retrieve vote counts from original uncentered data matrix
    Xs = []
    colors = []
    labels = []
    hatches = []
    for color, _, hatch in markers:
        Xs.append(X_original[np.array(affiliations) == color, idx2])
        colors.append(color)
        labels.append(color)
        hatches.append(hatch)

    _, _, patches = axes[i-1, 1].hist(
        Xs,
        color=colors,
        label=labels,
    )
    for patch_set, hatch in zip(patches, hatches):
        for patch in patch_set.patches:
            patch.set_hatch(hatch)

    axes[i-1,1].legend()
    axes[i-1,1].set_title(bills[idx2])

plt.subplots_adjust(hspace=0.5, wspace = 1)
fig.suptitle('Most Variance -- Least Variance', fontsize=16)
plt.show()
```

Most Variance -- Least Variance



1.4.1 TODO: Interpretation

Comment on the voting trends you observe in the plots above. In general, if a vote is contentious, what do you expect the plots to look like? What about if a vote is uncontentious?

TODO: If a vote is contentious, the majority of red and blue should be in different values. Otherwise, they share the same value. .

2 e) Infer political affiliation using top two PCA directions

We now consider a strategy to infer the political affiliation of two senators based on how they voted for the bills and considering the projection of their votes onto the two principal components.

In this part, we will compute the top two PCA directions for our given data after removing two specific senators. Then we will plot all the senators in 2D based on their projections on the top

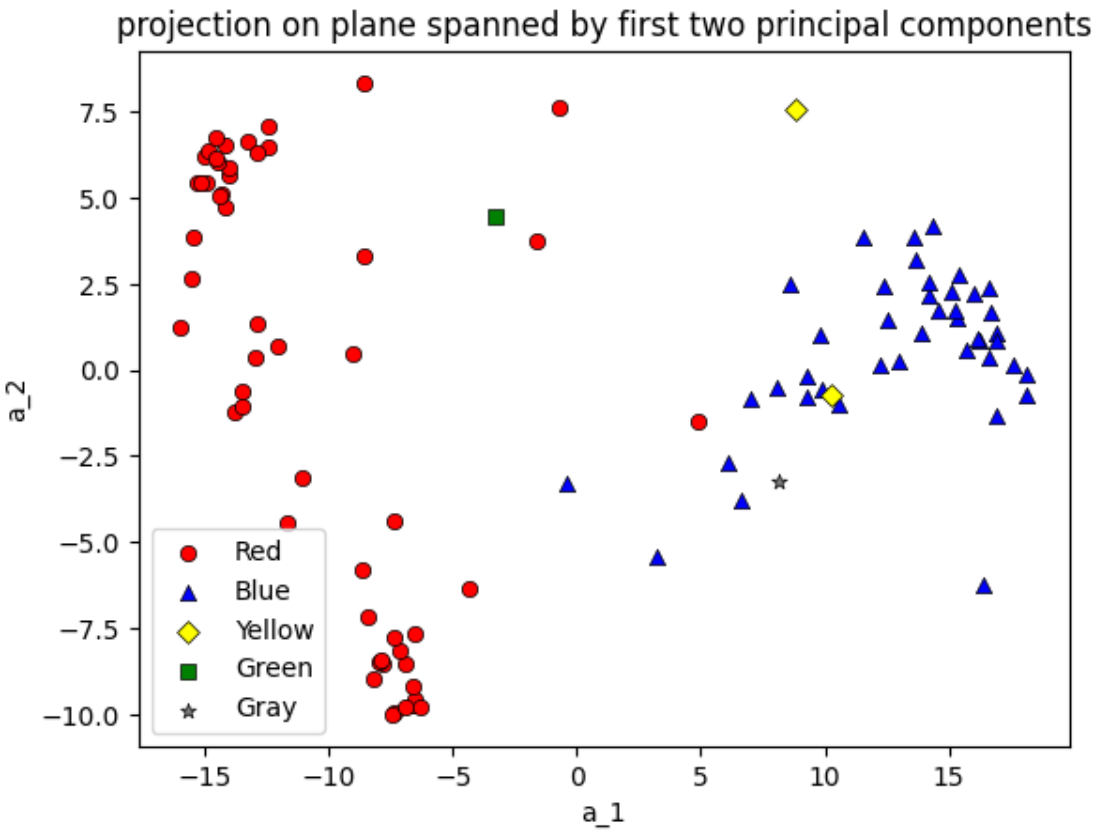
two directions as in part c). The two senators whose affiliation needs to be inferred are marked in green and gray. Can you infer the political affiliation (Red or Blue) of the green and gray senator by looking at the points in this 2D plane?

[48]:
```
X_train=np.delete(X,[0,1],0)
affiliations_train=affiliations[:]
affiliations_train[0]='Green'
affiliations_train[1]='Gray'


# this is just used for plotting, feel free to ignore
assert set(affiliations_train) == {"Red", "Blue", "Yellow", "Green", "Gray"}
# assign a marker and hatch to each affiliation
markers_train = [("Red", "o", "/"), ("Blue", "^", "-"), ("Yellow", "D", "+"),␣
 ↪("Green", "s", "x"), ("Gray", "*", ".")]
```

[49]:
```
# calculate projected data matrix and observe its shape
pca = PCA(n_components=2)
pca.fit(X_train)
projected = pca.transform(X)
print(projected.shape)
```

```
(100, 2)
```

[50]:
```
# plot projected data matrix
for aff, marker, _ in markers_train:
    plt.scatter(
        projected[np.array(affiliations_train) == aff, 0],
        projected[np.array(affiliations_train) == aff, 1],
        c=aff, marker=marker, edgecolors="black", linewidth=0.5, label=aff
    )
plt.legend()
plt.xlabel('a_1')
plt.ylabel('a_2')
plt.title('projection on plane spanned by first two principal components')
plt.show()
```



2.0.1 TODO: Interpretation

Based on the plot above, what is the likely affiliation of the Green senator? What is the likely affiliation of the Grey senator?

TODO: The green senator is more likely to be from red affiliation. And the grey from blue affiliation.

2.1 f) Finding extreme senators

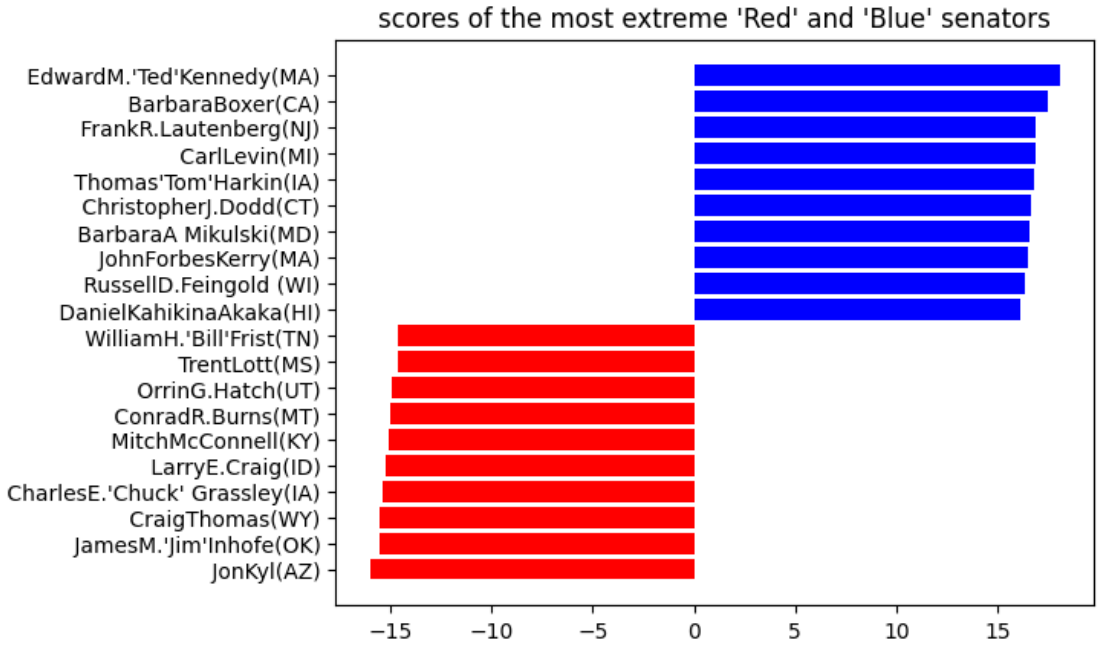
Lastly, let us return to our initial definition of $f(X, \vec{a})$, which assigns each senator a score. We will now use this value computed along the first principal component $\vec{a}$ to assign the following classifications to our senators:

- Senators with the top 10 most positive scores and top 10 most negative scores are classified as *most extreme*.
- Senators with the 20 scores closest to 0 are classified as *least extreme*.

In the final subproblem, we observe these scores and how they relate to party affiliation.

We first compute the most extreme senators:

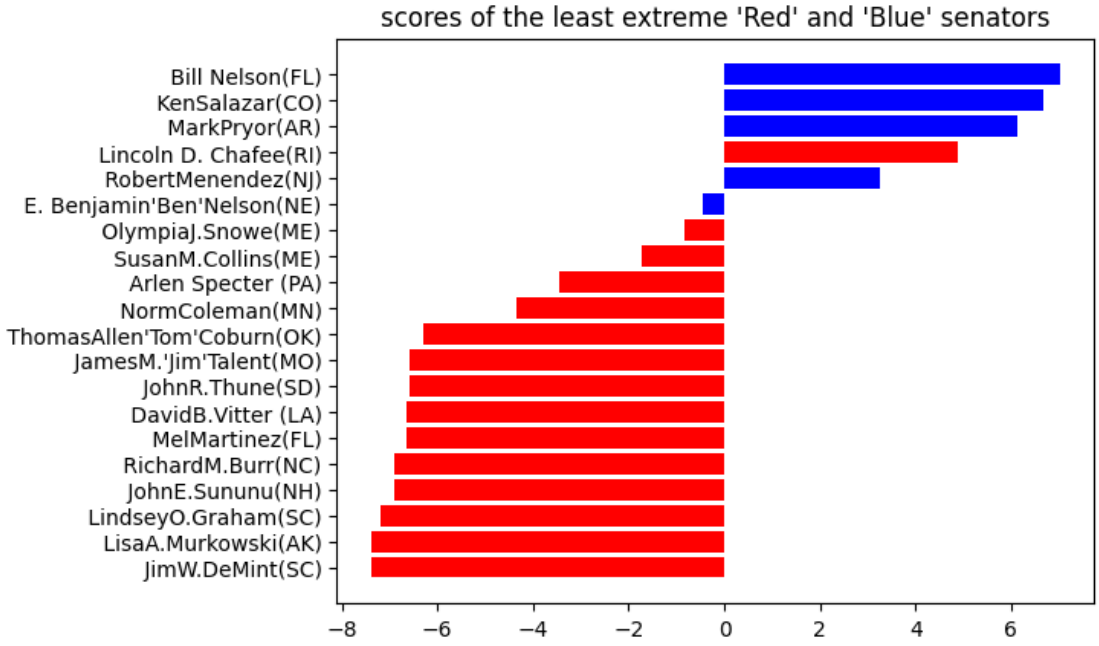
[51]:
```
# extract senator names
senators = senator_df.columns.values[3:]

# compute and sort senators scores and corresponding affiliations
senator_scores = f(X,a_1)
complete_sort_indices = np.argsort(senator_scores)
sort_indices = np.hstack([complete_sort_indices[:10], complete_sort_indices[-11:
 ↪-1]])
senators_sorted = senators[sort_indices]
senator_scores_sorted = senator_scores[sort_indices]
affiliations = np.array(affiliations)
affiliations_sorted = affiliations[sort_indices]

plt.barh(y = senators_sorted, width = senator_scores_sorted, color =␣
 ↪affiliations_sorted)
plt.title('scores of the most extreme \'Red\' and \'Blue\' senators')
plt.show()
```



And the least extreme senators:

[52]:
```
# compute and sort senators scores and corresponding affiliations
senator_scores = f(X,a_1)
complete_sort_indices = np.argsort(np.abs(senator_scores))[:20]
senator_scores_le= senator_scores[complete_sort_indices]
senators_le = senators[complete_sort_indices]
affiliations = np.array(affiliations)
```

```
affiliations_le = affiliations[complete_sort_indices]
sort_indices = np.argsort(senator_scores_le)
senators_sorted = senators_le[sort_indices]
senator_scores_sorted = senator_scores_le[sort_indices]
affiliations_sorted = affiliations_le[sort_indices]

plt.barh(y = senators_sorted, width = senator_scores_sorted, color =␣
 ↪affiliations_sorted)
plt.title('scores of the least extreme \'Red\' and \'Blue\' senators')
plt.show()
```



2.1.1 TODO: Interpretation

Comment on the sign of senators' scores and what they say about party affiliation for both the most and least extreme senators.

TODO: Most of the senators in blue have positive first principle component values while the ones in red have negative values. When the absolute value is large, the color is purely blue or red. But when it is close to zero, the red and blue start to intersect.

svd_transformation

February 17, 2026

1 Readme

Places where solutions are required are marked with **#TODO**
You will **NOT** need to modify any section not marked as #TODO to answer this question.
Make sure the helper file. svd_transformation_helper.py is in the same folder as this .ipynb
Make sure you have numpy, matplotlib and itertools packages installed for python

1.0.1 In this notebook:

Part (b) has 3 subparts i, ii, and iii
Part (c) has 4 subparts i, ii, iii and iv
Part (d) has 2 subparts i,ii
Part (e) has only 1 subpart

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
!gdown 1cV2RxKzE-02nGVMXi-O92dytf6-Cg-3D -O svd_transformation_helper.py
from svd_transformation_helper import visualize_function
from svd_transformation_helper import matrix_equals, is_orthonormal
```

```
Downloading…
From (original):
https://drive.google.com/uc?id=1cV2RxKzE-02nGVMXi-O92dytf6-Cg-3D
From (redirected): https://drive.google.com/uc?id=1cV2RxKzE-02nGVMXi-O92dytf6-
Cg-3D&confirm=t&uuid=aa75dbad-20db-4848-ab58-e7a663de25a3
To: /content/svd_transformation_helper.py
   0% 0.00/4.63k [00:00<?, ?B/s] 100% 4.63k/4.63k [00:00<00:00, 13.0MB/s]
```

[ ]:
```
DISABLE_CHECKS = False #Set this to True only if you get Value Errors about␣
 ↪inputs even
#when you are sure that what you are inputting is correct.
#WARNING: Setting this to True and entering wrong inputs can lead to all kinds␣
 ↪of crazy results/errors


def visualize(U = np.identity(2), D = np.ones(2), VT = np.identity(2),␣
 ↪num_grid_points_per_dim = 200,\
```

```
    disable_checks = DISABLE_CHECKS, show_original = True, show_VT = True,␣
↪show_DVT = True, show_UDVT = True):
    '''
    Inputs:
    A has singular value decomposition A = U np.diag(D) VT
    U: 2 x 2 orthonormal matrix represented as a np.array of shape (2,2)
    D: Diagonal entries corresponding to the diagonal matrix in SVD represented␣
↪as a np.array of shape (2,)
    VT: 2 x 2 orthonormal matrix represented as a np.array of shape (2,2)
    num_grid_points_per_dim: Spacing of points used to represent circle␣
↪(Decrease this if plotting is slow)
    disable_checks: If False then have checks in  place to make sure dimensions␣
↪of VT, U are correct, etc.
    show_original: If True plots original unit circle and basis vectors
    show_VT: If True plots transformation by VT
    show_DVT: If True plots transformation by DVT
    show_UDVT: If True plots transformation by UDVT
    '''

    visualize_function(U=U, D=D, VT=VT,␣
↪num_grid_points_per_dim=num_grid_points_per_dim,␣
↪disable_checks=disable_checks,\
                       show_original=show_original, show_VT=show_VT,␣
↪show_DVT=show_DVT, show_UDVT=show_UDVT)
```

2 Effect of the linear transformation by an orthonormal matrix V^T

A 2 x 2 orthonormal matrix can be viewed as a linear transformation that performs some combination of rotations and reflections. Note that both rotation and reflection are operations that preserve the length of vectors and the angle between them.

