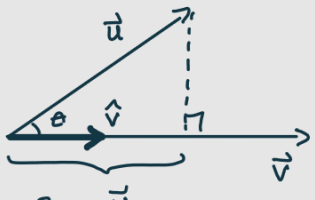
Projection of $\vec{u}$ on $\vec{v}$

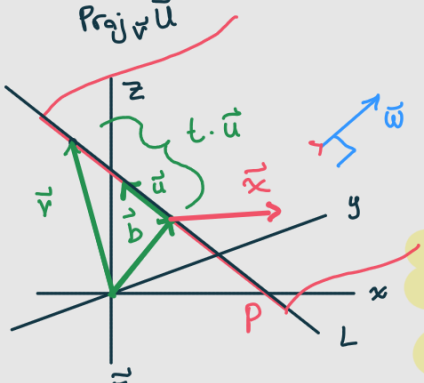


$\text{proj}_{\vec{v}}\vec{u} = l \cdot \hat{v}$ → scaling of unit vector in direction of $\vec{v}$

now, $\cos\theta = \|\text{Proj}_{\vec{v}}\vec{u}\| / \|\vec{u}\| = l\|\hat{v}\| / \|\vec{u}\| = l/\|\vec{u}\|$

and $\cos\theta = \frac{\langle\vec{u},\vec{v}\rangle}{\|\vec{u}\|\|\vec{v}\|} = \frac{\langle\vec{u},\vec{v}\rangle}{\|\vec{u}\|\|\hat{v}\|} = \frac{\langle\vec{u},\hat{v}\rangle}{\|\vec{u}\|}$

so $\frac{l}{\|\vec{u}\|} = \frac{\langle\vec{u},\vec{v}\rangle}{\|\vec{u}\|} \Rightarrow l = \langle\vec{u},\hat{v}\rangle$ and $\text{Proj}_{\vec{v}}\vec{u} = \langle\vec{u},\hat{v}\rangle\hat{v}$



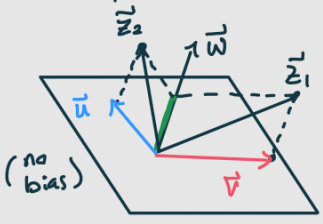
$L = \{\vec{v}\in\mathbb{R}^2 \mid \vec{v} = \vec{b} + t\vec{u},\ t\in\mathbb{R}\}$

$P = \{\vec{v}\in\mathbb{R}^3 \mid \vec{v} = \vec{b} + t\vec{u} + l\vec{x},\ t, l\in\mathbb{R} \text{ and } \vec{u},\vec{x}\in\mathbb{R}^3\}$

$P = \{\vec{v}\in\mathbb{R}^3 \mid \langle(\vec{v}-\vec{b}),\vec{\omega}\rangle = 0,\ \vec{b},\vec{\omega}\in\mathbb{R}^3\}$ *Normal vector form will always describe a n-1 dimensional hyperplane in $\mathbb{R}^n$

$= \{\vec{v}\in\mathbb{R}^3 \mid \langle\vec{v},\vec{\omega}\rangle - c = 0,\ \vec{\omega}\in\mathbb{R}^3 \wedge c\in\mathbb{R}\}$

Note* $(\vec{v}-\vec{b})\cdot\vec{\omega} = (\vec{v}\cdot\vec{\omega}) - (\vec{b}\cdot\vec{\omega}) = \vec{v}\cdot\omega - c$



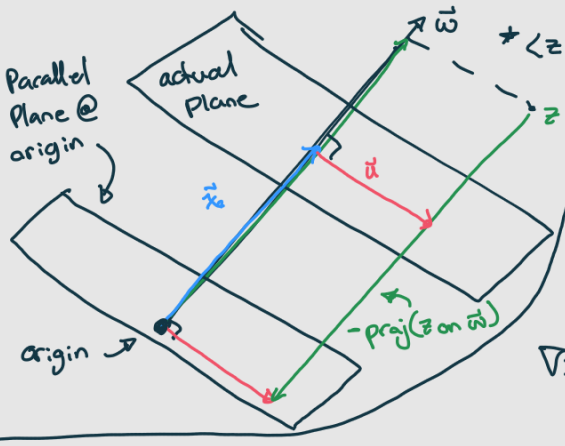
$\vec{\omega} = \vec{u}\times\vec{v}$, $\vec{u} = z_2 - \text{proj}(\vec{z}_2,\vec{\omega})$ & $\vec{v} = \vec{u}\times\vec{\omega}$ or $\vec{v} = z_1 - \text{proj}(\vec{z}_1,\vec{\omega})$

