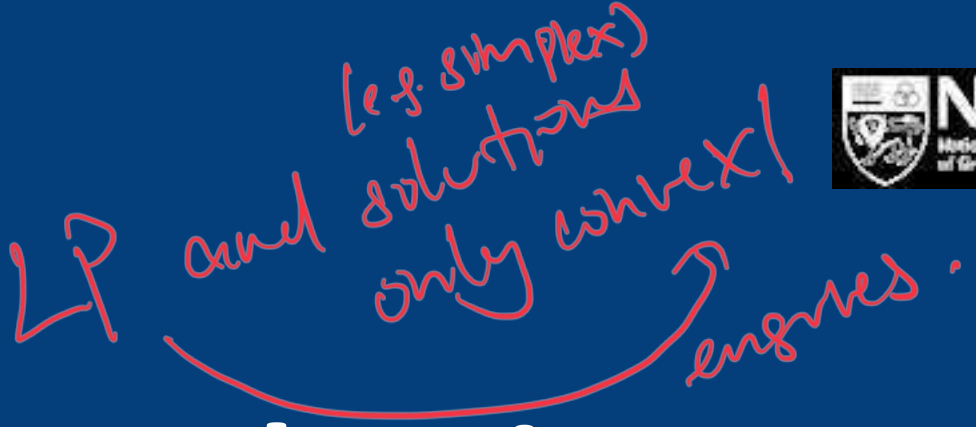


EE2213 Introduction to Artificial Intelligence

Lecture 5: Optimization I

Dr. Shaojing Fan
fanshaojing@nus.edu.sg

OVERVIEW OF COURSE CONTENTS

- **Introduction (Shaojing)**
  - What is AI
  - Applications of AI
  - AI agent
- **Search (Shaojing)**
  - Uninformed search algorithms: breadth-first, depth-first, uniform-cost(Dijkstra's algorithm)
  - Informed search algorithms: greedy best-first, A*
  - Applications
- **Optimization (Shaojing)**
  - Linear programming
  - Convex problems
  - Applications
- **Machine learning (Wang Si)**
  - Supervised and unsupervised learning: regression, classification, clustering
  - Neural networks and deep learning
  - Applications
- **Knowledge representation (Wang Si)**
  - Knowledge Representation and Reasoning
  - Propositional Logic
  - Applications
- **Ethical considerations (Shaojing)**
  - Bias in AI
  - Privacy concerns
  - Societal impact

Path optimization

Mathematical optimization

Agenda

- Background of linear programs
- Recap of linear algebra
- Solving linear programs (LP)
  -- The graphical method
  -- Different cases of LP outcomes
  -- The Simplex algorithm

Background

- **Optimization**

Choosing the best option from a set of options

- **Constrained optimization**

An optimization problem is constrained if solutions must satisfy some hard constraints on the values of the variables.

$$\underset{x}{\text{minimize}} \quad f(x)$$

$$subject\ to\ x \in X$$

set of possible values based on some constraint on x.

The set X is called the feasible set.

➢ Example: minimize x^2, s.t. $x \in [-1, 1]$

such that

Background

- **Constraint types**

Generally, constraints are formulated using two types

1. Equality constraints: $h(x) = 0$
2. Inequality constraints: $g(x) \leq 0$

Any optimization problem can be written as

$$\underset{x}{\text{minimize}} \quad f(x)$$

$$\text{subject to} \quad h_i(x) = 0 \text{ for all } i \text{ in } \{1, \ldots, l\}$$
$$g_j(x) \leq 0 \text{ for all } j \text{ in } \{1, \ldots, m\}$$

function output given an input $x \in X$

to be $=0$ or ≤ 0.

Linear programming problem

A special category of constrained optimization problem, where the constraints must be linear inequalities.

"**Programming**" originally meant "**resource planning**" in military context.

Now, "programming" simply means solving a mathematical problem.

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P139 – P140
Reference: https://roam.libraries.psu.edu/system/files/e-books/MATH484-Linear_Programming.pdf

Linear program example:

You want to plan your meals for the week to **minimize total cost** while meeting your nutritional needs for **calories** and **protein.**

Food	Cost ($)	Calories	Proteins (g)
Chicken	3	250	30
Rice	1	200	4
Vegetables	2	50	2

Decision variables:

- x_1: number of servings of chicken;
- x_2: number of servings of rice
- x_3: number of servings of vegetables

Objective:

$$\text{Minimize cost} = 3x_1 + 1x_2 + 2x_3$$

Constraints:

Calories ≥ 2000: $250x_1 + 200x_2 + 50x_3 \geq 2000$

Protein $\geq$50 grams: $30x_1 + 4x_2 + 2x_3 \geq 50$

Servings must be non-negative: $x_1,\ x_2,\ x_3 \geq 0$

min cost while adhering to constraints on x_1, x_2, x_3

Motivation

Why linear programming is important?

- A lot of problems can be formulated as linear programs, and
- There exist efficient methods to solve them
- or at least give good approximations.

Alan Cheng said linear problems are the easiest form, and all other problems can be reduced to or derived from a LP.

The transportation problem

There are ***I* factories** (index by i), each producing iPhone with a production capacity s_i (units)

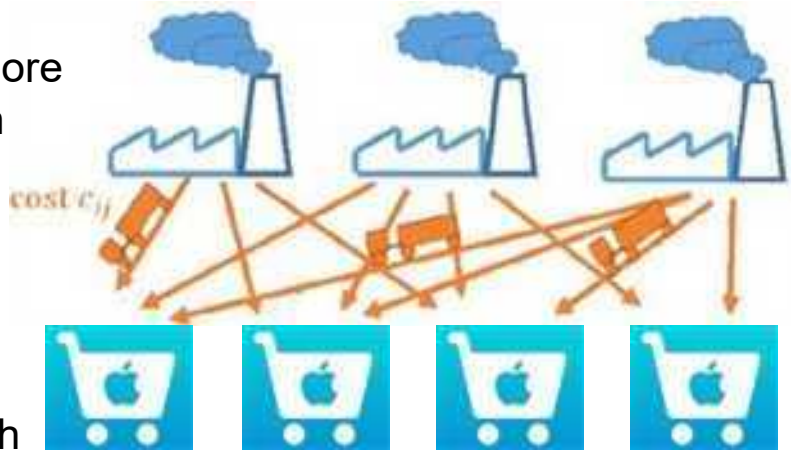
There are ***J markets*** (indexed by j, e,g, the Singapore market, the US market, etc) demand iPhones, each with a demand d_j (units)

The cost of transporting one unit of iPhone from factory i to market j is c_{ij}.

We want to decide how much to transport from each factory to each market in order to:

minimize the total transportation cost,

I factories
s_i supply from factory i



J markets
d_j demand from market j

subject to:

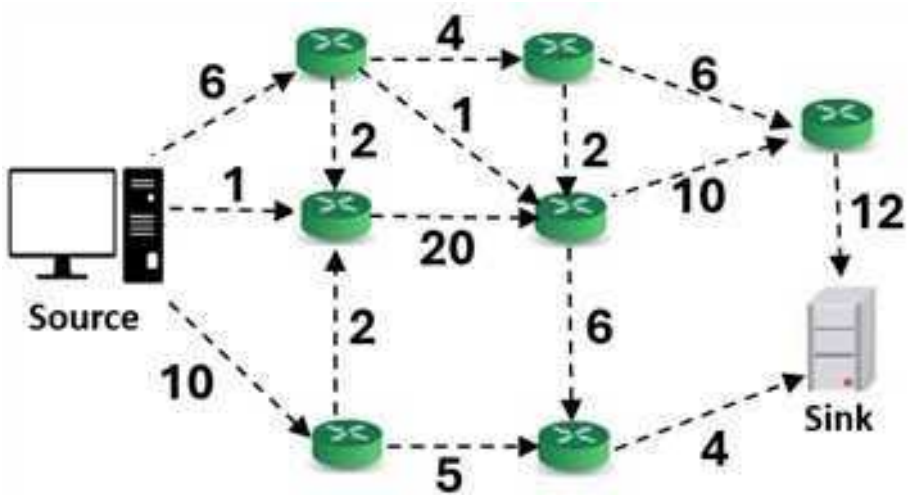
- *Supply constraints*: The total amount shipped from each factory must not exceed its supply.
- *Demand constraints*: The total amount received by each market must meet its demand.
- *Non-negativity*: Quantities shipped must be greater than or equal to zero.

The maximum flow problem

A network of nodes and links, each with a capacity (e.g., bandwidth limit).

A source sends data, a sink receives it.

Intermediate nodes (e.g., routers) must forward data without storing it.



Goal:

Maximize total data flow from source to sink

subject to:

- *Capacity limits on each link*
- *Flow conservation at all intermediate nodes (flow in = flow out)*
- *Non-negative flows*

Resource allocation problem 1

Maximize academic benefit by selecting the best course combination within limits.

- Choose courses within a maximum credit limit (20 credits)
- Stay within a weekly workload limit (40 hours)
- Avoid time slot conflicts
- Select only whole courses (integer decision)

Decision variables:

$x_i = 1$ if course i is selected, 0 otherwise

Objective Function:

Maximize $Z = b_1x_1 + b_2x_2 + \cdots + b_nx_n$ (b_i : benefit score of course i)

not only minimise

subject to:

- *Total credits*: $c_1x_1 + c_2x_2 + \cdots + c_nx_n \leq 20$
- *Total workload*: $w_1x_1 + w_2x_2 + \cdots + w_nx_n \leq 40$
- *No time conflicts*: $x_i + x_j \leq 1$ for overlapping courses
- $x_i \in \{0, 1\}$

Course	Credits	Workload per week	Time Slot	Benefit score
EE2213	4	10 hrs	Mon 10–12	9
MA2101	4	9 hrs	Mon 10–12	7
CDE2212	4	11 hrs	Tue 2–5	8
CE3201	2	5 hrs	Wed 3–5	5
EE3801	3	7 hrs	Thu 1–3	6

Problem formulation

x_1, x_2: Decision variables

Maximize $150x_1 + 200x_2$ ← Objective function

subject to:

$$x_1 + x_2 \leq 200$$
$$9x_1 + 3x_2 \leq 1240$$
$$12x_1 + 16x_2 \leq 2660$$
$$x_1, x_2 \geq 0$$

← Constraints

In linear program: objective function + constraints are **all linear**

Typically (not always): variables are non-negative

Linear programming uses only non-strict inequalities (≤, ≥) and equalities (=), not strict inequalities (<, >)

If variables are integer: system called Integer Linear Program (ILP)*.

→ prob since properties of int are used!

* ILP involves more advanced mathematical complexity, so we won't cover it in lectures, but you'll gain hands-on experience with small-scale integer programs during tutorials.

Agenda

- Background of linear programs
- Recap of linear algebra
- Solving linear programs (LP)
  -- The graphical method
  -- Different cases of LP outcomes
  -- The Simplex algorithm

Some linear algebra review

Concept	Definition	Notation	Dimension	Key characteristics
Scalar	A single numerical value	A single number (e.g., 5, -2.3, π)	1 × 1	- A fundamental, indivisible quantity. - Has magnitude only.
Variable	A symbol representing a quantity that can change or take on different numerical values.	A letter (e.g., x, y, z, w_1)	1 × 1	- Represents a point in space, a direction, or a collection of features. - Has both magnitude and direction. - A 1D array of scalars.
Vector	An ordered list (array) of numbers (scalars).	Bold lowercase letters (e.g., $\mathbf{x}$, $\mathbf{y}$), or with an arrow ($\vec{\mathbf{x}}$); written as a row or column (e.g., $\mathbf{x}$ = [2, -1, 4])	n × 1 or 1 × n	- Represents a point in space, a direction, or a collection of features. - Has both magnitude and direction. - A 1D array of scalars.
Matrix	A rectangular array of numbers (scalars) arranged in rows and columns.	Bold uppercase letters (e.g., $\mathbf{A}$, $\mathbf{M}$), e.g., $\mathbf{X}$ = [[1, 2], [3, 4]]	m × n	- Organizes multi-dimensional data. - Can represent linear transformations or systems of equations. - A 2D array of scalars.

Capital Sigma: $\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_{n-1} + x_n$ sum

Capital Pi: $\prod_{i=1}^{n} x_i = x_1 \cdot x_2 \cdot \ldots \cdot x_{n-1} \cdot x_n$ product.

Reference: http://www.cs.cmu.edu/~zkolter/course/linalg

Some linear algebra review

Vector or matrix **transpose**:

$$\begin{bmatrix} \cdots & x_{m1} \\ \ddots & \vdots \\ \cdots & x_{mn} \end{bmatrix}$$

$$\begin{bmatrix} x_{2,1} & x_{3,1} \\ x_{2,2} & x_{3,2} \\ x_{2,3} & x_{3,3} \end{bmatrix}$$

Some linear algebra review

Addition and subtraction of matrices and vectors is defined just as addition or subtraction of the elements, for $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$,

$$\mathbf{C} \in \mathbb{R}^{m \times n} = \mathbf{A} + \mathbf{B} \Leftrightarrow c_{ij} = a_{ij} + b_{ij}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad \mathbf{x} + \mathbf{y} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

- Example:

matrix addition / subtraction

$$\begin{bmatrix} 2 & 8 \\ 3 & 7 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 0 & 5 \end{bmatrix} = \begin{bmatrix} 2+1 & 8+2 \\ 3+0 & 7+5 \end{bmatrix} = \begin{bmatrix} 3 & 10 \\ 3 & 12 \end{bmatrix}$$

Some linear algebra review

Matrices multiplication: for two matrices $\mathbf{A} \in \mathbb{R}^{m \times d}, \mathbf{B} \in \mathbb{R}^{d \times h}$,

Dimension must match

$$\mathbf{C} \in \mathbb{R}^{m \times h} = \mathbf{AB} \Leftrightarrow c_{ij} = \sum_{k=1}^{d} a_{ik} b_{kj}$$

- Example:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, B = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}$$

matrix multiplication

$$A \times B = \begin{bmatrix} aj+bm+cp & ak+bn+cq & al+bo+cr \\ dj+em+fp & dk+en+fq & dl+eo+fr \\ gj+hm+ip & gk+hn+iq & gl+ho+ir \end{bmatrix}$$

row x col.

$$\begin{bmatrix} 2 & 4 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 & 1 & 5 \\ -2 & 1 & 3 \end{bmatrix} = \begin{bmatrix} -2 & 6 & 22 \\ -3 & 4 & 14 \end{bmatrix}$$

must same (col. col = no. rows)

(2×2) (2×3) (2×3)

resultant dimension

Source: https://notesbylex.com/matrix-multiplication

Some linear algebra review

Special case of matrices multiplication: ***inner product*** of two vectors, for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$,

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y} = \sum_{i=1}^{n} x_i y_i$$

dot product

- Example:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2$$

Special case: product of matrix and vector, for $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$,

$$\mathbf{A}\mathbf{x} \in \mathbb{R}^m = \begin{bmatrix} a_1^T x \\ \vdots \\ a_m^T x \end{bmatrix} = \sum_{i=1}^{n} a_i x_i$$

just use normal matrix multiplication

- Example:

$$\begin{pmatrix} a & b & c \\ l & m & n \\ r & s & t \end{pmatrix} \begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \longrightarrow \begin{pmatrix} aA_x + bA_y + cA_z \\ lA_y + mA_y + nA_z \\ rA_z + sA_z + tA_z \end{pmatrix}$$

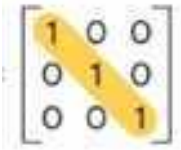
3x3 3x1

Source:
https://phys.libretexts.org/Courses/University_of_California_Davis/UCD%3A_Physics_9HB__Special_Relativity_and_Thermal_Statistical_Physics/3%3A_Spacetime/3.1%3A_Vector_Rotations

Some linear algebra review

The **identity matrix $\mathbf{I} \in \mathbb{R}^{n \times n}$** is a *square matrix* with:

- 1s on the diagonal
- 0s elsewhere



It acts like "**1**" in matrix multiplication, *i.e.*, for any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$:

$\mathbf{IA} = \mathbf{A}$ (with $\mathbf{I} \in \mathbb{R}^{m \times m}$)

$\mathbf{AI} = \mathbf{A}$ (with $\mathbf{I} \in \mathbb{R}^{n \times n}$)

it is what allows "divide" => inverse.

For a *square* matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, its **inverse matrix $\mathbf{A}^{-1}$** is the matrix such that,

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$$

Not all matrices have an inverse (they must be **square** and **non-singular**).

also it essentially means that it can be expressed as a linear combination

Linear function

- If a function f is linear, superposition extends to linear combinations of any number of vectors:
  $f(\alpha_1 \mathbf{x}_1 + \cdots + \alpha_k \mathbf{x}_k) = \alpha_1 f(\mathbf{x}_1) + \cdots + \alpha_k f(\mathbf{x}_k)$
  for any d vectors $\mathbf{x}_1 + \cdots + \mathbf{x}_k$, and any scalars $\alpha_1 + \cdots + \alpha_k$.

A function $f: \mathcal{R}^n \rightarrow \mathcal{R}$ is **linear** if it can be written as a weighted sum of variables:

$$f(\mathbf{x}) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = \mathbf{a}^T \mathbf{x}$$

- Example: $f(x_1, x_2) = 2x_1 - 7x_2 = \begin{bmatrix} 2 \\ -7 \end{bmatrix} [x_1, x_2]$

linear functions MA1512

This means it contains no products, no powers, and no nonlinear functions of variables.

Mathematically, this form arises because linear functions satisfy:

- Homogeneity (Scaling): $f(\boldsymbol{\lambda}\mathbf{x}) = \boldsymbol{\lambda} f(\mathbf{x})$
- Additivity (Adding): $f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$

Superposition and linearity

- The inner product function $f(\mathbf{x}) = \boldsymbol{a}^T \mathbf{x}$ defined in equation (1) (slide 9) satisfies the property

$$\begin{aligned} f(\alpha\mathbf{x} + \beta\mathbf{y}) &= \boldsymbol{a}^T(\alpha\mathbf{x} + \beta\mathbf{y}) \\ &= \boldsymbol{a}^T(\alpha\mathbf{x}) + \boldsymbol{a}^T(\beta\mathbf{y}) \\ &= \alpha(\boldsymbol{a}^T\mathbf{x}) + \beta(\boldsymbol{a}^T\mathbf{y}) \\ &= \alpha f(\mathbf{x}) + \beta f(\mathbf{y}) \end{aligned}$$

  for all d-vectors $\mathbf{x}$, $\mathbf{y}$, and all scalars α, β.

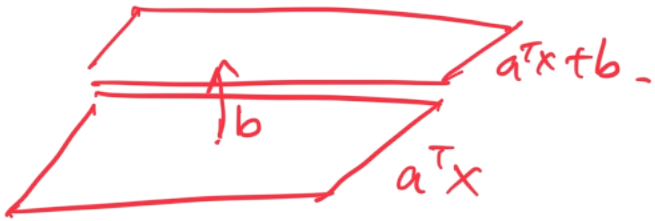
- This property is called superposition, which consists of homogeneity and additivity
- A function that satisfies the superposition property is called linear

Linear vs affine function

A function $f: \mathcal{R}^n \rightarrow \mathcal{R}$ is **affine** if and only if it can be written as:

$$f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} + b$$

where $\mathbf{a}, \ \mathbf{x} \in \mathcal{R}^n$, $b \in \mathcal{R}$



Here, b is a scalar, often referred to as the **offset** or **bias**.

- Example: $f(x_1, x_2) = 2x_1 - 7x_2 + 6 = \begin{bmatrix} 2 \\ -7 \end{bmatrix}[x_1, x_2] + 6$

An affine function is a linear function with an added constant term.

Linear vs affine function

An **affine function** is **not necessarily a linear function**, but every linear function is a special case of an affine function.

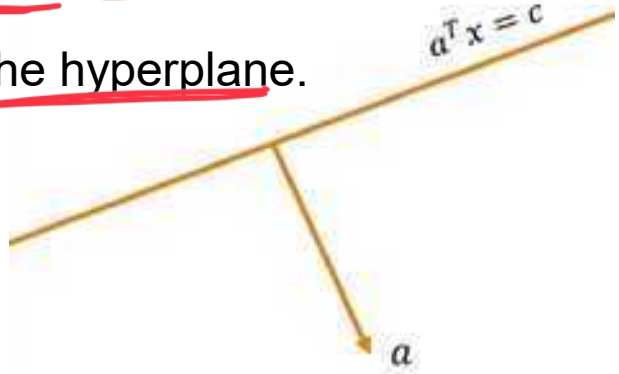
An **affine function** $f(x) = \mathbf{a}^T\mathbf{x} + b$ is a **linear function** if and only if the constant term $b = 0$.

or else it cannot be written as just a^Tx

Linear function: Geometric interpretation

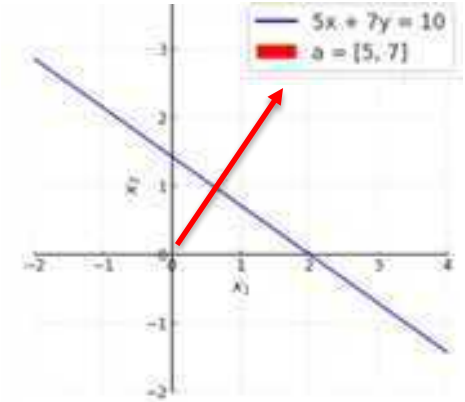
Geometric interpretation of $\mathbf{a}^T\mathbf{x} = c$

- The equation $\mathbf{a}^T\mathbf{x} = c$ represents a hyperplane in $\mathcal{R}^n$ (a line in $\mathcal{R}^2$, a plane in $\mathcal{R}^3$, etc.)
- The vector $\mathbf{a}$ is a normal (perpendicular) to the hyperplane.



- In 2D: $\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$ is a perpendicular to the line $a_1x_1 + a_2x_2 = c$

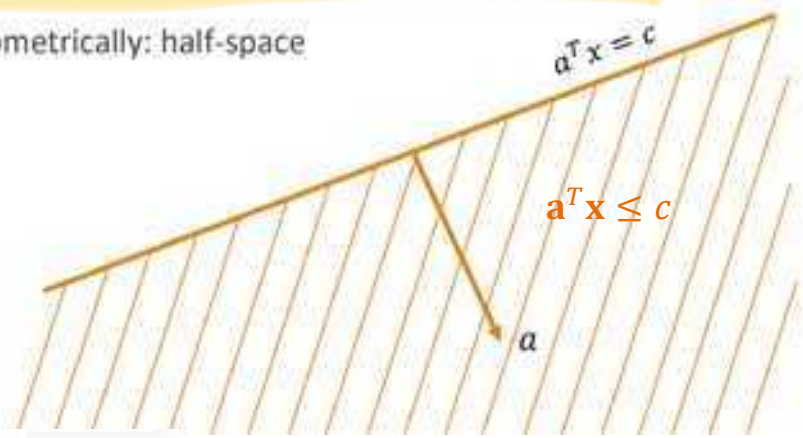
➢ Example: $5x_1 + 7x_2 = 10$



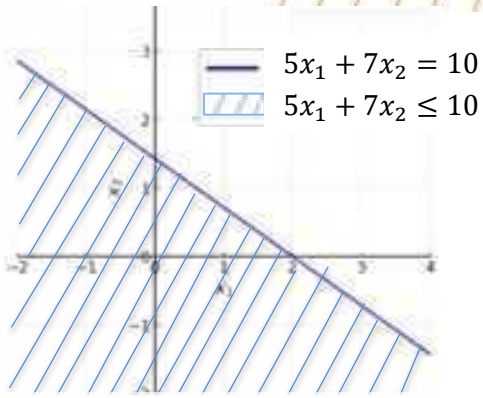
Linear inequality

The inequality $\mathbf{a}^T\mathbf{x} \leq c$ defines a half-space in geometry.

It includes all points x that lie on or below the hyperplane $\mathbf{a}^T\mathbf{x} = c$



- Example: $5x_1 + 7x_2 \leq 10$



Visualize in higher-dimensional space

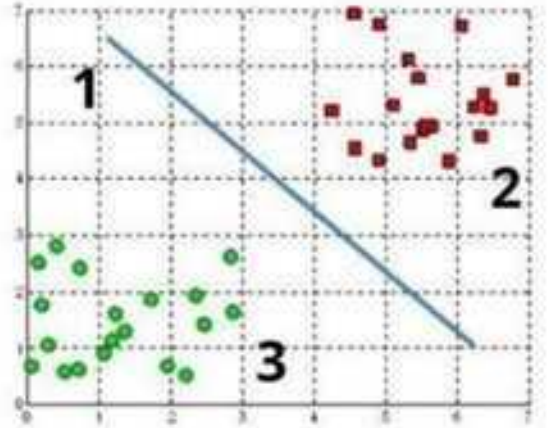
Hyperplane: The region defined by a linear equality $a_i x = b_i$

Half-space: The region defined by a linear inequality $a_i x \leq b_i$

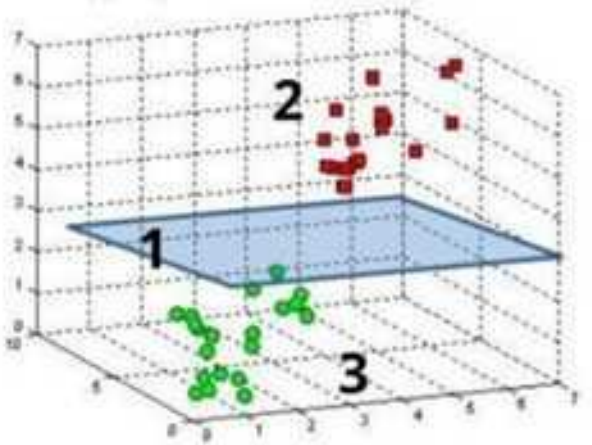
1 $\beta_o + \beta_1 X_1 + \beta_2 X_2 = 0$ → hyperplane ← 1 $\beta_o + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 = 0$

2 $\beta_o + \beta_1 X_1 + \beta_2 X_2 > 0$ → half-space ← 2 $\beta_o + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 > 0$

3 $\beta_o + \beta_1 X_1 + \beta_2 X_2 < 0$ → half-space ← 3 $\beta_o + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 \leq 0$



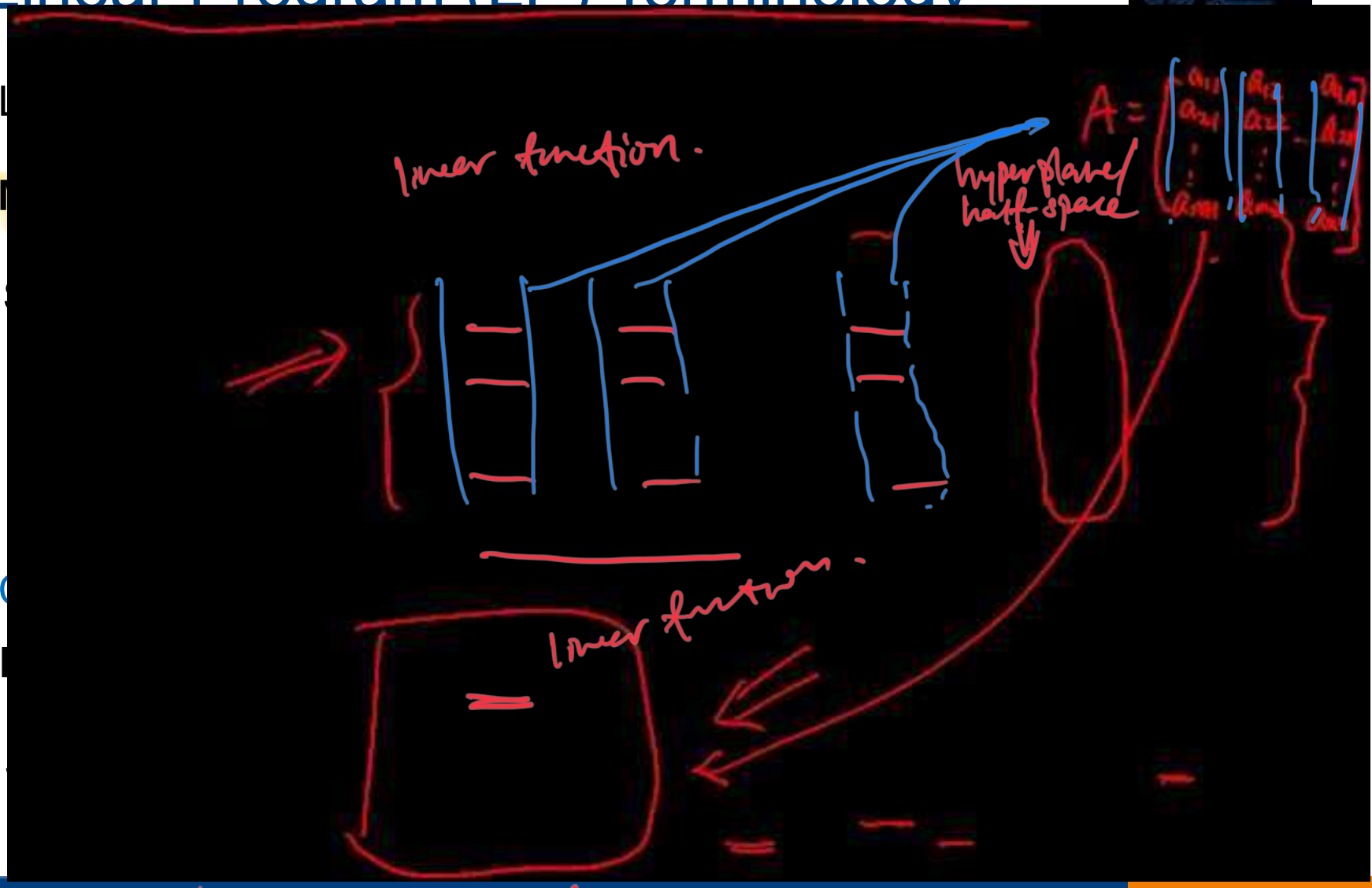
R2 Space, Hyperplane is a line



R3 Space, Hyperplane is a plane

Credit: https://www.enjoyalgorithms.com/blog/support-vector-machine-in-ml

Linear Program (LP) Terminology



Standard form

Linear Program (LP) Terminology

In matrix form, a linear program in standard form can be written as:

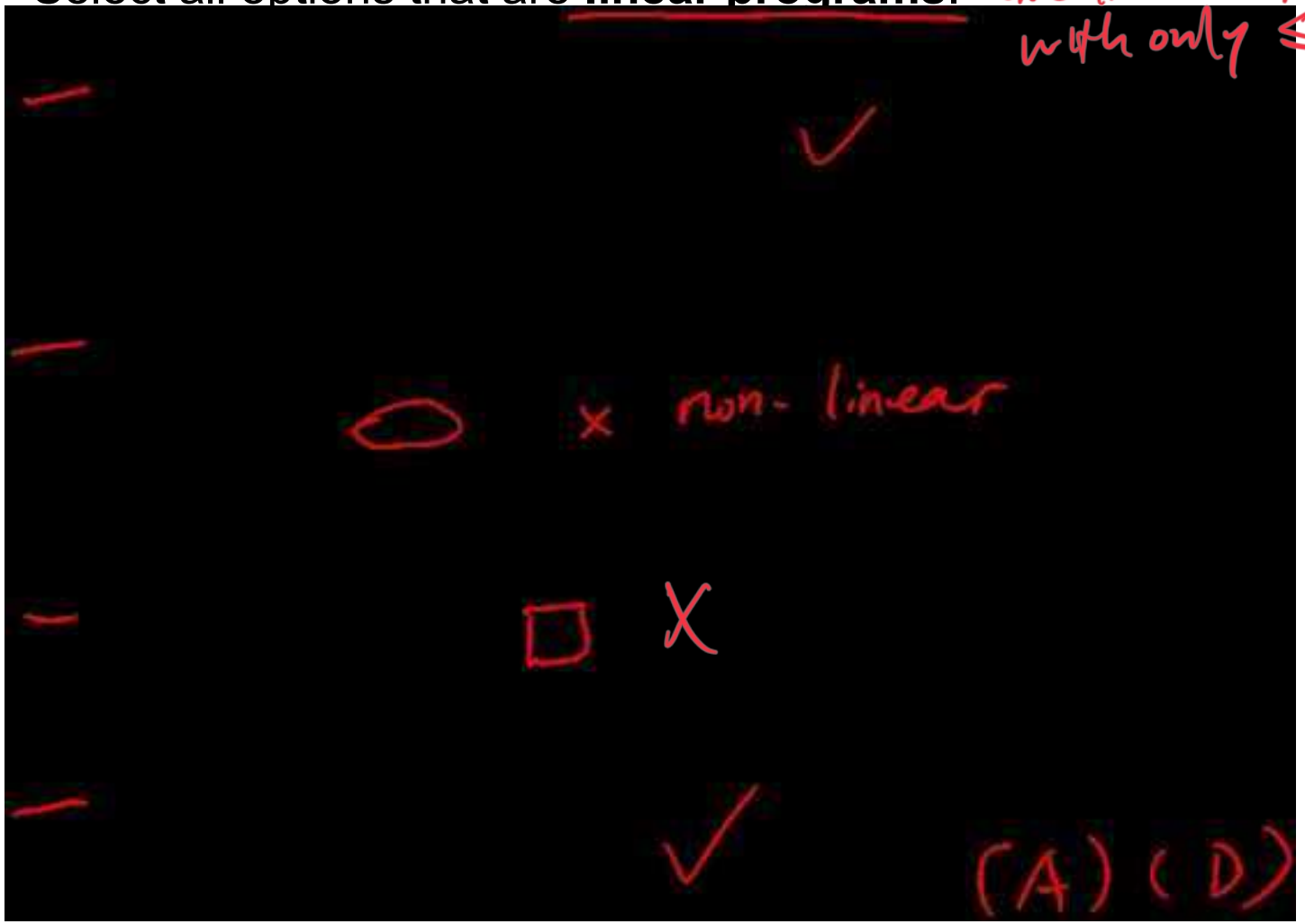
where

whose $\boldsymbol{i,j}$ element is a_{ij}

$c^T : [c_1\ c_2\ c_3\ \ldots]\ x$

Quick question 1

Select all options that are **linear programs**.