2.1 V^T as a rotation matrix

First we set V^T as a counter-clockwise rotation matrix.

2.1.1 (b) i: Fill in the function "get_RCC(theta)" to return a 2 x 2 matrix that, when applied to a vector x, rotates it by theta radians counter clockwise.

Example: If $V^T = RCC\left(\frac{\pi}{4}\right)$ and x $= \begin{bmatrix}1\\0\end{bmatrix}$, then,

$$V^T \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\end{bmatrix}$$

[ ]:
```
def get_RCC(theta):
    '''
```

```
    Returns a 2 x 2 orthonormal matrix that rotates x by theta radians␣
↪counter-clockwise
    '''

    RCC = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.
↪cos(theta)]])

  ␣
↪#########################################################################################
    #Some assertions (WARNING: Do not modify below code)
    if DISABLE_CHECKS is False:
        if not isinstance(RCC, np.ndarray) or isinstance(RCC, np.matrix):
                raise ValueError('RCC must be a np.ndarray')
        if len(RCC.shape) != 2 or (RCC.shape != np.array([2,2])).any():
                raise ValueError('RCC must have shape [2,2]')
    return RCC
```

2.1.2 get_RCC(theta) function test

If the function get_RCC(theta) is defined correctly then you should not get any ERROR statement here.

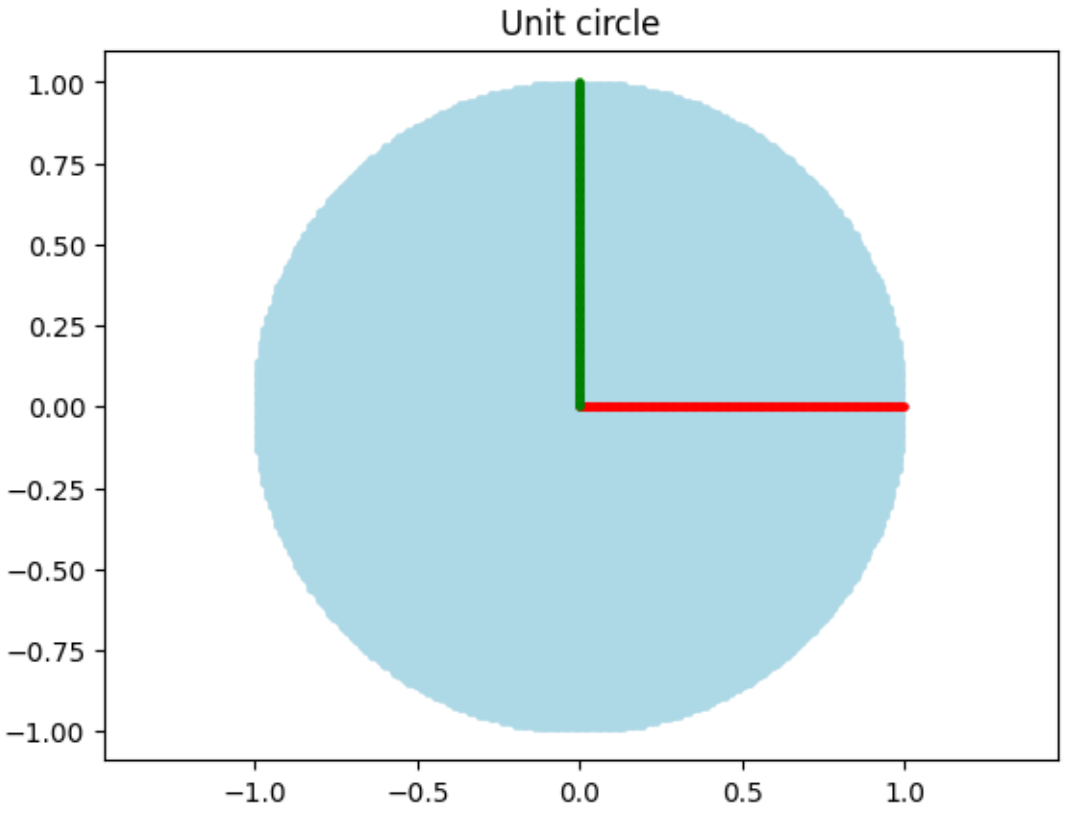
[ ]:
```
x = np.array([[1,0]]).T
V_test = get_RCC(np.pi/4)
y = V_test @ x
expected_y = np.array([[1/np.sqrt(2), 1/np.sqrt(2)]]).T
print("y:")
print(y)
print("Expected y:")
print(expected_y)
if not matrix_equals(y, expected_y):
    print("ERROR: y does not match expected_y. Check if function get_RCC(theta)␣
↪is completed correctly")
else:
    print("MATCHED: y matches expected_y!")
```

```
y:
[[0.70710678]
 [0.70710678]]
Expected y:
[[0.70710678]
 [0.70710678]]
MATCHED: y matches expected_y!
```

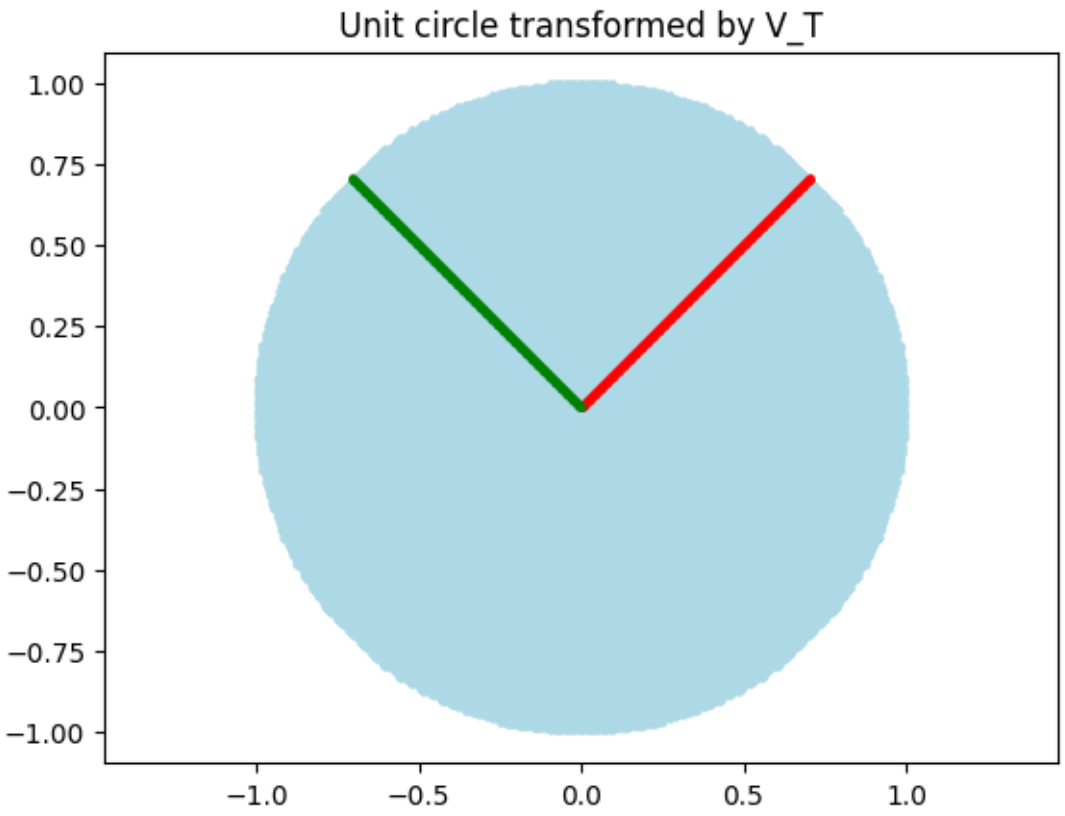
Next we observe how V^T transforms the unit circle and unit basis vectors when:

1) $V^T = RCC\left(\frac{\pi}{4}\right)$

[ ]:
```
VT_1 = get_RCC(np.pi/4)
visualize(VT = VT_1, show_DVT=False, show_UDVT=False)
```

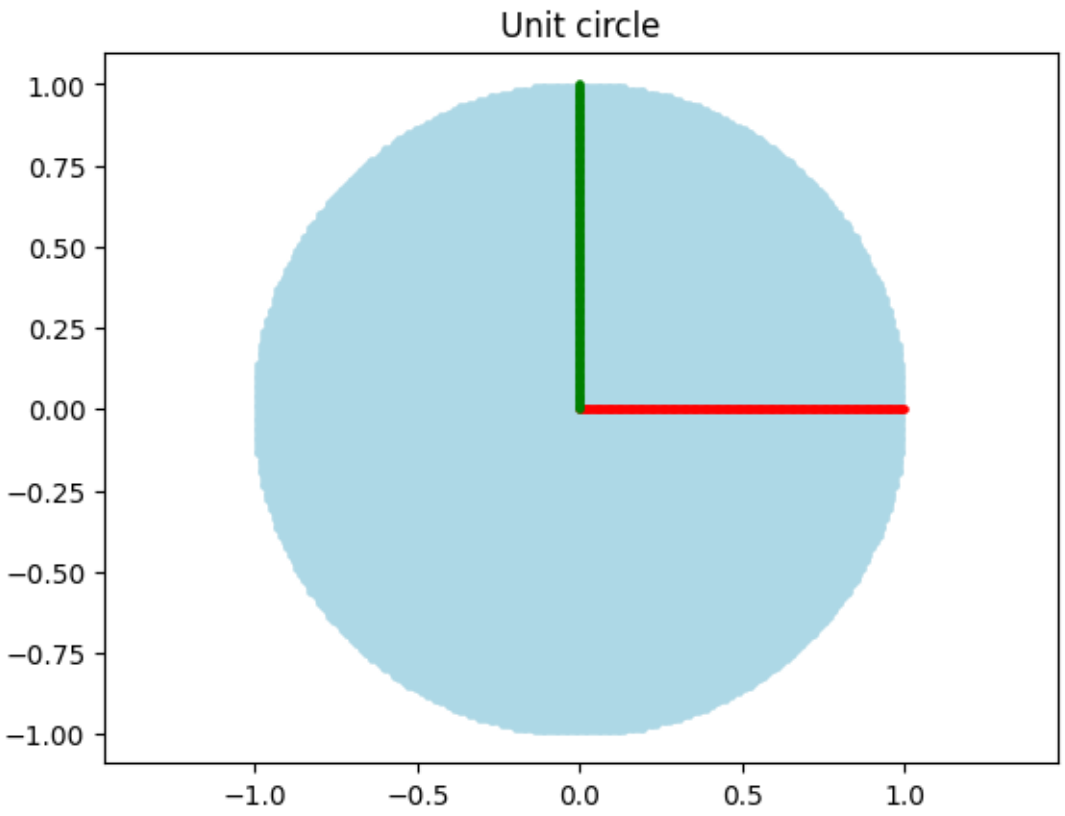



```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```

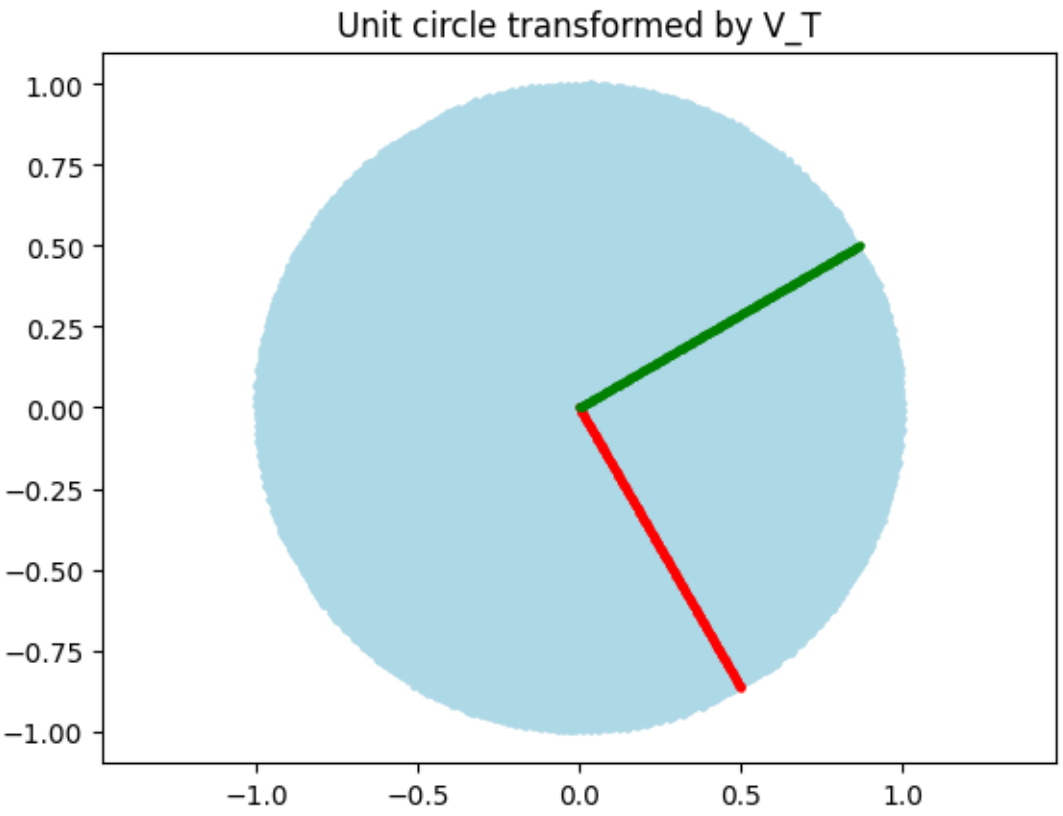


2) $V^T = RCC\left(\frac{-\pi}{3}\right)$

[ ]:
```
VT_2 = get_RCC(-np.pi/3)
visualize(VT = VT_2, show_DVT=False, show_UDVT=False)
```




```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```



Next we consider the case where V^T is a reflection matrix.

2.2 V^T as a reflection matrix

A reflection matrix is another type of orthonormal matrix.

2.2.1 (b) ii: Fill in the function "get_RFx()" to return a 2 x 2 matrix that when applied to a vector x reflects it about the x-axis.