So now $P = \{\vec{x}\in\mathbb{R}^3 \mid \vec{x} = a\vec{u} + b\vec{v},\ a,b\in\mathbb{R}\} \longleftrightarrow P = \{\vec{x}\in\mathbb{R}^3 \mid \langle\vec{x},\vec{\omega}\rangle = 0\}$

(with bias) $P = \{\vec{x}\in\mathbb{R}^3 \mid \langle\vec{x},\vec{\omega}\rangle + b = 0\}$ take $\vec{x}_0\in P$ s.t. $\vec{x}_0 = c\cdot\vec{\omega}$, $c\in\mathbb{R}$

So $\langle x_0,\vec{\omega}\rangle + b = 0 \rightarrow \langle(c\cdot\vec{\omega}),\vec{\omega}\rangle = -b \rightarrow c = \frac{-b}{\|\vec{\omega}\|} \rightarrow \vec{x}_0 = \left(\frac{-b}{\|\vec{\omega}\|^2}\right)\vec{\omega} \rightarrow$ $\|\vec{x}_0\|$ = distance b/w Plane & origin

$\therefore \vec{u} = \vec{z} - \text{proj}(\vec{z}\text{ on }\vec{\omega}) + \vec{x}_0 \in P = \{\vec{u}\in\mathbb{R}^3 \mid \langle u,\vec{\omega}\rangle + b = 0,\ b\in\mathbb{R},\vec{\omega}\in\mathbb{R}^3\}$



* $\langle z,\hat{\omega}\rangle + b = 0$ → on plane, $\langle z,\hat{\omega}\rangle + b < 0$ → below plane

$\langle z,\vec{\omega}\rangle + b > 0$ → above plane

Potential Field: $f: \mathbb{R}^M \mapsto \mathbb{R}^K$, $K = 1$ (scalar functions over vectors)

Vector Field: $f: \mathbb{R}^M \mapsto \mathbb{R}^K$, $K > 1$

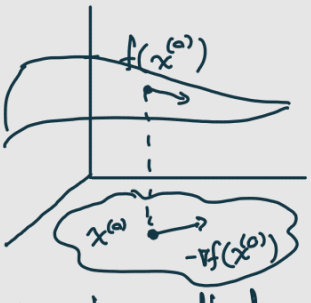
$\nabla f\left(\begin{bmatrix}\vdots\\ x_i\\ \vdots\end{bmatrix}\right) = \begin{bmatrix}\vdots\\ \partial f/x_i\\ \vdots\end{bmatrix}$ $\frac{\partial f}{\partial x_i}$ is r.o.c in f w.r.t. x_i

f is a potential field, ∇f is a vector field

$\nabla f(\bar{x})$ is direction of maximal increase @ $\bar{x}$, $|\nabla f(\bar{x})|$ is magnitude of maximal increase @ $\bar{x}$

Compositions of known functions are analytical: $f\left(\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}\right) = x_1 + 2x_2 + \sin(x_3)$

Gradient Descent: Minimize f by moving in $\mathbb{R}^n$ $f\left(\begin{bmatrix}\vdots\\ x_i\\ \vdots\end{bmatrix}\right) = \frac{\text{rand}[x_i]}{x_{i+1}\cdots}$ not analytical



$-\nabla f(x^{(a)})$ is direction of maximal decrease @ $\vec{x}_a$ & ϵ is some small step

let $x^{(a)}$ = random initialization location in $\mathbb{R}^n$

for i in range(n):