$a, b \geq 0$

Agenda

- Background of linear programs
- Recap of linear algebra
- Solving linear programs (LP)
  -- The graphical method
  -- Different cases of LP outcomes
  -- The Simplex algorithm

Graphical method

For LPs with only two decision variables, we often solve them using the **graphical method**.

Consider the bakery production example:

Maximize $x_1 - 2x_2$

Subject to $x_1 \geq 0$

$x_2 \geq 0$

$3x_1 + 2x_2 \leq 6$

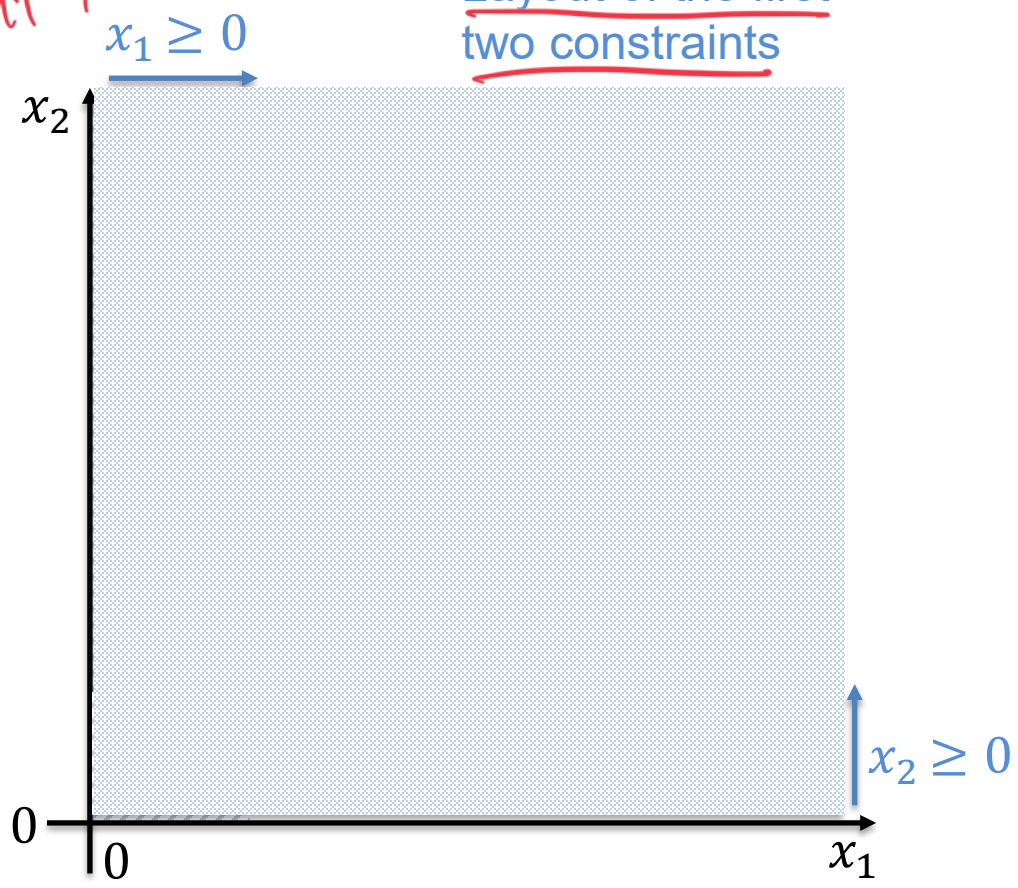
$2x_1 + 3x_2 \leq 6$

Graphical method

Maximize $x_1 - 2x_2$

Subject to

$x_1 \geq 0$
$x_2 \geq 0$
$3x_1 + 2x_2 \leq 6$
$2x_1 + 3x_2 \leq 6$



Layout of the first two constraints

Graphical method

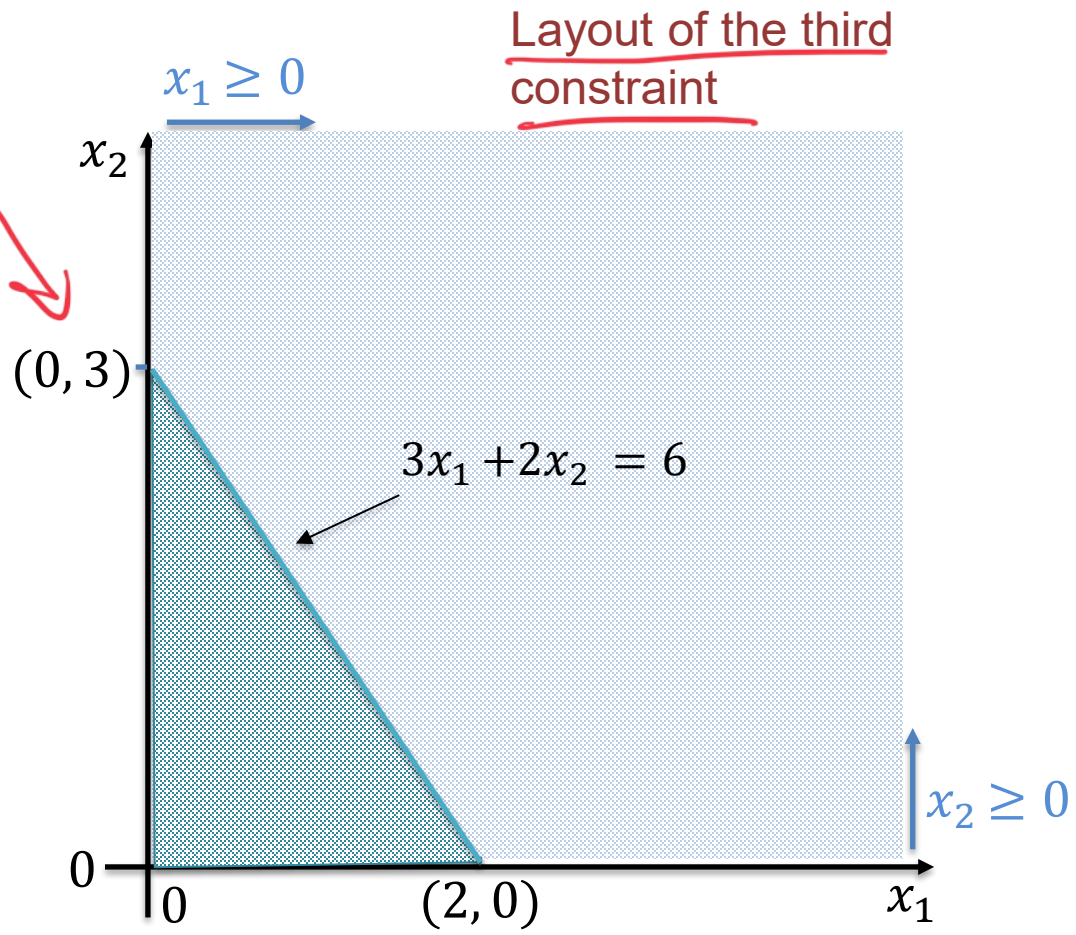
Maximize $x_1 - 2x_2$

Subject to $x_1 \geq 0$

$x_2 \geq 0$

$3x_1 + 2x_2 \leq 6$

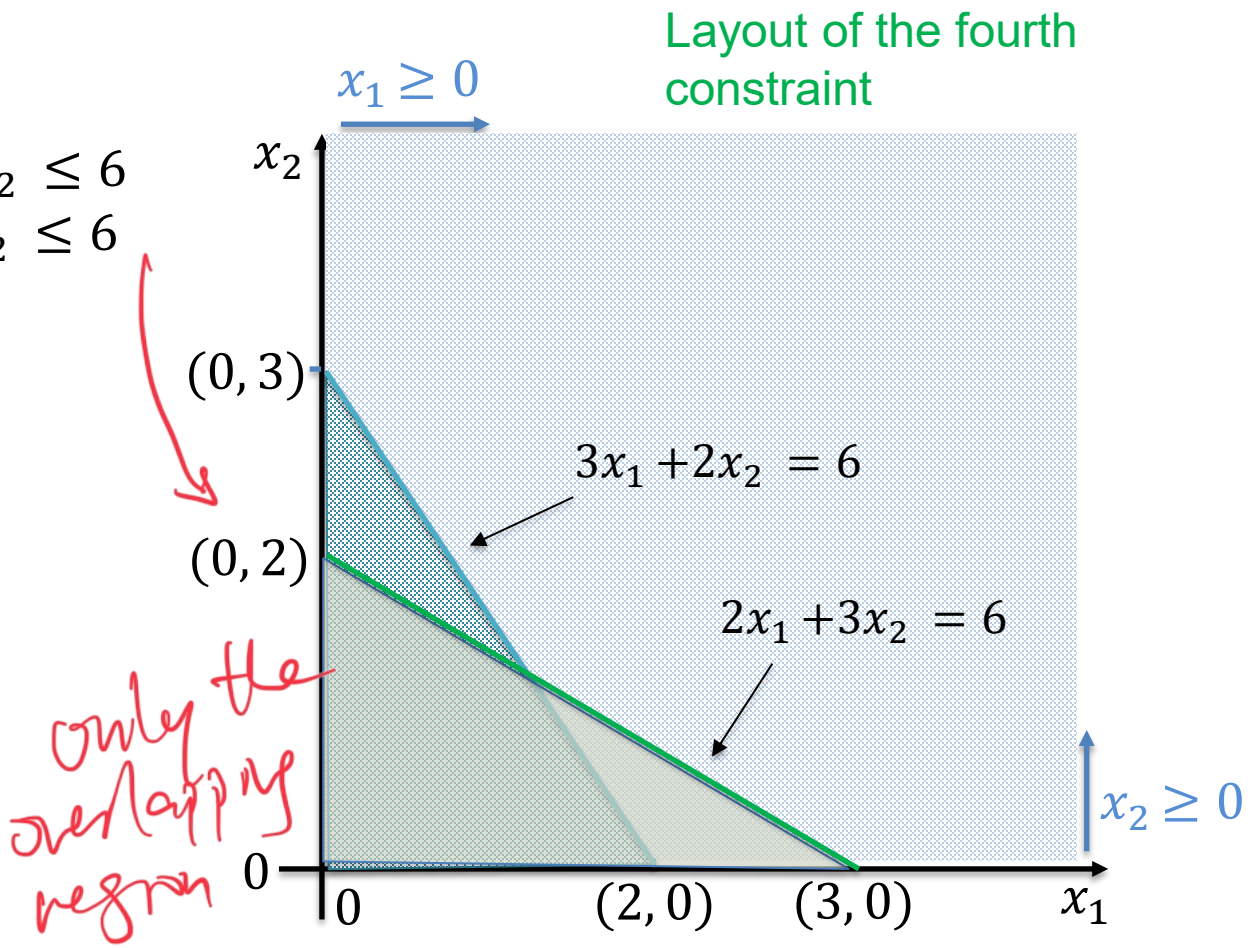
$2x_1 + 3x_2 \leq 6$



Graphical method

Maximize $x_1 - 2x_2$

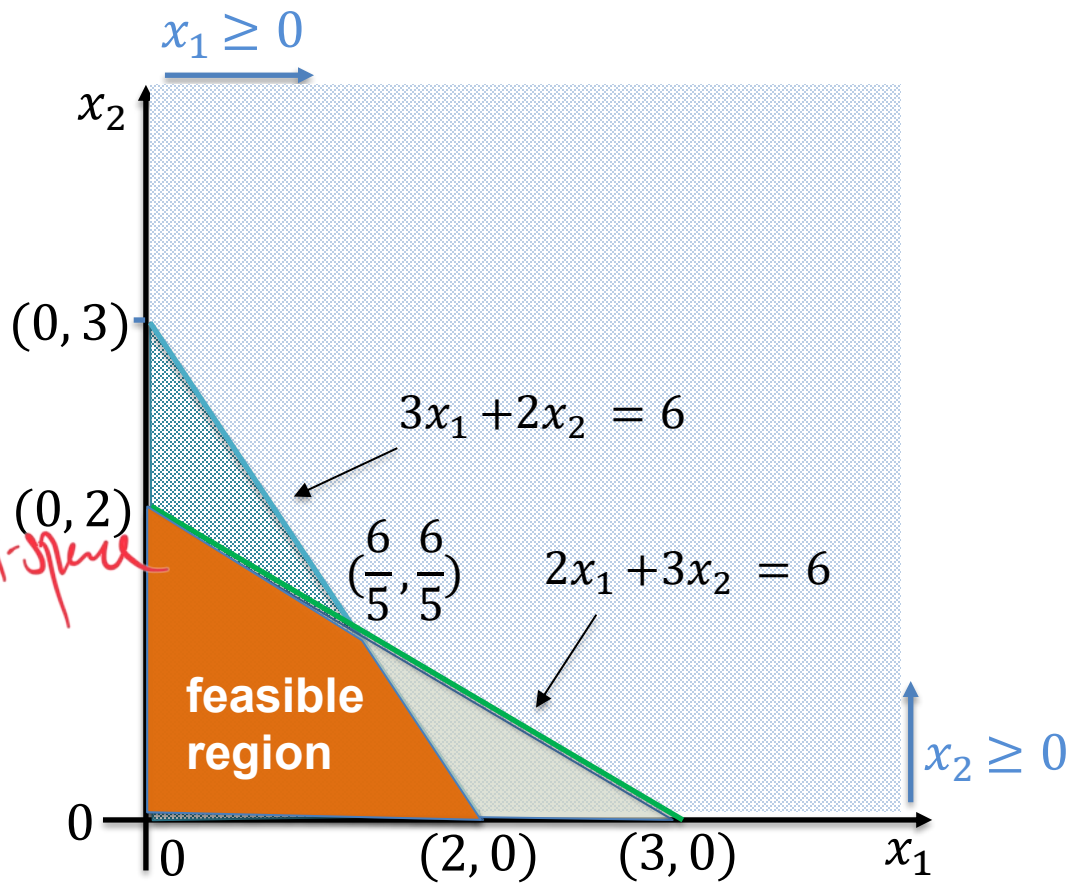
Subject to $x_1 \geq 0$
$x_2 \geq 0$
$3x_1 + 2x_2 \leq 6$
$2x_1 + 3x_2 \leq 6$



Graphical method

Maximize $x_1 - 2x_2$

Subject to
$x_1 \geq 0$
$x_2 \geq 0$
$3x_1 + 2x_2 \leq 6$
$2x_1 + 3x_2 \leq 6$



overlaps all half-spaces

Feasible region: the set of all points that satisfy all the constraints of a LP problem.

Graphical method

Maximize $x_1 - 2x_2$

Subject to $x_1 \geq 0$

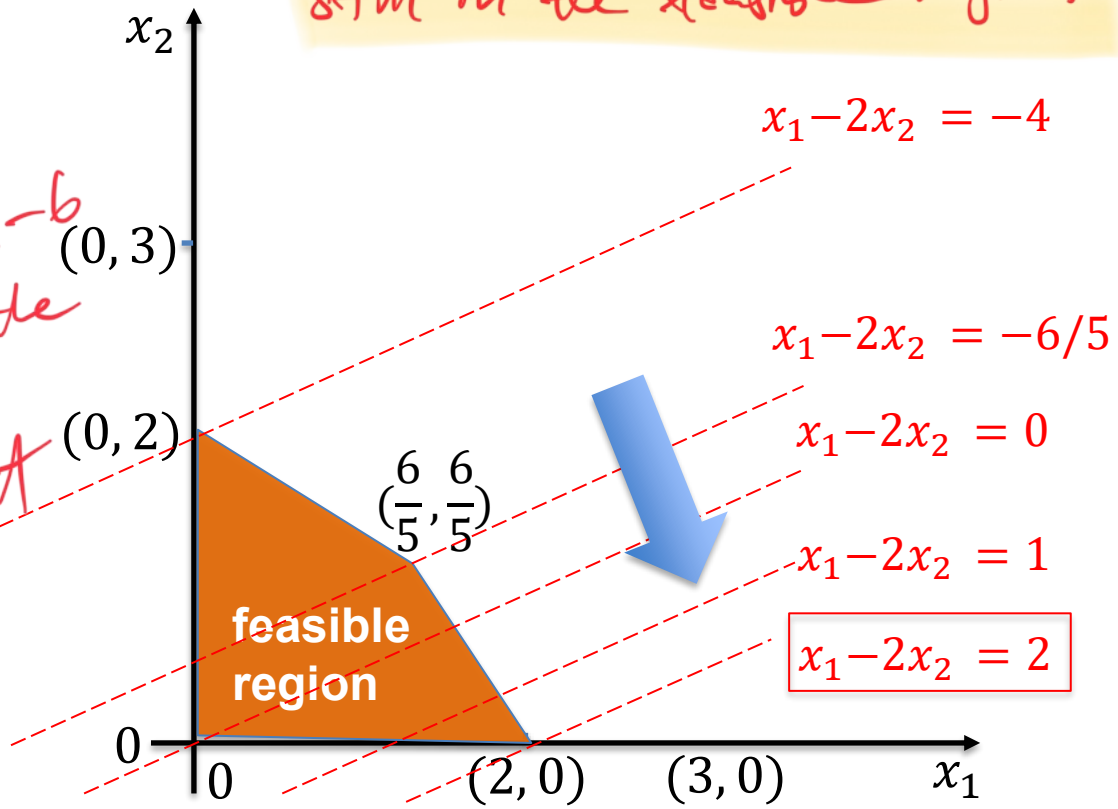
$x_2 \geq 0$

$$\begin{cases} 3x_1 + 2x_2 \leq 6 \\ 2x_1 + 3x_2 \leq 6 \end{cases}$$

Let $y = x_1 - 2x_2$

find highest value of y that is still on the feasible region

Let $y = 3x_1 + 2x_2 - 6$ and draw the line to get the constraint



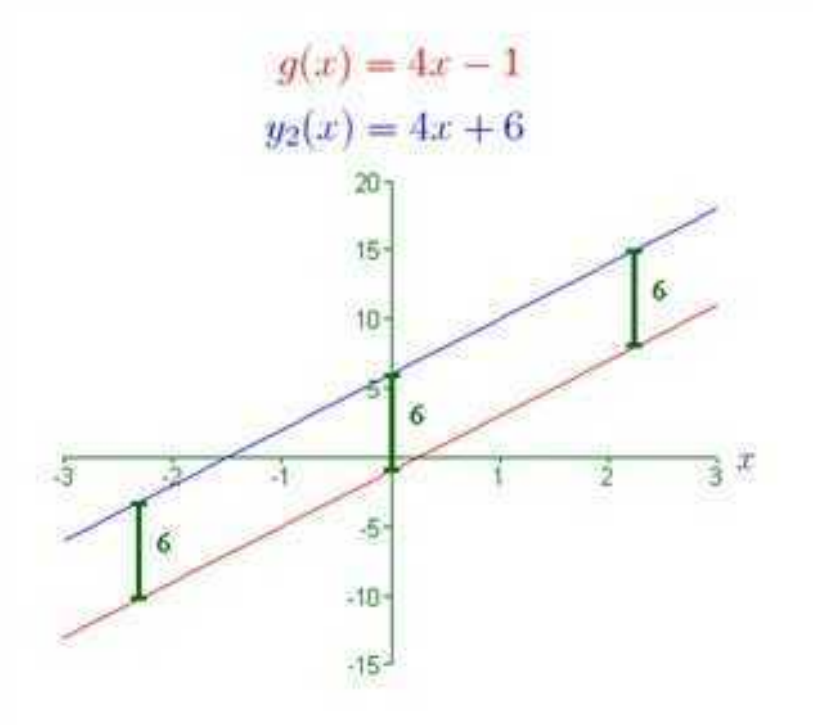
Objective = 2

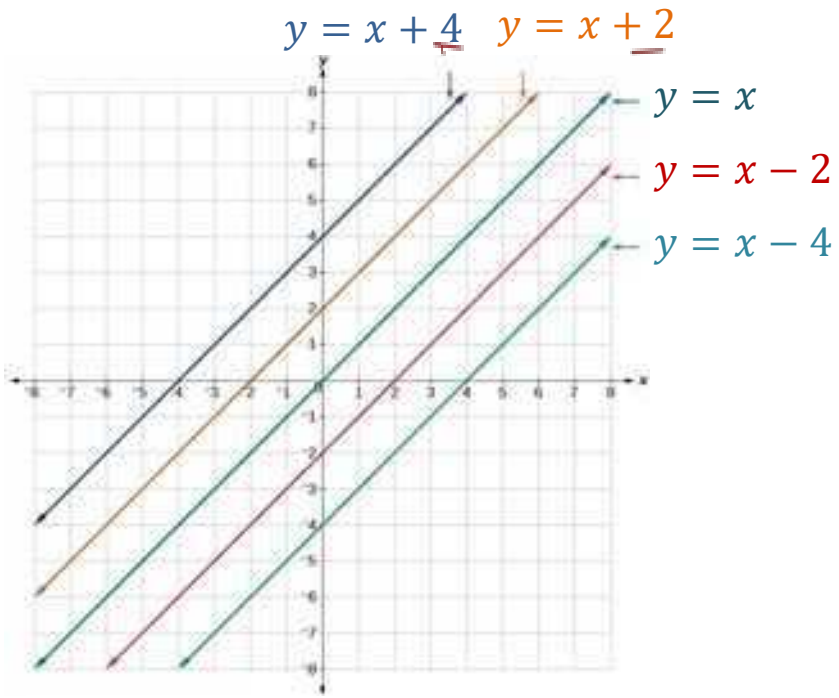
Recap: vertical and horizontal shift

graph: $y = f(x)$

$y = f(x) + k$ vertical shift by k units

$y = f(x + h)$ horizontal shift by h units

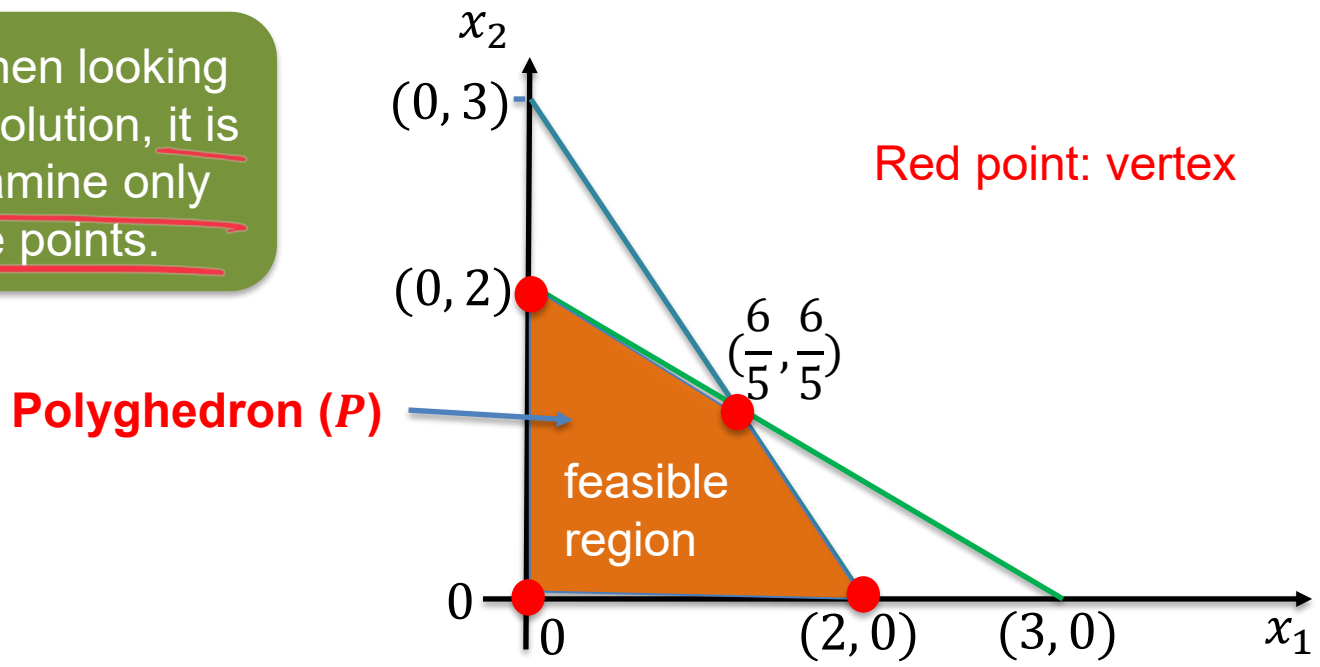




Fundamental theorem of LP

If a linear program has an optimal solution, then there exists at least one optimal solution at a **vertex (corner point)** of the feasible region (or along an edge between vertices).

Implication: When looking for an optimal solution, it is enough to examine only the extreme points.

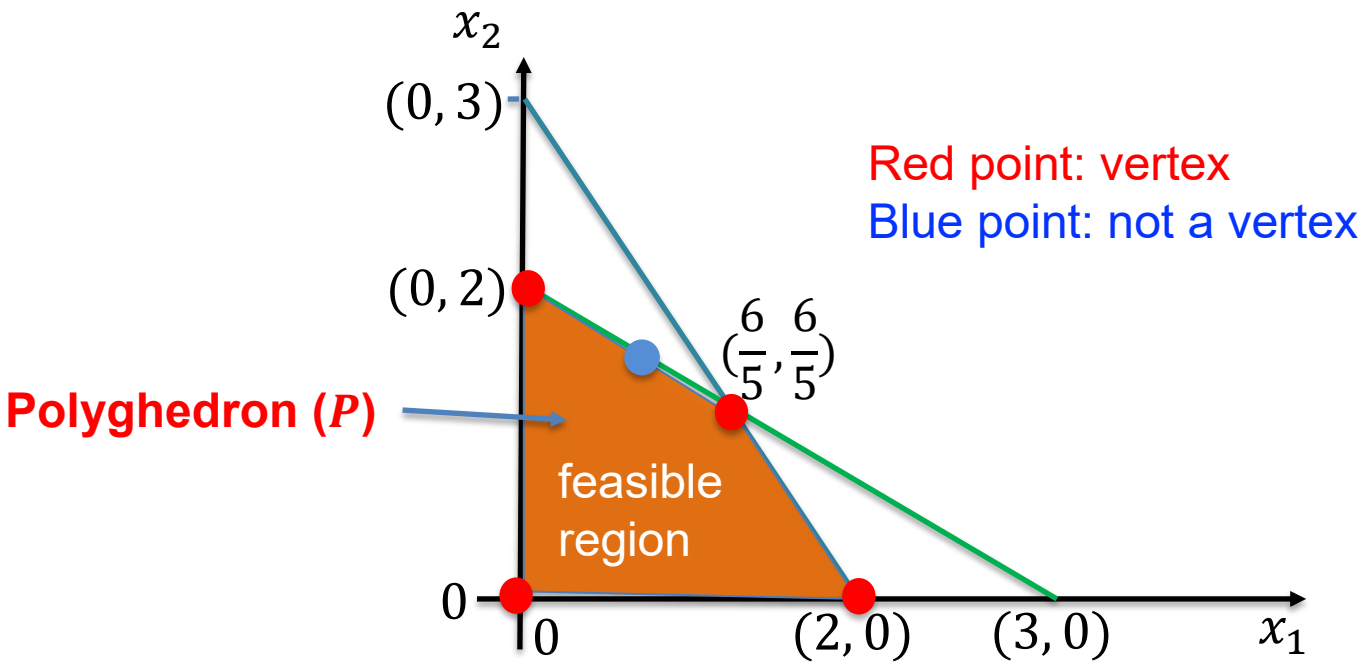


For proof, see
https://www.princeton.edu/~aaa/Public/Teaching/ORF363_COS323/F14/ORF363_COS323_F14_Lec11.pdf P13-P14

Terminology

A **vertex** (or **corner point**) is a point on the boundary of the feasible region where two or more constraints *intersect*; **the optimal solution of an LP (if it exists) occurs at one of these vertices**.

Polyhedron (P): the geometric shape formed by the intersection of all linear constraints; it represents the feasible region in LP



Summary of graphical method

- Works best for LPs with **two variables**.

Steps:

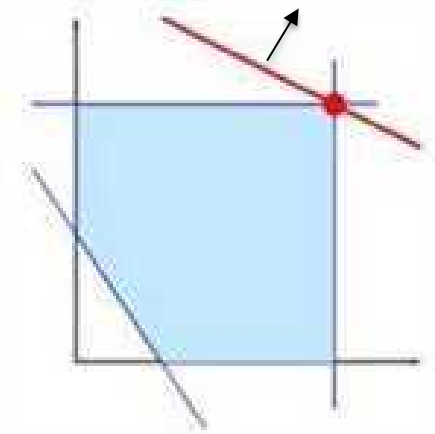
- Plot all constraints to identify the **feasible region**.
- Draw **objective function level lines** (parallel lines for different values).
- Move the level lines in the direction of optimization (maximize/minimize).
- **Optimal solution** occurs at a **vertex** (corner point) of the feasible region.

Limitations: Works best for 2 variables (easy to draw). For higher dimensions, use **Simplex or other algorithms**.

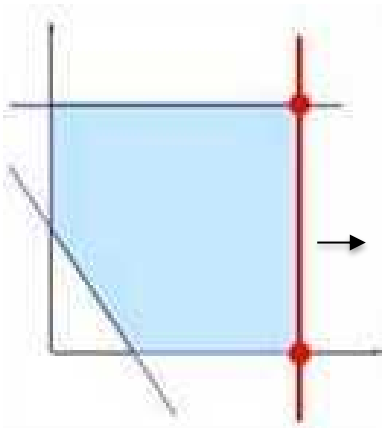
Agenda

- Background of linear programs
- Recap of linear algebra
- Solving linear programs (LP)
  -- The graphical method
  -- Different cases of LP outcomes
  -- The Simplex algorithm

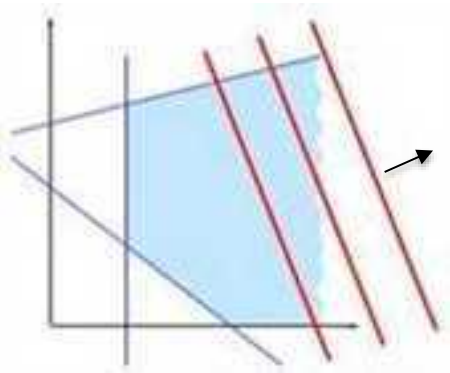
Quick question 2: Different cases of LP



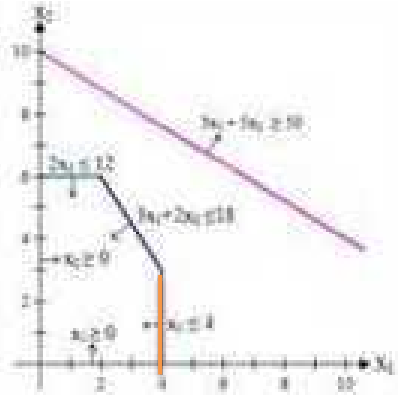
(a)



(b)



(c)



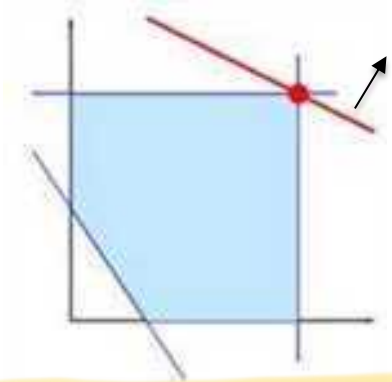
(d)

For each case, consider:

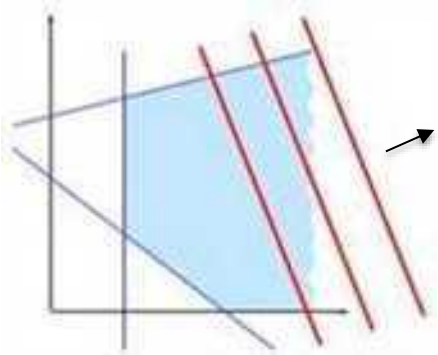
- Does an optimal solution exist?
- If yes, is it unique?
- If not, why not?

Blue shaded area: feasible region
Red lines: objective function levels

Different cases of LP outcomes

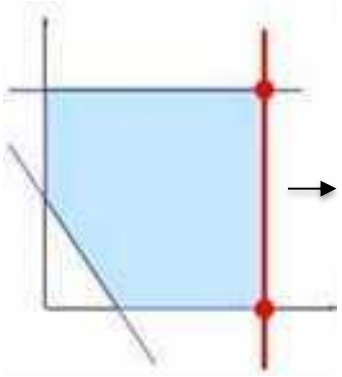


(a) Single optimal solution

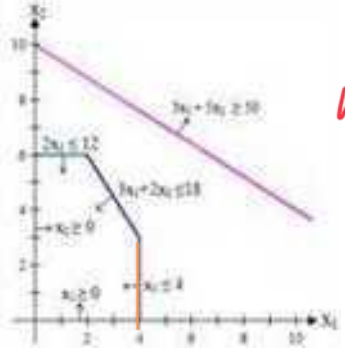


maximizing, unbounded will go to infinity.