Example: If $V^T = RFx()$ and $x = \begin{bmatrix}1\\1\end{bmatrix}$, then,

$V^T \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}1\\-1\end{bmatrix}$

[ ]:
```
def get_RFx():
    '''
    Returns a 2 x 2 orthonormal matrix that reflects about x-axis
    '''


    RFx = np.array([[1,0], [0,-1]])
```

```
␣
↪#############################################################################################
    #Some assertions (WARNING: Do not modify below code)
    if DISABLE_CHECKS is False:
        if not isinstance(RFx, np.ndarray) or isinstance(RFx, np.matrix):
                raise ValueError('RFx must be a np.ndarray')
        if len(RFx.shape) != 2 or (RFx.shape != np.array([2,2])).any():
                raise ValueError('RFx must have shape [2,2]')
    return RFx
```

2.2.2 get_RFx() function test

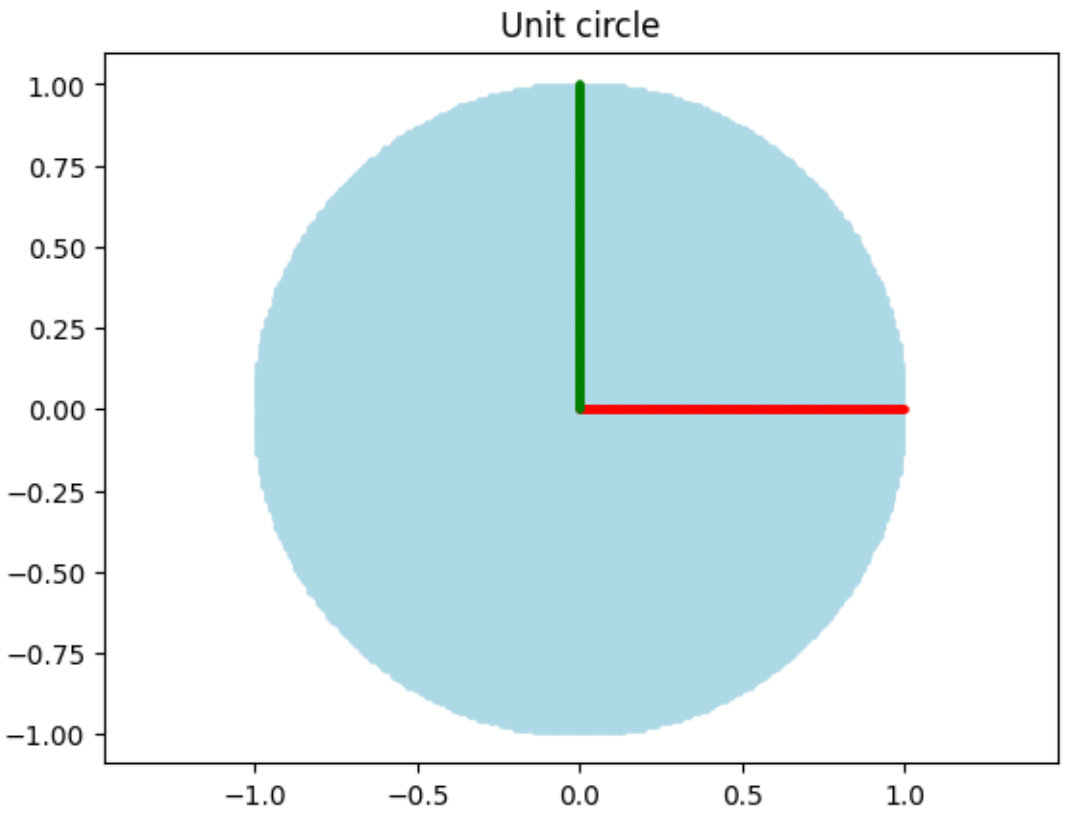
If the function get_RFx() is defined correctly then you should see a MATCHED statement here.

[ ]:
```
x = np.array([[1,1]]).T
V_test = get_RFx()
y = V_test @ x
expected_y = np.array([[1, -1]]).T
print("y:")
print(y)
print("Expected y:")
print(expected_y)
if not matrix_equals(y, expected_y):
    print("ERROR: y does not match expected_y. Check if function get_RFx() is␣
↪completed correctly")
else:
    print("MATCHED: y matches expected_y!")
```

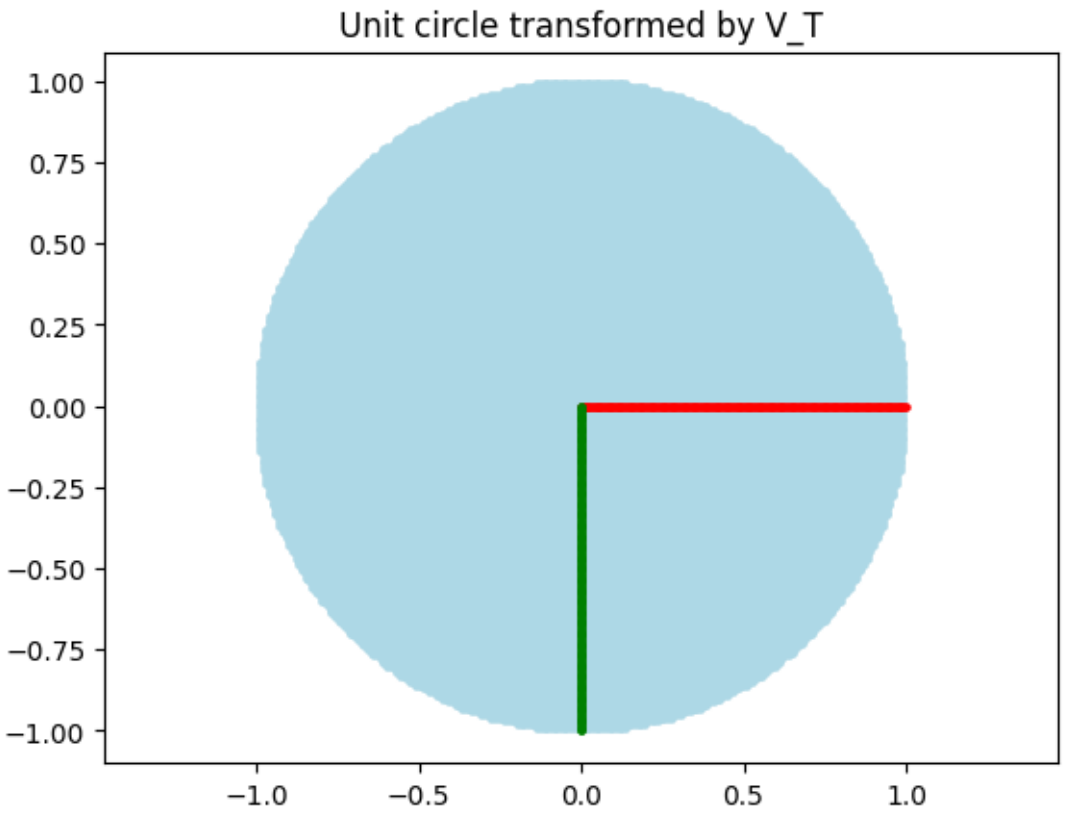
```
y:
[[ 1]
 [-1]]
Expected y:
[[ 1]
 [-1]]
MATCHED: y matches expected_y!
```

$V^T = RFx()$

[ ]:
```
VT_3 = get_RFx()
visualize(VT = VT_3, show_DVT=False, show_UDVT=False)
```




```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```



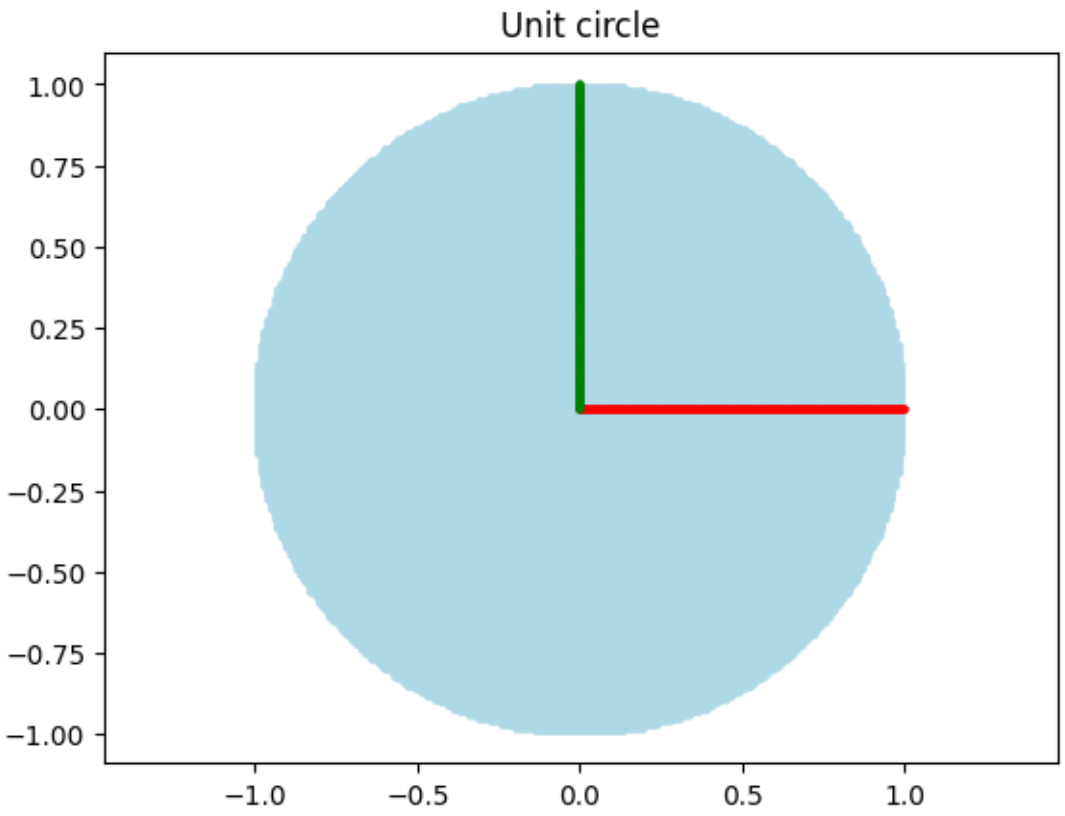
2.3 V^T as a composition of reflection and rotation matrix

In general an orthonormal transformation can be viewed as compositions of rotation and reflection operators. Next we observe the effect of setting

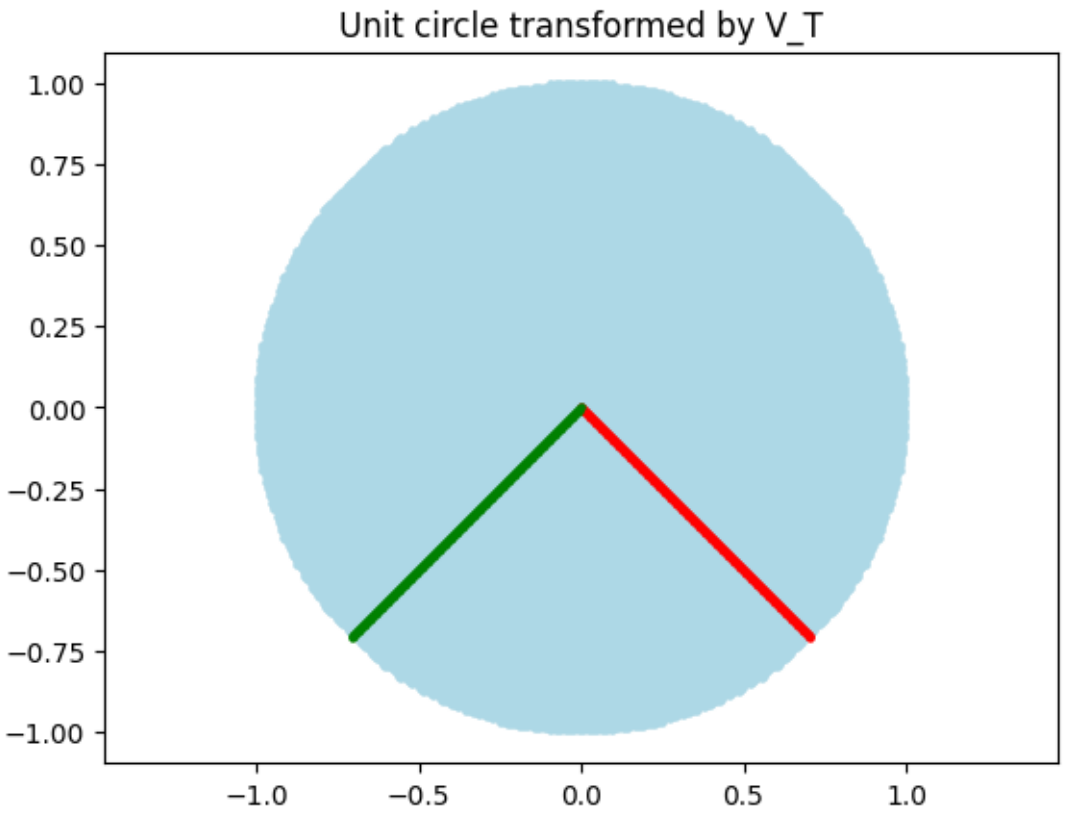
$V^T = RFx() \cdot RCC\left(\frac{\pi}{4}\right)$

[ ]:
```
VT_4 = VT_3 @ VT_1
#Check that VT_4 is still orthonormal
print("VT_4 is orthonormal?: ", is_orthonormal(VT_4))
visualize(VT = VT_4, show_DVT=False, show_UDVT=False)
```