$\quad x^{(i+1)} = x^{(i)} - \epsilon\nabla f(x^{(i)})$

Momentum method: $dx_i = -\epsilon\nabla f(x^{(i)}) + \beta dx^{(i-1)}$ ↳ if previous update was too large, $dx^{(i-1)}$ will be in the opposite direction from $-\epsilon\nabla f(x^{(i)})$ so $dx^{(i)}$ slows down, if $dx^{(i-1)}$ same direction, $dx^{(i)}$ speed up

$x^{(i-1)} \xrightarrow{dx^{i-1}} x^{(i)} \xrightarrow{dx^{i}} x^{(i+1)}$

(2,3) @ (3,) is a compatible operation, output = (2,)

V.backward() → v.backward() *requires retain_graph = True

If y is part of a computational graph either created using torch.no_grad() use y.detach.numpy

```
# ==============================
# INDEXING & MANIPULATION
# ==============================
# Joins a sequence of arrays along an existing axis
# Example: np.concatenate([np.array([[1, 2], [3, 4]]), np.array([[5, 6], [7, 8]])], axis=0)
# Result: array([[1, 2], [3, 4], [5, 6], [7, 8]])
np.concatenate(arrays: list[np.ndarray], axis: int = 0) -> np.ndarray

# Returns the indices of non-zero elements
# Example: np.nonzero(np.array([[0, 1, 2], [0, 3, 0]]))
# Result: (array([0, 0, 1]), array([1, 2, 1]))
np.nonzero(a: np.ndarray) -> tuple[np.ndarray, ...]

# Repeats elements of an array along a specified axis
# Example: np.repeat(np.array([[1, 2], [3, 4]]), repeats=2, axis=0)
# Result: array([[1, 2], [1, 2], [3, 4], [3, 4]])
np.repeat(a: np.ndarray, repeats: int, axis: int = None) -> np.ndarray

# Reshapes an array without changing its data
# Example: np.reshape(np.array([[1, 2, 3, 4], [5, 6, 7, 8]]), newshape=(4, 2))
# Result: array([[1, 2], [3, 4], [5, 6], [7, 8]])
np.reshape(a: np.ndarray, newshape: tuple[int, ...]) -> np.ndarray

# Splits an array into multiple sub-arrays along a specified axis
# Example: np.split(np.array([[1, 2, 3, 4], [5, 6, 7, 8]]), 2, axis=1)
# Result: [array([[1, 2], [5, 6]]), array([[3, 4], [7, 8]])]
np.split(ary: np.ndarray, indices_or_sections: int, axis: int = 0) -> list[np.ndarray]
```

```
    # Splits an array into multiple sub-arrays along a specified axis
    # Example: np.split(np.array([[1, 2, 3, 4], [5, 6, 7, 8]]), 2, axis=1)
    # Result: [array([[1, 2], [5, 6]]), array([[3, 4], [7, 8]])]
    np.split(ary: np.ndarray, indices_or_sections: int, axis: int = 0) -> list[np.ndarr

    # Removes single-dimensional entries from the shape of an array
    # Example: np.squeeze(np.array([[[2], [3], [4]]]))
    # Result: array([[2, 3, 4]])
    np.squeeze(a: np.ndarray, axis: int = None) -> np.ndarray

    # Constructs an array by repeating the input
    # Example: np.tile(np.array([[1, 2], [3, 4]]), reps=(2, 2))
    # Result: array([[1, 2, 1, 2], [3, 4, 3, 4], [1, 2, 1, 2], [3, 4, 3, 4]])
    np.tile(A: np.ndarray, reps: tuple[int, ...]) -> np.ndarray

    # Converts a flat index into a tuple of coordinate indices
    # Example: np.unravel_index([2, 5, 7], (3, 3))
    # Result: (array([0, 1, 2]), array([2, 2, 1]))
    np.unravel_index(indices: int, dims: tuple[int, ...]) -> tuple[int, ...]
    # Unravel - Custom/contextual usage example (details depend on context)
    # example
    x = np.array([[3, 5, 2, 4],
                  [7, 6, 8, 8],
                  [1, 6, 7, 7]])
    #Bottom right:
    x[1:3,-1:-3:-1] # = [[8,8]
    #                    [7,7]]
```