(c) No optimal solution



(b) Infinite number of optimal solution



no feasible region

(d) No optimal solution

Three types of LPs

Every LP falls into one of these cases:

Infeasible — No solution satisfies all constraints.

Unbounded — Objective can improve indefinitely within the feasible region.

Finitely optimal — An optimal solution exists and is finite.

If finitely optimal, the solution can be:

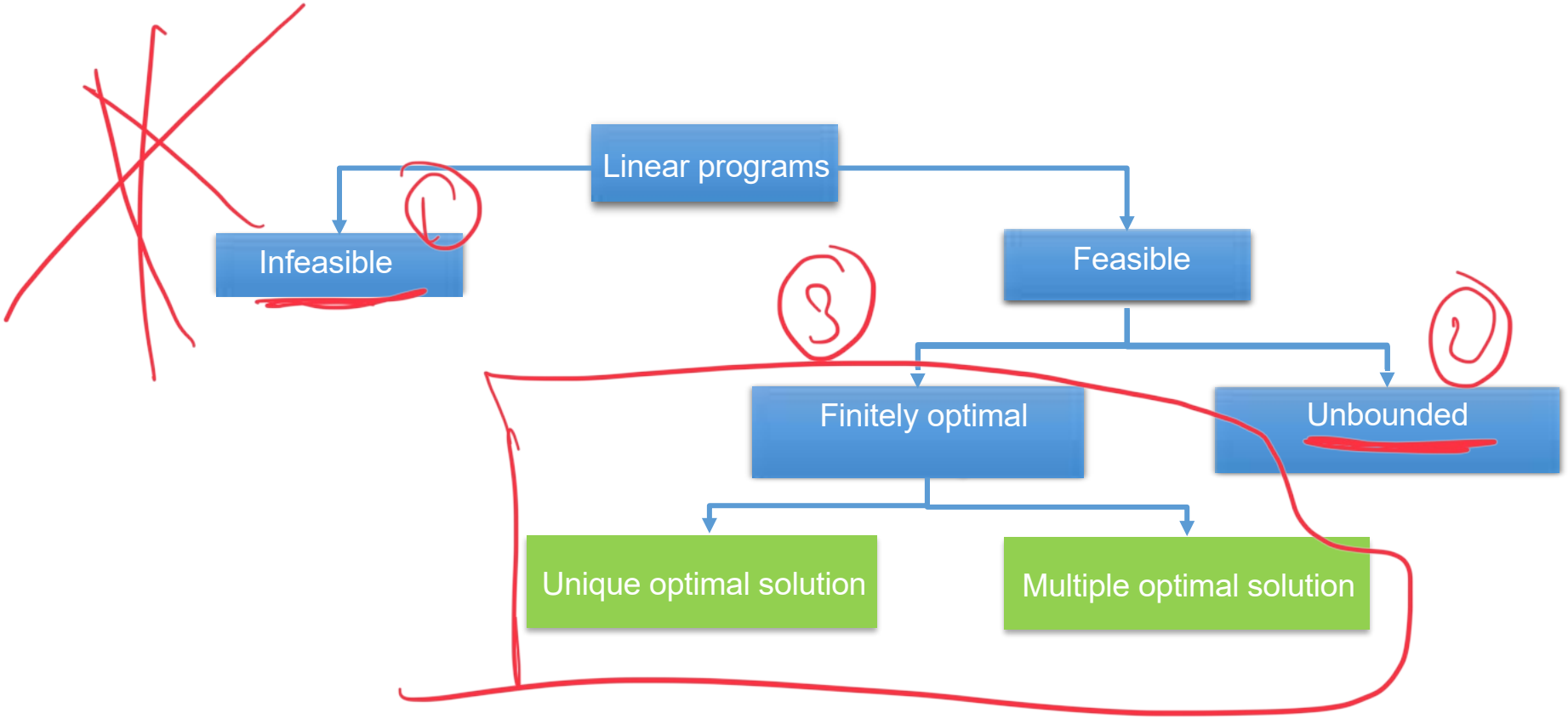
Unique, or

Multiple optimal solutions along an edge or face of the feasible region.

optimal line is on an edge

for higher dimensions.

Recap: Different cases of LP outcomes



Example: Infeasible problem

An LP is infeasible if its feasible region is empty.

Try out yourself online: https://www.desmos.com/calculator

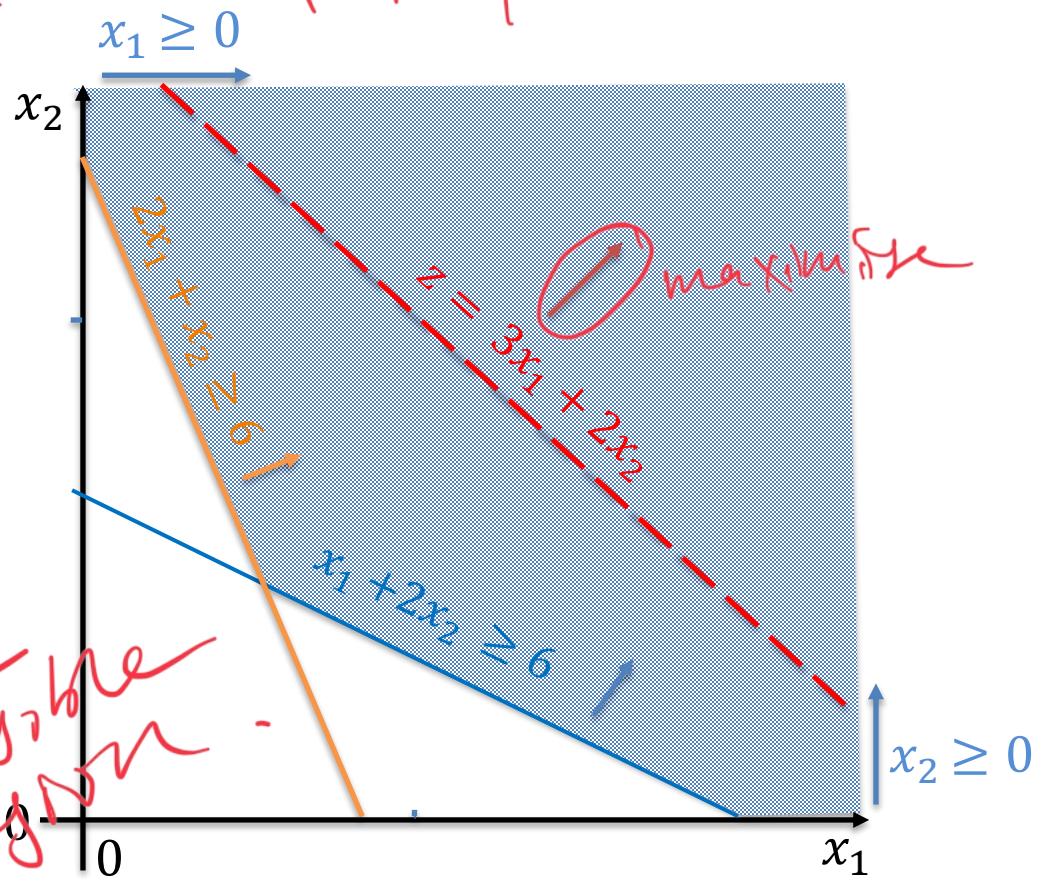
Example: Unbounded problem

An LP is unbounded if for any feasible solution, there is another feasible solution that is better.

Maximize $z = x_1 + x_2$

subject to:

$$x_1 + 2x_2 \geq 6$$
$$2x_1 + x_2 \geq 6$$
$$x_1, x_2 \geq 0$$



All bounded problems are feasible because there is a feasible region.

Example: Unbounded problem (cont.)

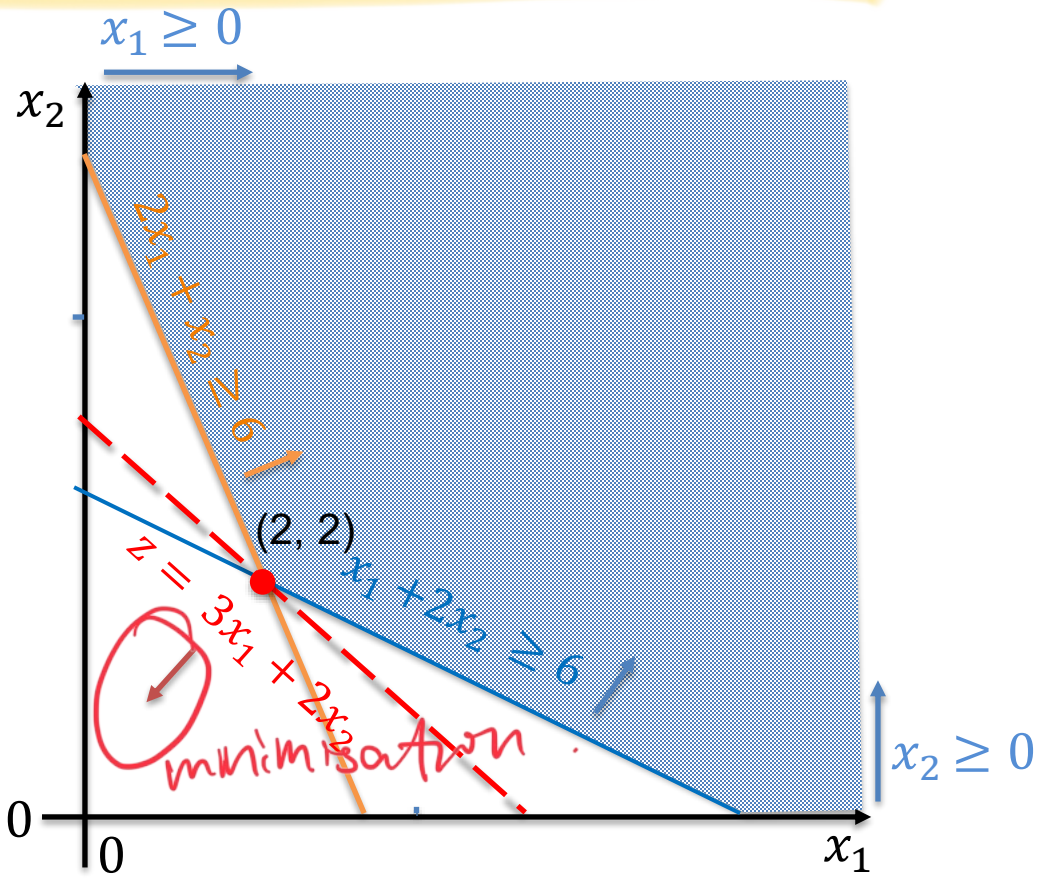
Clarify a common misconception:
An unbounded feasible region does *not* mean the LP is unbounded.

Minimize $z = x_1 + x_2$

subject to:

$$x_1 + 2x_2 \geq 6$$
$$2x_1 + x_2 \geq 6$$
$$x_1, x_2 \geq 0$$

The optimal solution is:
$x_1 = 2, \quad x_2 = 2$
$z_{min} = 4$



Example: Multiple optimal solution

An LP may have multiple optimal solutions.

False: In a linear program with two variables, if the slope of the level set of the objective function is different from all constraint boundary slopes, the optimal solution must occur at a single vertex of the feasible region (only slope, and have multiple)

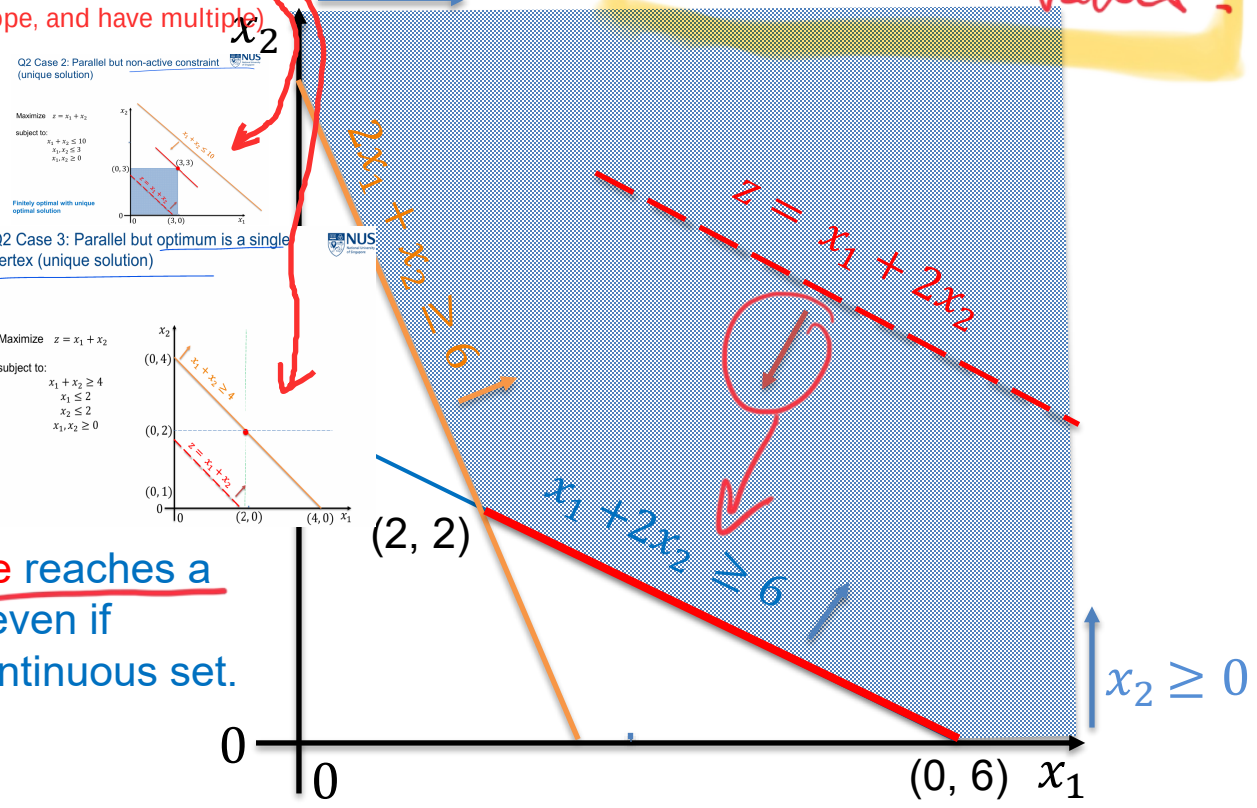
Minimize $z = x_1 + 2x_2$

subject to:

$$x_1 + 2x_2 \geq 6$$
$$2x_1 + x_2 \geq 6$$
$$x_1, x_2 \geq 0$$

$z_{min} = 6$

Finitely Optimal: Objective reaches a finite maximum/minimum, even if optimal solutions form a continuous set.



↳ can be single vertex, but x_1, x_2 are not single, they have a set of values.

Instead of vertex, the obj. func. reached a constraint line

Optimal solution vs. Optimal objective value

- **Optimal Solution:**
  The value of the decision variable(s) that gives the best result for the objective function.
  Tells us *where* the minimum or maximum occurs.
- **Optimal Objective Value:**
  The *best value* achieved by the objective function.
  Tells us *what* the minimum or maximum value is.

Optimal solution: $x^* = \underset{x \in \mathcal{F}}{\operatorname{argmin}} f(x)$

Optimal objective value $f^* = \min_{x \in \mathcal{F}} f(x) = f(x^*)$

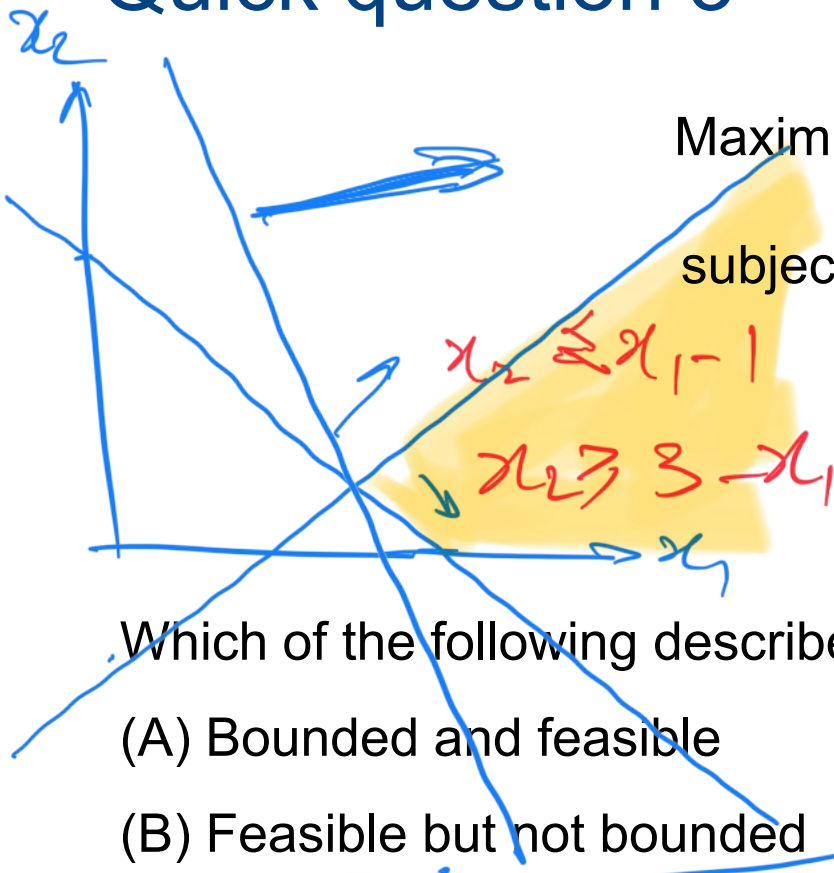
- Example:

  maximize x^2, s.t. $x \in [-2, 2]$

  Optimal solution: $-2, 2$

  Optimal objective value 4

Quick question 3



Maximize $z = 2x_1 + x_2$

change to y = mx+c form to draw more easily

subject to:

$$x_1 - x_2 \geq 1$$
$$x_1 + x_2 \geq 3$$
$$x_1, x_2 \geq 0$$

Which of the following describes the feasible region and the LP?

(A) Bounded and feasible

(B) Feasible but not bounded

(C) Bounded but not feasible

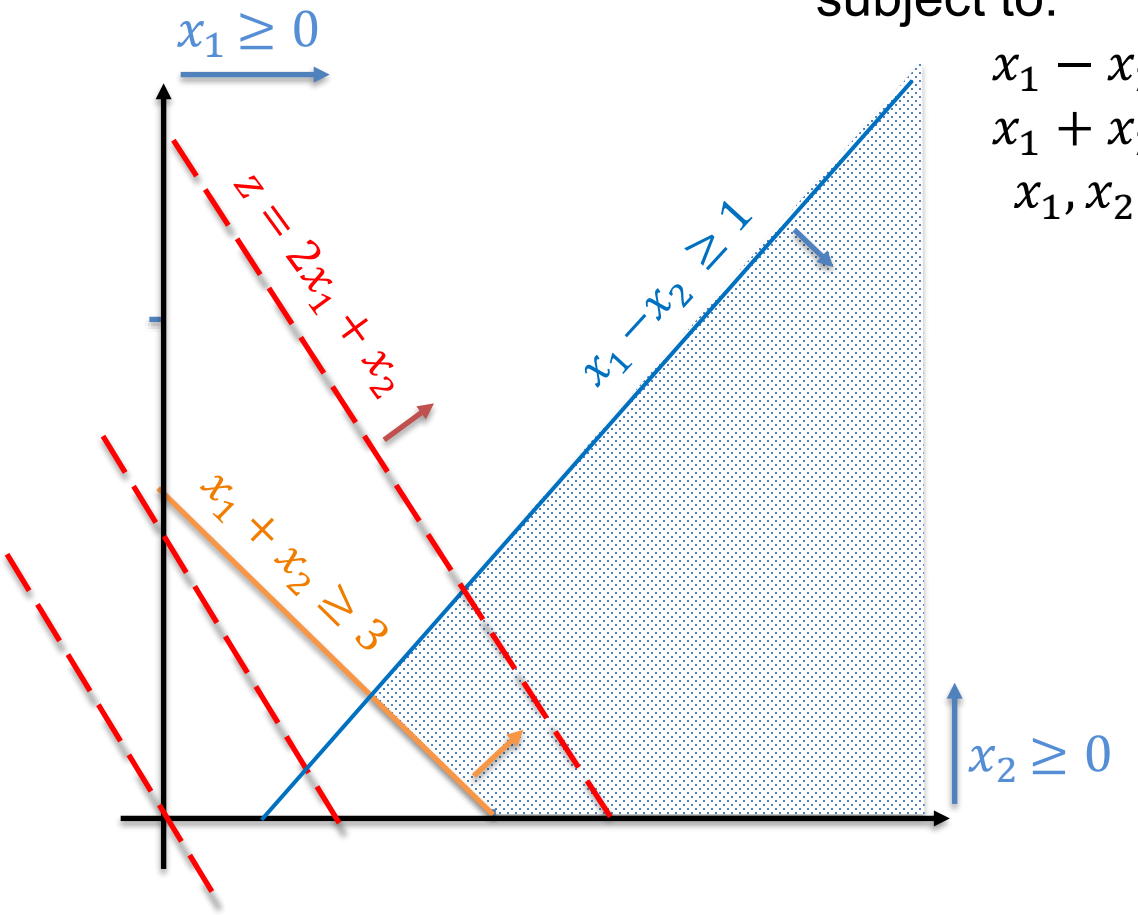
(D) Neither bounded nor feasible

Quick question 3

Maximize $z = 2x_1 + x_2$

subject to:

$$x_1 - x_2 \geq 1$$
$$x_1 + x_2 \geq 3$$
$$x_1, x_2 \geq 0$$

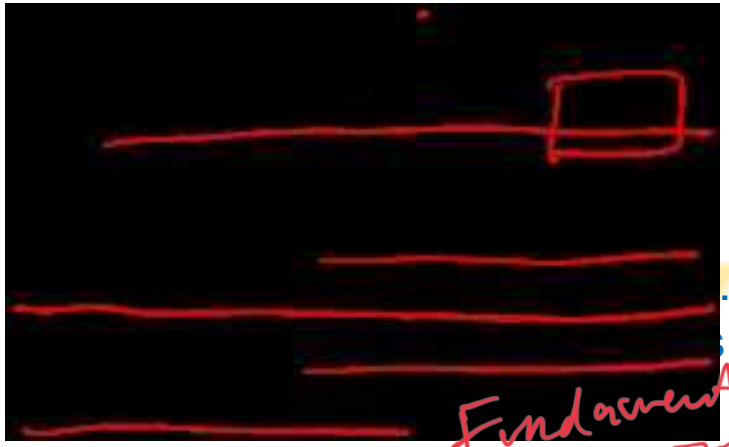


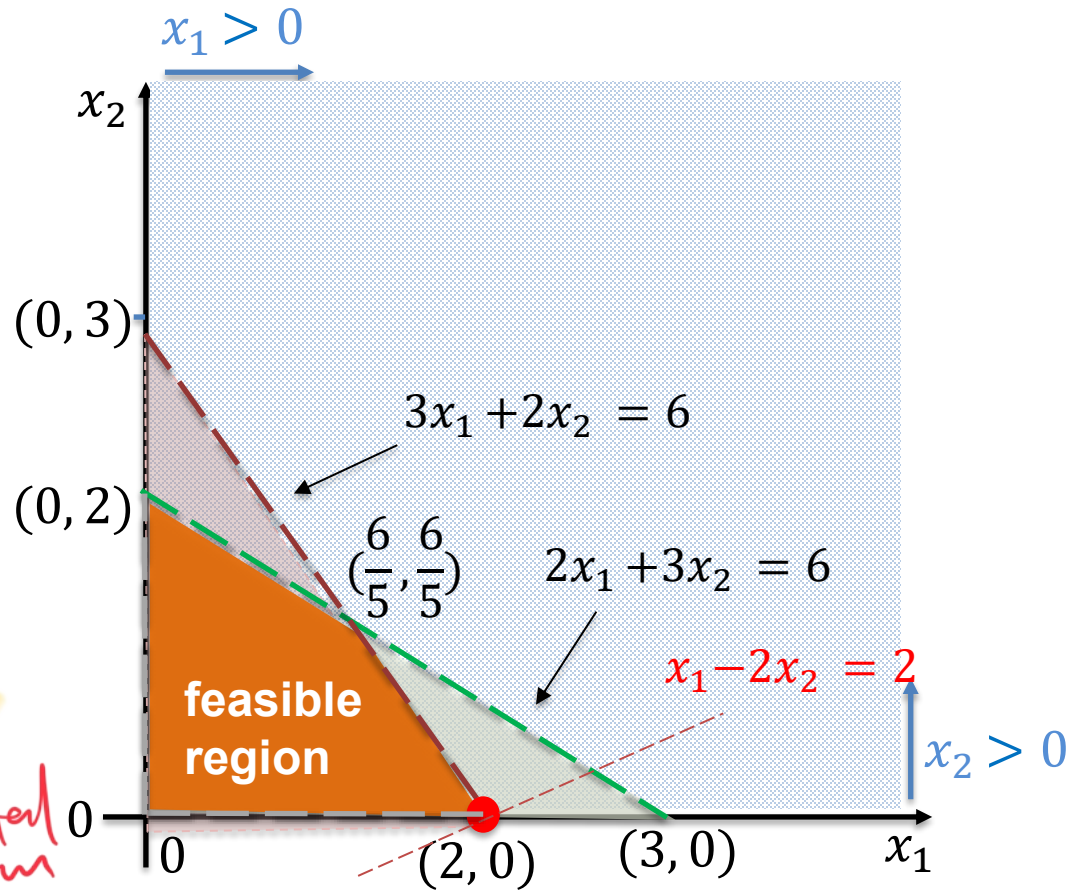
Ans: feasible but not bounded

Why not strict inequality?

Maximize $x_1 - 2x_2$

Subject to: $x_1 > 0$
$x_2 > 0$
$3x_1 + 2x_2 < 6$
$2x_1 + 3x_2 < 6$





Fundamental theorem of LP.

Recap: Linear programmes (LP)

An optimization problem that aims to **maximize or minimize a linear objective function**, subject to **linear equality and/or inequality constraints**.

$$\begin{aligned} \text{Maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{Subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Solving LPs:

Graphical method: applicable when there are only two variables.

Simplex algorithm: efficient for large-scale LPs

Fundamental theorem: Optimal solutions occur on the boundary of the feasible region — typically at its vertices (or along an edge between vertices).

Q1. In a linear programming (LP) problem, which of the following must always be true?

(A) The feasible region is bounded.

(B) The objective function and all constraints are linear.

(C) The optimal solution is always at the intersection of all constraints.

(D) The constraints can be strict inequalities as well as non-strict inequalities.

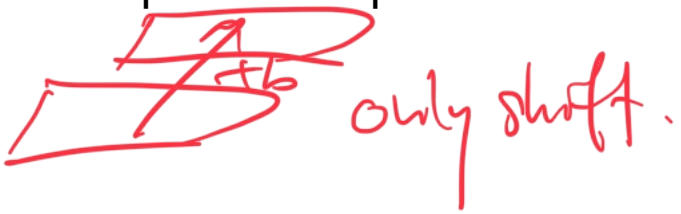
Linear vs. affine in LP

Strict mathematics

- Linear function: passes through the origin (e.g., $f(x_1, x_2) = 3x_1 - 2x_2$).
- Affine function: linear part + constant term (e.g., $x_1 + x_2 \leq 15$).

Why we still say "linear programming"

- In optimization, "linear" includes affine functions.
- Both constraints and objective functions may have constant terms.
- Constants don't change the feasible region shape or the optimal solution (only shift values).



Takeaway: In LP, *linear = linear/affine*.

Q2 True or false: If an LP is neither infeasible nor unbounded, it is finitely optimal.

- True
- False

only 3 types.
infeasible
unbounded.
finitely optimal.

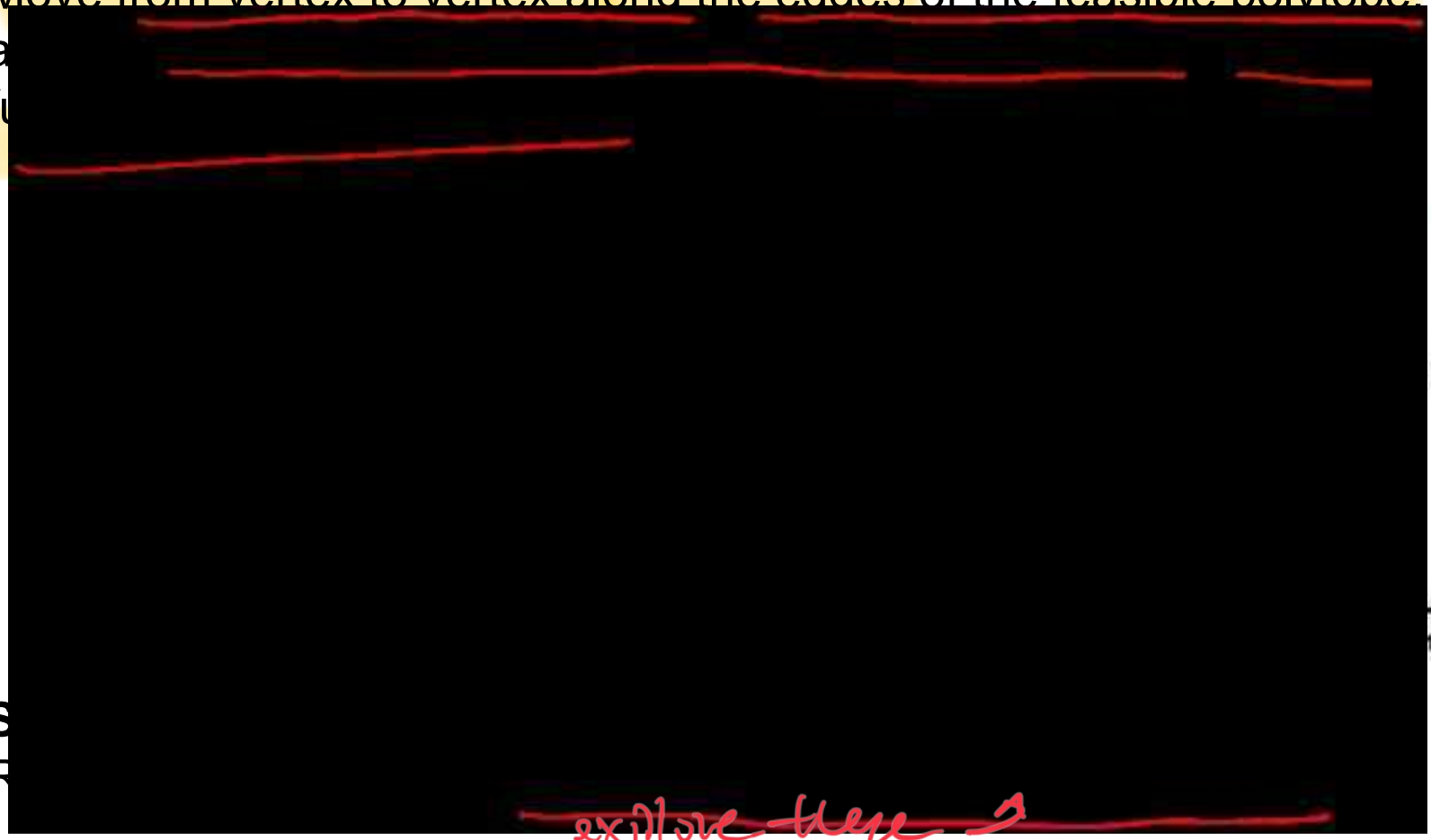
Agenda

- Background of linear programs
- Recap of linear algebra
- Solving linear programs (LP)
  -- The graphical method
  -- Different cases of LP outcomes
  -- The Simplex algorithm

Simplex algorithm

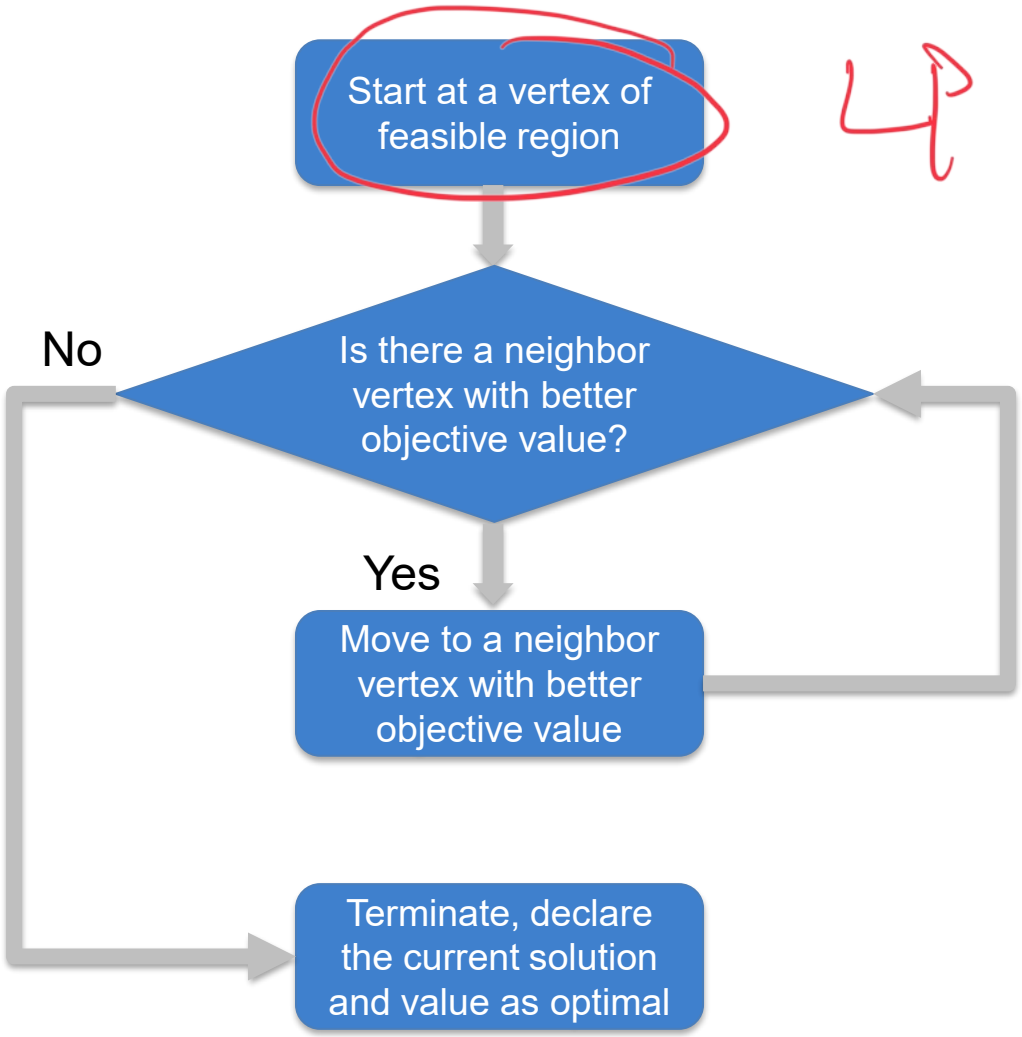
Basic idea:

Move from vertex to vertex along the edges of the feasible polytope,
a
fu

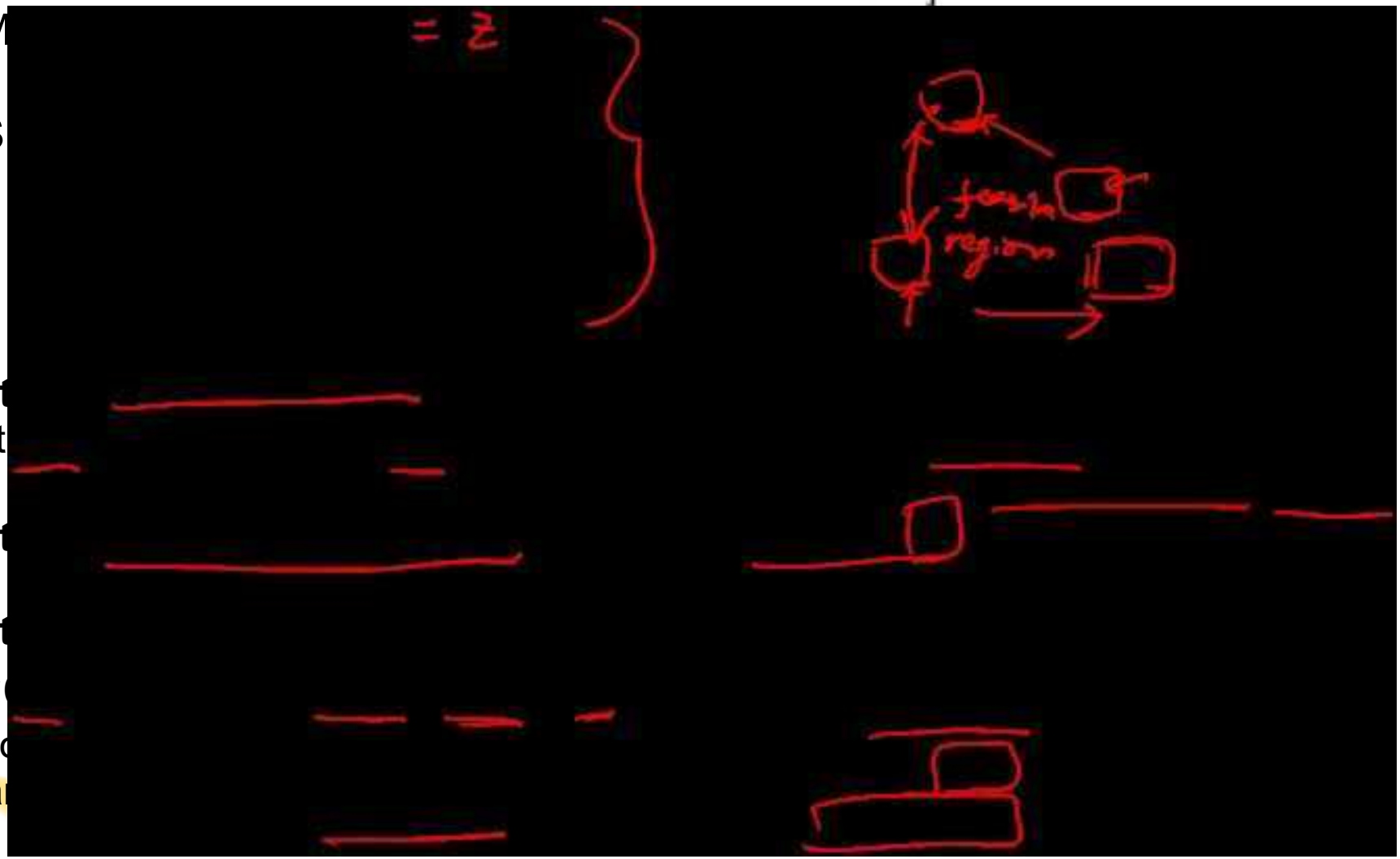


S
R

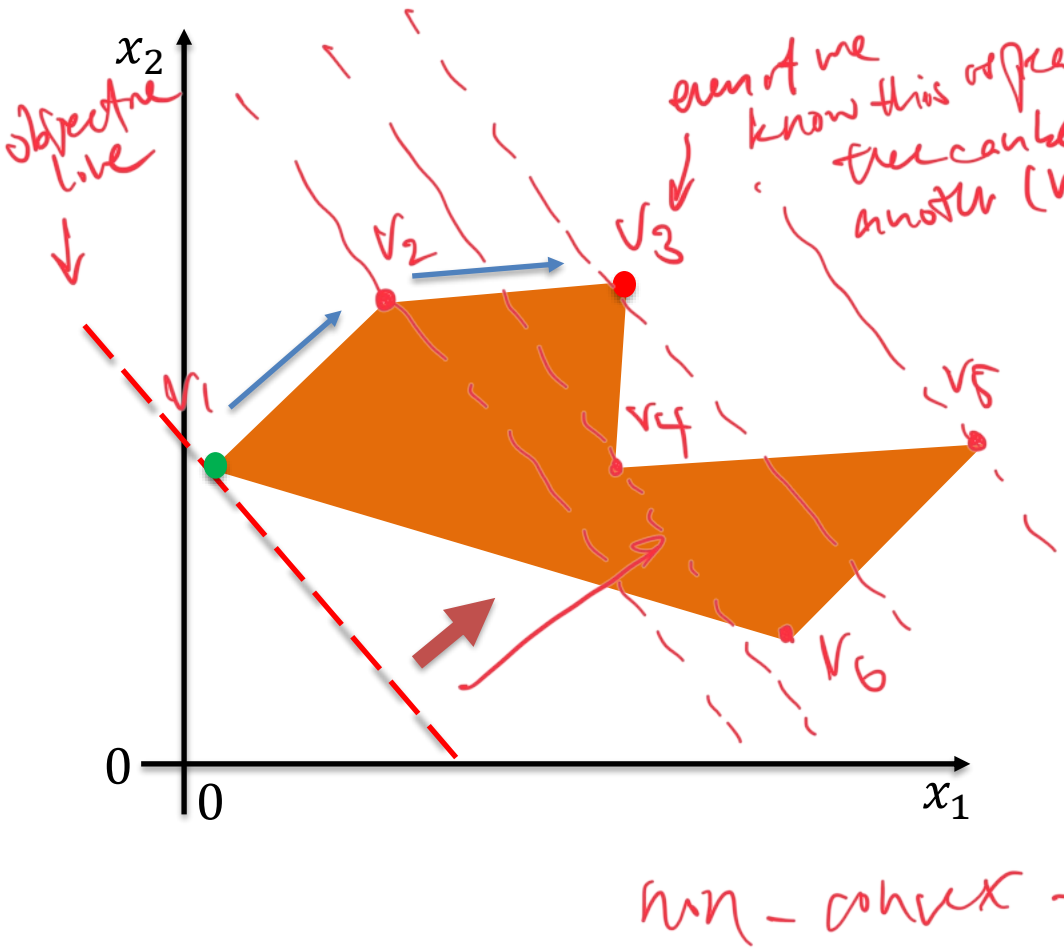
Simplex algorithm implementation



Simplex algorithm: Example



Will Simplex algorithm terminate early?



The Simplex algorithm is specifically designed for linear programs, where the feasible region is guaranteed to be **convex**.

Non-linear constraints can create non-convex regions, where multiple local optima may exist, and the global optimum may occur at interior points rather than at vertices. In such cases, the Simplex algorithm cannot be applied.

Summary

A **linear program (LP)** is an optimization problem that aims to maximize or minimize a linear objective function, subject to linear equality and/or inequality constraints.

Solving LPs:

Graphical method: works best when there are only two variables.

Simplex algorithm: efficient for large-scale LPs

Key insight: Optimal solutions occur on the boundary of the feasible region — typically at its vertices.

EE2213 Introduction to Artificial Intelligence

Lecture 6

Dr. Shaojing Fan
fanshaojing@nus.edu.sg

OVERVIEW OF COURSE CONTENTS

- **Introduction (Shaojing)**
  - What is AI
  - Applications of AI
  - AI agent
- **Search (Shaojing)**
  - Uninformed search algorithms: breadth-first, depth-first, uniform-cost(Dijkstra's algorithm)
  - Informed search algorithms: greedy best-first, A*
  - Applications

Path optimization

- **Optimization (Shaojing)**
  - Linear programming
  - **Convex problems**
  - **Applications**

Mathematical optimization

- **Machine learning (Wang Si)**
  - Supervised and unsupervised learning: regression, classification, clustering
  - Neural networks and deep learning
  - Applications
- **Knowledge representation (Wang Si)**
  - Knowledge Representation and Reasoning
  - Propositional Logic
  - Applications
- **Ethical considerations (Shaojing)**
  - Bias in AI
  - Privacy concerns
  - Societal impact

Recap: Linear programs (LP)

An optimization problem that aims to **maximize or minimize a linear objective function**, subject to **linear equality and/or inequality constraints**.

$$\begin{aligned} &\text{Maximize} && \mathbf{c}^T\mathbf{x} \\ &\text{Subject to} && \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & && \mathbf{x} \geq 0 \end{aligned}$$

Solving LPs:

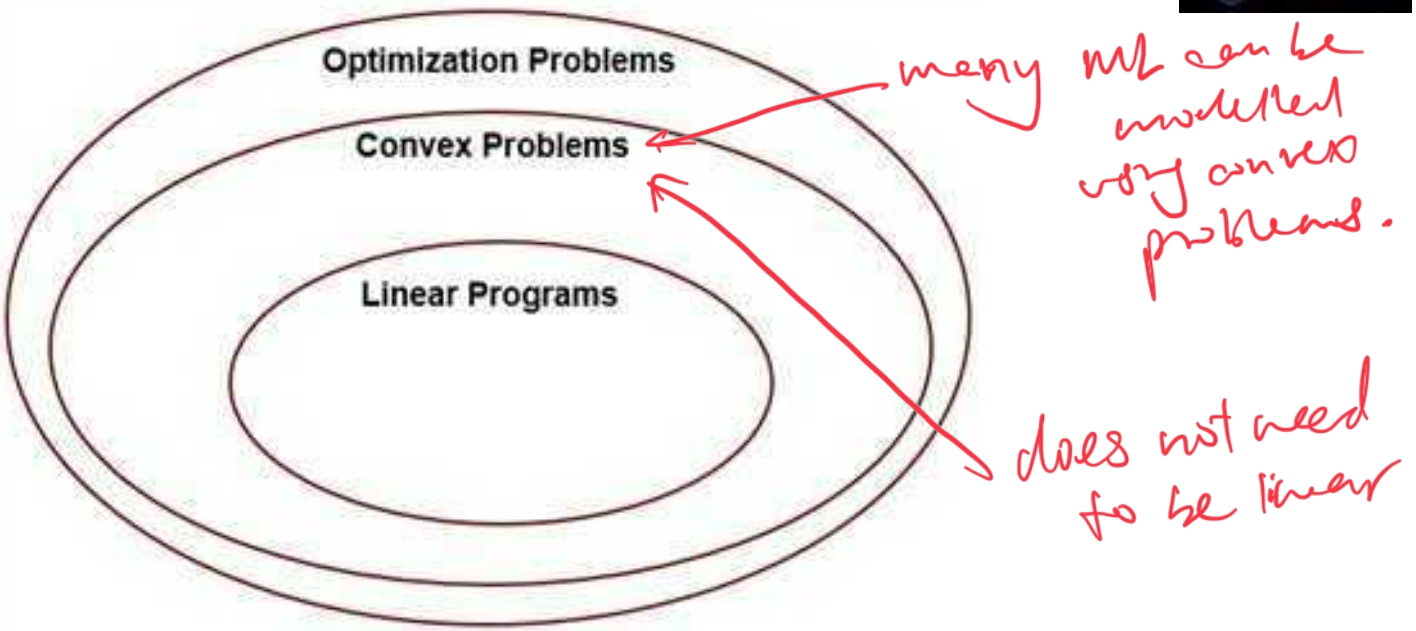
Graphical method: applicable when there are only two variables.

Simplex algorithm: efficient for large-scale LPs

LP.

Fundamental theorem in LP: Optimal solutions occur on the boundary of the feasible region — typically at its vertices (or along an edge between vertices).

Linear programs vs convex problems

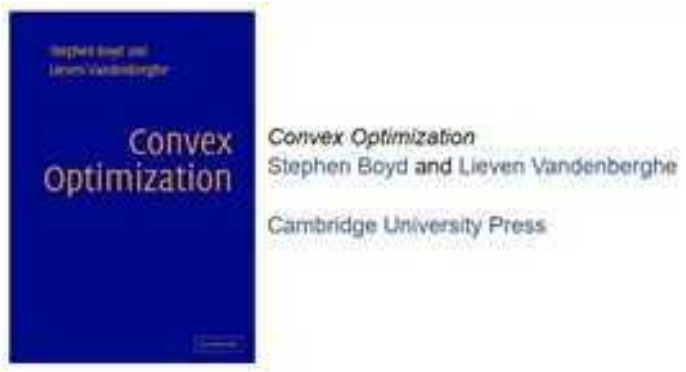


Linear programming is probably the most widely studied and broadly useful method for optimization. It is a special case of the more general problem of **convex optimization**, which allows the constraint region to be any convex region and the objective to be any function that is convex within the constraint region. Under certain conditions, convex optimization problems are also polynomially solvable and may be feasible in practice with thousands of variables. Several important problems in machine learning and control theory can be formulated as convex optimization problems (see Chapter 21).

Reference: Stuart Russell & Peter Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.) P140

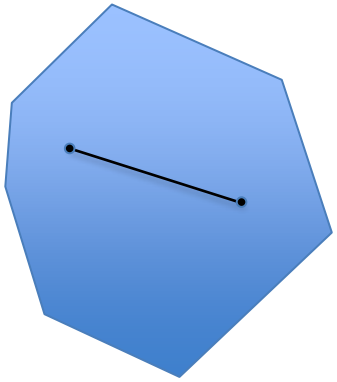
Agenda

- Background: Convex sets and convex functions
- Convex optimization problems: Real-world examples
- Solving convex problems: Gradient descent

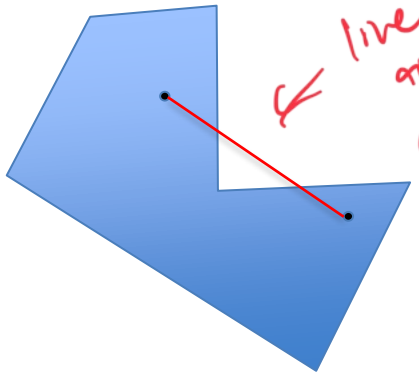


Reference (modern bible of convex optimization): Stephen Boyd & Lieven Vandenberghe, Convex Optimization, https://web.stanford.edu/~boyd/cvxbook/

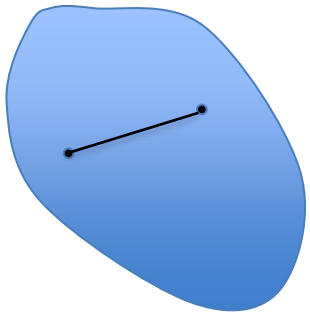
Convex set: Definition



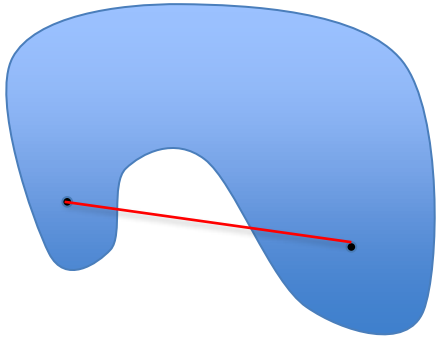
Convex



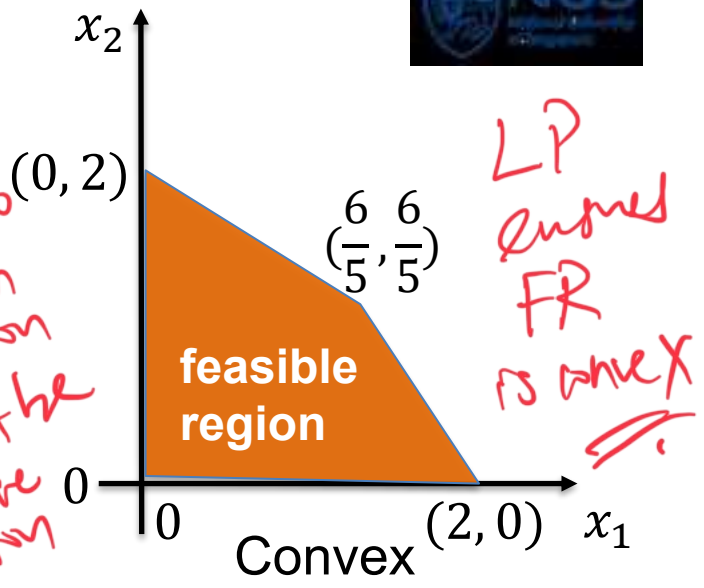
Non-convex



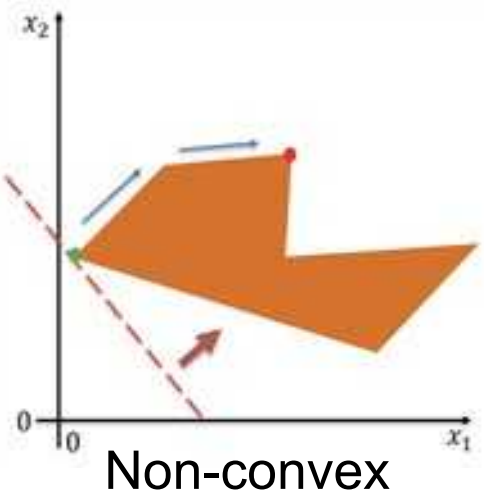
Convex



Non-convex



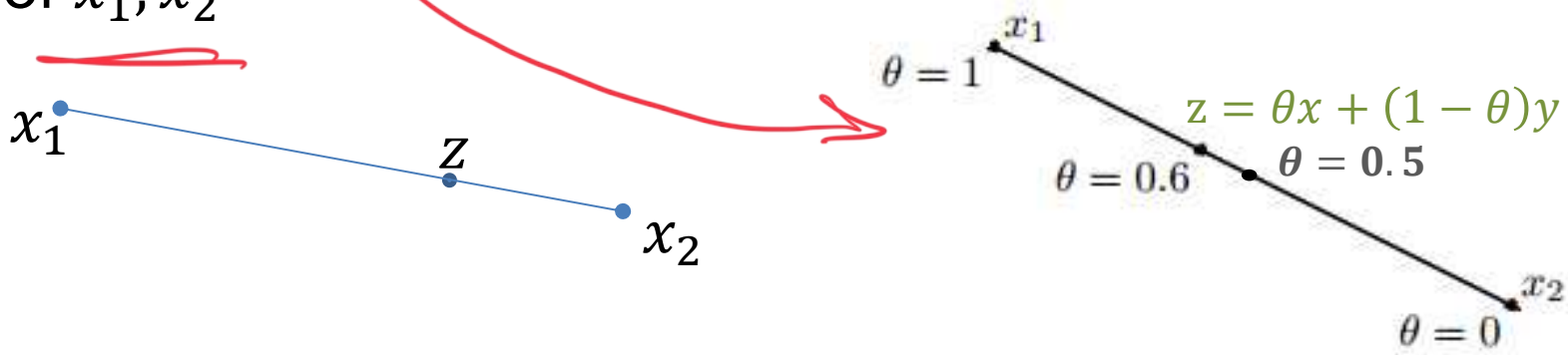
Convex



Non-convex

Convex combination: Definition

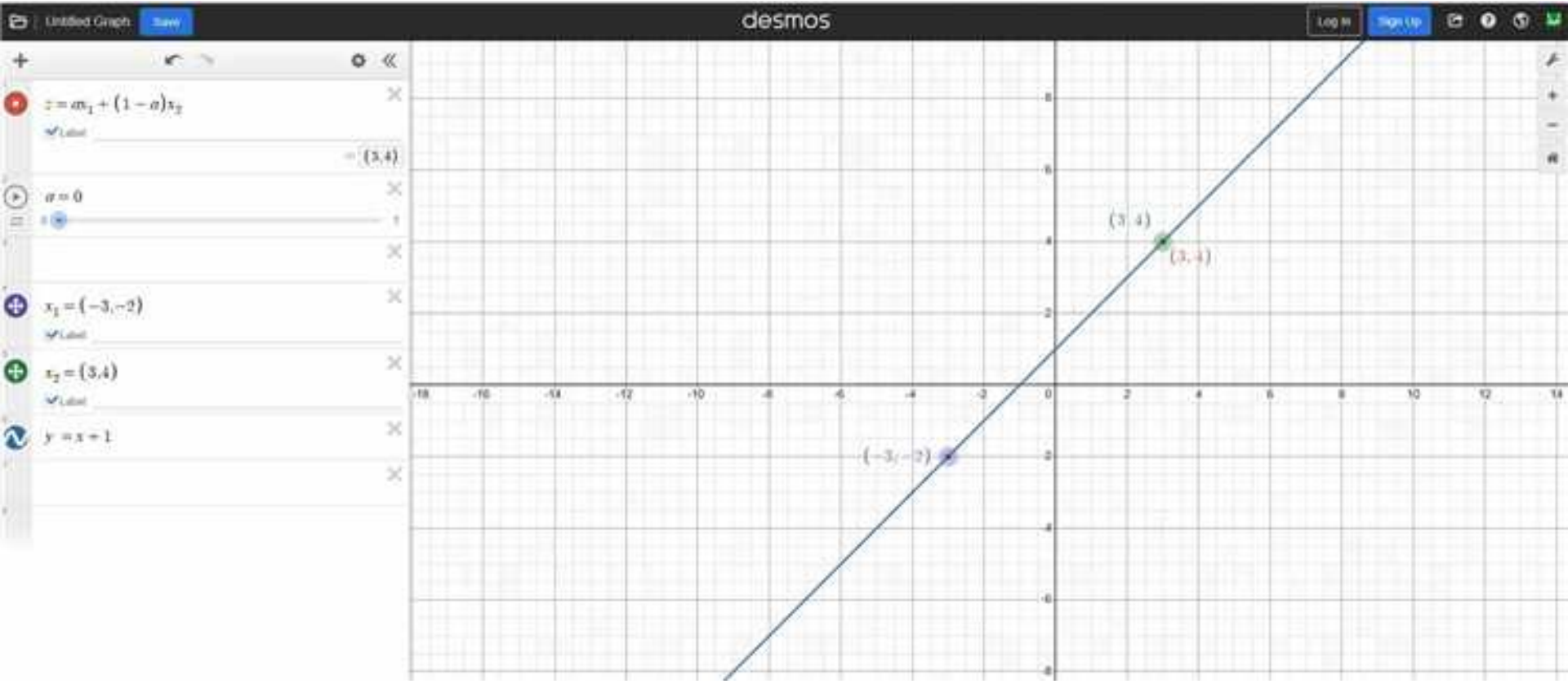
- A point between two points
- Given $x_1, x_2 \in \mathbb{R}^n$, a *convex combination* of them is any point of the form $z = \theta x_1 + (1-\theta)x_2$ where $\theta \in [0,1]$
- When $\theta \in (0,1)$, z is called a strict convex combination of x_1, x_2



A convex combination is just a fancy way of saying "a point between two points."

Convex combination: Definition

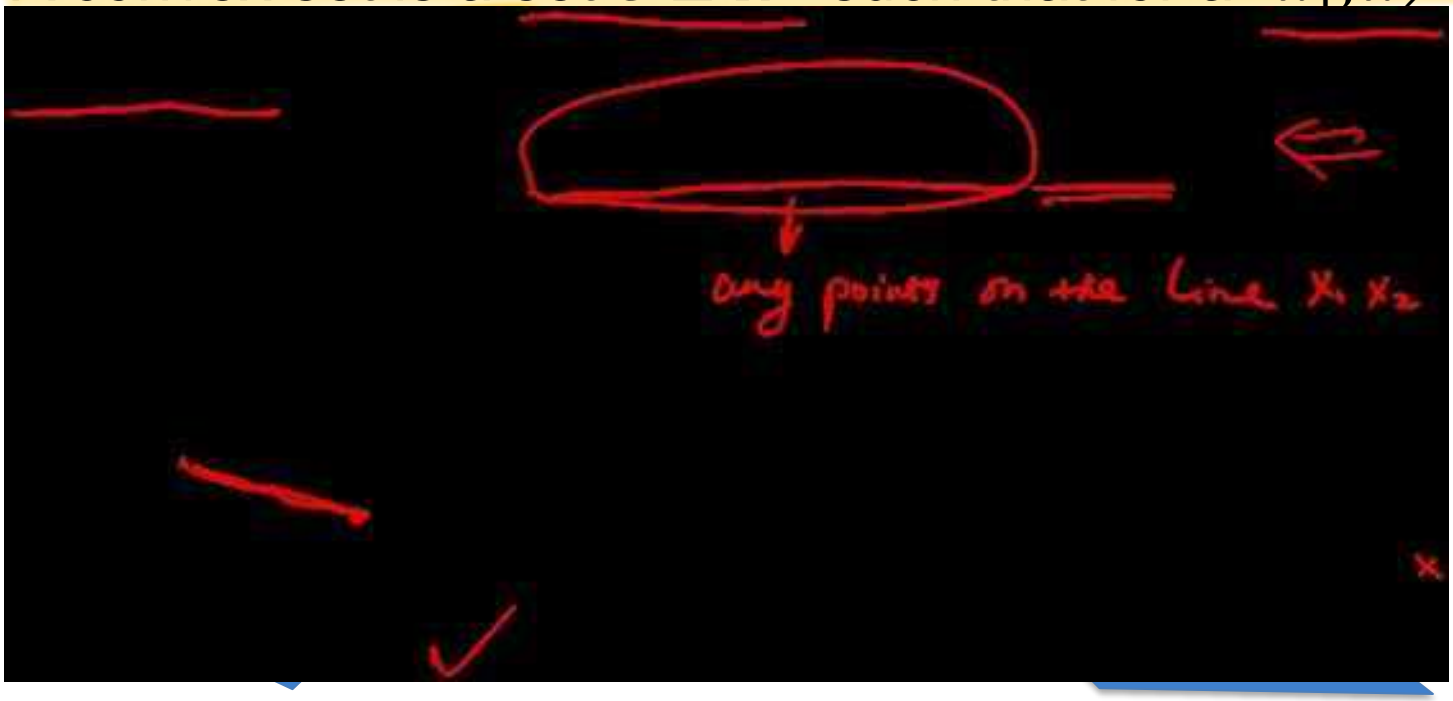
Given $x_1, x_2 \in \mathbb{R}^n$, a *convex combination* of them is any point of the form $z = \theta x_1 + (1-\theta)x_2$ where $\theta \in [0,1]$



Animation: https://www.desmos.com/calculator

Convex set: Definition

A convex set is a set $C \subseteq R^n$ such that for all $x_1, x_2 \in C$ and



Convex set

Non-convex set

Quick question 1

Which of the following sets are convex?

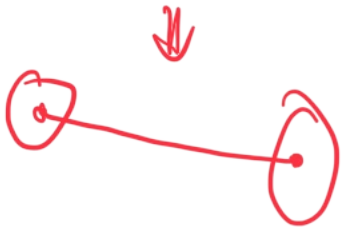
A: $C = \mathbb{R}^n$ whole space

B: $C = \emptyset$ null space

C: $C = \{x_0\},\ x_0 \in \mathbb{R}^n$ single point

D: $C = C_1 \cap C_2$, where C_1 and C_2 are convex sets

E: $C = C_1 \cup C_2$, where C_1 and C_2 are convex sets



Quick question 1

Which of the following sets are convex?

A: $C = \mathbb{R}^n$ ✓

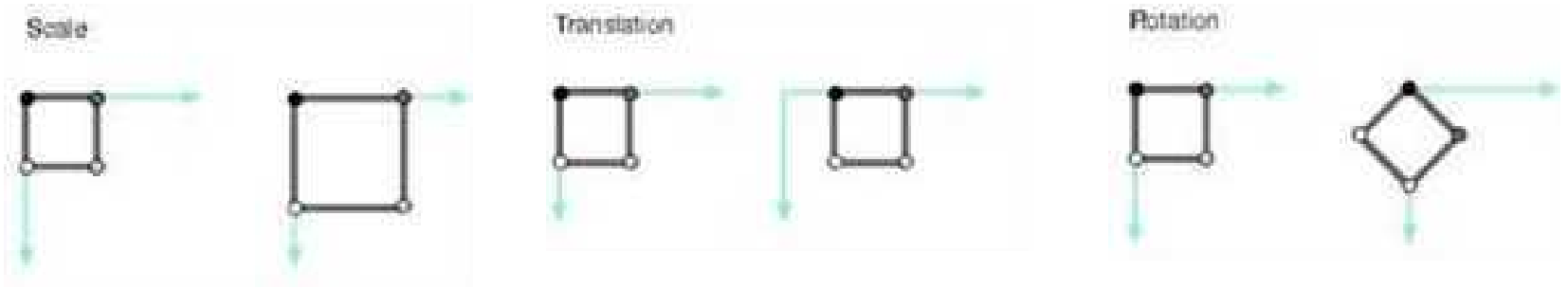
B: $C = \emptyset$ ✓

C: $C = \{x_0\},\ x_0 \in \mathbb{R}^n$ ✓

D: $C = C_1 \cap C_2$, where C_1 and C_2 are convex sets ✓

Convex set: Properties

- The empty set $\emptyset$ and $\mathbb{R}^n$ are both convex.
- Intersections of convex sets are convex.
- Convexity is **preserved** under three common operations:
  - **Scaling:** Multiplying all points by a positive scalar
  - **Translation:** Shifting to a new position (adding a constant vector to all points)
  - **Rotation:** Rotating them around an axis



Convex function: Definition

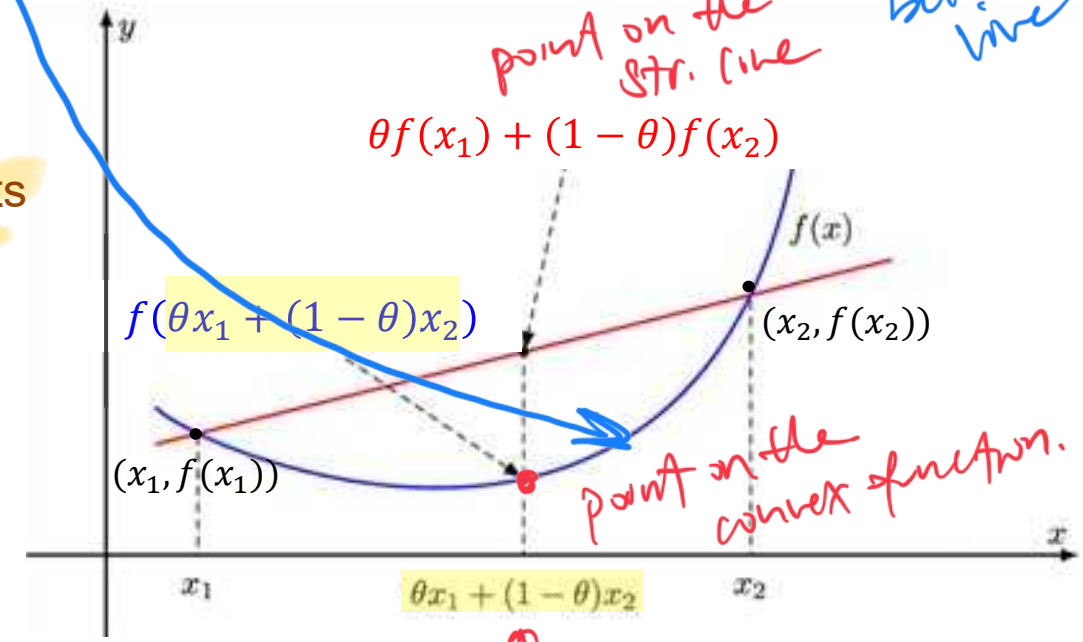
Let C be a convex set. A function $f: C \to \mathbb{R}$ is convex in C if $\forall x_1, x_2 \in C, \forall \theta \in [0,1]$,

$$f(\theta x_1 + (1-\theta)x_2) \leq \theta f(x_1) + (1-\theta) f(x_2)$$

If $C = \mathbb{R}^n$, we simply say f is a convex function.