```
VT_4 is orthonormal?:  True
```

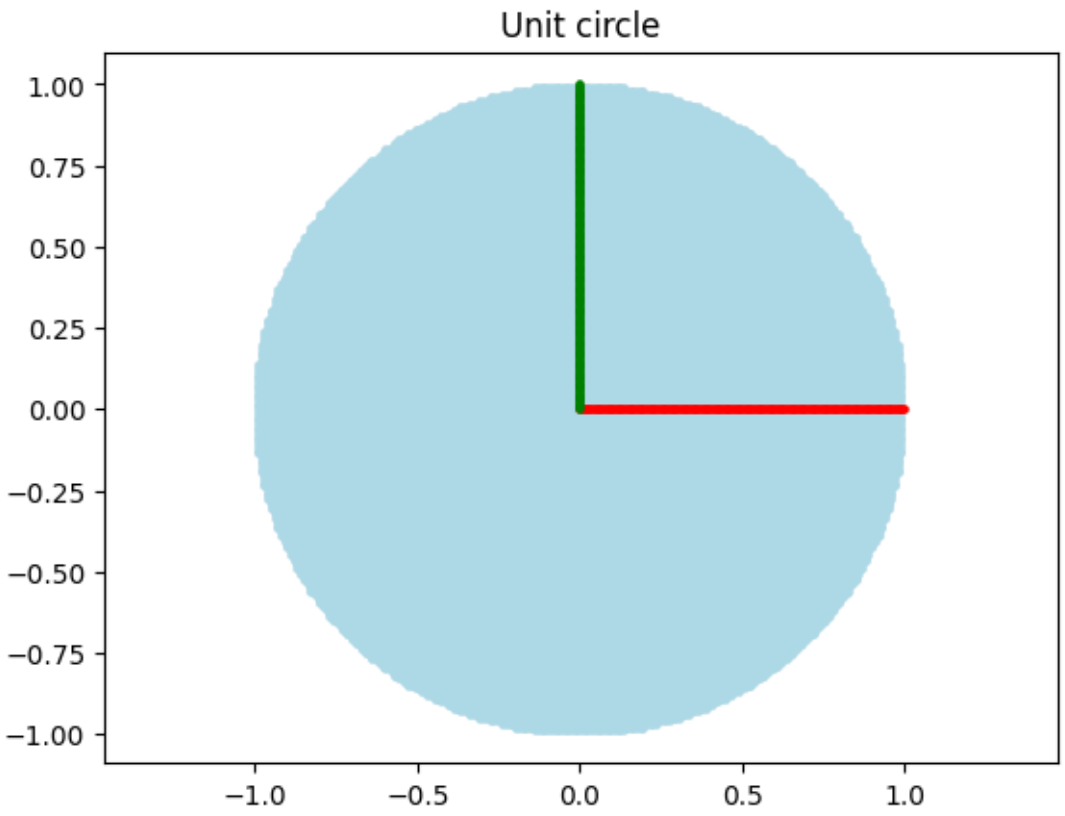



```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```

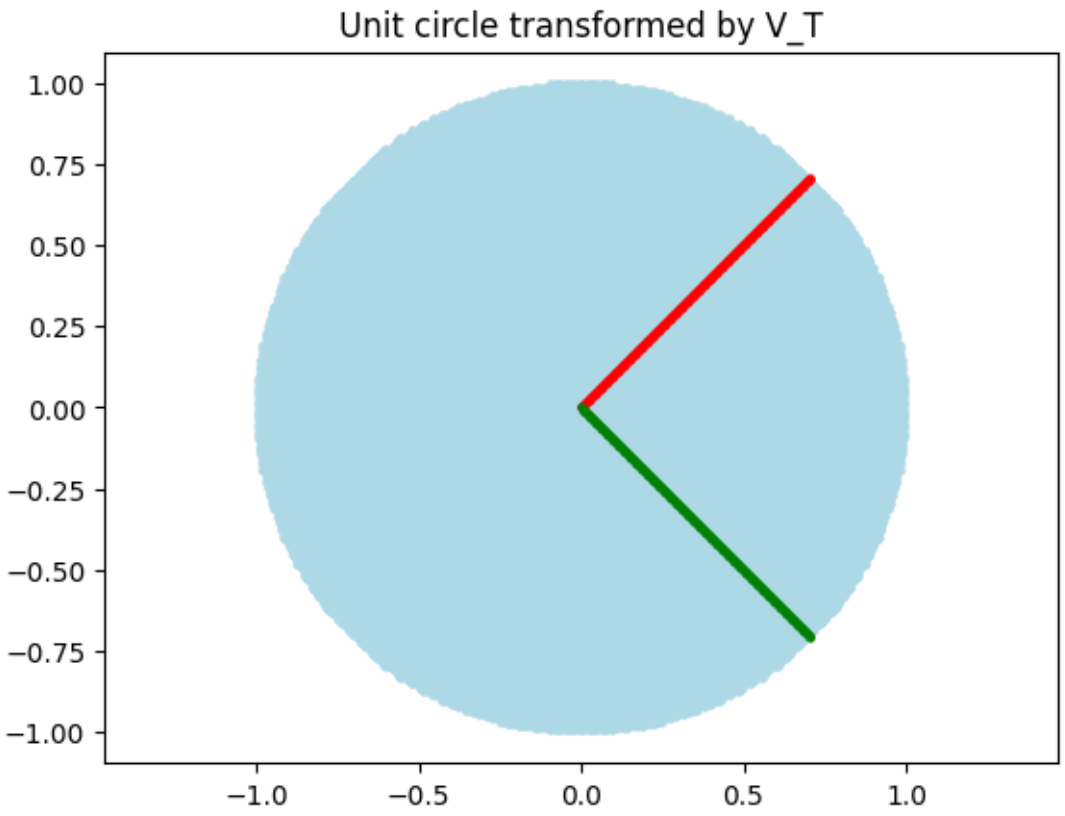


2.3.1 (b) iii: Comment on the effect of $V^T = RCC\left(\frac{\pi}{4}\right) \cdot RFx()$. Is it same as the case when $V^T = RFx() \cdot RCC\left(\frac{\pi}{4}\right)$?

[ ]:
```
VT_5 = VT_1 @ VT_3
visualize(VT = VT_5, show_DVT=False, show_UDVT=False)
```




```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```



No, they are not commutative.

3 Effect of linear transformation by diagonal matrix D

The diagonal matrix D with entries σ_1 and σ_2, transforms the unit circle into an ellipse with x direction scaled by σ_1 and y direction scaled by σ_2.

If $\sigma_1 > \sigma_2$, then the major axis of the ellipse will be along the x-axis.

If $\sigma_1 < \sigma_2$, then the major axis of the ellipse will be along the y-axis.

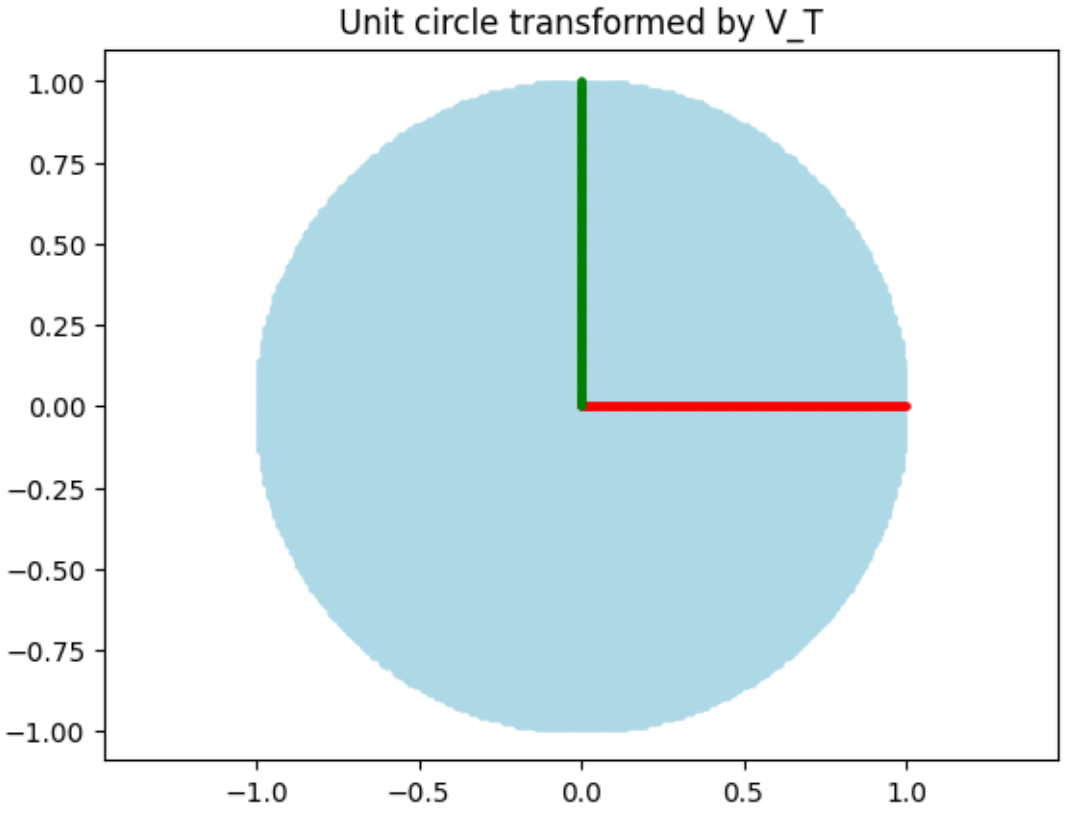
If $\sigma_1 = \sigma_2$, then the ellipse will have both axis equal (i.e it is a circle).

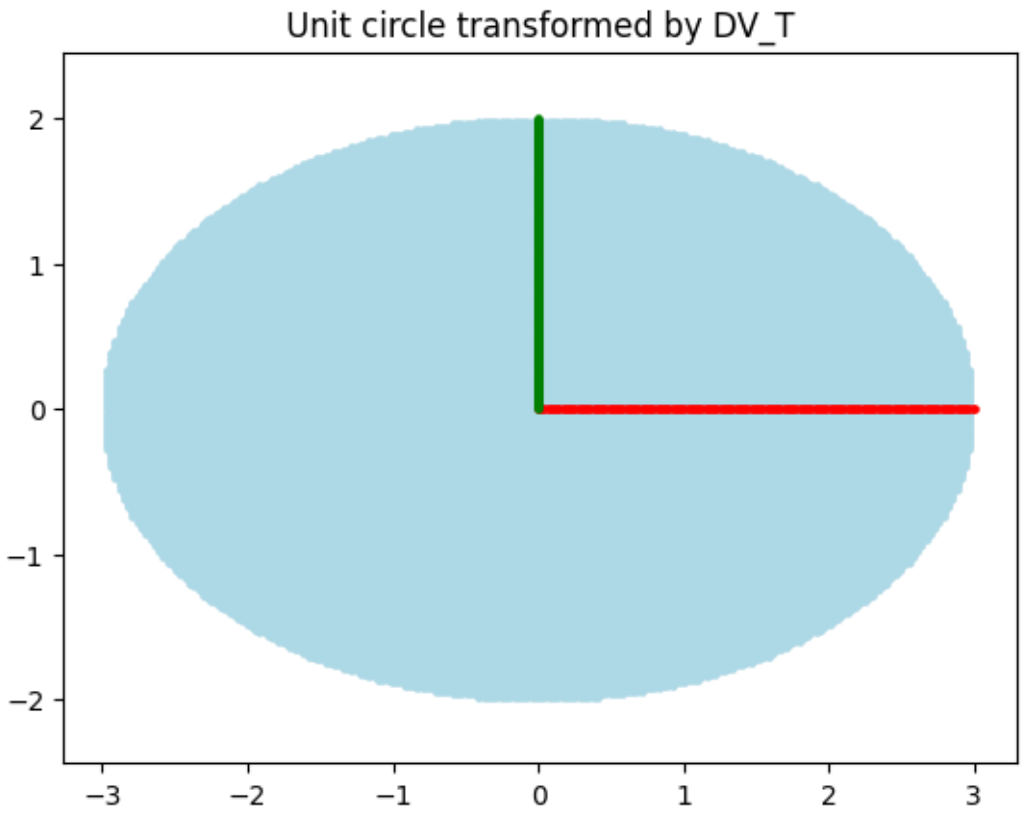
Note that multiplying by D, does not rotate or reflect points in any way. It is a purely scaling operation where different directions get scaled by different values based on entries of D.

3.0.1 (c) i: Comment on the length of major and minor axis of the ellipse and their orientation with respect to X and Y axis when D has entries [3, 2]. Here V is identity.

[ ]:
```
D_1 = np.array([3, 2])
visualize( D = D_1, show_original=False, show_UDVT=False)
```

```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```



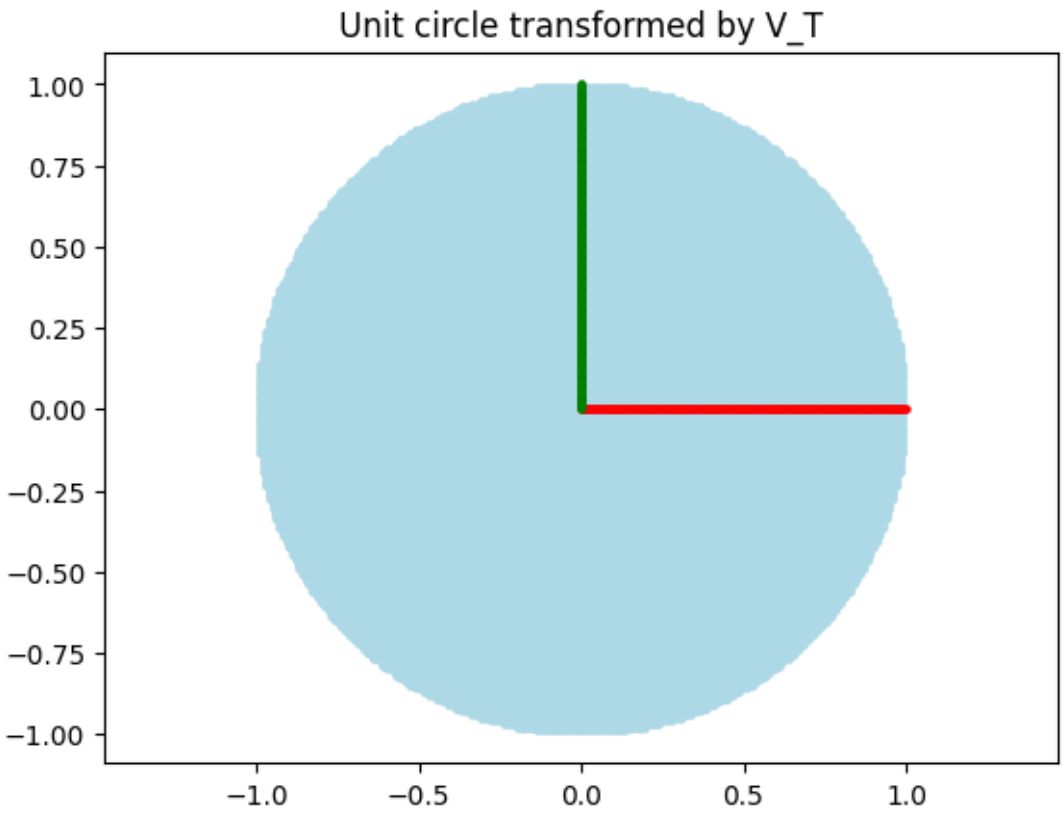


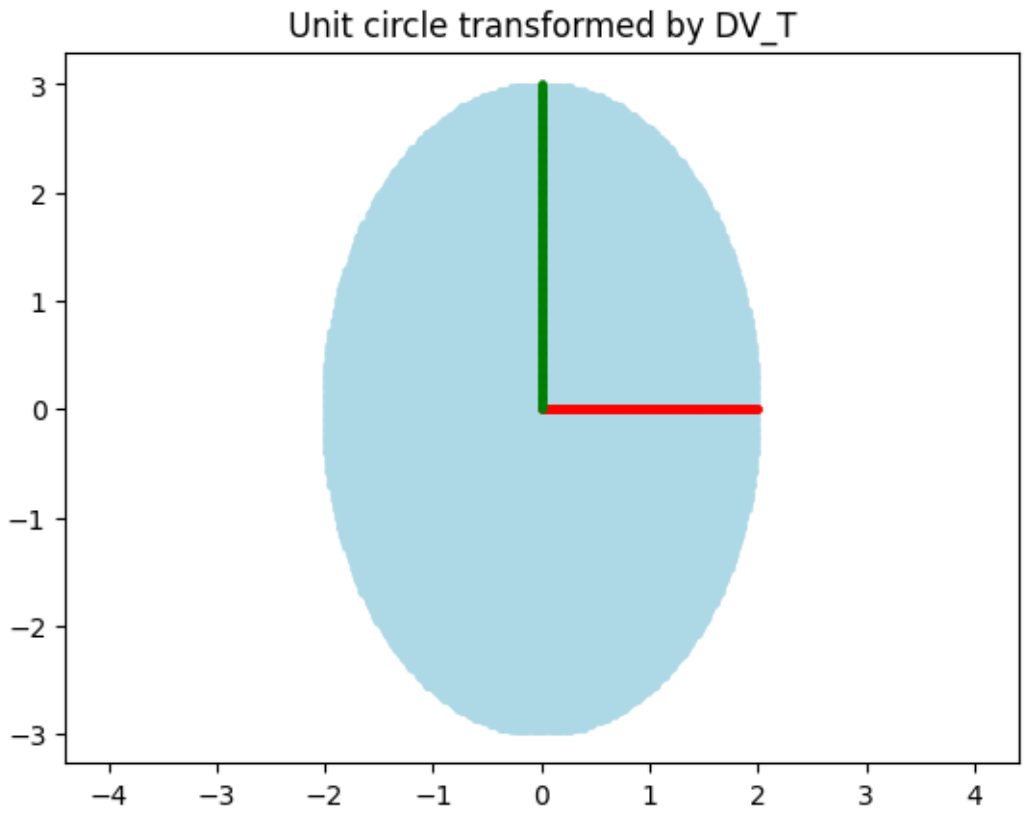
The orientation of the axis is the same, but the lengths have been changed proportionally to the diagonal elements in D. Larger diagonal element correspond longer axis.