- if $A.shape \neq B.shape$
  1. runtime error
  2. **manually** adjust the shape using `np.reshape` or `np.repeat`
  3. **broadcasting** is applied using the following rules:
     - if `len(A.shape)` $\neq$ `len(B.shape)`, smaller array compared to larger array along dimensions **right-to-left**
     - leading dimensions of size 1 prepended to smaller array until same size as larger array
     - dimension size mismatch with one axis size=1, this axis is repeated to match the larger axis size
     - dimension size mismatch where no axes size=1, error

```
# ==============================
# LINEAR ALGEBRA FUNCTIONS
# ==============================
# Computes the determinant of an array
np.linalg.det(a: np.ndarray) -> float
# Computes the dot product of two arrays
np.dot(a: np.ndarray, b: np.ndarray) -> np.ndarray
# Computes the eigenvalues and right eigenvectors of a square matrix
np.linalg.eig(a: np.ndarray) -> tuple[np.ndarray, np.ndarray]
# Computes the inverse of a matrix
np.linalg.inv(a: np.ndarray) -> np.ndarray

# ==============================
# RANDOMIZATION FUNCTIONS
# ==============================
# Generates random numbers from a normal (Gaussian) distribution
# loc: Mean of the distribution
# scale: Standard deviation of the distribution
# size: Output shape
np.random.normal(loc: float = 0.0, scale: float = 1.0, size: tuple[i

# Generates random numbers from a uniform distribution
# low: Lower boundary of the output interval
# high: Upper boundary of the output interval
# size: Output shape
np.random.uniform(low: float = 0.0, high: float = 1.0, size: tuple[i
```

```
# ==============================
# BROADCASTING RULES
# ==============================
a = np.random.random((1,2,3,4,5))
b = np.ones((■(1,2,3))
print((a+b).shape)#error

a = np.random.random((1,2,3,4,5))
b = np.ones((1,2,3,1,1))
print((a+b).shape) # = (1, 2, 3, 4, 5)

a = np.random.random((1,2,3,4,5))
b = np.ones((■(3,1,5))
print((a+b).shape) # = (1, 2, 3, 4, 5)
```

```
# ==============================
# BOOLEAN MASKING
# ==============================
# math | CS | Bio
grades = np.array([
    [90, 80, 75],
    [93, 95, 87],
    [67, 98, 88],
    [77, 89, 80],
    [93, 97, 95],
])
names = np.array([
    'Jack',
    'Jill',
    'Joe',
    'Jason',
    'Jennifer',
])
# All 90+ CS students
names[grades[:,1]>90]
```