Any chord between two points on the function always lies above the function.

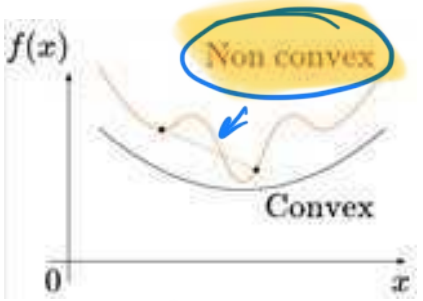
Value in the middle point is lower than the average of the two endpoint values.



Convex function: Definition (cont.)

Let C be a convex set. A function $f: C \to \mathbb{R}$ is convex in C if $\forall x_1, x_2 \in C, \forall \theta \in [0,1]$,

$$f(\theta x_1 + (1-\theta)x_2) \leq \theta f(x_1) + (1-\theta) f(x_2)$$



If $C = \mathbb{R}^n$, we simply say f is a convex function.

Any chord between two points on the function always lies above the function.

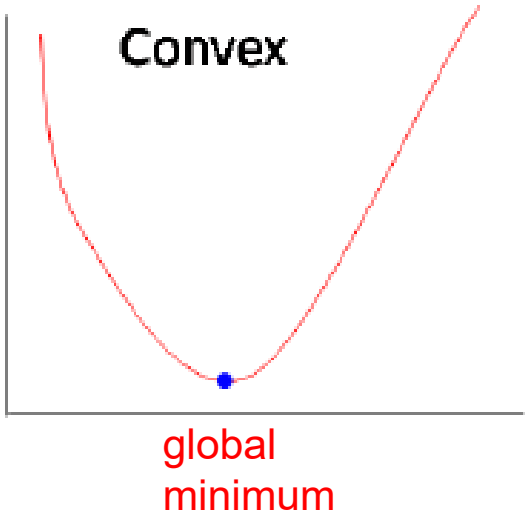
Convex function: Properties

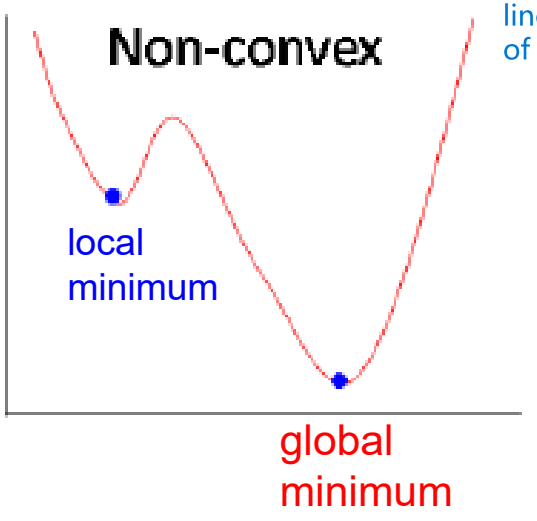
- Sum of convex functions is convex
  - $af(x) + bg(x)$ is convex for convex functions f, g and $a, b > 0$.
- Convexity is preserved under affine transformation
  - If $f(\mathbf{x}) = g(\boldsymbol{A}\mathbf{x} + \mathbf{b})$, g is convex, then $f(\mathbf{x})$ is convex.

  where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$
- Any local minimum is a global minimum

the product of two convex functions is NOT always convex

linear/affine functions are special cases of convex functions





Recap: Affine transformation

It's a type of transformation that preserves points, straight lines, and planes. Crucially, it also preserves parallelism (lines that are parallel before the transformation remain parallel after).

An affine transformation is a combination of a **linear transformation** (like rotation, scaling, reflection) and a **translation**(shifting).

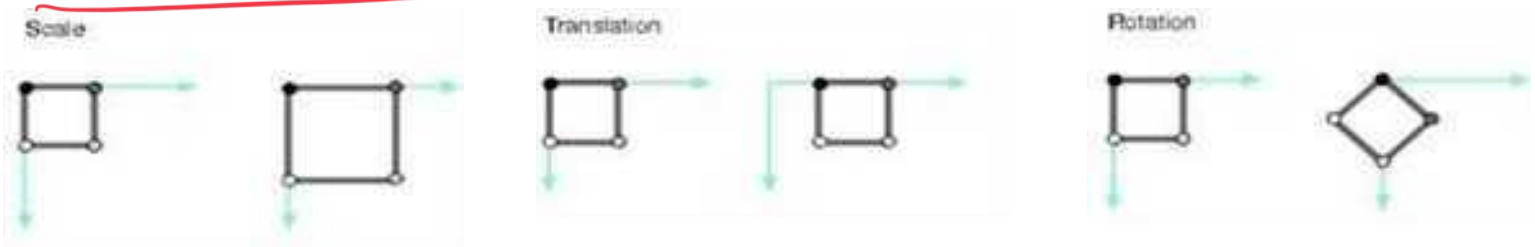
$$\mathbf{y} = \boldsymbol{A}\mathbf{x} + \mathbf{b}$$

where $\boldsymbol{A}$: matrix (rotation, scaling, etc.)

$\mathbf{b}$: vector (shift)

Why important?

- Convexity is preserved under affine transformations.
- That means if a set or function is convex, and you apply *scaling, rotation, or translation*, it **stays convex**.



Recap: Affine transformation

Example of affine transformation: Rotation the **input space**

Take the convex function: $f(x_1, x_2) = {x_1}^2 + 2{x_2}^2$
Let's rotate the ***input*** by 45 degrees. The rotation matrix is

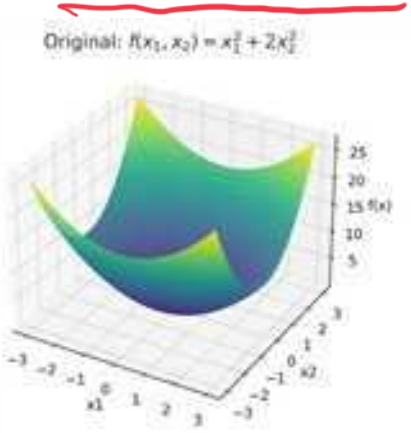
$$\boldsymbol{R} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$$

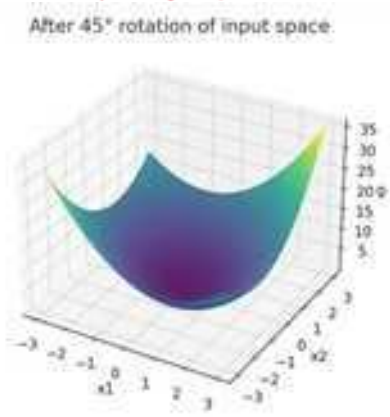
Rotate the input: $\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \boldsymbol{R}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} x_1 - x_2 \\ x_1 + x_2 \end{bmatrix}$, then $u_1 = \frac{x_1 - x_2}{\sqrt{2}}$, $u_2 = \frac{x_1 + x_2}{\sqrt{2}}$

Define the rotated function $g(\mathbf{x}) = f(\boldsymbol{R}\mathbf{x}) = f(\mathbf{u})$, $g(\mathbf{x}) = {u_1}^2 + 2{u_2}^2$

Substitute u_1, u_2 and expand

$$g(\mathbf{x}) = \left(\frac{x_1 - x_2}{\sqrt{2}}\right)^2 + 2\left(\frac{x_1 + x_2}{\sqrt{2}}\right)^2 = \frac{3}{2}{x_1}^2 + x_1 x_2 + \frac{3}{2}{x_2}^2$$



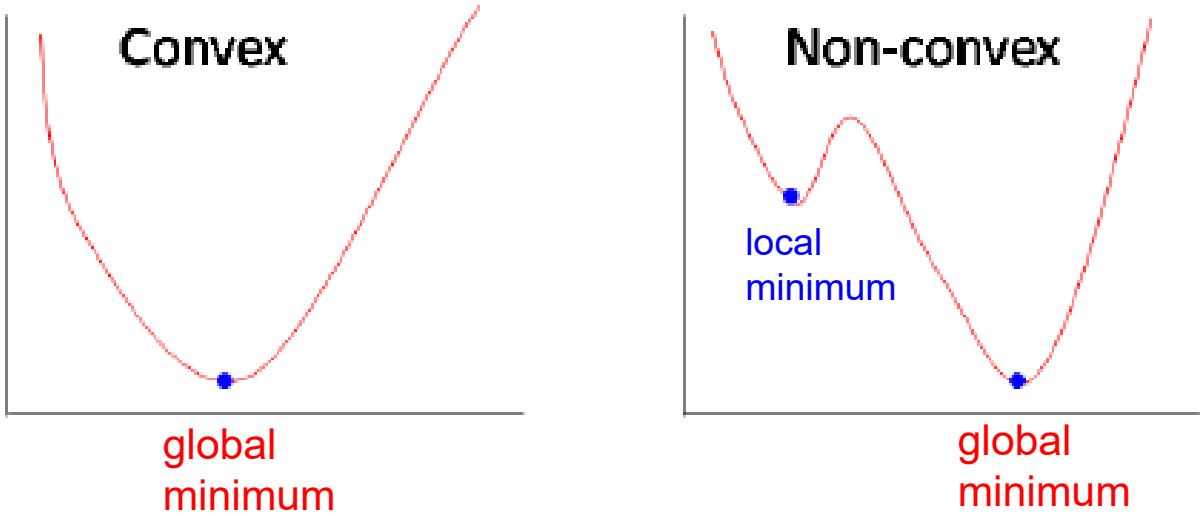


Tools: https://www.desmos.com/3d

Convex function: Properties

- Sum of convex functions is convex
  - $af(x) + bg(x)$ is convex for convex functions f, g and $a, b > 0$.
- Convexity is preserved under affine transformation
  - If $f(\mathbf{x}) = g(\boldsymbol{A}\mathbf{x} + \mathbf{b})$, g is convex, then $f(\mathbf{x})$ is convex.

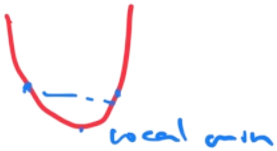
  where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$
- Any local minimum is a global minimum



Quick question 2

Which of the following functions are convex?

A: $f(x) = x^2 + 2$

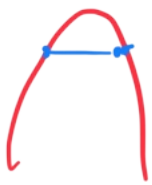


B: $f(x) = 3x + 4$ LP (affine function) is a special case of convex

C: $f(x) = \sin(x)$ no local min

D: $f(x) = 3$ constant is also special case

E: $f(x) = -3x^2$



→ concave function

Quick question 2

Which of the following functions are convex?

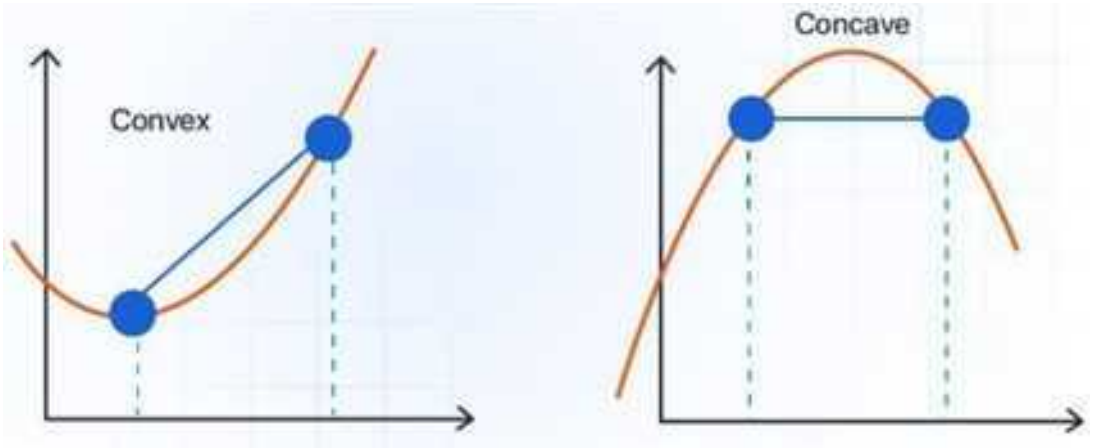
A: $f(x) = x^2 + 2$ ✓

B: $f(x) = 3x + 4$ ✓

C: $f(x) = \sin(x)$ ✗

D: $f(x) = 3$ ✓

E: $f(x) = -3x^2$ ✗



Examples of convex functions

Linear/affine functions:

$$f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} + b$$

where $\mathbf{a},\ \mathbf{x} \in \mathcal{R}^n$, $b \in \mathcal{R}$

Quadratic functions:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} + d$$

where $\mathbf{a},\ \mathbf{c},\ \mathbf{x} \in \mathcal{R}^n$, $d \in \mathcal{R}$, $\mathbf{Q} \in \mathcal{R}^{n\times n}$ is a symmetric matrix

Example: $f(x) = 2x^2 + 7x - 3$

Agenda

- Background: Convex sets and convex functions
- Convex optimization problems: Real-world examples
- Solving convex problems: Gradient descent

Convex optimization problem: Example 1

Problem (Wi-Fi placement to serve many devices)

You're placing one Wi-Fi router in a 4×4 m room. There are m devices at known locations $u_1, \ldots u_m \in \mathcal{R}^2$ (phones, laptops, smart TV, etc). You want the router to give good average performance, so you minimize the sum of squared distances $(dist_{Euclidean})$ from the router position, $x = (x_1, x_2)$ to the devices. The router must be placed inside the living area: the half-plane behind the divider $x_1 + x_2 \geq 3$ and inside the room $0 \leq x_1, x_2 \leq 4$

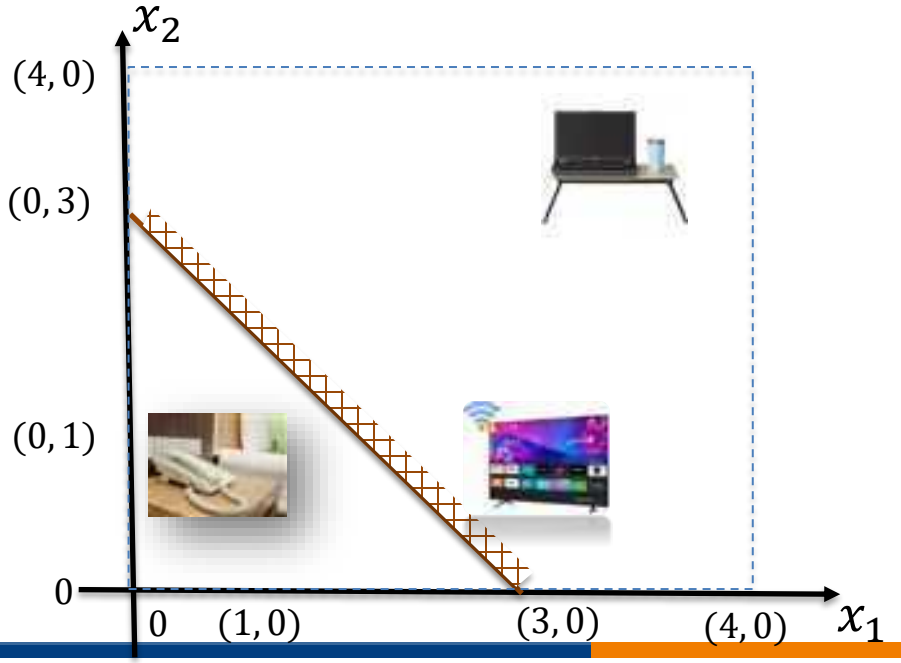
Formulation:

$$\min_{x \in \mathcal{R}^2} f(x) = \sum_{i=1}^{m} \|x - u_i\|_2^2$$

s. t. $x_1 + x_2 \geq 3$,

$0 \leq x_1 \leq 4$

$0 \leq x_2 \leq 4$



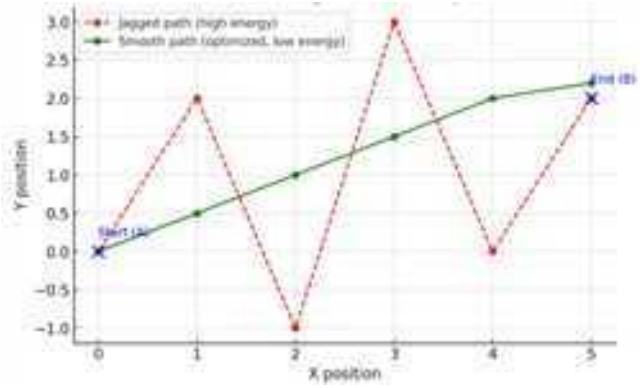
$\|x - u_i\|_2 = \sqrt{(x_1 - u_{i1})^2 + (x_2 - u_{i2})^2}$ (the Euclidean distance between the router and device i)

Convex optimization problem: Example 2

Problem (Minimizing energy cost in robot path planning)
Example: A delivery robot wants to move from point A to point B smoothly, using *as little energy as possible*. *Sharp turns cost more energy*.

Minimizing the sum of squared differences in positions, which *penalizes rapid movement changes*:

$$\min_{\mathbf{x}} \sum_{i=2}^{n-1} \|\mathbf{x}_{i+1} - 2\mathbf{x}_i + \mathbf{x}_{i-1}\|^2$$



$\mathbf{x}_i$: the robot's position at step i

manhatton distance.

$\|x - y\|_1 = the\ Manhatton\ norm\ (a.k.a.\ L_1\ norm)$

Second difference: $(\mathbf{x}_{i+1} - \mathbf{x}_i) - (\mathbf{x}_i - \mathbf{x}_{i-1}) = \mathbf{x}_{i+1} - 2\mathbf{x}_i + \mathbf{x}_{i-1}$

Convex optimization problem: Example 3 (machine learning context)

Example: I want to sell my 1,700 ft² house—how much should I ask for?

Historical records.

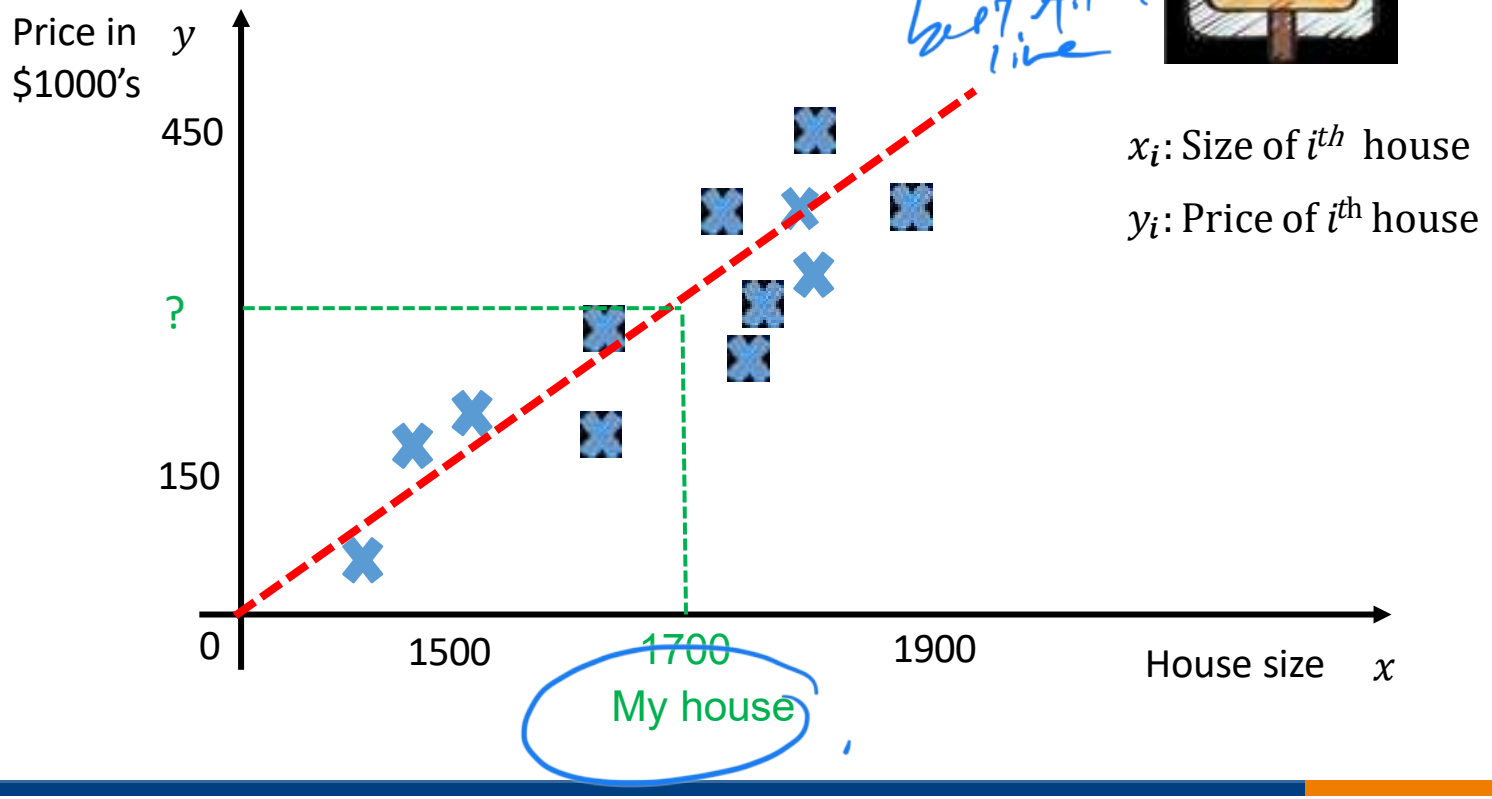
My model

House size (feet^2)	Sold price ($1000)
1227	128
1360	145
1600	260
1780	306
1900	450
2107	460
…..	…..

predict cost of any house

Convex optimization problem: Example 3 (machine learning context)

Example: I want to sell my 1,700 ft² house—how much should I ask for?

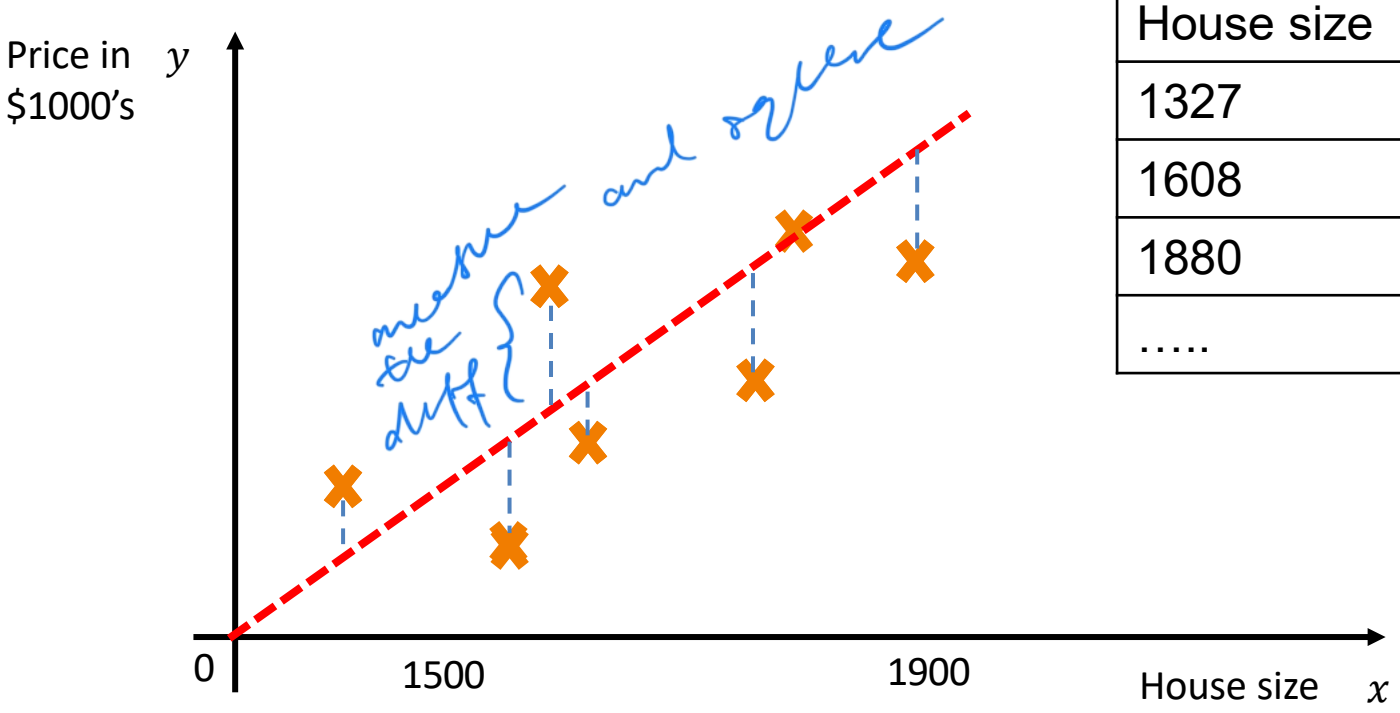


Convex optimization problem: Example 3 (machine learning context)

How accurate is my prediction?

Historical records (used as validation data):

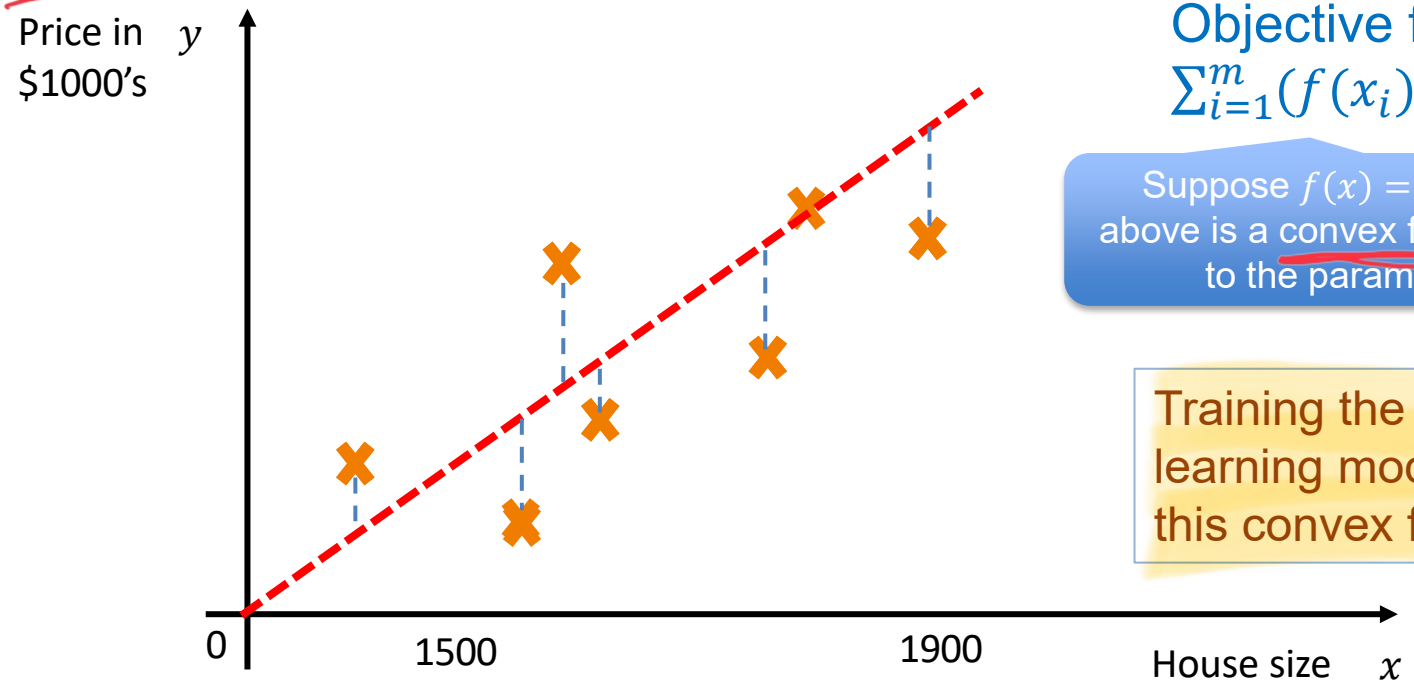
House size	Sold price
1327	148
1608	265
1880	324
…..	…..



Convex optimization problem: Example 3 (machine learning context)

Given m data points (x_1, y_1), (x_2, y_2), (x_m, y_m), linear regression finds a function $f(x)$ (a mapping from x to y) by minimizing the total squared errors $\min_f \sum_{i=1}^{m}(f(x_i) - y_i)^2$

That is, it finds the best-fitting line by minimizing the differences between predicted and actual values.



Objective function:
$\sum_{i=1}^{m}(f(x_i) - y_i)^2$

Suppose $f(x) = ax + b$. Then the above is a convex function with respect to the parameters a and b.

Training the machine learning model = minimizing this convex function

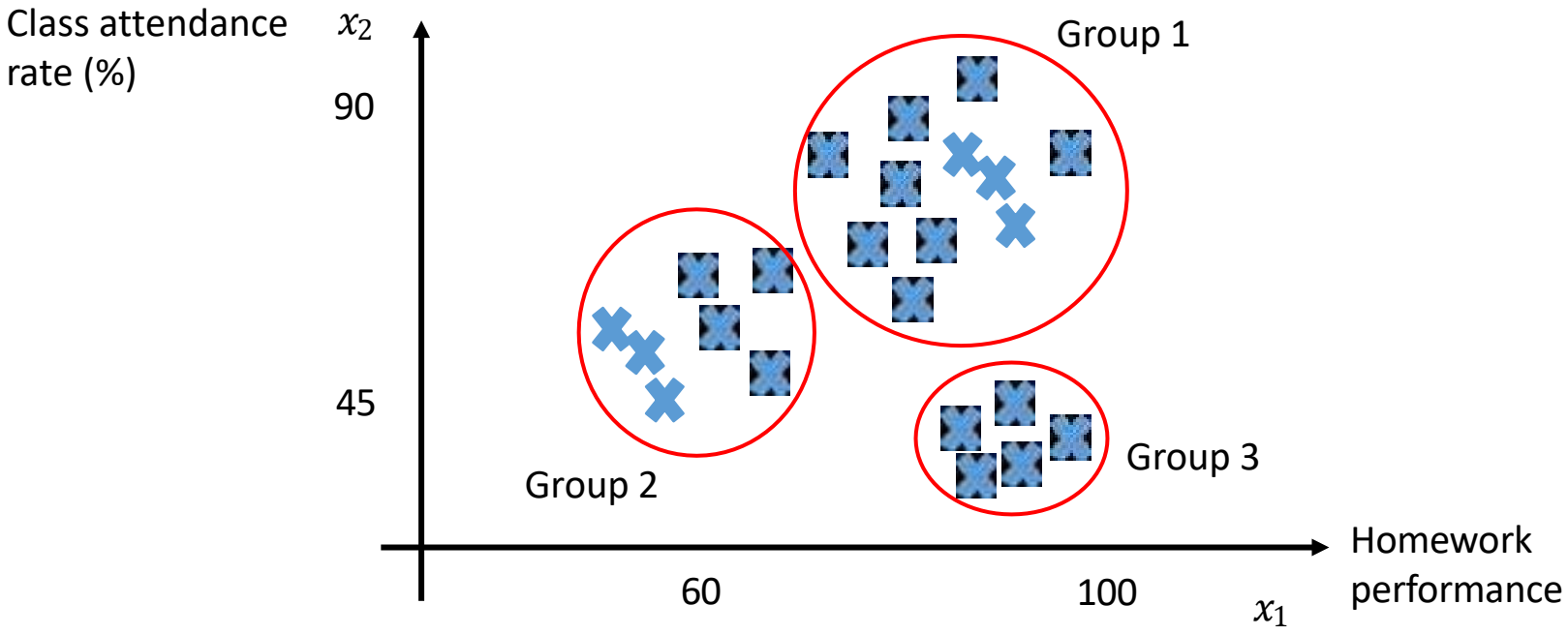
Convex optimization problem: Example 4 (machine learning context)

Example: How many types of students are there in EE2213?

$$x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \end{bmatrix}$$

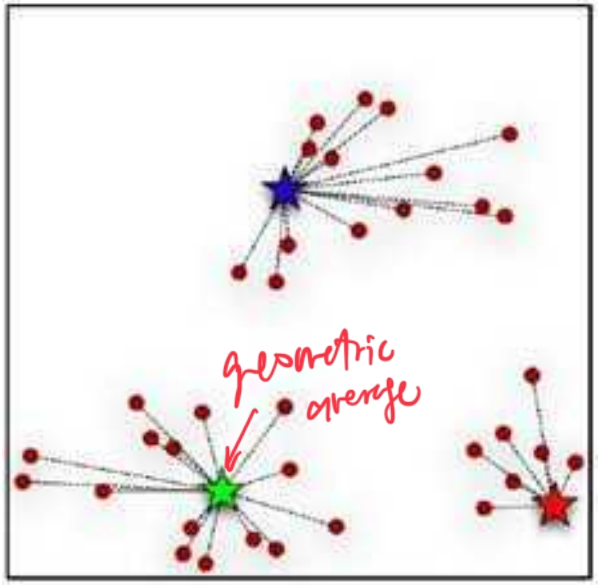
x_{i1}: Homework performance of i^{th} student in EE2213

x_{i2}: Attendance rate of i^{th} student in EE2213



Convex optimization problem: Example 4 (machine learning context)

How good is the clustering result?