3.0.2 (c) ii: Comment on the length of major and minor axis of the ellipse and their orientation with respect to X and Y axis when D has entries [2, 3]. Here V is the identity matrix.

[ ]:
```
D_2 = np.array([2, 3])
visualize( D = D_2,  show_original=False, show_UDVT=False)
```

```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```


Unit circle transformed by V_T
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0

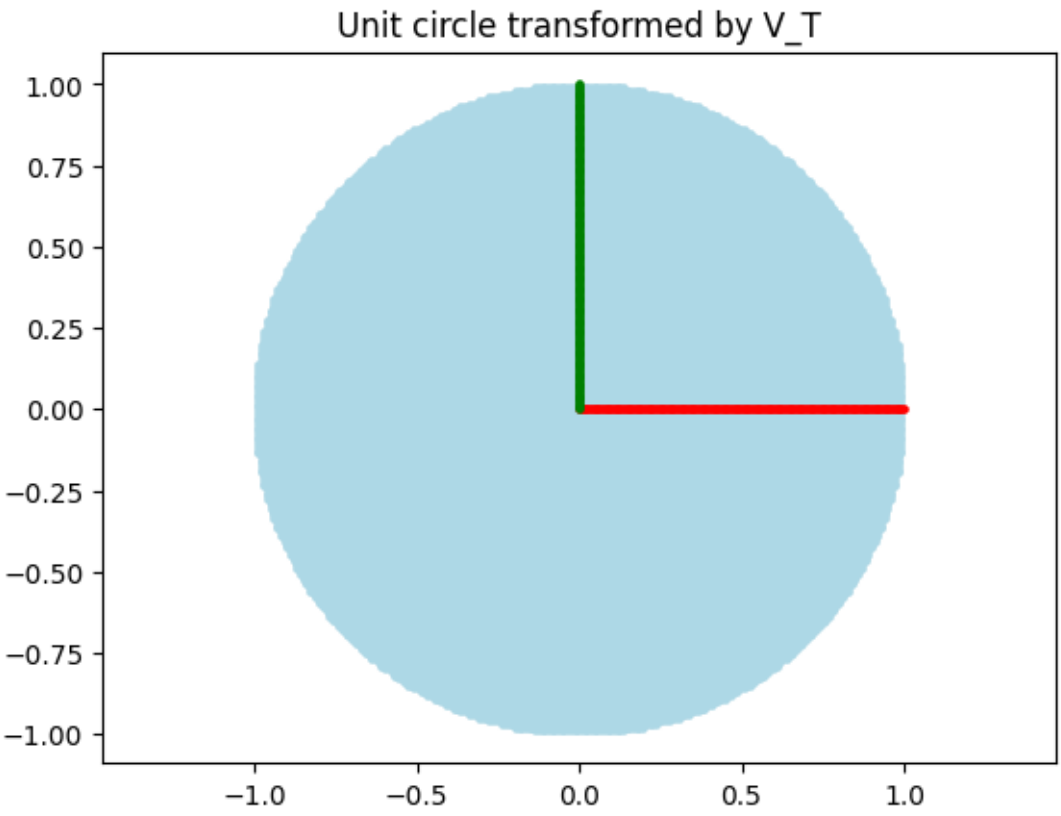


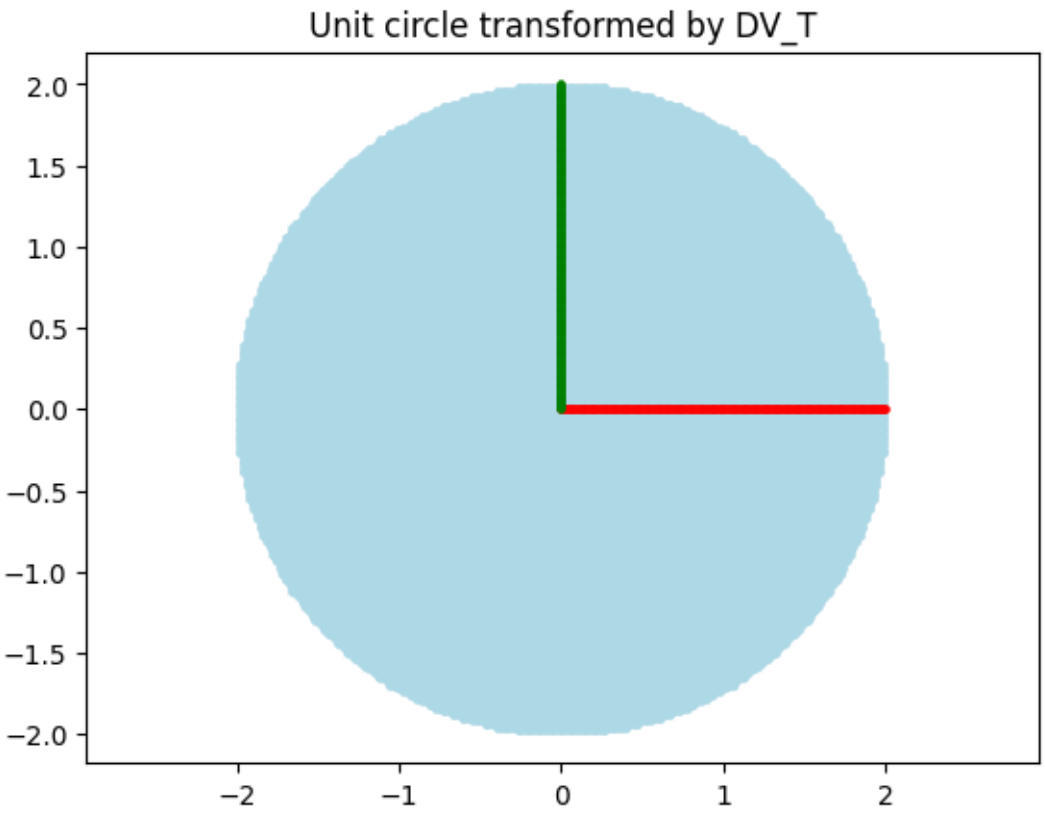
Still, the green and red axis stay the same. The major axis is the green line.

3.0.3 (c) iii: What can you say about the ellipse when D has entries [2, 2]? Here V is the identity matrix.

[ ]:
```
D_3 = np.array([2, 2])
visualize( D = D_3, show_original=False, show_UDVT=False)
```

```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```


Unit circle transformed by V_T
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0

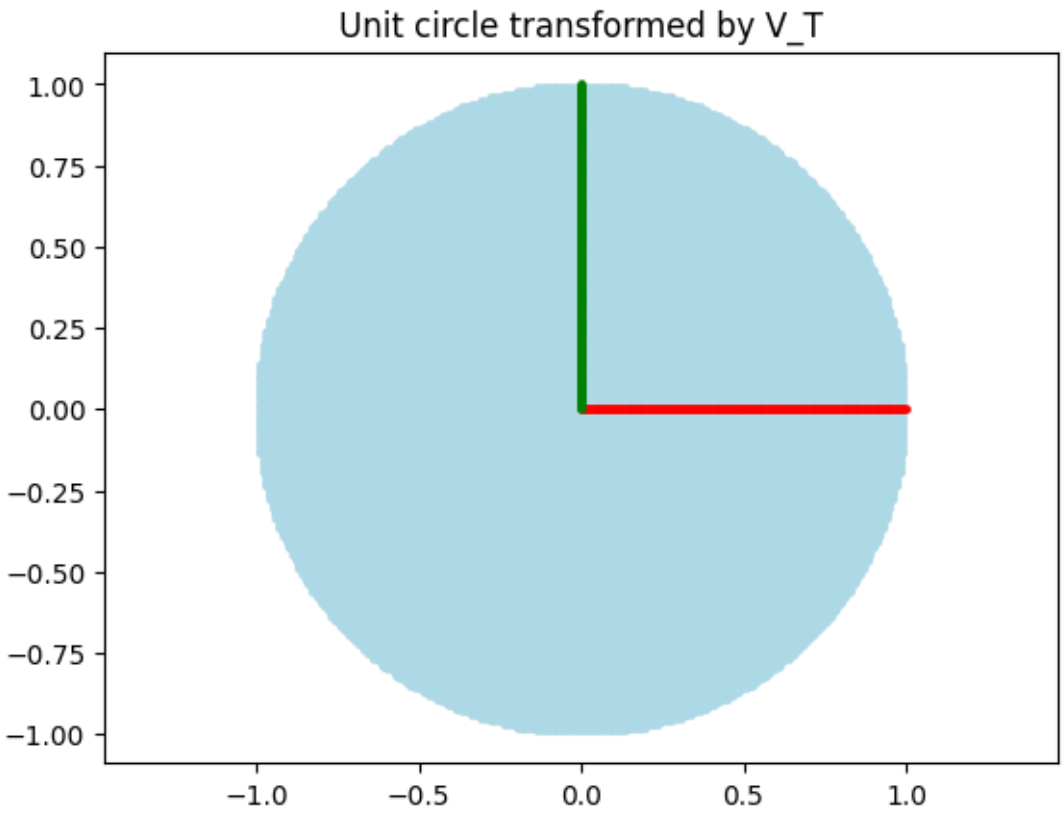


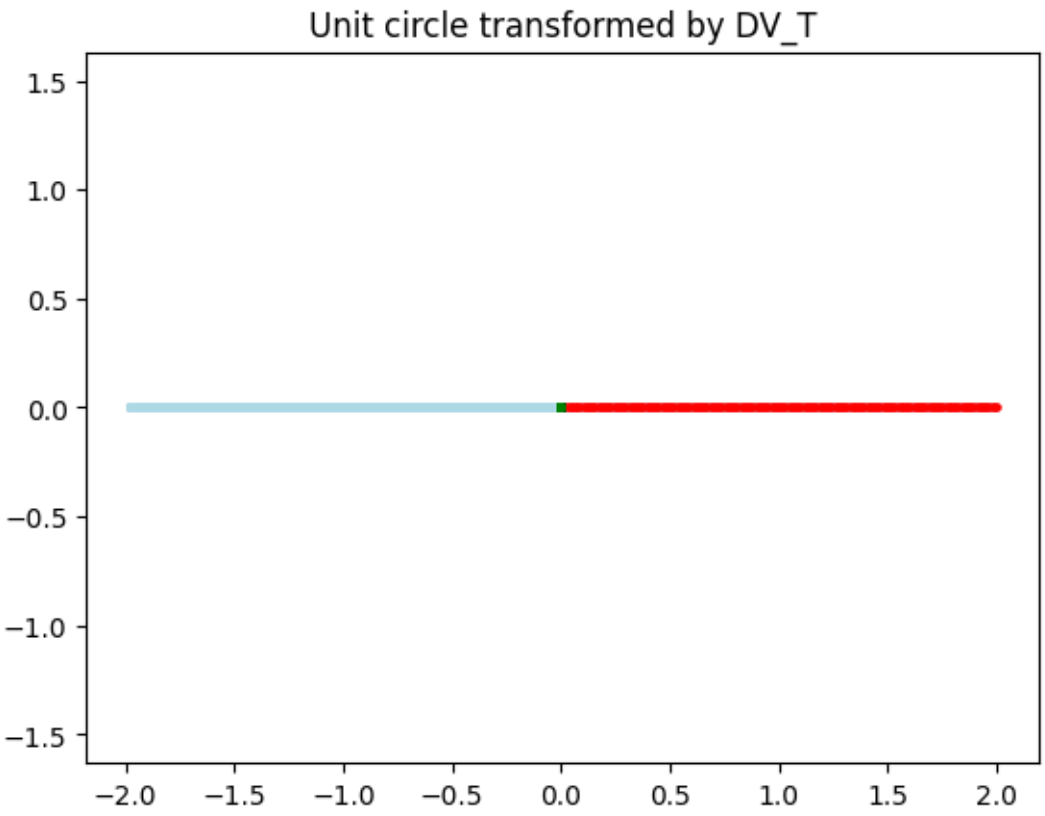
The whole unit circle is magnified by 2, with orientation the same.

3.0.4 (c) iv: What can you say about the ellipse when D has entries [2, 0]? Here V is the identity matrix.