0.0s

```
array(['Jill', 'Joe', 'Jenn
```

```
# ==============================
# VISUALIZATION FUNCTIONS
# ==============================
# Creates a new figure for plotting
plt.figure(figsize: tuple[float, float] = None) -> plt.Figure
# Adds a legend to the plot
plt.legend(loc: str = 'best') -> None
# Plots data on a 2D plane
plt.plot(*args, **kwargs) -> list[plt.Line2D]
# Displays the figure (renders the plot)
plt.show() -> None
# Sets the title of the plot
plt.title(label: str) -> None
# Sets the x-axis label
plt.xlabel(xlabel: str) -> None
# Sets the y-axis label
plt.ylabel(ylabel: str) -> None
# Sets the scaling for the y-axis
plt.yscale(value: str) -> None
```

```
# ==============================
# MATHEMATICAL OPERATIONS
# ==============================
# Returns the absolute values of the elements in an array
np.abs(x: np.ndarray) -> np.ndarray
# Adds two arrays element-wise
np.add(x1: np.ndarray, x2: np.ndarray) -> np.ndarray
# Divides elements of one array by another, element-wise
np.divide(x1: np.ndarray, x2: np.ndarray) -> np.ndarray
# Performs floor division, element-wise
np.floor_divide(x1: np.ndarray, x2: np.ndarray) -> np.ndarray
# Multiplies two arrays element-wise
np.multiply(x1: np.ndarray, x2: np.ndarray) -> np.ndarray
# Raises each element of an array to a specified power
np.power(x1: np.ndarray, x2: np.ndarray) -> np.ndarray
# Returns the sign of each element in the array (-1 for negative, 1 for positive, 0 for zero)
np.sign(x: np.ndarray) -> np.ndarray
# Subtracts one array from another element-wise
np.subtract(x1: np.ndarray, x2: np.ndarray) -> np.ndarray
# Computes the sine of each element in the array (input in radians)
np.sin(x: np.ndarray) -> np.ndarray
```

```
# ==============================
# AGGREGATION FUNCTIONS
# ==============================
# Checks if all elements along a given axis evaluate to True
np.all(a: np.ndarray, axis: int = None) -> np.ndarray
# Checks if any element along a given axis evaluates to True
np.any(a: np.ndarray, axis: int = None) -> np.ndarray
# Returns the indices of the maximum values along a specified axis
np.argmax(a: np.ndarray, axis: int = None) -> np.ndarray
# Returns the indices of the minimum values along a specified axis
np.argmin(a: np.ndarray, axis: int = None) -> np.ndarray
# Returns the maximum value of an array along a specified axis
np.max(a: np.ndarray, axis: int = None) -> np.ndarray
# Element-wise maximum of two arrays
np.maximum(x1: np.ndarray, x2: np.ndarray) -> np.ndarray
# Computes the median of the elements along a specified axis
np.median(a: np.ndarray, axis: int = None) -> np.ndarray
# Returns the minimum value of an array along a specified axis
np.min(a: np.ndarray, axis: int = None) -> np.ndarray
# Computes the nth percentile of the elements along a specified axis
np.percentile(a: np.ndarray, q: float, axis: int = None) -> np.ndarray
# Returns the product of array elements along a specified axis
np.prod(a: np.ndarray, axis: int = None) -> np.ndarray
# Returns the sum of array elements along a specified axis
np.sum(a: np.ndarray, axis: int = None) -> np.ndarray
# Computes the variance of the elements along a specified axis
np.var(a: np.ndarray, axis: int = None) -> np.ndarray
# Computes the standard deviation of the elements along a specified axis
np.std(a: np.ndarray, axis: int = None) -> np.ndarray
# Computes the arithmetic mean of the elements along a specified axis
# Example Input: np.mean(np.array([[1, 2, 3], [4, 5, 6]]), axis=0)
# Example Output: array([2.5, 3.5, 4.5])
# mean calculated along dimenion 0 (rows) -> gets mean for each column
np.mean(a: np.ndarray, axis: int = None) -> np.ndarray
```

```
# ==============================
# NUMPY ARRAY CREATION FUNCTIONS
# ==============================
# Creates an array with a range of values from start to stop with a given step size
np.arange(start: int, stop: int, step: int) -> np.ndarray
# Creates an array from the given object (list, tuple, etc.), with an optional data type
np.array(object: Any, dtype: np.dtype = None) -> np.ndarray
# Creates an array filled with a specified value, given the shape
np.full(shape: tuple[int, ...], fill_value: Any, dtype: np.dtype = None) -> np.ndarray
# Creates an array of evenly spaced values between start and stop with a specified number of elements
np.linspace(start: float, stop: float, num: int = 50) -> np.ndarray
# Creates an array filled with ones, given the shape
np.ones(shape: tuple[int, ...], dtype: np.dtype = None) -> np.ndarray
# Creates an array of ones with the same shape and type as a given array
np.ones_like(a: np.ndarray, dtype: np.dtype = None) -> np.ndarray
# Generates random numbers from a uniform distribution between 0 and 1
np.random.random(size: tuple[int, ...] = None) -> np.ndarray
# Generates random numbers from a standard normal distribution
np.random.randn(*shape: int) -> np.ndarray
# Creates an array filled with zeros, given the shape
np.zeros(shape: tuple[int, ...], dtype: np.dtype = None) -> np.ndarray
# Creates an array of zeros with the same shape and type as a given array
np.zeros_like(a: np.ndarray, dtype: np.dtype = None) -> np.ndarray
```