We assign each data point x_i to a cluster center c_k , and minimize the total **within-cluster distance**, measured by the **sum of squared distances** between each point and its assigned center:

$$\min_{c_k} \sum_{i=1}^{m} \sum_{k=1}^{K} w_{ik} \|x_i - c_k\|^2$$

Convex function with respect to c_k

Training the model at this step = minimizing this convex function with respect to the centers c_k

$w_{ik} = 1$ if point x_i is assigned to cluster k; otherwise 0. w_{ik} is fixed here, and we hope to update c_k.

Agenda

- Background: Convex sets and convex functions
- Convex optimization problems: Real-world examples
- Solving convex problems: Gradient descent

Convex optimization: Gradient descent

can only be applied to

- Continuous & differentiable setting
  - -- Iteratively update $\mathbf{x}$ to improve the objective
  - -- Use gradient-based method (e.g., gradient descent)

cannot

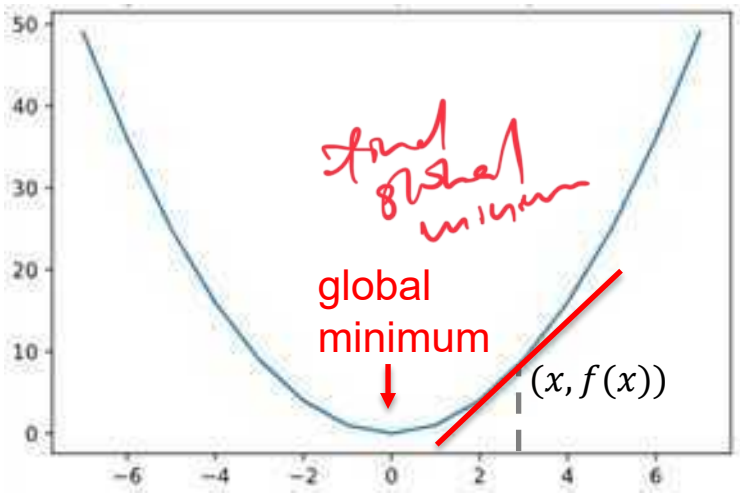
- Discrete setting *(not covered in this course)*
  - -- Local search or integer programming
  - -- Requires different techniques, often problem-specific

Motivation for gradient descent

Goal: Minimize a **convex** objective function $f(\mathbf{x})$

-- $\mathbf{x} \in \mathcal{R}^n$ is a vector of continuous variables

-- $f(\mathbf{x})$ is differentiable



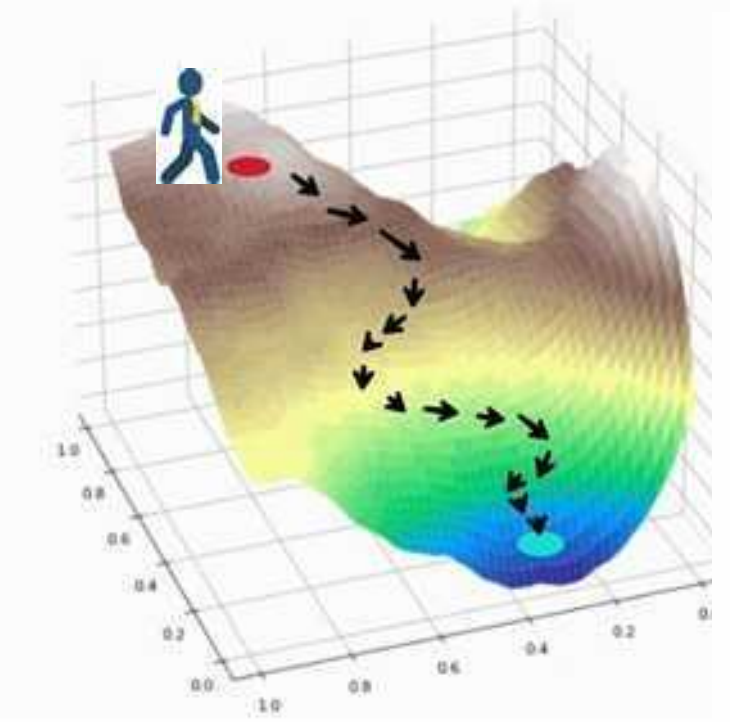
Why use gradient?

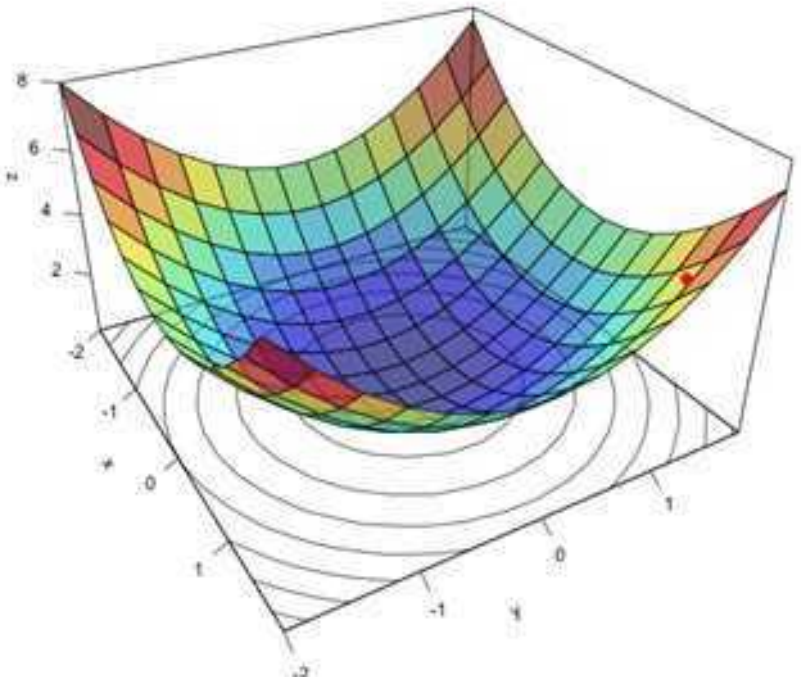
--The **gradient** $\nabla f(\mathbf{x})$ points in the direction of **steepest increase**

--To minimize, we move in the **opposite direction**

Motivation for gradient descent

Gradient descent solves convex optimization by iteratively moving toward the global minimum along the direction of steepest decrease.



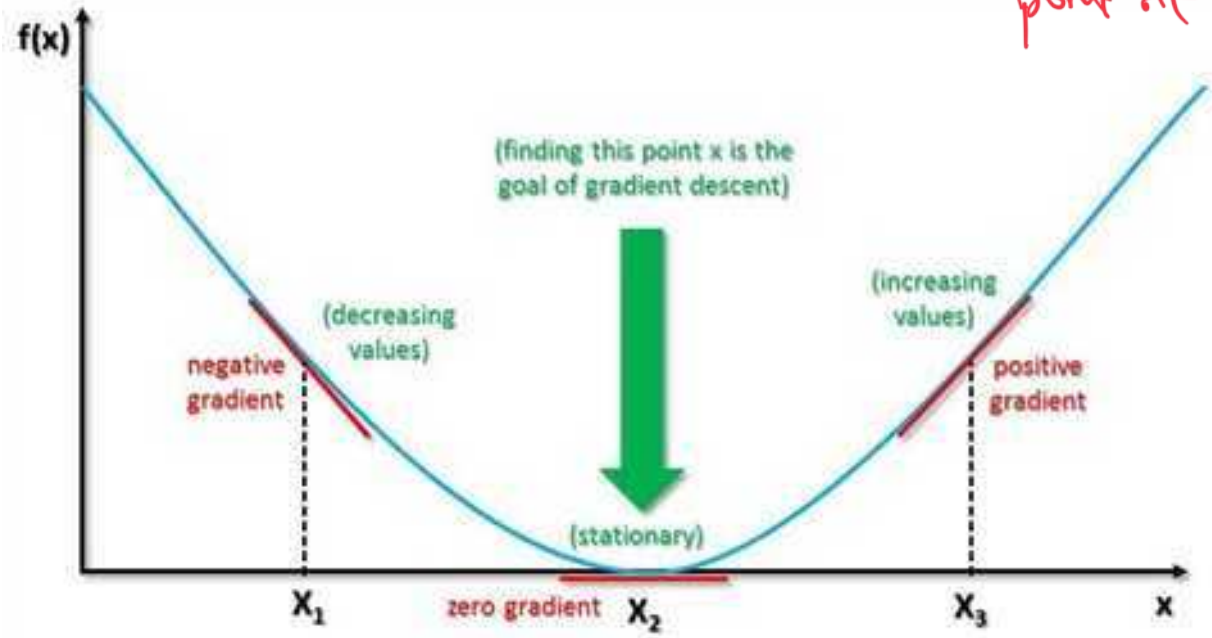


Animation sources: https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

Recap: Derivation and gradient

The **derivative** $f'(\mathbf{x})$ of a function $f(\mathbf{x})$ tells how fast the function increases or decreases.

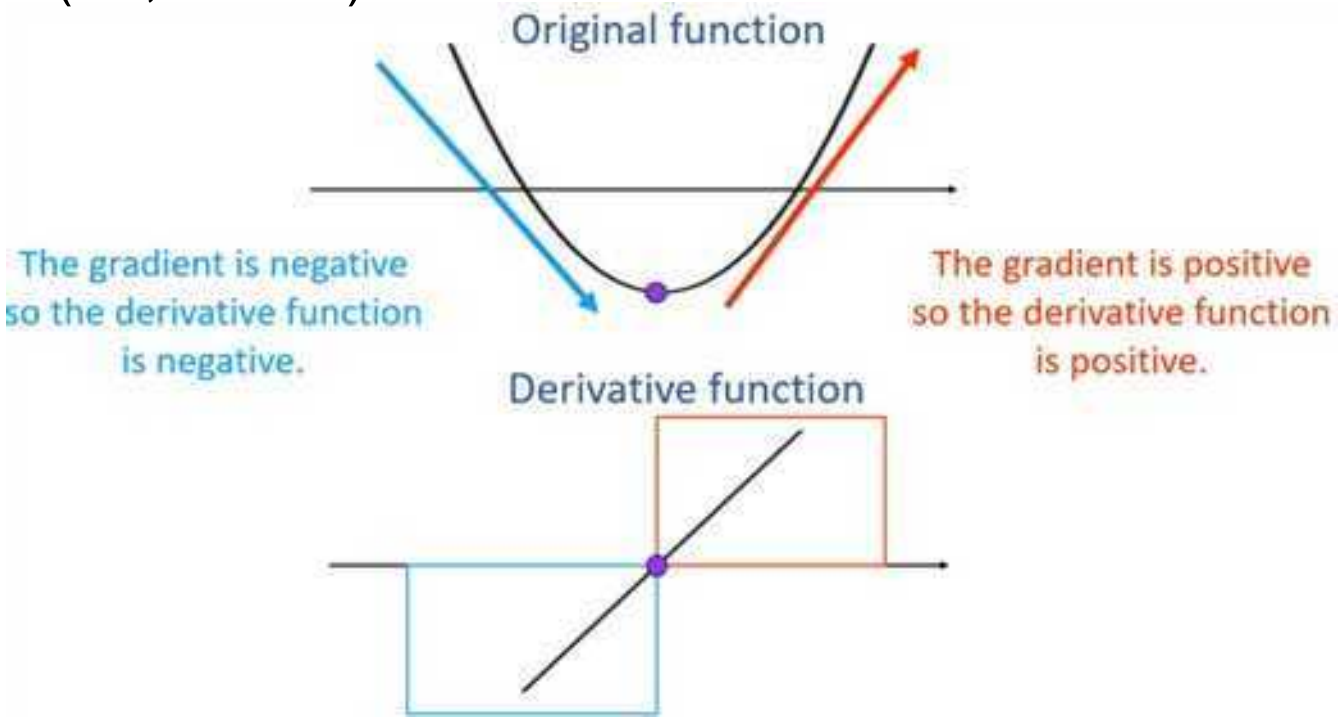
- If $f'(\mathbf{x})$ is constant, the function grows at a *steady* rate.
- If $f'(\mathbf{x}) > 0$, the function is *increasing* at $\mathbf{x}$.
- If $f'(\mathbf{x}) < 0$, the function is *decreasing* at $\mathbf{x}$.
- If $f'(\mathbf{x}) = 0$, the slope is *flat* — possibly a maximum or minimum.



Reference: http://www.big-data.tips/gradient-descent

Recap: Derivation and gradient

- The **derivative** $f'(\mathbf{x})$ of a function $f(\mathbf{x})$ tells how fast the function increases or decreases.
- The process of finding a derivative is called **differentiation.**
- The **gradient** is a generalization of the derivative for functions with **multiple inputs** (*i.e.*, vectors).



Reference: https://mathsathome.com/sketching-the-derivative/

Recap: Derivation and gradient

- The **gradient** of a function is a vector of **partial derivatives**.

If $f(\mathbf{x})$ is a scalar function (*i.e.*, $f: \mathbb{R}^n \to \mathbb{R}$) of n variables, and $\mathbf{x}$ is a $n \times 1$ vector, then differentiating $f(\mathbf{x})$ with respect to $\mathbf{x}$ yields a $n \times 1$ vector:

$$\frac{df(\mathbf{x})}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

This is called the **gradient** of $f(\mathbf{x})$ and is often denoted by $\nabla_x f$

$$\nabla_{\mathbf{x}} f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 3 \end{bmatrix}$$

➢ Example:

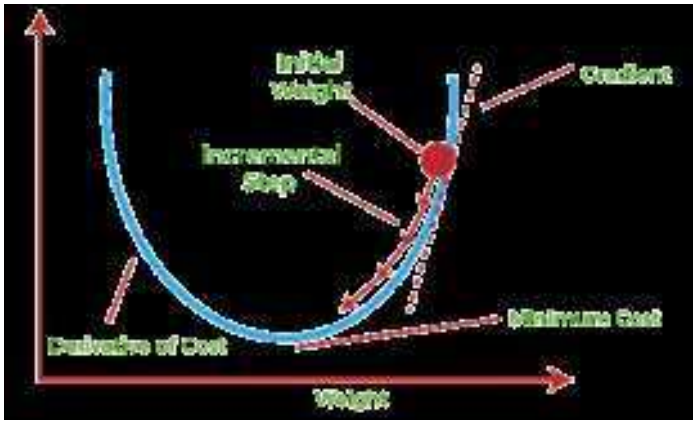
Let $f(\mathbf{x}) = {x_1}^2 + 3x_2$ Then, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $\nabla_{\mathbf{x}} f = \begin{bmatrix} 2x_1 \\ 3 \end{bmatrix}$

Reference: https://mathsathome.com/sketching-the-derivative/

Gradient descent algorithm

Suppose we want to minimize $f(\mathbf{x})$ with respect to $\mathbf{x} = [x_1, x_2, \cdots, x_n]^T$

Gradient $\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$



$\nabla_{\mathbf{x}} f(\mathbf{x})$ is a vector and function of $\mathbf{x}$

is a vector

$\nabla_{\mathbf{x}} f(\mathbf{x})$ points in the direction where f **increases most rapidly**.

So $-\nabla_{\mathbf{x}} f(\mathbf{x})$ is the direction of steepest descent

↳ update the vector

Gradient descent algorithm

Gradient Descent:

Initialize $\mathbf{x}_0$ and learning rate η

while true **do**

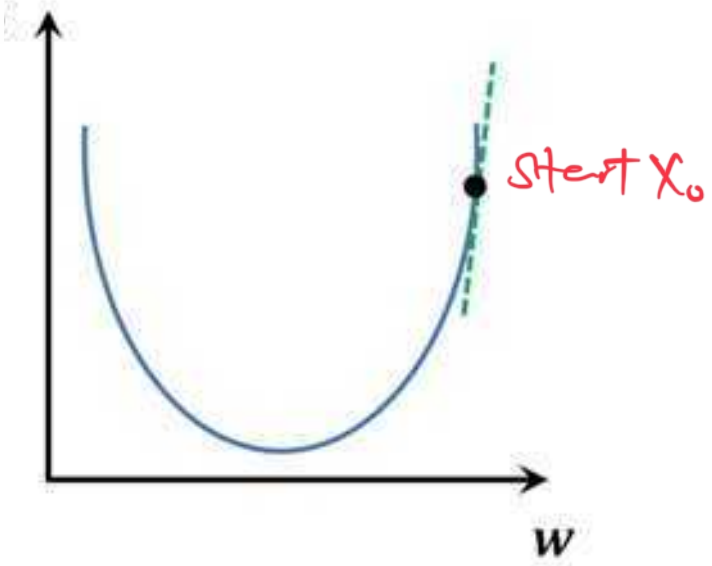
  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$

  if converge **then**

    return $\mathbf{x}_{k+1}$

  end

end



Animation sources: https://datahacker.rs/003-pytorch-how-to-implement-linear-regression-in-pytorch/

Gradient descent algorithm

Key points:

- $-\eta \nabla_x f(\mathbf{x}_k)$: The product of the *learning rate* and the *gradient*, which determines the ***direction*** and ***step size*** to move towards minimizing the cost.

- Common stopping criteria:
  - Maximum number of iterations is reached.
  - The percentage or absolute change in f is below a threshold.
  - The percentage or absolute change in the variable $\mathbf{x}$ is below a threshold.

e.g. good convergence.

Gradient Descent:

Initialize $\mathbf{x}_0$ and learning rate η

while true **do**

  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$

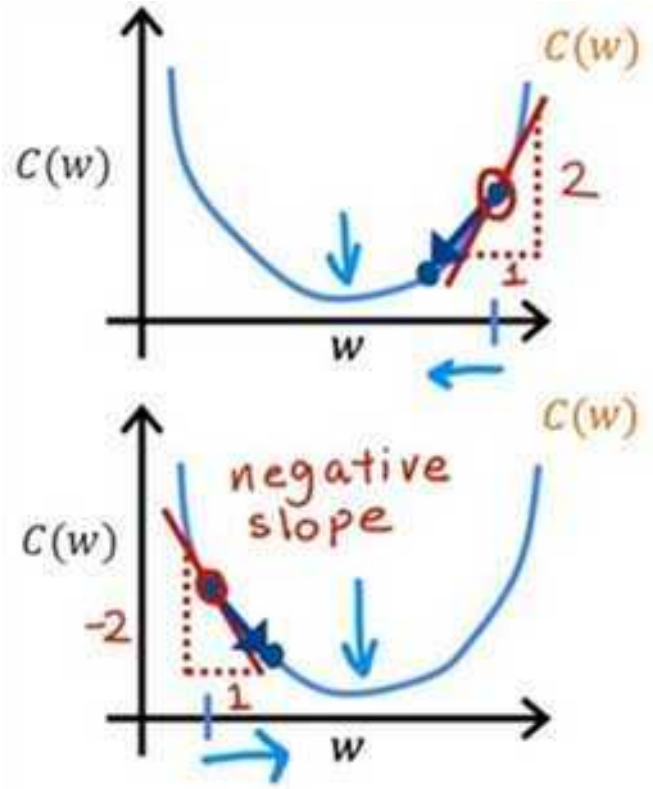
  if converge **then**

    return $\mathbf{x}_{k+1}$

  end

end

Gradient descent algorithm



Gradient Descent:
Initialize $\mathbf{x}_0$ and learning rate η
while true **do**
  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$
  if converge **then**
    return $\mathbf{x}_{k+1}$
  end
end

+ve or -ve

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta \nabla_{\mathbf{w}} C(\mathbf{w}_k)$$

η is positive

$$w = w - \eta \cdot (\text{positive number})$$

w_{i+1} $\Rightarrow$ w will go down (left)

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta \nabla_{\mathbf{w}} C(\mathbf{w}_k)$$

$$w = w - \eta \cdot (\text{negative number})$$

w_{i+1} $\Rightarrow$ w will go up (right)

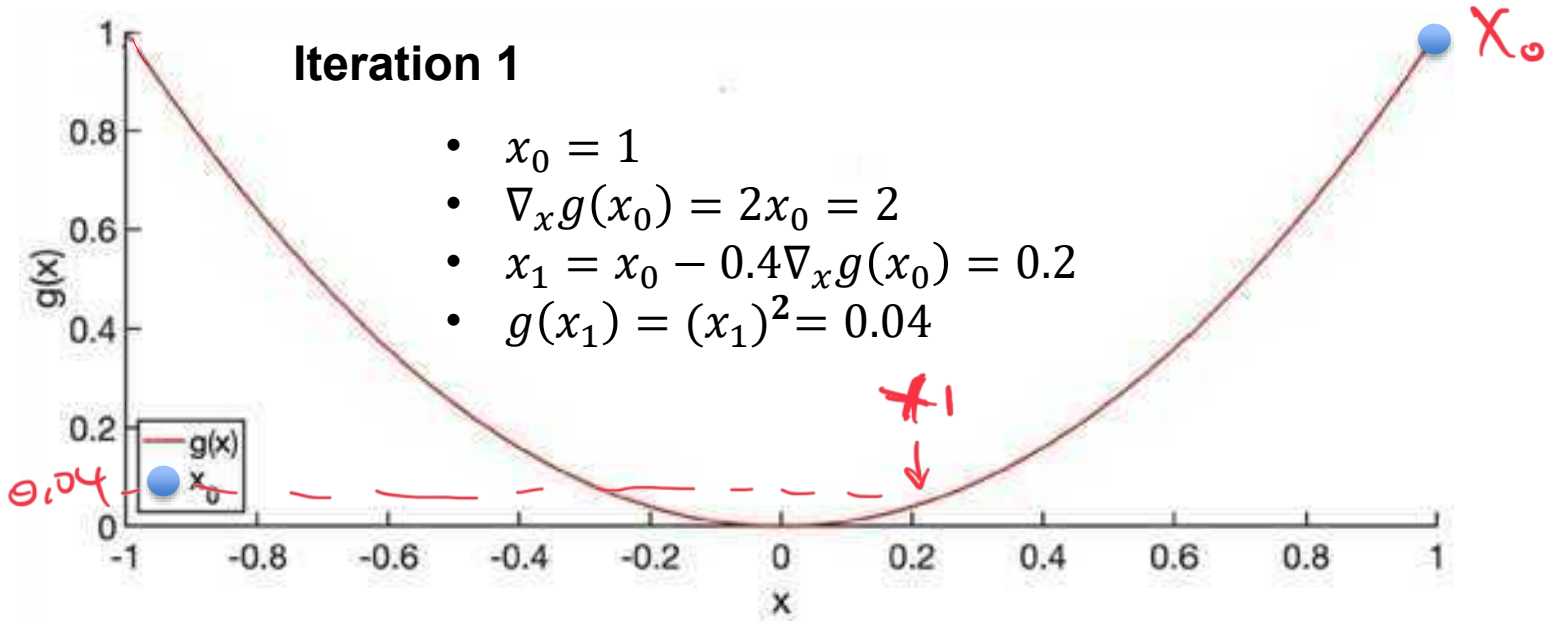
Modified from https://www.youtube.com/watch?v=PKm61nrqpCA by Dr Andrew Ng

Gradient descent (example)

Qn: Find x to minimize $g(x) = x^2$.
Initialize $x_0 = 1$, learning rate $\eta = 0.4$

Gradient Descent:
Initialize $\mathbf{x}_0$ and learning rate η
while true **do**
  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$
  if converge **then**
    return $\mathbf{x}_{k+1}$
  end
end

1. Compute the gradient: $\nabla_x g(x) = 2x$
2. At each iteration, $x_{k+1} \leftarrow x_k - \eta \nabla_x g(x_k)$

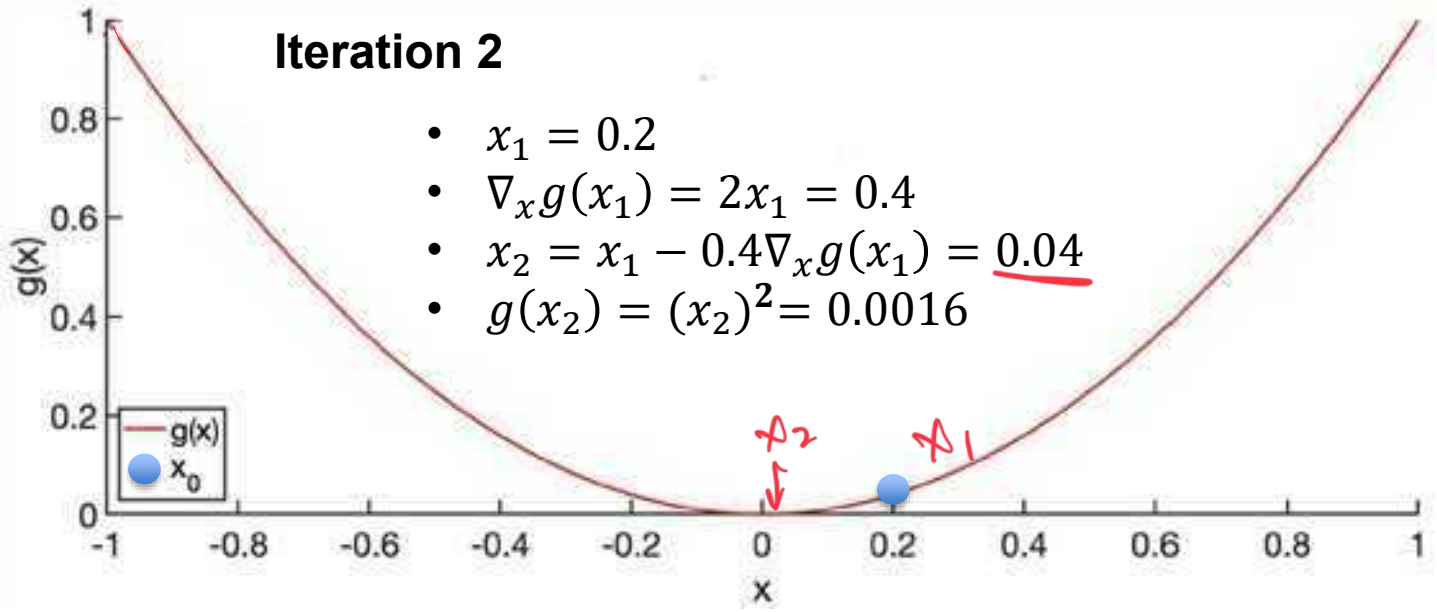


Gradient descent (example)

Qn: Find x to minimize $g(x) = x^2$.
Initialize $x_0 = 1$, learning rate $\eta = 0.4$

1. Compute the gradient: $\nabla_x g(x) = 2x$
2. At each iteration, $x_{k+1} \leftarrow x_k - \eta \nabla_x g(x_k)$

Gradient Descent:
Initialize $\mathbf{x}_0$ and learning rate η
while true **do**
  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$
  if converge **then**
    return $\mathbf{x}_{k+1}$
  end
end

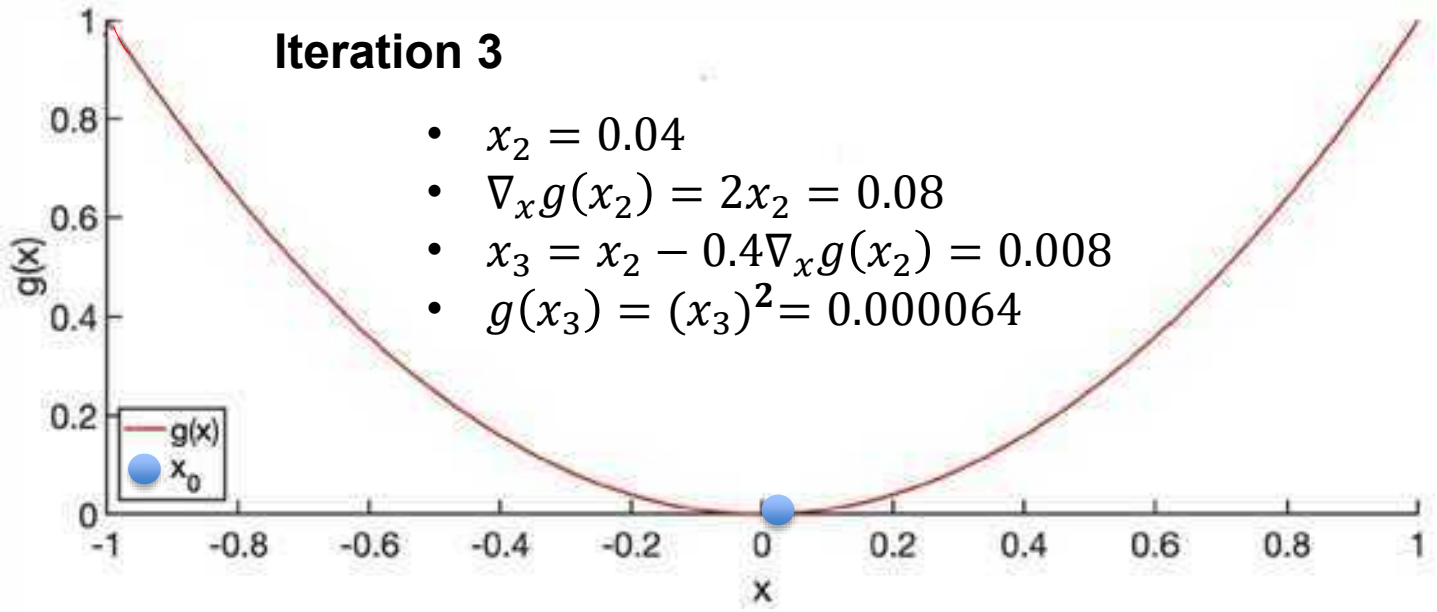


Gradient descent (example)

Qn: Find x to minimize $g(x) = x^2$.
Initialize $x_0 = 1$, learning rate $\eta = 0.4$

1. Compute the gradient: $\nabla_x g(x) = 2x$
2. At each iteration, $x_{k+1} \leftarrow x_k - \eta \nabla_x g(x_k)$

Gradient Descent:
Initialize $\mathbf{x}_0$ and learning rate η
while true **do**
  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$
  if converge **then**
    return $\mathbf{x}_{k+1}$
  end
end

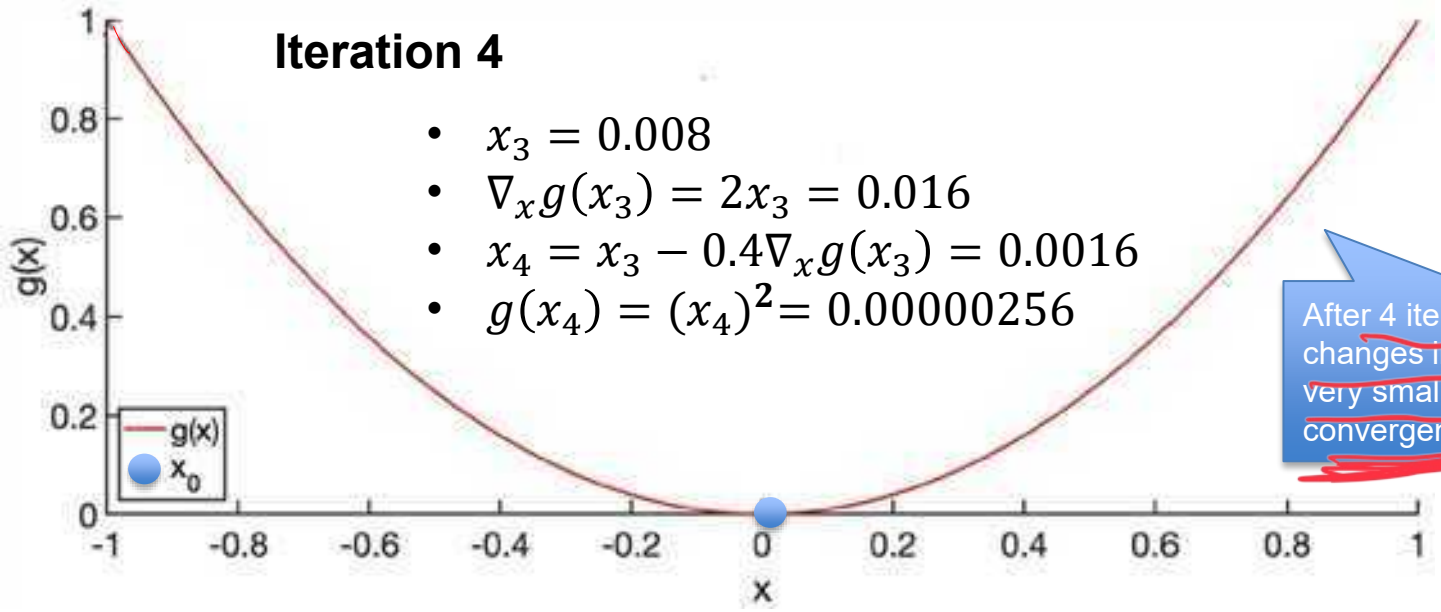


Gradient descent (example)

Qn: Find x to minimize $g(x) = x^2$.
Initialize $x_0 = 1$, learning rate $\eta = 0.4$

1. Compute the gradient: $\nabla_x g(x) = 2x$
2. At each iteration, $x_{k+1} \leftarrow x_k - \eta \nabla_x g(x_k)$

Gradient Descent:
Initialize $\mathbf{x}_0$ and learning rate η
while true **do**
  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$
  if converge **then**
    return $\mathbf{x}_{k+1}$
  end
end



Reference (similar animation): https://www.youtube.com/watch?v=huyBoeTkm8I

Gradient descent algorithm overview

Gradient descent takes **larger steps** when the slope is **steep**, allowing for faster progress in the initial phases of optimization.