[ ]:
```
D_4 = np.array([2, 0])
visualize( D = D_4, show_original=False, show_UDVT=False)
```

```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```


Unit circle transformed by V_T
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0



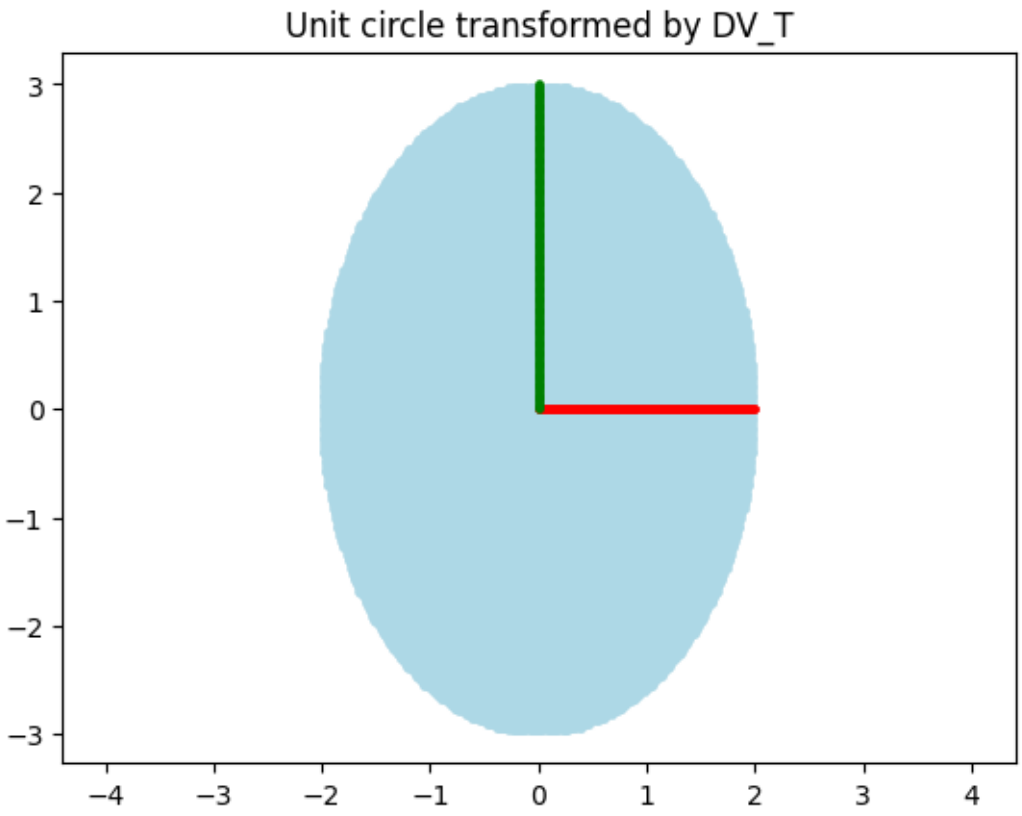
The dimension is reduced to 1. We project the ellipse to a line on x-axis.

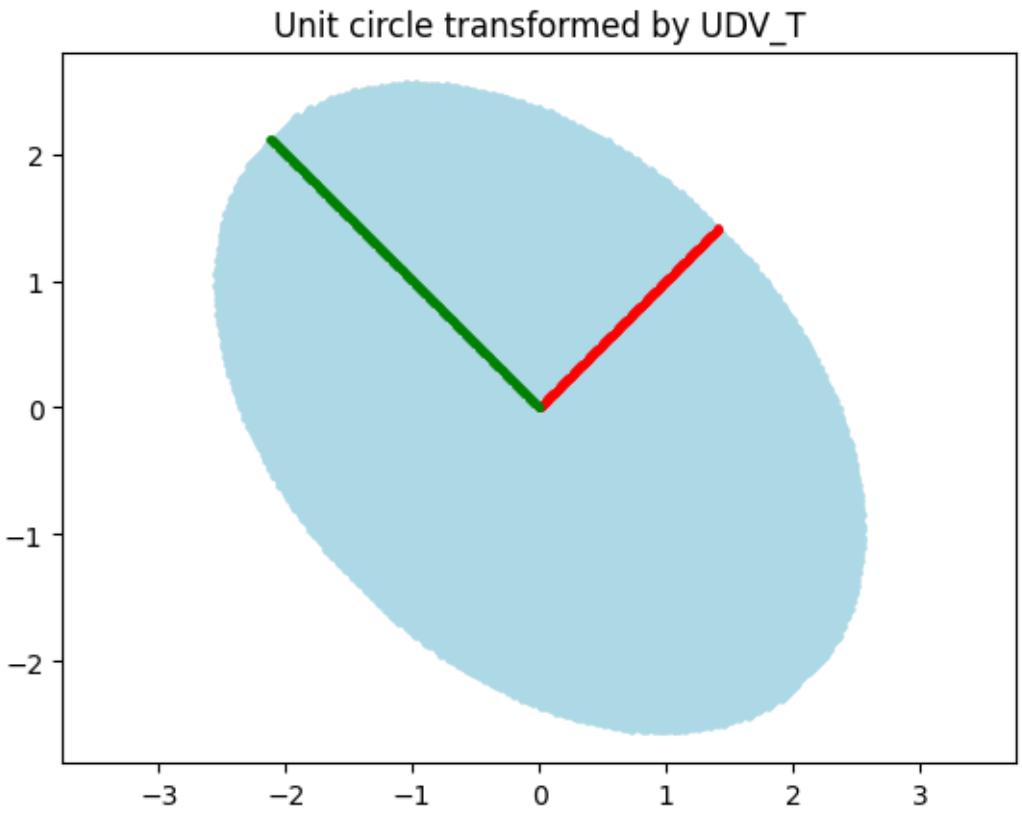
4 Effect of the linear transformation by an orthonormal matrix U

As we saw before for V^T, a 2 x 2 orthonormal matrix can be viewed as a linear transformation that performs some combination of rotations and reflections.

4.0.1 (d) i: Comment on the effect of $U = RCC\left(\frac{\pi}{4}\right)$ as in cell below. The value of D is in the code below and V is the identity matrix. What happened to the ellipse? Did the length of the major and minor axis change?

[ ]:
```
U_1 = get_RCC(np.pi/4)
visualize( U = U_1, D =np.array([2,3]), show_original=False, show_VT=False)
```

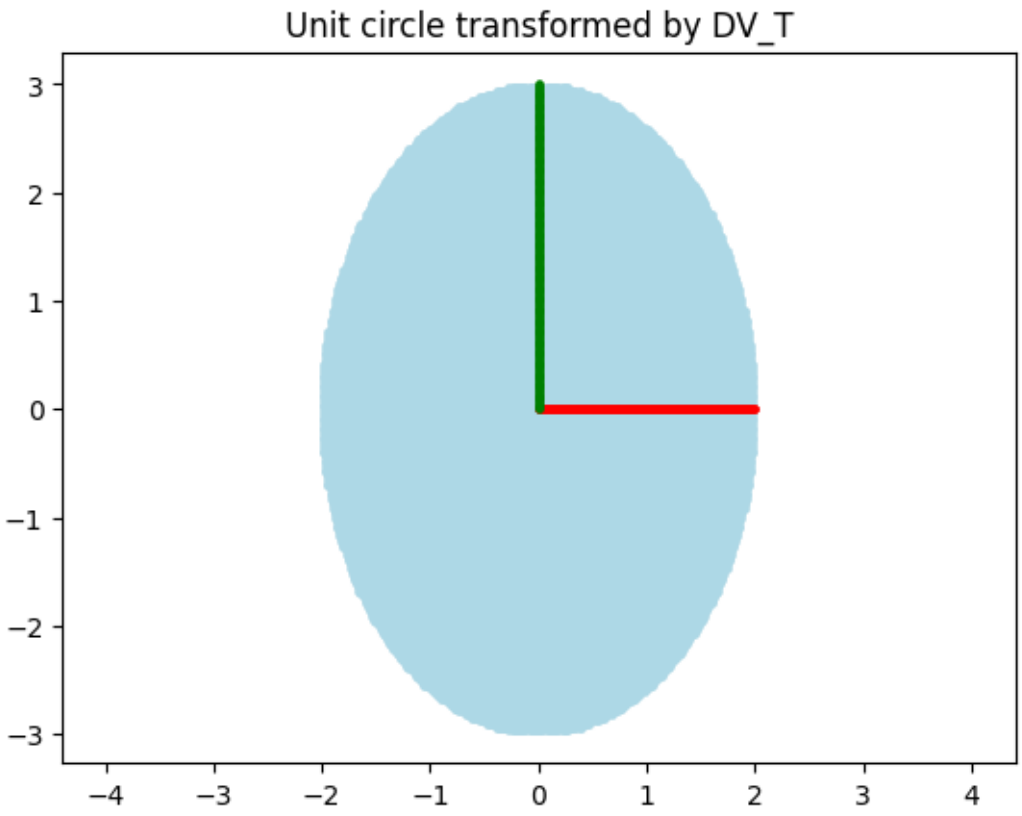

Unit circle transformed by DV_T
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3
4

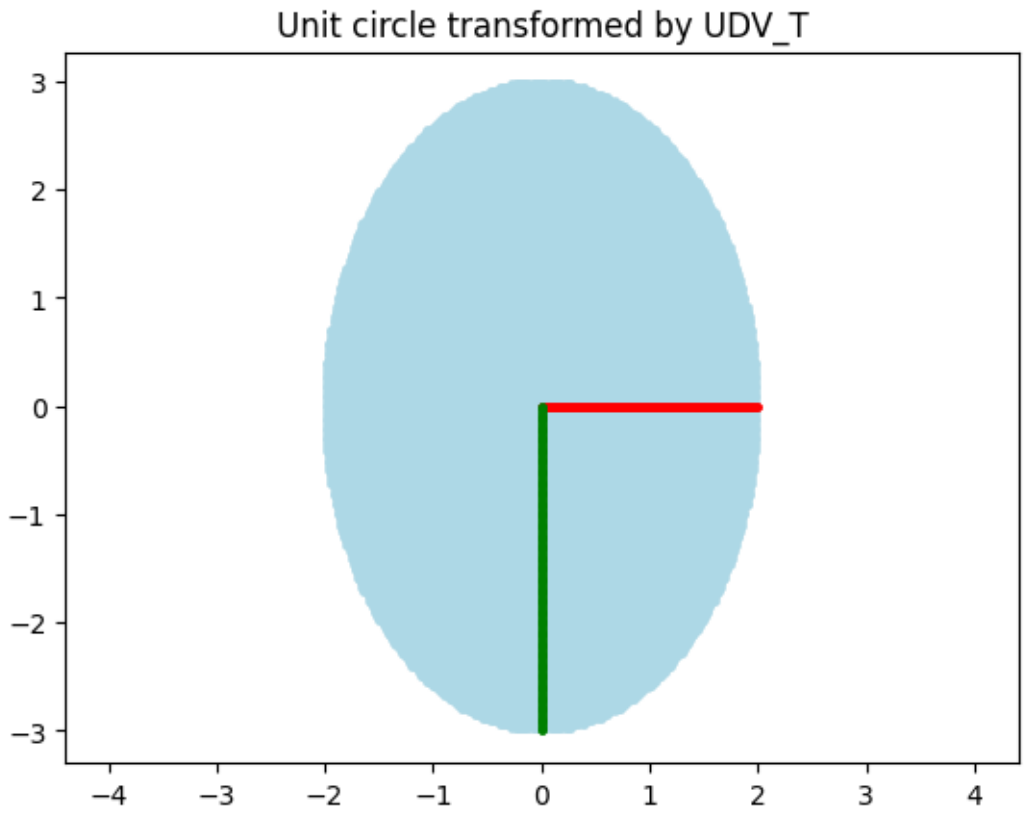


The ellipse is rotated by $ / 4 $. No, the magnitude of it stays the same.

4.0.2 (d) ii: Comment on the effect of $U = RFx()$ as in cell below. The value of D is in the code below and V is the identity matrix. What happened to the ellipse? Did length of major and minor axis change?

[ ]:
```
U_2 = get_RFx()
visualize( U = U_2, D =np.array([2,3]), show_original=False, show_VT=False)
```


Unit circle transformed by DV_T
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3
4

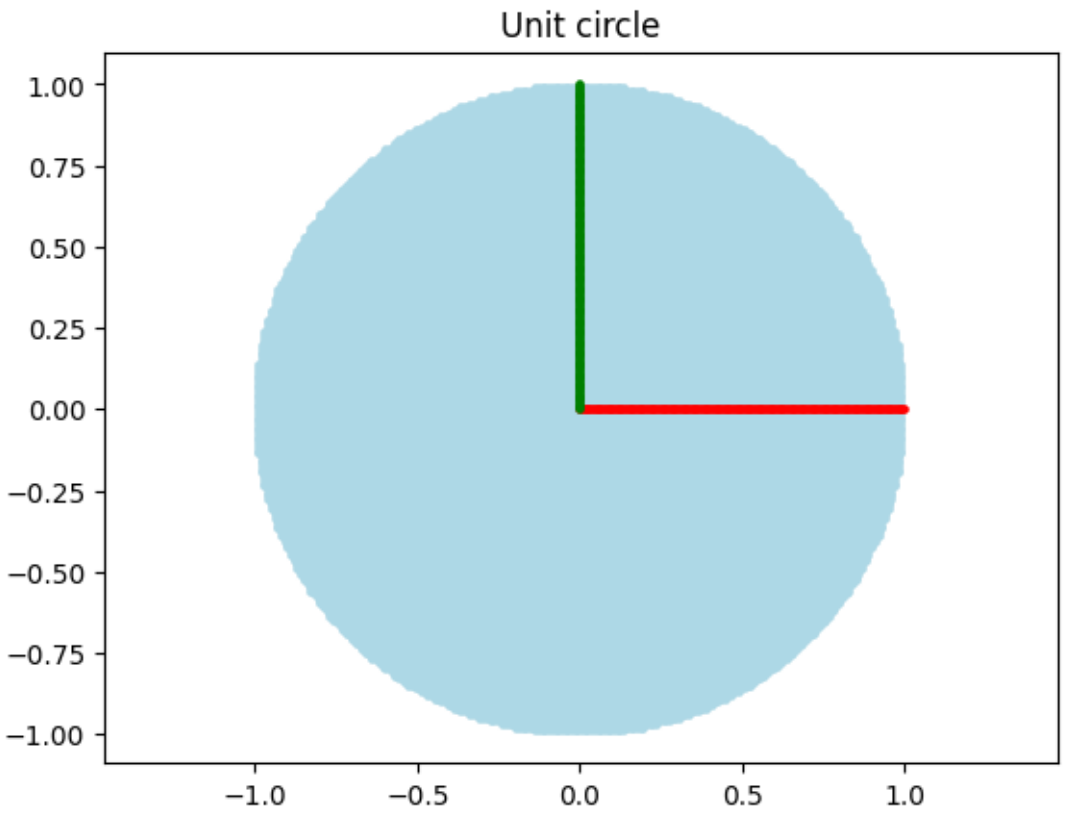


It is reflected by x-axis. No, the magnitude is the same.

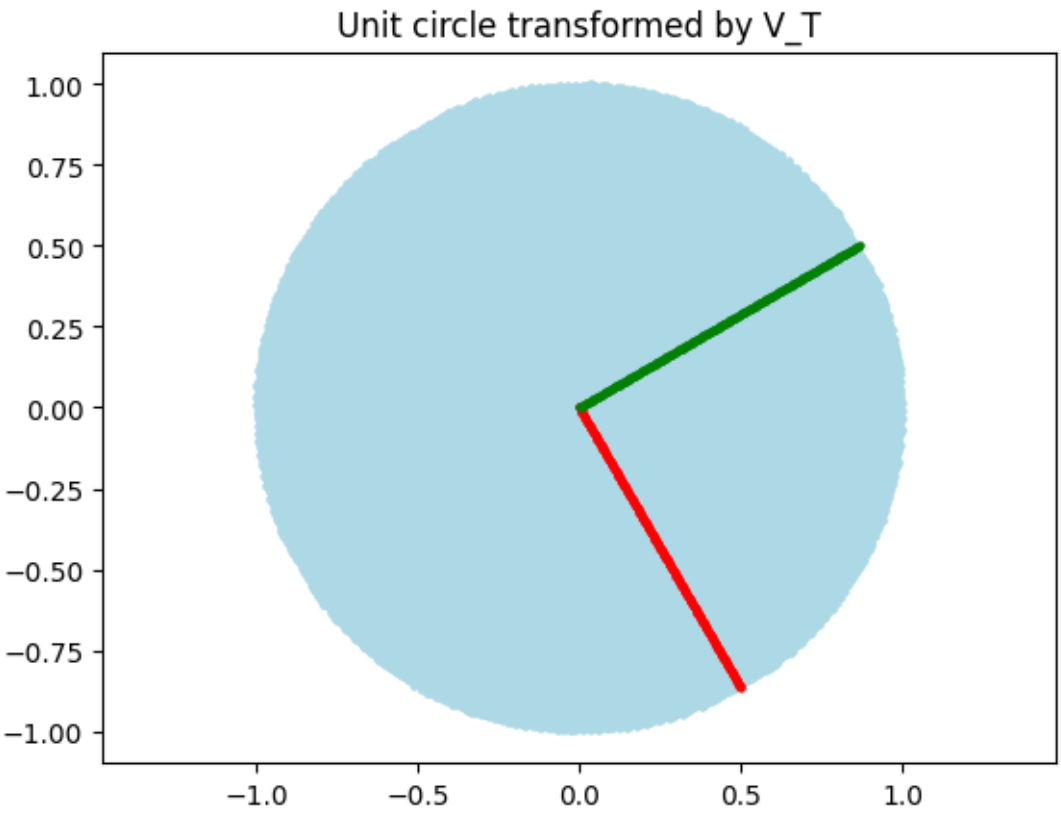
5 Putting everything together. Effect of linear transformation by UDV^T

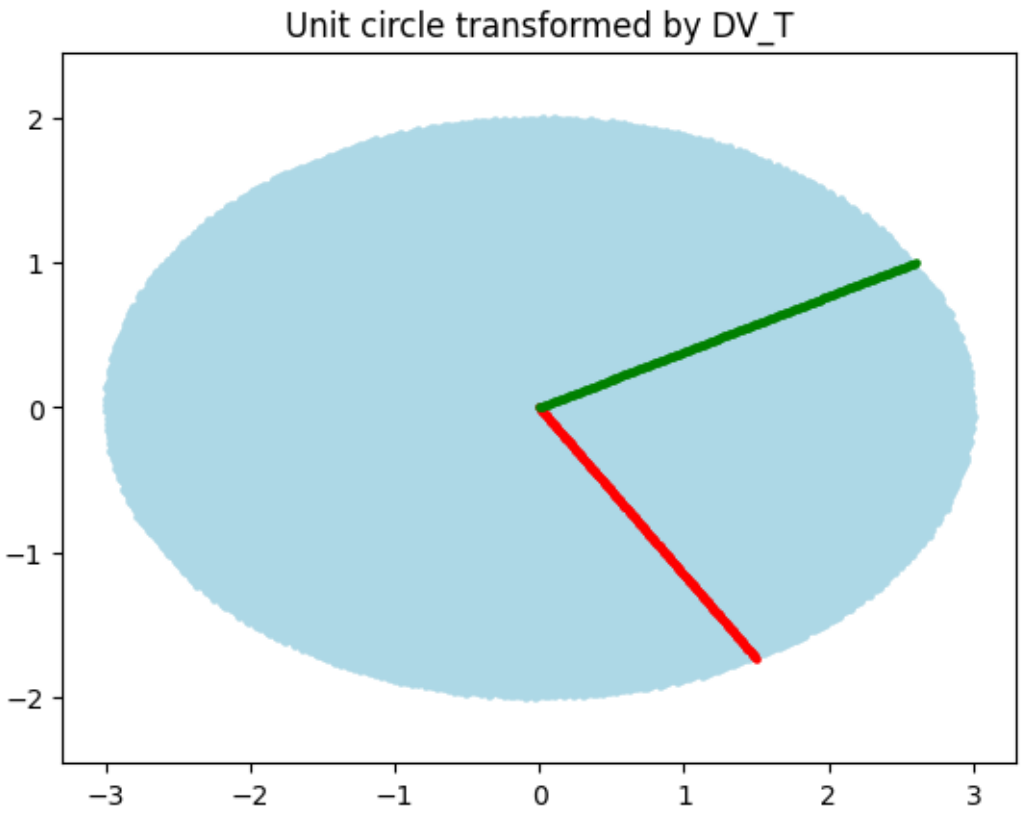
5.0.1 Case I

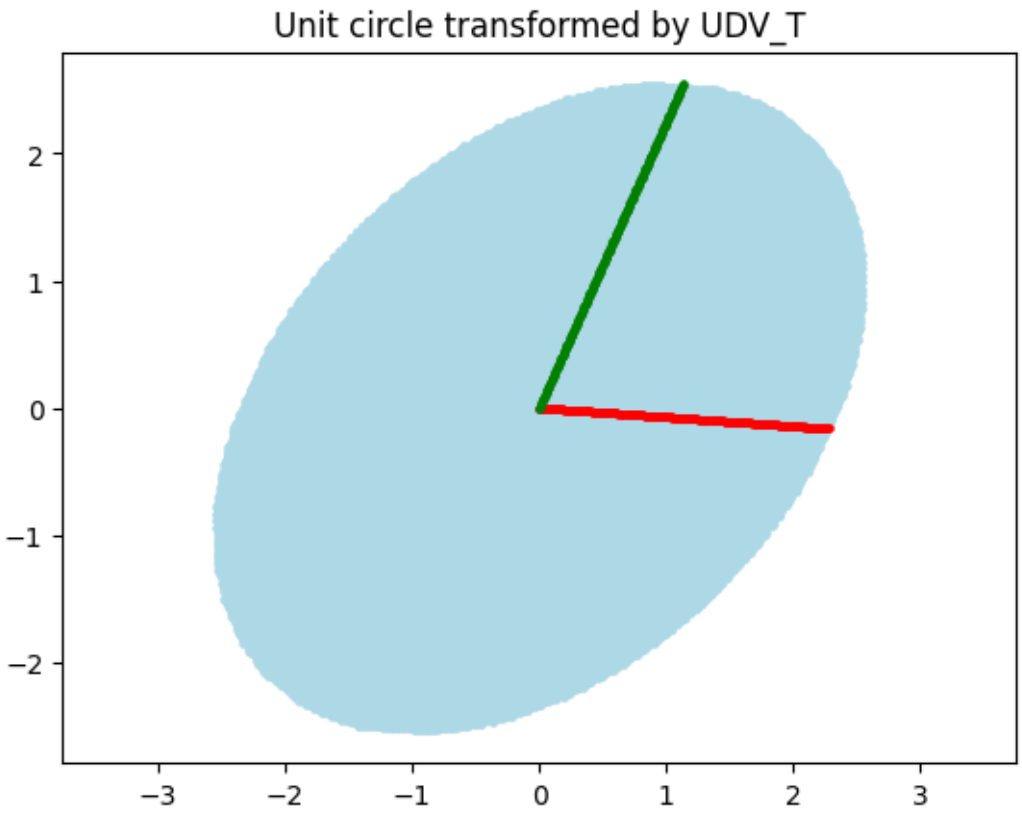
[ ]:
```
U = get_RCC(np.pi/4)
VT = get_RCC(-np.pi/3)
D = np.array([3,2])
visualize(U = U, VT= VT, D=D)
```




```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```


Unit circle transformed by V_T
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0

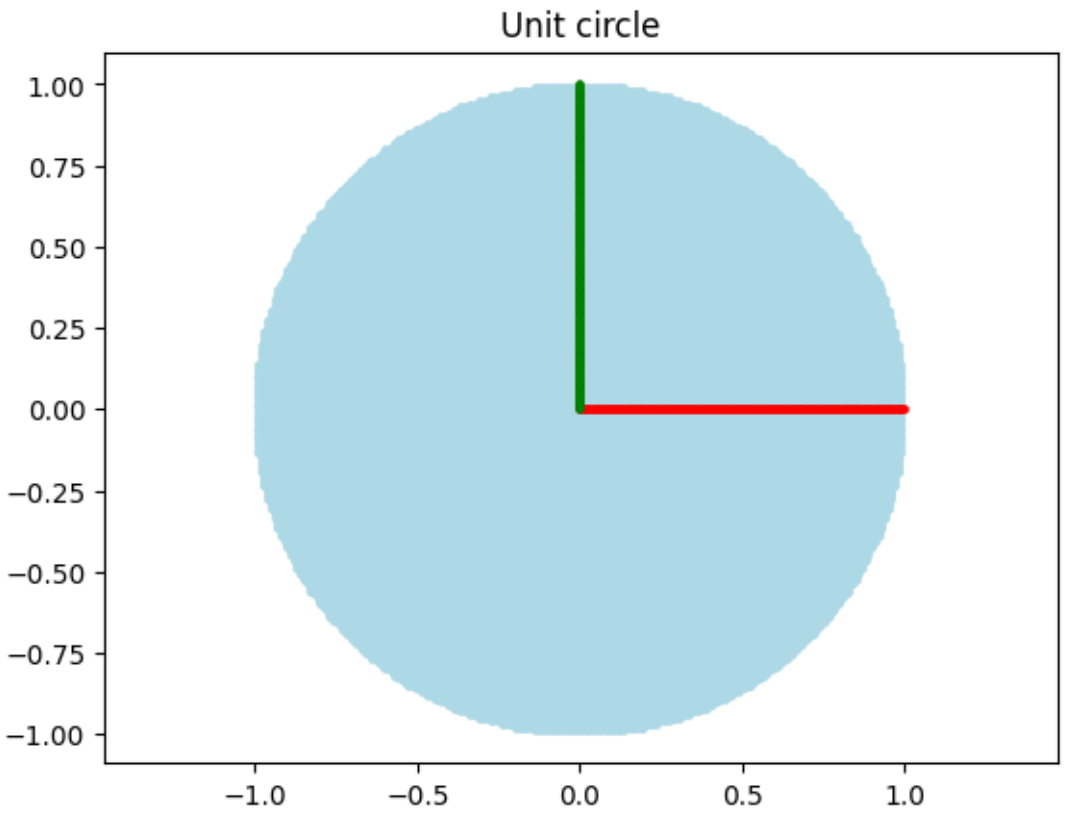

Unit circle transformed by DV_T
2
1
0
−1
−2
−3
−2
−1
0
1
2
3



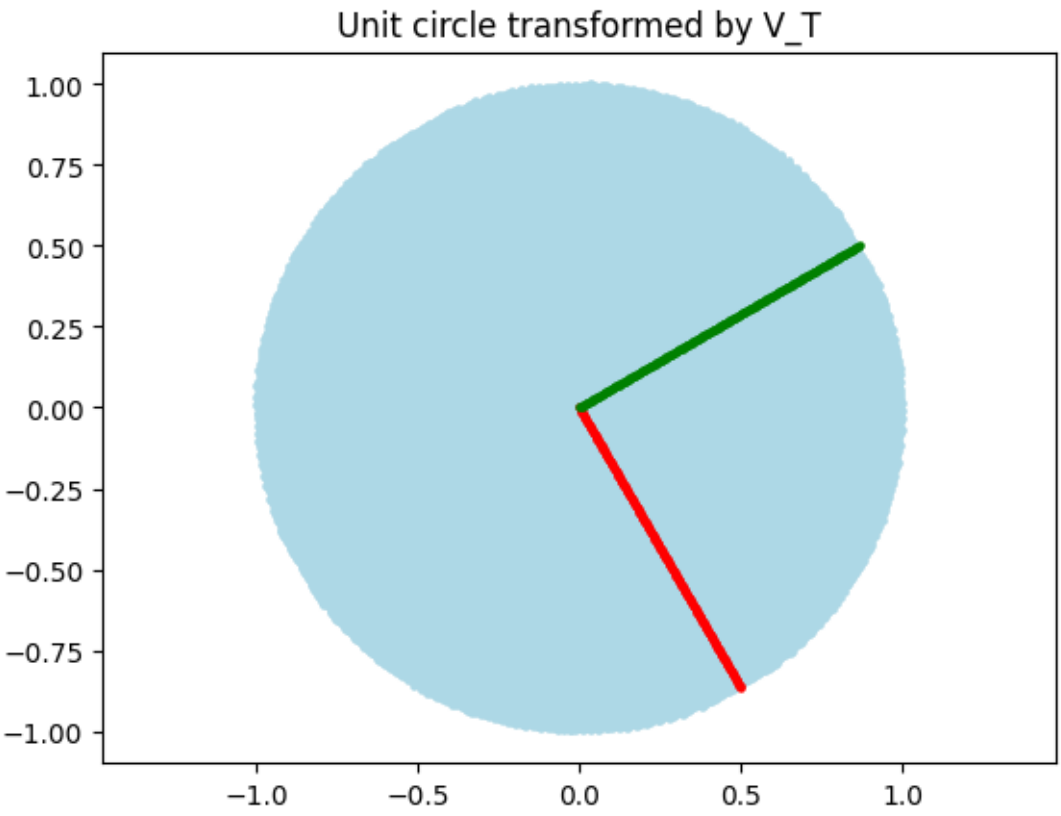
The above figures show the transformation after each step.