As the slope **flattens**, gradient descent takes **smaller steps**, enabling precise convergence towards the minimum ("Goal") without overshooting.

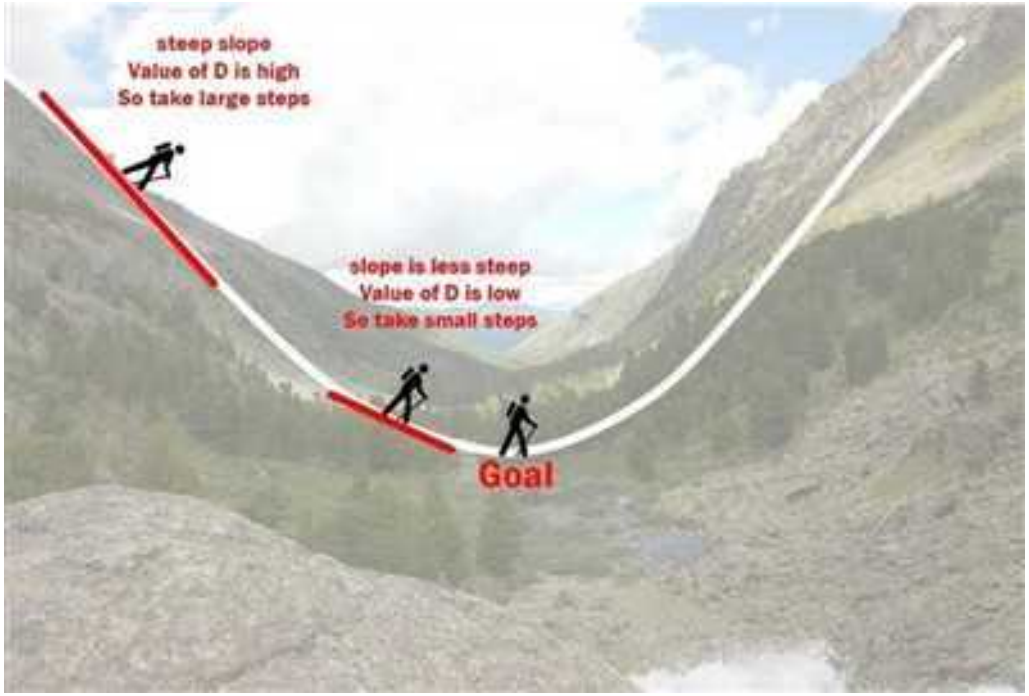


Image source: https://www.inspiritai.com/

Class practice 1: Gradient descent

Qn: Minimize $f(x) = (x_1 - 1)^2 + (x_2 - 2)^2$

with respect to $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

Initialize: $x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Learning rate: $\eta = 0.1$

Gradient $\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$

Gradient Descent:
Initialize $\mathbf{x}_0$ and learning rate η
while true **do**
  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$
  if converge **then**
    return $\mathbf{x}_{k+1}$
  end
end

Class practice 1: Gradient descent

Qn: Minimize $f(x) = (x_1 - 1)^2 + (x_2 - 2)^2$

Initialize: $x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ Learning rate: $\eta = 0.1$

Gradient Descent:

Initialize $\mathbf{x}_0$ and learning rate η

while true **do**

  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$

  if converge **then**

    return $\mathbf{x}_{k+1}$

  end

end

Step 1: Compute gradient

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2(x_1 - 1) \\ 2(x_2 - 2) \end{bmatrix}$$

rmb wrt to x_1, then wrt to x_2

Step 2: **First iteration**:

$$\nabla_{\mathbf{x}} f(\mathbf{x}_0) = \begin{bmatrix} 2(0 - 1) \\ 2(0 - 2) \end{bmatrix} = \begin{bmatrix} -2 \\ -4 \end{bmatrix}$$

Update:

$$\mathbf{x}_1 = \mathbf{x}_0 - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - 0.1 \begin{bmatrix} -2 \\ -4 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}$$

Class practice 1: Gradient descent

Qn: Minimize $f(x) = (x_1 - 1)^2 + (x_2 - 2)^2$

Initialize: $x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ Learning rate: $\eta = 0.1$

Gradient Descent:

Initialize $\mathbf{x}_0$ and learning rate η

while true **do**

  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$

  if converge **then**

    return $\mathbf{x}_{k+1}$

  end

end

Step 1: Compute gradient

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2(x_1 - 1) \\ 2(x_2 - 2) \end{bmatrix}$$

Step 3: **Second iteration:**

Current position: $\mathbf{x}_1 = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}$

Gradient at $\mathbf{x}_1$:

$$\nabla_{\mathbf{x}} f(\mathbf{x}_1) = \begin{bmatrix} 2(0.2 - 1) \\ 2(0.4 - 2) \end{bmatrix} = \begin{bmatrix} -1.6 \\ -3.2 \end{bmatrix}$$

Update:

$$\mathbf{x}_2 = \mathbf{x}_1 - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_1) = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix} - 0.1 \begin{bmatrix} -1.6 \\ -3.2 \end{bmatrix} = \begin{bmatrix} 0.36 \\ 0.72 \end{bmatrix}$$

Class practice 1: Gradient descent

Qn: Minimize $f(x) = (x_1 - 1)^2 + (x_2 - 2)^2$

Initialize: $x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ Learning rate: $\eta = 0.1$

Gradient Descent:

Initialize $\mathbf{x}_0$ and learning rate η

while true **do**

  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$

  if converge **then**

    return $\mathbf{x}_{k+1}$

  end

end

Step 1: Compute gradient

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2(x_1 - 1) \\ 2(x_2 - 2) \end{bmatrix}$$

Step 4: **Third iteration:**

Current position: $\mathbf{x}_2 = \begin{bmatrix} 0.36 \\ 0.72 \end{bmatrix}$

Gradient at $\mathbf{x}_2$:

$$\nabla_{\mathbf{x}} f(\mathbf{x}_2) = \begin{bmatrix} 2(0.36 - 1) \\ 2(0.72 - 2) \end{bmatrix} = \begin{bmatrix} -1.28 \\ -2.56 \end{bmatrix}$$

Update:

$$\mathbf{x}_3 = \mathbf{x}_2 - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_2) = \begin{bmatrix} 0.36 \\ 0.72 \end{bmatrix} - 0.1 \begin{bmatrix} -1.28 \\ -2.56 \end{bmatrix} = \begin{bmatrix} 0.49 \\ 0.98 \end{bmatrix}$$

After 3 iterations, we are at (0.49, 0.98), continuously approaching the optimal solution at (1, 2).

Visualization in Python

Class practice 2: Gradient descent

Qn: Suppose we are minimizing $g(x) = x^2$. We initialize x to be 1. Using gradient descent with learning rate $\eta = 1$, perform the first few steps of the algorithm and observe what happens.

Does the value of x get closer to the minimum, or does it keep bouncing back and forth? Why?

large η will not go to minimum. can cause overshoot

Gradient Descent:

Initialize $\mathbf{x}_0$ and learning rate η

while true **do**

  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$

  if converge **then**

    return $\mathbf{x}_{k+1}$

  end

end

Class practice: Gradient descent

Qn: Minimizing $g(x) = x^2$. Initialize $x_0 = 1$. Learning rate $\eta = 1$

1. Compute the gradient: $\nabla_x g(x) = 2x$
2. At each iteration, $x_{k+1} \leftarrow x_k - \eta \nabla_x g(x_k)$

Iteration t	x_t	$\nabla_x g(x_t)$	$g(x_t)$
0	1	2	1
1	-1	-2	1
2	1	2	1
3	-1	-2	1
…	…	…	…

Large learning rates can cause gradient descent to overshoot and diverge.
Choose the learning rate carefully for stable convergence

Gradient descent algorithm

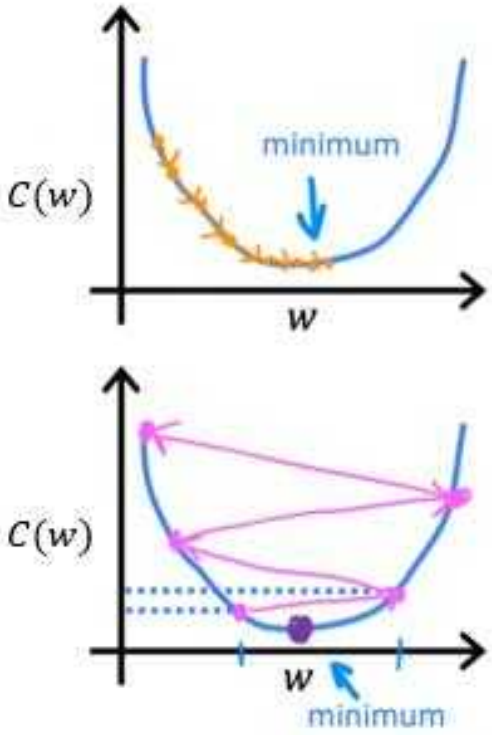
Choose suitable learning rate:

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta \nabla_{\mathbf{w}} C(\mathbf{w}_k)$$

If η is too small...
Gradient descent may be slow.

If η is too large...

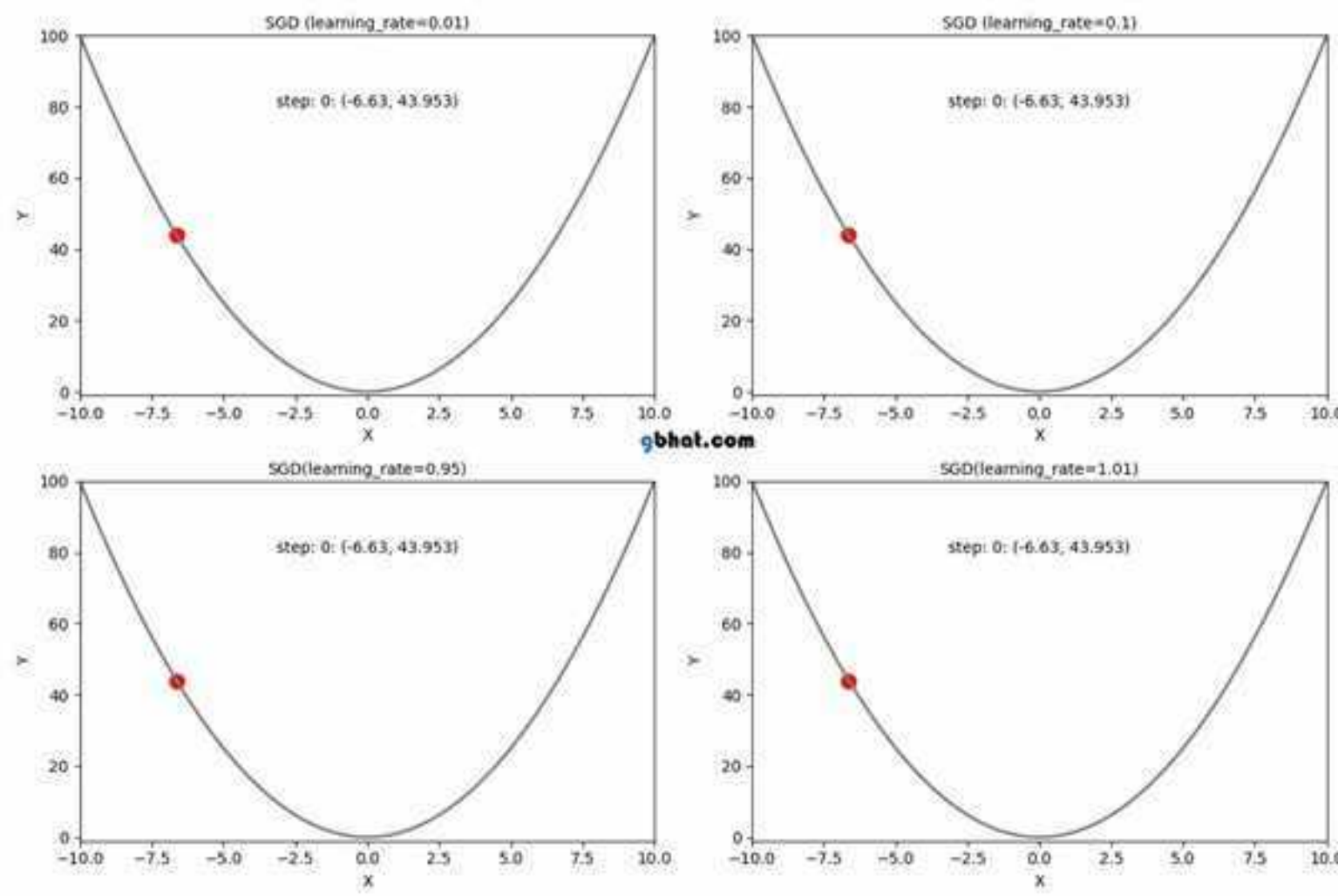
Gradient descent may:
- Overshoot, never reach minimum
- Fail to converge, diverge



Source: https://www.youtube.com/watch?v=k0h8emRAAHE

Gradient descent algorithm

Different learning rates comparison



Source: https://gbhat.com/machine_learning/gradient_descent_learning_rates.html

Convex optimization: How to solve

- **Unconstrained** and **differentiable** convex functions:
  Gradient descent converges to the **global minimum** with a sufficiently small step size (i.e., learning rate η).

- **Constrained** and **differentiable** convex functions:
  Projected gradient descent is used when the domain is constrained to a convex set.

- **Non-differentiable** convex functions (*not covered in this course*):
  ϵ-subgradient methods can be applied

* In non-convex functions, gradient descent might get stuck in local minima but sometimes finds good solutions in practice.

Projected gradient descent

Projected Gradient Descent:

Initialize $\mathbf{x}_0$ and learning rate η

while true **do**

  Compute:

    $\mathbf{z}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$

    $\mathbf{x}_{k+1} \leftarrow \Pi_C(\mathbf{z}_{k+1})$ // *Project* $\mathbf{z}_{k+1}$ *onto constraint set*

  if converge **then**

    return $\mathbf{x}_{k+1}$

  end

end

$\Pi_C(\mathrm{z})$: the projection of a point z onto a convex set C

e.g., $\Pi_C(\mathbf{z}) = \underset{\mathbf{x} \in C}{\operatorname{argmin}} \|\mathbf{x} - \mathbf{z}\|_2$

Take the point z (which might be outside the constraint set C), and find the *closest point* in C to it — using *Euclidean distance* (L2 norm), denoted by $\|\bullet\|$.

Projected gradient descent: Example

Qn: Minimize $g(x) = (x-2)^2$.

Subject to: $x \geq 0$ (feasible region $C = [0, +\infty)$)

Initialize $x_0 = -1$, learning rate $\eta = 0.15$

1. Compute the gradient: $\nabla_x g(x) = 2x - 4$
2. At each iteration: $x_{k+1} \leftarrow x_k - \eta \nabla_x g(x_k)$ // gradient step (tentative)

   $\mathbf{x}_{k+1} \leftarrow \Pi_C(\mathbf{z}_{k+1})$ // projection step

Iteration 1

Gradient step (tentative):

- $x_0 = -1$
- $\nabla_x g(x_0) = 2x_0 - 4 = -6$
- $z_1 = x_0 - 0.15\nabla_x g(x_0) = -0.1$

Projection step:

Since $z_1 = -0.1 < 0$, we project it to its closet point in feasible region C

$x_1 \leftarrow \Pi_C(-0.1) = 0$

Projected gradient descent: Example

Qn: Minimize $g(x) = (x-2)^2$.

Subject to: $x \geq 0$ (feasible region $C = [0, +\infty)$)

Initialize $x_0 = -1$, learning rate $\eta = 0.15$

1. Compute the gradient: $\nabla_x g(x) = 2x - 4$
2. At each iteration: $x_{k+1} \leftarrow x_k - \eta \nabla_x g(x_k)$ // gradient step (tentative)

   $\mathbf{x}_{k+1} \leftarrow \Pi_C(\mathbf{z}_{k+1})$ // projection step

Iteration 2

Gradient step (tentative):

- $x_1 = 0$
- $\nabla_x g(x_1) = 2x_1 - 4 = -4$
- $z_2 = x_1 - 0.15\nabla_x g(x_1) = 0.6$

Projection step:

Since $z_2 = 0.6 \geq 0$, its projection is itself

$x_2 \leftarrow \Pi_C(0.6) = 0.6$

Projected gradient descent: Example

Qn: Minimize $g(x) = (x-2)^2$.
Subject to: $x \geq 0$ (feasible region $C = [0, +\infty)$)
Initialize $x_0 = -1$, learning rate $\eta = 0.15$

1. Compute the gradient: $\nabla_x g(x) = 2x - 4$
2. At each iteration: $x_{k+1} \leftarrow x_k - \eta \nabla_x g(x_k)$ // gradient step (tentative)
$\mathbf{x}_{k+1} \leftarrow \Pi_C(\mathbf{z}_{k+1})$ // projection step

Iteration 3

Gradient step (tentative):

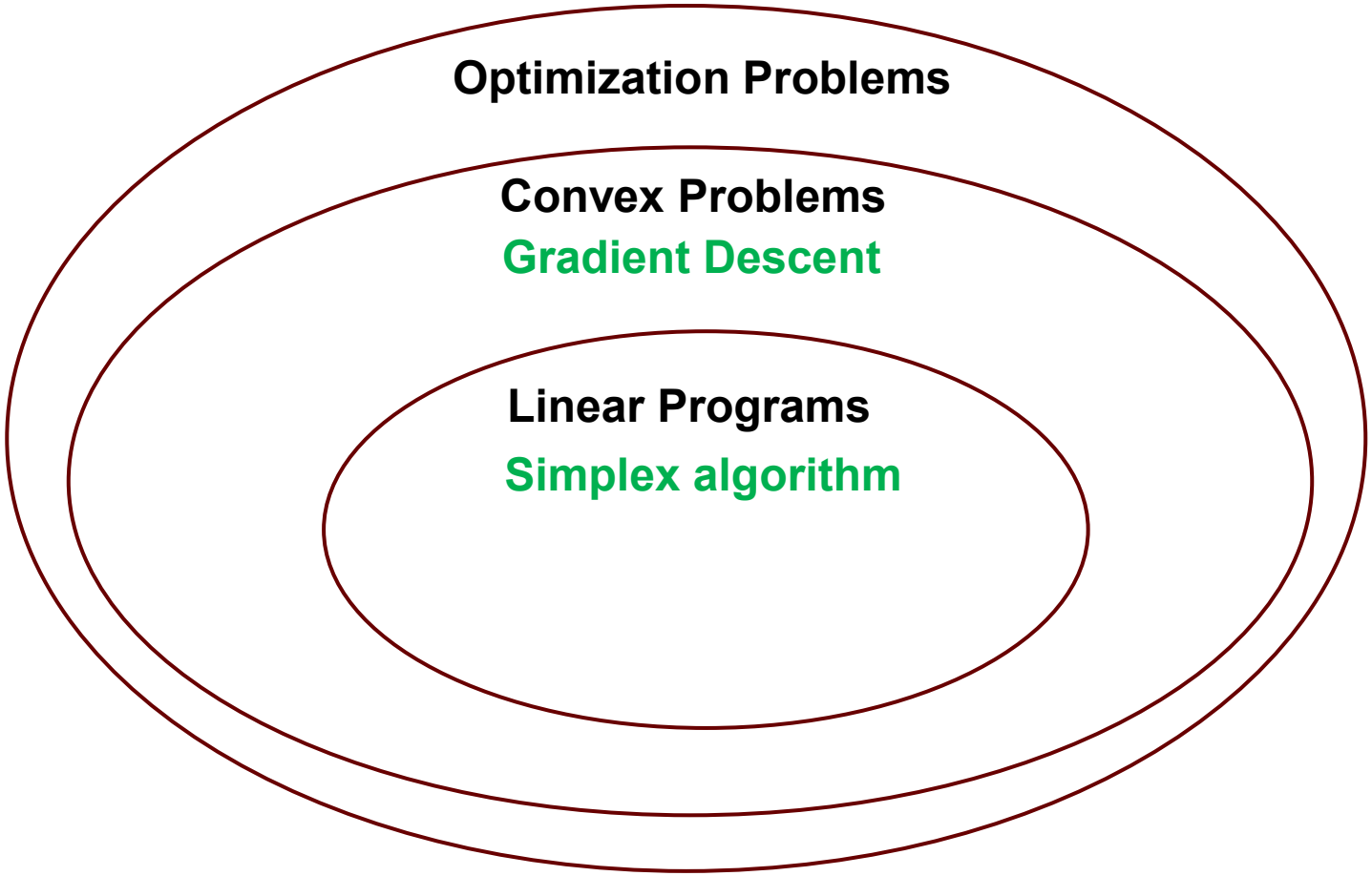
- $x_2 = 0.6$
- $\nabla_x g(x_2) = 2x_2 - 4 = -2.8$
- $z_3 = x_2 - 0.15\nabla_x g(x_2) = 1.02$

Projection step:

Since $z_3 = 1.02 \geq 0$, its projection is itself
$x_3 \leftarrow \Pi_C(1.02) = 1.02$

After 3 iterations, we are at 1.02, continuously approaching the optimal solution at x=2.

Summary



.\venv\Scripts\activate

EE2213 Introduction to Artificial Intelligence

Tutorial 4 (Lecture 5)

Dr. Shaojing Fan
fanshaojing@nus.edu.sg

Q3 (Production resource allocation problem)

A company wants to maximize its monthly profit by deciding how many units of two products to produce each month.

Products: Product A and Product B
Profit per unit: $40 for Product A, $30 for Product B
Constraints:

- Product A requires 2 hours of machine time per unit.
- Product B requires 1 hour of machine time per unit
- The company has at most 100 hours of machine time available each month.
- To fulfill a contract, the company must produce at least 10 units of Product B each month.
- The company has limited storage space and shipping capacity, allowing it to handle only 40 finished units per month.

(i) Formulate the problem mathematically
(ii) Solve using graphical method (show feasible region and optimal point)
(iii) Implement in Python using CVXPY or PuLP library

Q3 (i) Problem formulating

x_A: number of units of Product A produced,
x_B: number of units of Product B produced

Maximize $\mathrm{z} = 40x_A + 30x_B$ (total profit)

Subject to:
$2x_A + x_B \leq 100$ (machine time limit)
$x_B \geq 10$ (minimum requirement for Product B)
$x_A + x_B \leq 40$ (production capacity)
$x_A, x_B \geq 0$ (non-negativity)

Q3 (ii) Use graphical method to solve the problem

Maximize $z = 40x_A + 30x_B$

Subject to: $2x_A + x_B \leq 100$

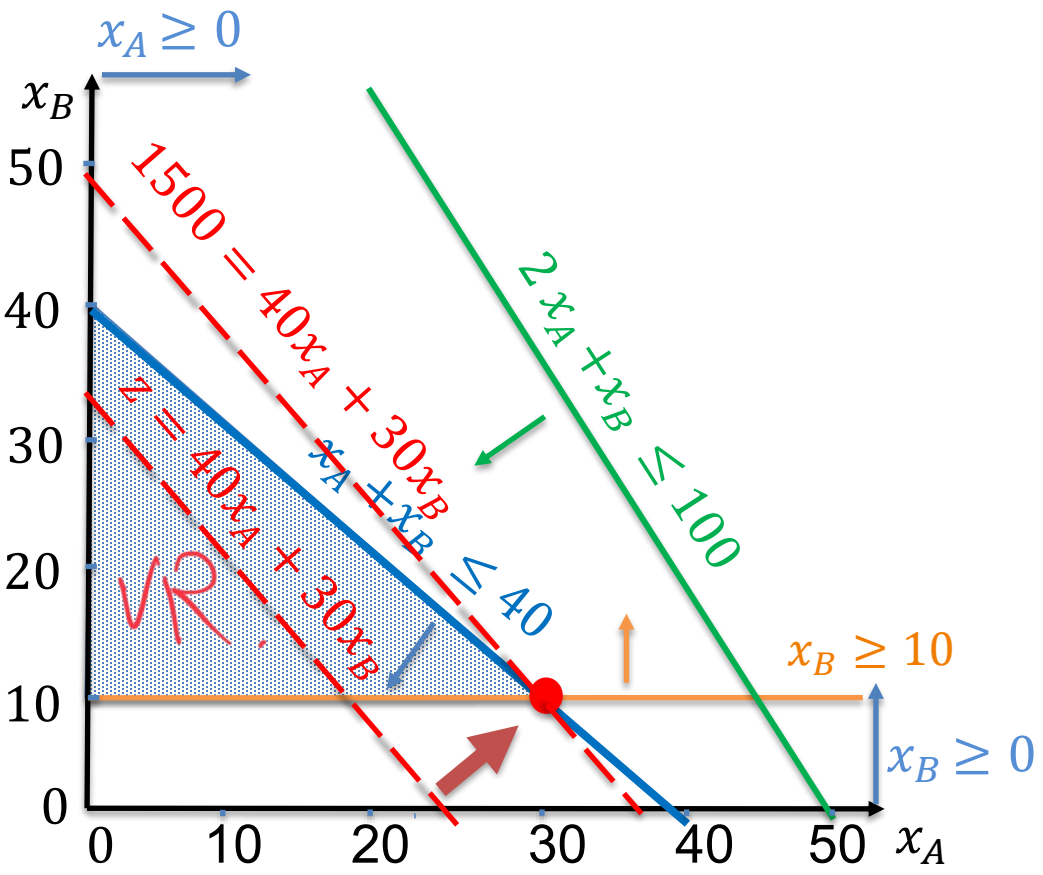
$x_B \geq 10$

$x_A + x_B \leq 40$

$x_A, x_B \geq 0$

vertex (x_A, x_B)	$z = 40x_A + 30x_B$
(0, 10)	300
(0, 40)	1200
(30,10)	1500

After evaluating all vertices, the maximum profit: 1500 ($x_A = 30, x_B = 10$)



Optimal solutions occur on the boundary of the feasible region — typically at its vertices (or along an edge between vertices).

Resources: https://www.desmos.com/calculator

Q3 (iii) Solving the problem in Python using a library like CVXPY

! pip install cvxpy.

Introduction to CVXPY library:

convex pi

- CVXPY is a Python library* for modeling and solving convex optimization problems, including linear and integer programs.
- It provides a simple and readable way to define decision variables, objective functions, and constraints in a mathematical style.
- It supports a variety of powerful solvers (including simplex-based solvers such as HiGHS) to solve problems under the hood.

not incl. in Anaconda by default

*CVXPY is not included in Anaconda by default — you'll need to install it separately.
Reference: https://www.cvxpy.org/api_reference/cvxpy.html

Problem formulation

x_1, x_2: Decision variables

Maximize $150x_1 + 200x_2$ ← Objective function

subject to:

$$x_1 + x_2 \leq 200$$
$$9x_1 + 3x_2 \leq 1240$$
$$12x_1 + 16x_2 \leq 2660$$
$$x_1, x_2 \geq 0$$

← Constraints

Q3 (iii) Solving the problem in Python using a library like CVXPY


```python
# Import libraries
import matplotlib.pyplot as plt # For plotting
import cvxpy as cp  # For solving linear programs
import numpy as np  # For numerical computations (used for plotting here)

# 1. Define variables
x_A = cp.Variable(name="product_A", integer=True)    # integer variable
x_B = cp.Variable(name="product_B", integer=True)    # integer variable

# 2. Define objective
objective = cp.Maximize(40 * x_A + 30 * x_B)

# 3. Define constraints
constraints = [
    x_A >= 0,              # Non-negativity for A
    x_B >= 0,              # Non-negativity for B
    2 * x_A + 1 * x_B <= 100,   # Machine time limit
    x_B >= 10,                  # Minimum product B
    x_A + x_B <= 40             # Production capacity
]

# 4. Creates the CVXPY optimization problem by combining the objective and the constraints
prob = cp.Problem(objective, constraints)

# 5. Solve the LP problem
prob.solve()

# 6. Results
xA_opt = x_A.value # Get the optimal value of decision variable x_A
xB_opt = x_B.value # Get the optimal value of decision variable x_B
max_profit = prob.value # Get the maximum profit from the objective function
problem_status = prob.status # Get the solution status
```

You'll need to install CVXPY separately

Decision variables

Objective function

Constraints

Solve

Q3 (iii) Solving the problem in Python using a library like CVXPY (cont.)

```python
# Import libraries
import matplotlib.pyplot as plt # For plotting
import cvxpy as cp  # For solving linear programs
import numpy as np  # For numerical computations (used for plotting here)

# 1. Define variables
x_A = cp.Variable(name="product_A", integer=True)
x_B = cp.Variable(name="product_B", integer=True)

# 2. Define objective
objective = cp.Maximize(40 * x_A + 30 * x_B)

# 3. Define constraints
constraints = [
    x_A >= 0,              # Non-negativity for A
    x_B >= 0,              # Non-negativity for B
    2 * x_A + 1 * x_B <= 100,   # Machine time limit
    x_B >= 10,             # Minimum product B
    x_A + x_B <= 40        # Production capacity
]

# 4. Creates the CVXPY optimization proble
prob = cp.Problem(objective, constraints)

# 5. Solve the LP problem
prob.solve()

# 6. Results
xA_opt = x_A.value # Get the optimal value of decisi
xB_opt = x_B.value # Get the optimal value of decisi
max_profit = prob.value # Get the maximum profit fro
problem_status = prob.status # Get the solution stat
```

cp.Variable(name="x", integer=True): Define optimization variables.
-name: (string) a label to the variable (optional, for readability)
-integer: (boolean) if True, the variable is restricted to be integers values; otherwise it is continuous by default.

cp.Maximize(): Define objective, tells CVXPY that we want to **maximize** the expression inside

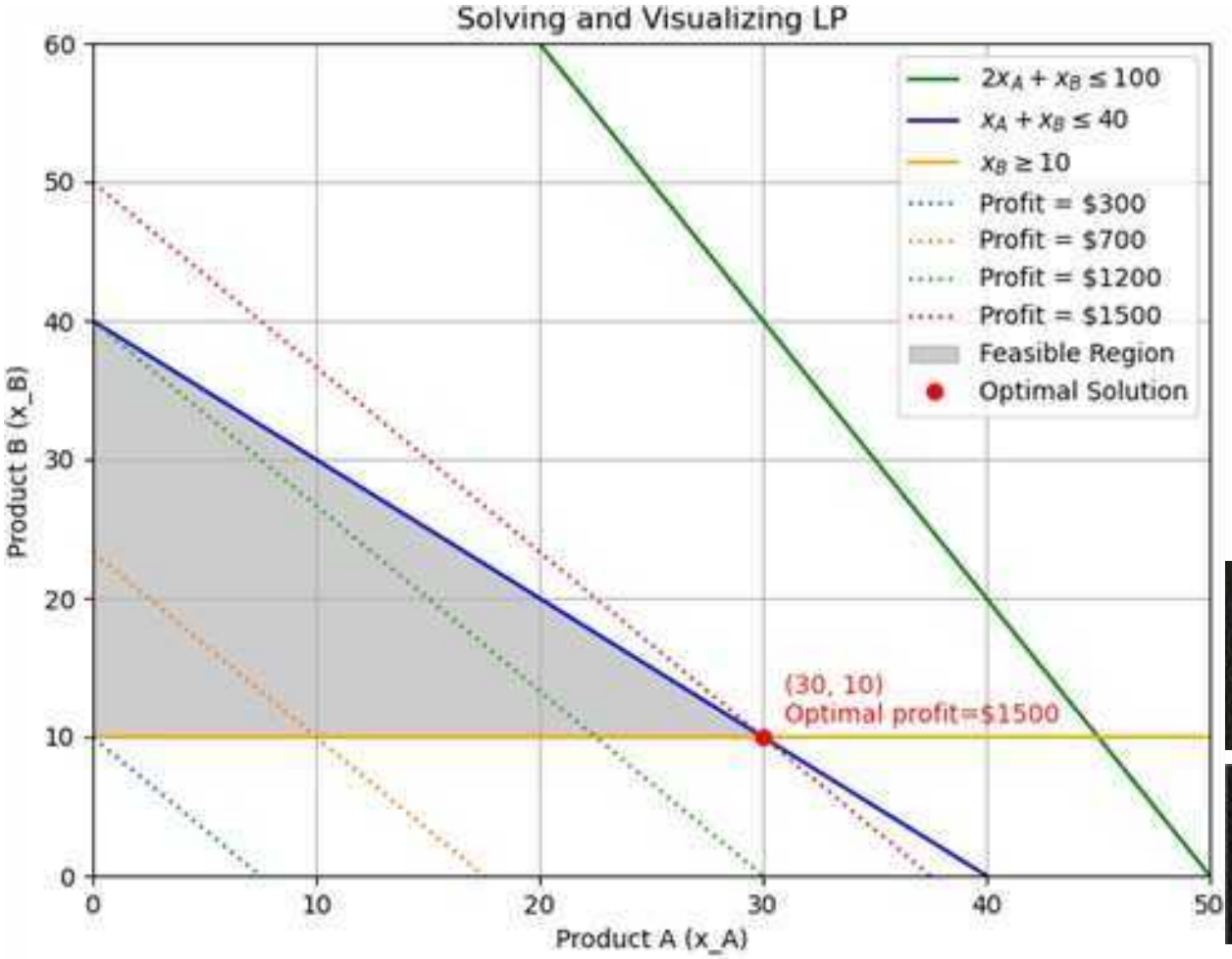
.constraints = []: a list that will hold all the problem's constraints.
-each item in the list is a CVXPY expression representing a constraint.
-internally, CVXPY stores these constraints as **objects** that know the left-hand side (e.g., x_A+x_B), the inequality type (e.g., <=), the right-hand side (e.g., 40)
-when you pass the list to prob.solve(), CVXPY interprets all the objects together as the **feasible region** for the optimization.

cp.Problem(objective, constraints**):** creates a CVXPY **problem object**. It combines the objective and constraints.