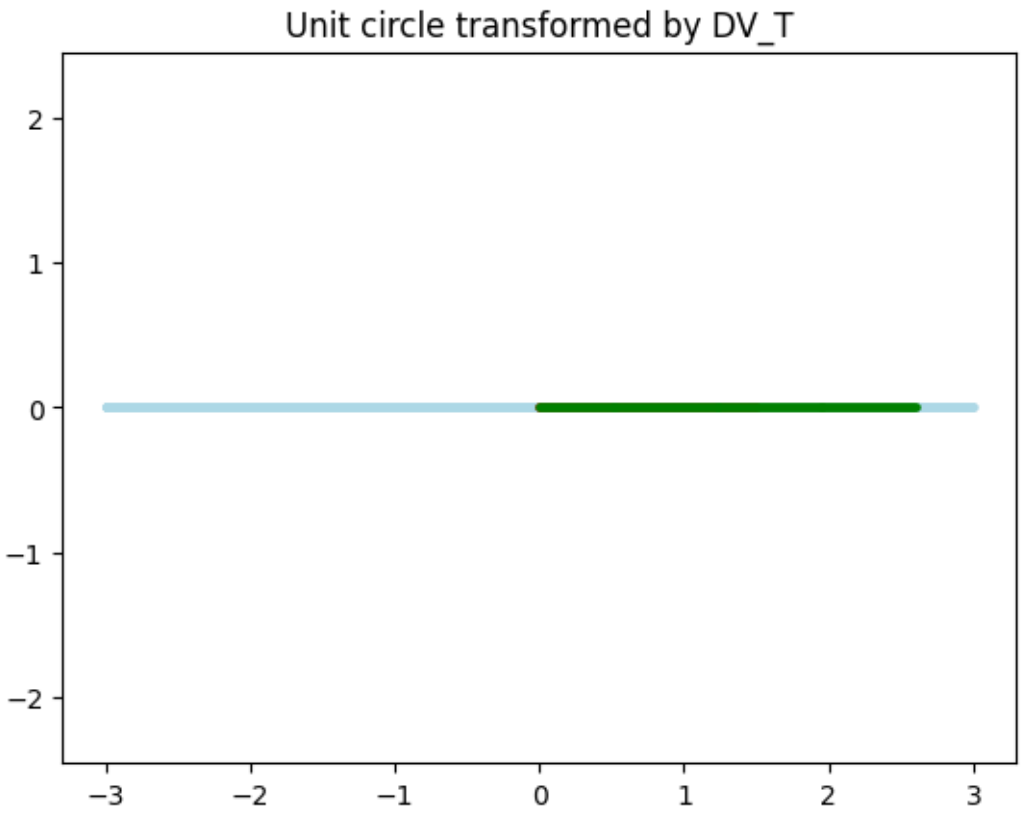
5.0.2 Case II

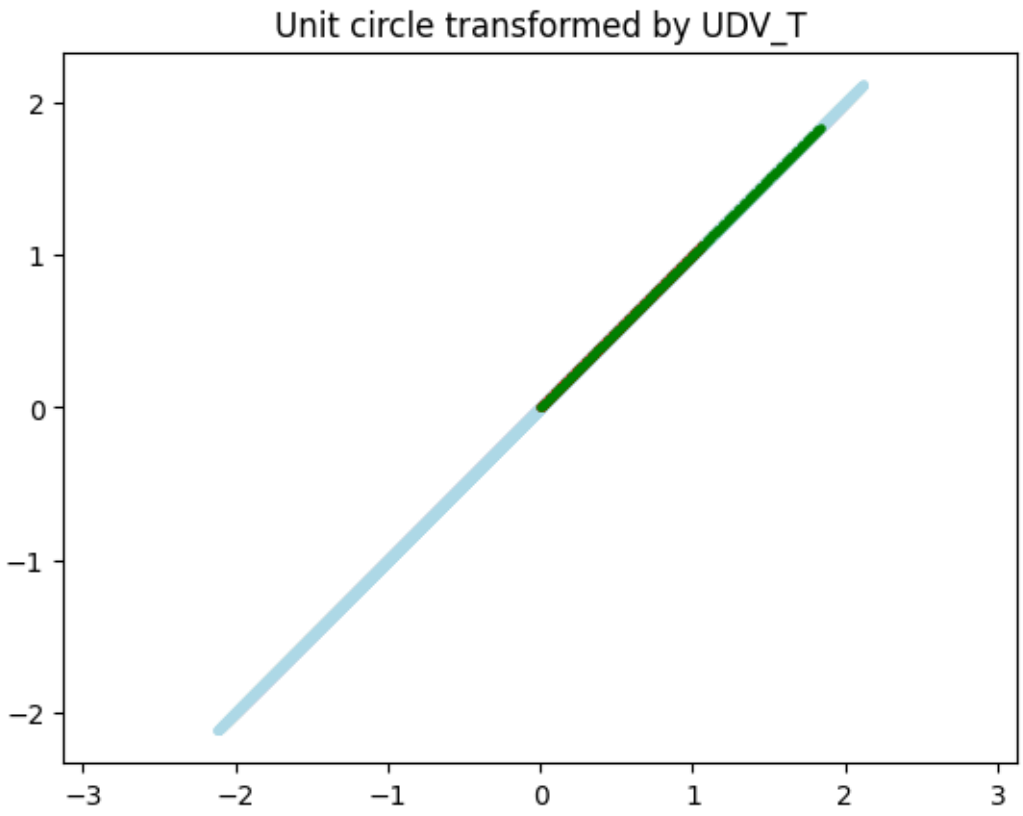
[ ]:
```
U = get_RCC(np.pi/4)
VT = get_RCC(-np.pi/3)
D = np.array([3,0])
visualize(U = U, VT= VT, D=D)
```




```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```


Unit circle transformed by V_T
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0

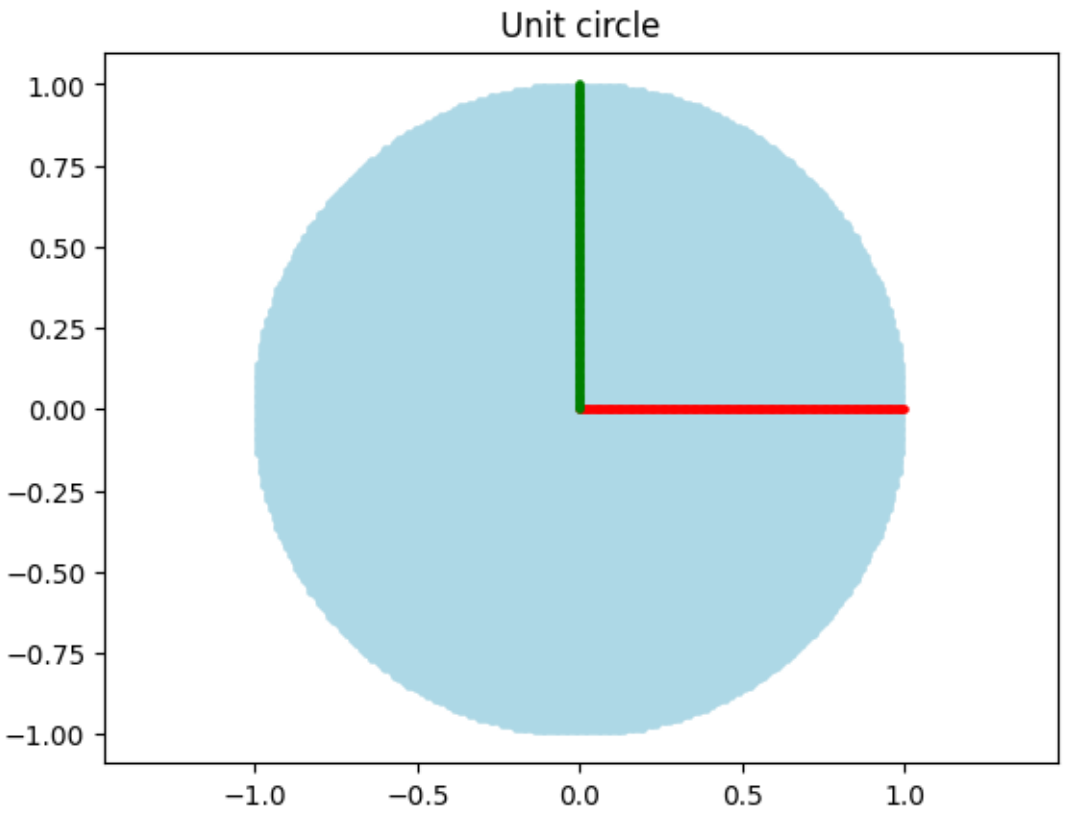

Unit circle transformed by DV_T
2
1
0
−1
−2
−3
−2
−1
0
1
2
3



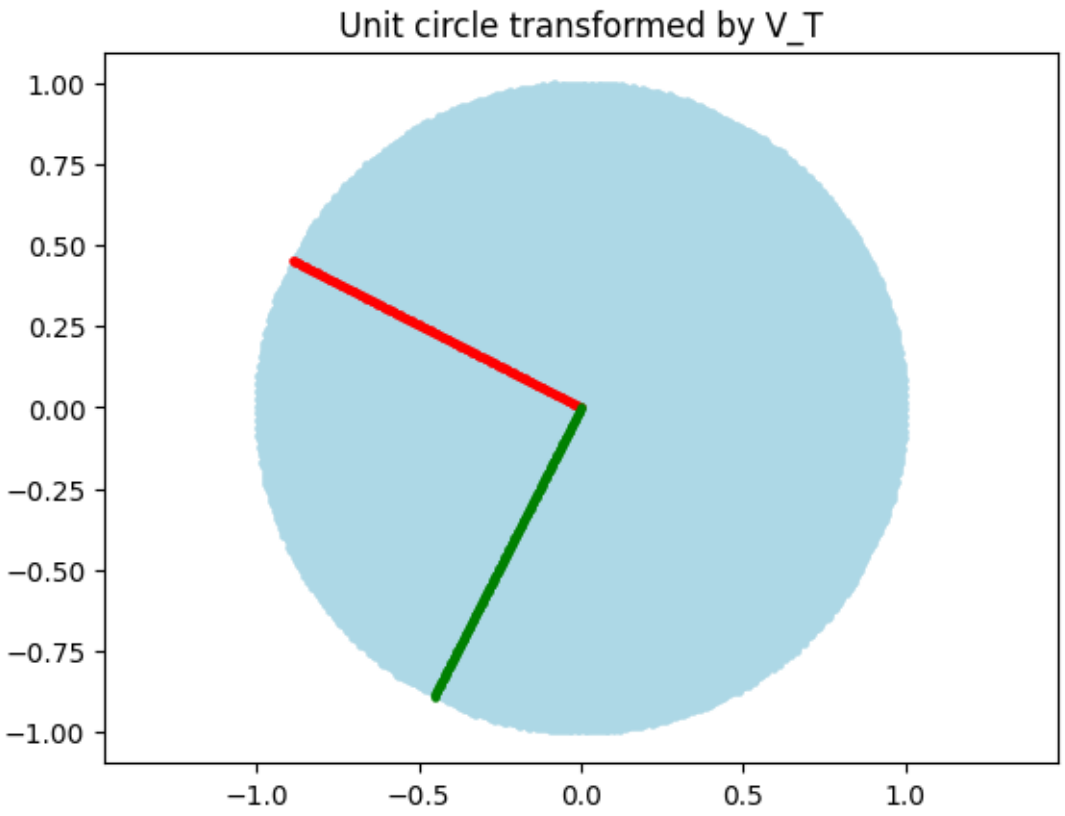
The above figures show the transformation after each step.

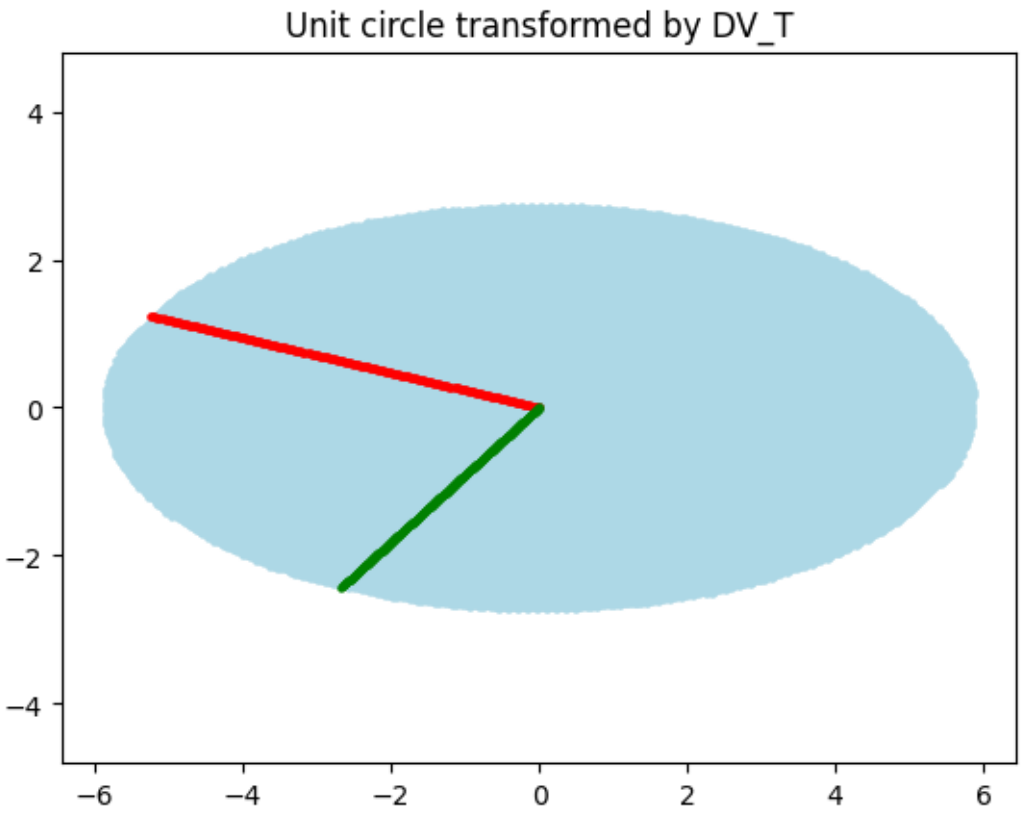
5.0.3 Case III

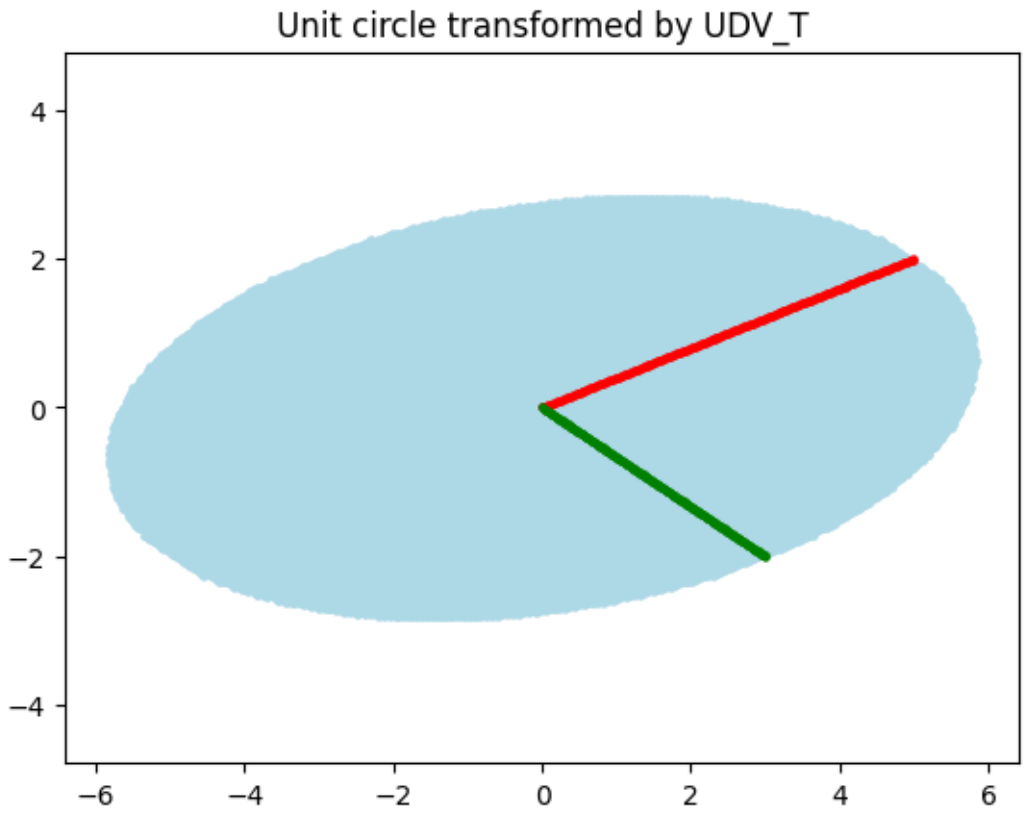
[ ]:
```
A = np.array([[5, 3], [2, -2]])
U,D,VT = np.linalg.svd(A)
visualize(U = U, D=D, VT=VT)
```




```
WARNING:matplotlib.axes._base:Ignoring fixed x limits to fulfill fixed data
aspect with adjustable data limits.
```


Unit circle transformed by V_T
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0


Unit circle transformed by DV_T
4
2
0
−2
−4
−6
−4
−2
0
2
4
6



5.0.4 (e): For case III, based on the figures obtained by running the cell, answer the following questions:

1) Is V^T a pure rotation, pure reflection or combination of both?

2) Let σ_1 and σ_2 denote the entries of the diagonal matrix in SVD of A, with $\sigma_1 > \sigma_2$. What is an approximate value of $\frac{\sigma_1}{\sigma_2}$?

3) Is U a pure rotation, pure reflection or combination of both?

1) $V^\top$ is pure rotation

2) We can approximate it by the ratio of the lengths of the long axis and short axis of the ellipse after the transformation.

3) A combination of both.

6 Exploration Area (Not part of homework question)

You are free to visualize the effect of the SVD transformation on the unit circle for whatever matrix you desire

[ ]:
```
# #Sample format 1
# U = get_RCC(np.pi/4)
# VT = get_RCC(-np.pi/3)
# D = np.array([3,2])
# visualize(U = U, VT= VT, D=D)
```

[ ]:
```
# #Sample format 2
# A = np.array([[5, 3], [2, -2]])
# U,D,VT = np.linalg.svd(A)
# visualize(U = U, D=D, VT=VT)
```

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```

```