.solve(): runs the solver to find the best solution. For linear program, the solver may use the Simplex algorithm.

.value (for variables): gives the values of the decision variables.
.value (for problem): Gives the optimal value of the objective function.
.status: Tells you what happened when solving:
'optimal' → found a solution
'infeasible' → no solution satisfies all constraints
'unbounded' → objective can improve indefinitely

Q3 (iii) Solving the problem in Python using a library like CVXPY (cont.)



Maximize $z = 40x_A + 30x_B$

Subject to: $2x_A + x_B \leq 100$

$x_B \geq 10$

$x_A + x_B \leq 40$

$x_A, x_B \geq 0$

will get some floating point and. e.g. 29.999999... 10.000000... data precision ing.

```
# Output results
print("\nSolution Status:", problem_status)
print("Optimal number of units to produce:")
print("Product A:", xA_opt)
print("Product B:", xB_opt)
print("Maximum Profit: $", max_profit)
```

```
Solution Status: optimal
Optimal number of units to produce:
Product A: 30.0
Product B: 10.0
Maximum Profit: $ 1500.0
```

*Visualization code is included in the shared Python file. While it's not required for exams, visualization is a valuable tool in real-world practice---it helps us better interpret results and understand model performance.

Q3 Code highlights – Key Python libraries


```
import matplotlib.pyplot as plt
from cvxpy as cp
import numpy as np
```

•**matplotlib**: A library for creating visualizations such as charts and plots to help interpret data and results.

•**cvxpy**: A Python library for **linear and convex optimization**, allowing us to define and solve optimization problems.

•**numpy**: A fundamental package for numerical computing in Python, offering support for arrays, math functions, and linear algebra.

By convention, we import libraries with short names for convenience: *numpy* as *np, cvxpy* as *cp.*

Q3 Code highlights – CVXPY library


```
from cvxpy as cp

cp.Variable(name=None, integer=False)
Defines a decision variable in the optimization problem.
    • name: variable identifier (optional, e.g., "x_A").
    • integer: if True, variable is restricted to integer values;
      otherwise it is continuous (default).


Example:
x = cp.Variable(name="product x", integer=True)
creates an integer variable x
```

```python
# 1. Define variables
x_A = cp.Variable(name="product_A", integer=True)    # integer variable
x_B = cp.Variable(name="product_B", integer=True)    # integer variable
```

Reference: CVXPY-tutorial-TUT4.ipynb

Reference: https://www.cvxpy.org/api_reference/cvxpy.html

Q3 Code highlights – CVXPY library (cont.)

`cp.Maximize() / cp.Minimize()`

Specifies the objective function of the optimization problem.
cp.Maximize(expr) → maximize the expression expr.
cp.Minimize(expr) → minimize the expression expr.
For example: objective = cp.Maximize(40 * x_A + 30 * x_B)

`constraints = []`

Defines a list to hold all the problem's constraints.
Each item in the list is a CVXPY expression representing a rule for the variables. Start empty, then add constraints like inequalities or equalities. For example:

```
constraints = [
    x >= 0,        # Non-negativity
    x + y <= 10    # Total limit
]
```

```
# 2. Define objective
objective = cp.Maximize(40 * x_A + 30 * x_B)

# 3. Define constraints
constraints = [
    x_A >= 0,              # Non-negativity for A
    x_B >= 0,              # Non-negativity for B
    2 * x_A + 1 * x_B <= 100,   # Machine time limit
    x_B >= 10,             # Minimum product B
    x_A + x_B <= 40        # Production capacity
]
```

Q3 Code highlights – CVXPY library (cont.)

`cp.Problem(objective, constraints)`
Combines the objective and constraints into a CVXPY problem object.
This object represents the entire optimization problem.
For example:
`prob = cp.Problem(objective, constraints)`

`prob.solve()`
Tells CVXPY to actually solve the optimization problem.
CVXPY passes the problem to an underlying solver (like HiGHS, ECOS, or others). *For linear programs, the solver may use the Simplex algorithm.*
After solving, we get:
Optimal variable values: x.value, y.value, …
Optimal objective value: prob.value
Solution status: prob.status (optimal, infeasable, unbounded)

```
# 4. Creates the CVXPY optimization problem by combining the objective and the constraints
prob = cp.Problem(objective, constraints)

# 5. Solve the LP problem
prob.solve()
```

Reference: You may refer to simplex_algorithm.py for simplex algorithm implementation

Q3 Code highlights – CVXPY library (cont.)

Get results:

.value (for variables)
Gives the values of the decision variables that produce the optimal objective value.

.value (for problem)
Gives the optimal value of the objective function.

.status
Tells you what happened when solving:
'*optimal*' : found a solution
'*infeasible*': no solution satisfies all constraints
'*unbounded*' : objective can improve indefinitely

```
# 6. Results
xA_opt = x_A.value # Get the optimal value of decision variable x_A
xB_opt = x_B.value # Get the optimal value of decision variable x_B
max_profit = prob.value # Get the maximum profit from the objective function
problem_status = prob.status # Get the solution status
```

In CVXPY (and most LP solvers), .status only tells you whether a solution was found. It does not reveal whether the solution is *unique* or if *multiple* optimal solutions exist.

Integer Linear Programming (brief introduction)

Maximize y

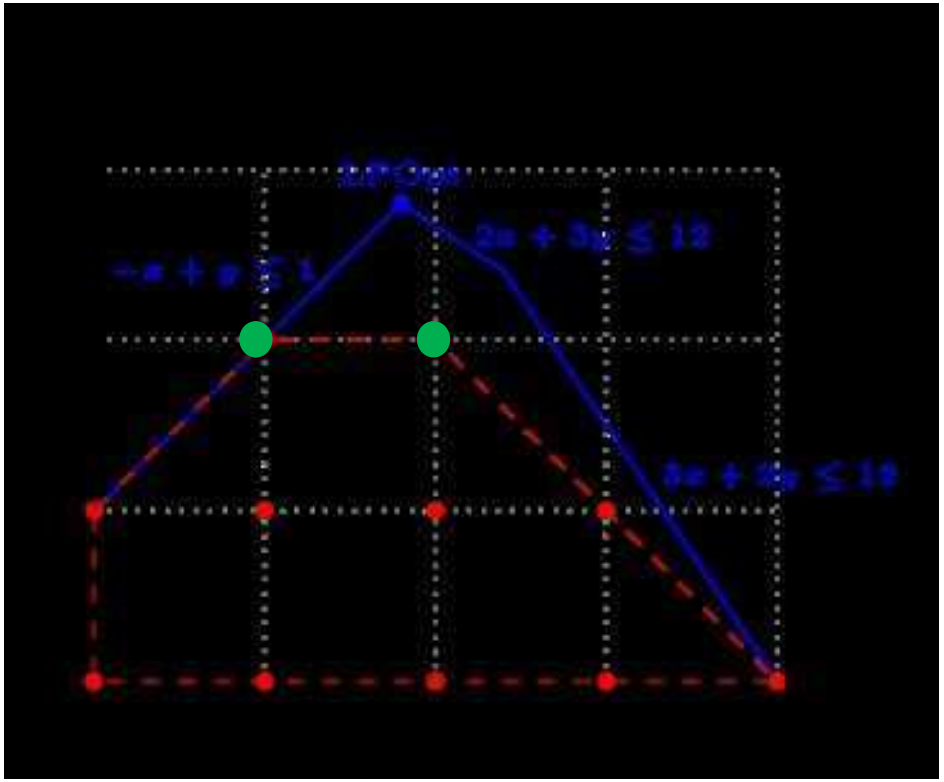
Subject to $-x + y \leq 1$

$3x + 2y \leq 12$

$2x + 3y \leq 12$

$x, y \geq 0$

$x, y \in \mathbb{Z}$



Challenge: The optimal integer linear programming (ILP) solution must be an integer point, which can lie **inside** the LP's feasible region, not just on its boundary.

Solution Strategy: We first solve the easier **LP Relaxation** (ignoring integer constraints) to get a bound, then use algorithms like branch and bound to find the best integer solution.

Reference: https://web.mit.edu/15.053/www/AMP-Chapter-09.pdf

LP vs ILP

Feature	Linear Programming (LP)	Integer Linear Programming (ILP)
Decision Variables	Continuous (any real number)	Integer (whole numbers only)
Feasible Region	A continuous region (includes all points inside and on the boundaries)	Discrete set of points inside the feasible region or on the boundary
Optimal Solution Location	At a vertex (or along an edge) of the feasible region	Can be at a vertex (on the edge) or inside the region
Solution method	Relatively easy — efficient algorithms (like Simplex, Interior Point)	Harder — often requires special methods (Branch & Bound, Cutting Planes)
Practical Examples	Resource allocation with divisible quantities (e.g., fractional production, mixing fuels, maximum flow, diet problem, transportation problem)	Situations requiring whole numbers (e.g., number of buses, yes/no decisions, stock trading with indivisible units)

Q3 (iii) Method 2: Solving the problem in Python using a library like PuLP

Introduction to PuLP library:

- PuLP is a Python library* for modeling and solving linear and integer programming problems.
- It offers a simple, readable way to define decision variables, objective functions, and constraints.
- It solves linear programs using powerful solvers, such as the Simplex-based CBC.

*PuLP is not included in Anaconda by default — you'll need to install it separately.
Reference: https://coin-or.github.io/pulp/guides/index.html

Q3 (iii) Method 2: Solving the problem in Python using a library like PuLP

```python
# Import libraries
import matplotlib.pyplot as plt # For plotting
import numpy as np  # For numerical computations (used for plotting here)
from pulp import *  # PuLP: Python library for modeling linear programming problems

# Define the LP problem
prob = LpProblem("Maximize_Profit", LpMaximize)

# Decision variables
x_A = LpVariable("x_A", lowBound=0)
x_B = LpVariable("x_B", lowBound=0)

# Objective function: Maximize profit
prob += 40 * x_A + 30 * x_B         # Total profit

# Constraints
prob += 2 * x_A + 1 * x_B <= 100    # Machine time
prob += x_B >= 10                   # Minimum product B
prob += x_A + x_B <= 40             # Production capacity

# Solve the LP problem
prob.solve()

# Results
xA_opt = value(x_A) # Get the optimal value of decision variable x_A
xB_opt = value(x_B) # Get the optimal value of decision variable x_B
max_profit = value(prob.objective) # Get the maximum profit from the objective function
```

Decision variables

Objective function

specific way to add expression to prob.

Constraints

Solve

Q3 (iii) Method 2: Solving the problem in Python using a library like PuLP


```
# Import libraries
import matplotlib.pyplot as plt # For plotting
import numpy as np  # For numerical computations (used for plotting here)
from pulp import *  # PuLP: Python library for modeling linear programming problems

# Define the LP problem
prob = LpProblem("Maximize_Profit", LpMaximize)

# Decision variables
x_A = LpVariable("x_A", lowBound=0)
x_B = LpVariable("x_B", lowBound=0)

# Objective function: Maximize profit
prob += 40 * x_A + 30 * x_B         # Total profit

# Constraints
prob += 2 * x_A + 1 * x_B <= 100    # Machine time
prob += x_B >= 10                   # Minimum product B
prob += x_A + x_B <= 40             # Production capacity

# Solve the LP problem
prob.solve()

# Results
xA_opt = value(x_A) # Get the optimal value of decision variable x_A
xB_opt = value(x_B) # Get the optimal value of decision variable x_B
max_profit = value(prob.objective) # Get the maximum profit from the objective function
```

LpProblem(name, sense): creates a new LP problem.
- name: just a label (optional, for readability)
- sense: *LpMaximize* (to maximize) or *LpMinimize* (to minimize)

LpVariable(name, lowBound, upBound, cat): defines a variable for LP
- name: variable name (string)
- lowBound: lower limit (e.g., 0 for non-negativity)
- cat: category of variable (e.g., Continuous, Integer). Default is Continuous.

"**+=**" is used in PuLP to add an expression to the LP model
Here, we are adding the objective function and constraints to 'prob'

.solve(): runs the built-in solver to find the optimal solution (usually using the Simplex algorithm)

.value(variable or expression): Extracts the numerical value from a *solved* variable or objective function.

Q3 Code highlights – PuLP library


```
prob = LpProblem("Maximize_Profit", LpMaximize)
```

Creates a new LP problem named "Maximize_Profit" with the goal to maximize the objective function, and stores it in the variable 'prob'.

```
LpProblem(name, sense)
```

Creates a new linear programming problem.

- `name:` a label for the problem (for reference).
- `sense:` specifies whether to maximize (`LpMaximize`) or minimize (`LpMinimize`) the objective.

Reference: Pulp-tutorial-TUT4.ipynb

For more information, check the PuLP documentation: https://coin-or.github.io/pulp/main/includeme.html

Q3 Code highlights – PuLP library (cont.)


```
x_A = LpVariable("x_A", lowBound=0)
x_B = LpVariable("x_B", lowBound=0)
```

Creates two decision variables named "X_A" and "X_B" for the LP problem, with constraint X_A, X_B ≥ 0 (non-negativity).

```
LpVariable(name, lowBound=None, upBound=None, cat='Continuous')
```

Defines a decision variable in the model.

- name: variable identifier (e.g., "x_A").
- lowBound: minimum value (e.g., 0 for non-negative).
- upBound: maximum value (optional).
- cat: variable type – 'Continuous' (default), 'Integer', or 'Binary'.

Example:

```
x = LpVariable("x", lowBound=0, upBound=10, cat='Integer')
```

creates an integer variable x with 0 ≤ x ≤ 10.

Q3 Code highlights – PuLP library (cont.)

```
prob += 40 * x_A + 30 * x_B
Add the objective function (to maximize) to the LP problem.
```

```
prob += 2 * x_A + 1 * x_B <= 100
prob += x_B >= 10
prob += x_A + x_B <= 40
Add constraints to the LP problem.
```

Context	Meaning of `+=`
Standard Python	Adds and updates the value of a variable. Example: `x += 1` means `x = x + 1`.
PuLP	Adds an expression (objective or constraint) to the LP problem model. Example: `prob += 40 * x_A + 30 * x_B` adds the objective function to the LP.

Q3 Code highlights – PuLP library (cont.)

`prob.solve()`
Solve the LP problem.

`.solve()`
- Runs the solver to find the optimal solution to the defined problem.
- By default, PuLP uses CBC (Coin-or Branch and Cut) as the solver, which is included with PuLP.
- CBC typically applies the *Simplex algorithm* for linear programs.
- This step performs all the necessary computations to maximize or minimize the objective while satisfying the constraints.

Q3 Code highlights – PuLP library (cont.)


```
xA_opt = value(x_A)
xB_opt = value(x_B)
```

Get the optimal values of decision variables x_A, X_B and store them in xA_opt and xB_opt.

```
max_profit = value(prob.objective)
```

Get the maximum profit from the objective function (i.e., the maximum profit).

```
value(variable or expression)
```

- Extracts the numerical value from a *solved* variable or objective function.
- Used to get the *final results*, such as optimal quantities or maximum profit.

Q3 Code highlights – PuLP library (cont.)

`.status`

The `.status` attribute of a PuLP problem tells you the result of the solver, that is, what happened when it tried to solve your linear program.

PuLP represents status codes as constants in pulp.constants, such as:

- LpStatusOptimal
- LpStatusUnbounded
- LpStatusInfeasible

more than convex.

prob.status (numeric)	LpStatus[prob.status]	Meaning	Case	Action needed
1	Optimal	Solution found that meets all constraints	Feasible and finitely optimal	Use the result
-2	Unbounded	Objective can go to $+\infty$ or $-\infty$	Feasible but unbounded	Check for missing constraints
-1	Infeasible	No solution satisfies all constraints	Infeasible	Re-express or relax constraints
0	Other	Solver failed or returned unexpected result	--	Debug model or solver setup
-3	Undefined	Solver failed or returned unexpected result	--	Debug model or solver setup

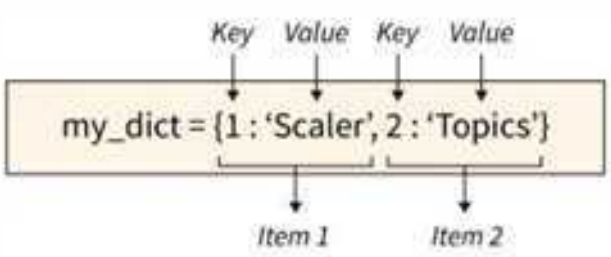
prob is the variable that represents the entire linear programming model in **our** code

In PuLP (and most LP solvers), .status only indicates whether an optimal solution was found, it does not reveal whether the solution is unique or if multiple optimal solutions exist. That is, prob.status == LpStatusOptimal simply confirms that an optimal solution was found, without specifying how many such solutions there may be.

Q3 Code highlights – PuLP library (cont.)

Item	Type	Description
.status	Integer	gives a number that tells what happened when the solver tried to solve your problem. (e.g., 0, 1, -1, -2, or -3)
LpStatus	Dictionary	PuLP's built-in Python dictionary that maps status codes to their meanings
LpStatus[status]	String	turns that number into a readable message like "Optimal" or "Infeasible".

Python dictionary:




```
If my_dict = {1: 'Topics', 2: 'Join', 'key3': 'Scaler'}:
Then my_dict[2] == 'Join'
And my_dict['key3'] == 'Scaler'
```

```
LpStatus = {
    -3: "Undefined",      # Solver could not determine the problem status
    -2: "Unbounded",      # Objective value can increase without limit
    -1: "Infeasible",     # No solution satisfies all constraints
     0: "Not Solved",     # Problem was defined but not solved yet
     1: "Optimal"         # Solver found the best solution under given constraints
}
```

Reference: https://www.w3schools.com/python/python_dictionaries.asp

Q3 PuLP vs CVXPY library

Feature	PuLP	CVXPY
Problem Focus	Linear and integer LP/ILP	General convex optimization (LP, QP, SDP, etc.)
Integer/Binary Variables	Directly supported	Supported, but sometimes may need specific solvers
Syntax Style	Problem-focused: LpProblem, LpVariable, += for objective and constraints	Mathematical-style: cp.Problem(cp.Maximize(...), [constraints])
Solver Integration	Built-in (CBC), GLPK, CPLEX, Gurobi	Many solvers; sometimes requires explicit selection

Q4 The given linear program is:

Minimize $z = 2x_1 + x_2$

subject to:

$$x_1 + x_2 \geq 5$$
$$x_1 + 2x_2 \geq 6$$
$$x_1, x_2 \geq 0$$

A. Bounded and feasible

B. Feasible but not bounded

C. Bounded but not feasible

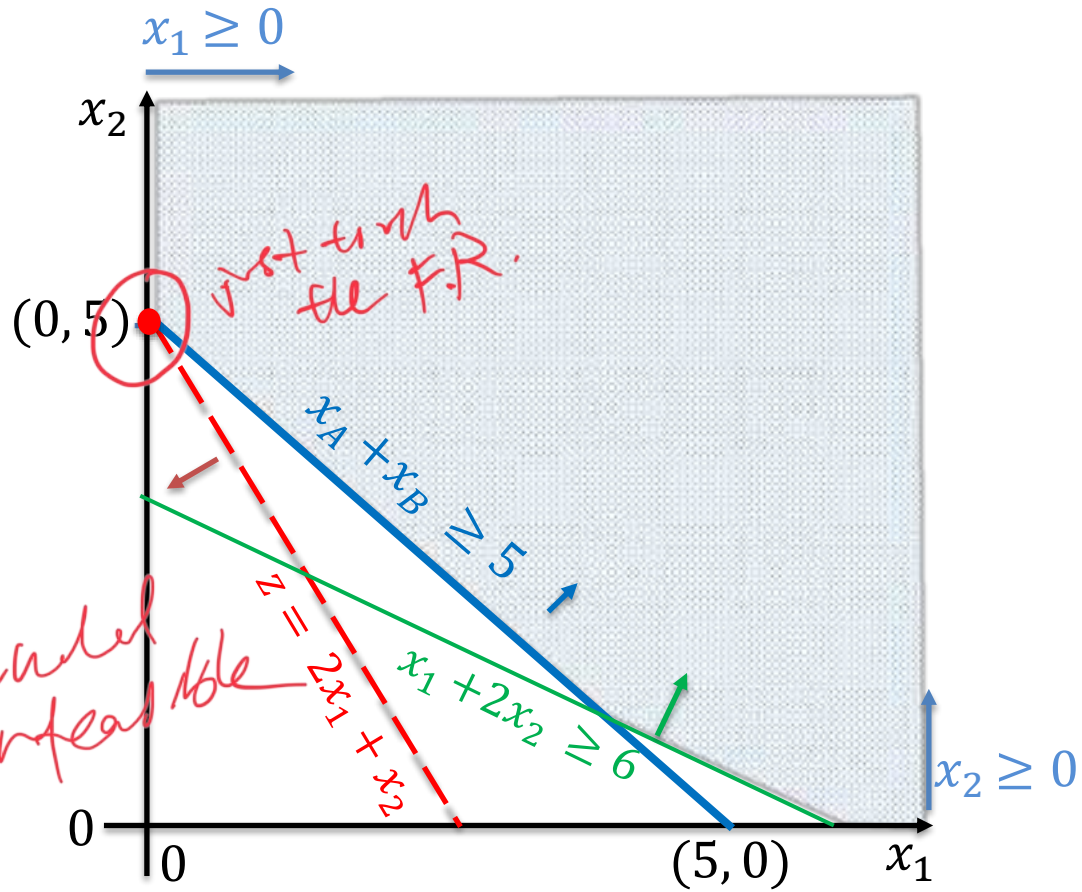
D. Neither bounded nor feasible

Q4 The given linear program is:

Minimize $z = 2x_1 + x_2$

subject to:

$$x_1 + x_2 \geq 5$$
$$x_1 + 2x_2 \geq 6$$
$$x_1, x_2 \geq 0$$



Method 1: Graphical method

optimal
⇒ not unbounded
not infeasible

Ans: Bounded and feasible

$x_1 = 0,\ x_2 = 5,\quad z = 5$

The LP is bounded, although the feasible region is unbounded

Resources: https://www.desmos.com/calculator

Q4 Method 2: Using Python


```python
import cvxpy as cp  # For solving linear programs

# 1. Define variables
x_1 = cp.Variable()  # defines a continuous decision variable x_1.
x_2 = cp.Variable()  # defines a continuous decision variable x_2.

# 2. Define objective
objective = cp.Minimize(2 * x_1 + x_2)

# 3. Define constraints
constraints = [
    x_1 >= 0,
    x_2 >= 0,
    x_1 + x_2 >= 5,
    x_1 + 2 * x_2 >= 6
]

# 4. Create a CVXPY problem object
prob = cp.Problem(objective, constraints)

# 5. Solve the LP problem
prob.solve()
```

Minimize $z = 2x_1 + x_2$

subject to:

$$x_1 + x_2 \geq 5$$
$$x_1 + 2x_2 \geq 6$$
$$x_1, x_2 \geq 0$$

cannot integer = true

```
Solution Status: optimal
x1: -9.328082293135338e-11
x2: 5.000000000322774
Minimum objective value: 5.000000000136212
```

*The tiny differences occur because CVXPY uses numerical solvers with floating-point arithmetic, so it returns very small approximations close to the exact graphical solution rather than exact integers like 0 or 5.

Q4' The given linear program is:

Maximize $z = 2x_1 + x_2$

subject to:

$$x_1 + x_2 \geq 5$$
$$x_1 + 2x_2 \geq 6$$
$$x_1, x_2 \geq 0$$

A. Bounded and feasible

B. Feasible but not bounded

C. Bounded but not feasible

D. Neither bounded nor feasible

As long as unbounded ⇒ feasible since there must be a feasible region

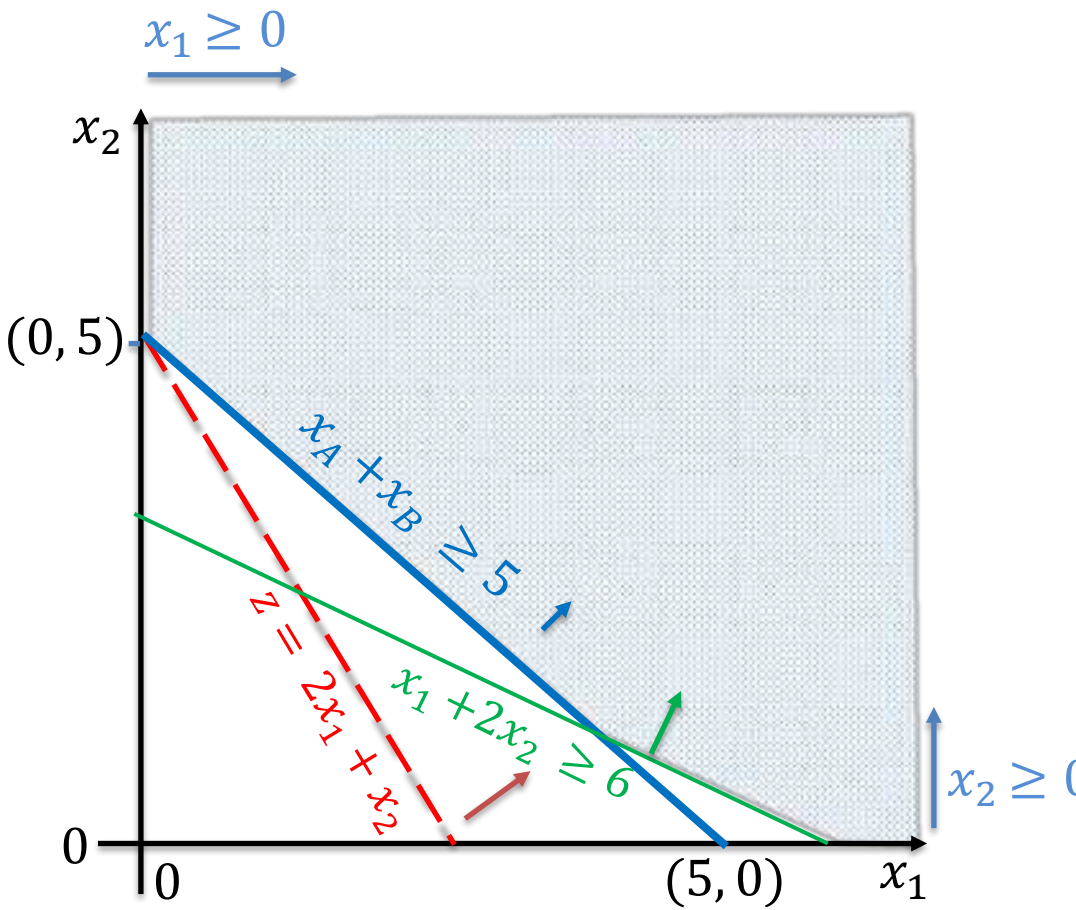
Q4' The given linear program is:

Maximize $z = 2x_1 + x_2$

subject to:

$$x_1 + x_2 \geq 5$$
$$x_1 + 2x_2 \geq 6$$
$$x_1, x_2 \geq 0$$

Method 1: Graphical method



Ans: Feasible but not bounded

Q4' The given linear program is:

Maximize $z = 2x_1 + x_2$

subject to:

$$x_1 + x_2 \geq 5$$
$$x_1, x_2 \geq 0$$

- Bounded and feasible
- Feasible but not bounded ✓
- Bounded but not feasible
- Neither bounded nor feasible

Q5 Discrete resource allocation problem

Imagine you are a Year 3 student planning your semester courses. Your goal is to maximize your benefit by carefully selecting the best combination of courses, but you have several important limits to consider:

- You can take up to **20 credits** in total for the semester.
- Your weekly workload should not exceed **40 hours** to maintain a healthy balance.
- You need to avoid **time slot conflicts** so your classes don't overlap.
- Due to personal and cognitive limits, you want to take **no more than 3 courses**.
- You can only enroll in **whole courses**—no partial enrollment allowed.

The course information is shown in the table below.

How would you select your courses to get the most academic benefit while respecting these constraints?

ID	Course	Credits	Workload per week	Time Slot	Benefit score*
1	EE2213	4	10 hrs	Mon 10 AM –12 PM	9
2	MA2101	4	9 hrs	Mon 10 AM –12 PM	7
3	CDE2212	4	11 hrs	Tue 2 PM – 5 PM	8
4	CE3201	2	5 hrs	Wed 3 PM – 5 PM	5
5	EE3801	3	7 hrs	Thu 1 PM – 3 PM	6

* Suppose the benefit score for each course is estimated based on a combination of factors such as academic return, personal interest, and career relevance.

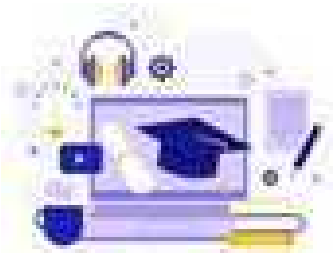
Q5 Formulating the problem

Maximize benefit by selecting the best course combination within limits.

- Choose courses within a maximum credit limit (20 credits)
- Stay within a weekly workload limit (40 hours)
- Avoid time slot conflicts
- Can select no more than 3 courses due to personal time and cognitive limits.
- Select only whole courses (integer decision)

Decision variables:

$x_i = 1$ if course i is selected, 0 otherwise, $i = 1, \dots, 5$



Objective Function (maximize academic benefit):

Maximize $Z = 9x_1 + 7x_2 + 8x_3 + 5x_4 + 6x_5$

subject to:

- *Total credits*: $4x_1 + 4x_2 + 4x_3 + 2x_4 + 3x_5 \leq 20$
- *Total weekly workload*: $10x_1 + 9x_2 + 11x_3 + 5x_4 + 7x_5 \leq 40$
- *No time conflicts*: $x_1 + x_2 \leq 1$ (EE2213 and MA2102 conflict (both Mon 10-12))
- *Total no. of courses:* $x_1 + x_2 + x_3 + x_4 + x_5 \leq 3$
- *Binary decision:* $x_i \in \{0, 1\} \quad i = 1, \dots, 5$

ID	Course	Credits	Workload per week	Time Slot	Benefit score
1	EE2213	4	10 hrs	Mon 10–12	9
2	MA2101	4	9 hrs	Mon 10–12	7
3	CDE2212	4	11 hrs	Tue 2–5	8
4	CE3201	2	5 hrs	Wed 3–5	5
5	EE3801	3	7 hrs	Thu 1–3	6

Q5 Method 1: Brute-force method

There are only 5 binary decision variables (x_1 to x_5), each taking values from {0, 1}.

That gives a total of $2^5 = 32$ possible combinations of course selections.

- Checks whether each combination satisfies all the constraints
- Evaluates the objective function (total benefit) for valid combinations.
- Returns the one with the highest academic benefit.

When the number of variables becomes large, for example, 20 courses:

20 binary decisions → 2^{20} = 1,048,576 combinations

That becomes computationally expensive.

Q5 Method 2: Using CVXPY library

```python
# Import libraries
import cvxpy as cp

# Decision variables: 1 if course is selected, 0 otherwise
x1 = cp.Variable(boolean=True, name='EE2213')
x2 = cp.Variable(boolean=True, name='MA2101')
x3 = cp.Variable(boolean=True, name='CDE2212')
x4 = cp.Variable(boolean=True, name='CE3201')
x5 = cp.Variable(boolean=True, name='EE3801')

# Objective function: maximize total benefit
objective = cp.Maximize(9*x1 + 7*x2 + 8*x3 + 5*x4 + 6*x5)

# Constraints
constraints = [
    4*x1 + 4*x2 + 4*x3 + 2*x4 + 3*x5 <= 20,    # Credit limit
    10*x1 + 9*x2 + 11*x3 + 5*x4 + 7*x5 <= 40,  # Workload limit
    x1 + x2 <= 1,                              # Time conflict
    x1 + x2 + x3 + x4 + x5 <= 3                # Limit on total number of courses
]

# Define the problem
prob = cp.Problem(objective, constraints)

# Solve the problem
prob.solve()
```

cp.Variable(boolean=True, name="x"): Define optimization variables.
-boolean=True: the variable can only take values {0, 1}; often used when our decision is yes/no choice.
-name: (string) a label to the variable (optional, for readability)

cp.Maximize(): Define objective, tells CVXPY that we want to **maximize** the expression inside

.constraints = []: a list that will hold all the problem's constraints.
-each item in the list is a CVXPY expression representing a constraint.

cp.Problem(objective, constraints**):** creates a CVXPY problem object.

.solve(): runs the solver to find the best solution.

Q5 Method 2: Using CVXPY library (cont.)

```python
# Output results
status_text = prob.status

print("Solver Status:", status_text)

# Show selected courses if solution is optimal
if status_text == "optimal":
    print("Selected courses:")
    for var in [x1, x2, x3, x4, x5]:
        if var.value == 1:
            print(var.name(), "is selected")
    print("Total benefit:", prob.value)
elif status_text == "unbounded":
    print("The problem is unbounded. Add more constraints.")
elif status_text == "infeasible":
    print("The problem has no feasible solution.")
else:
    print("Solver returned an undefined status. Please check your model.")
```

.value (for variables): gives the values of the decision variables.
.value (for problem): Gives the optimal value of the objective function.
.status: Tells you what happened when solving:
'optimal' → found a solution
'infeasible' → no solution satisfies all constraints
'unbounded' → objective can improve indefinitely

Q5 Method 3: Using PuLP library


```
# Import libraries
from pulp import *

# Define the problem
prob = LpProblem("Course_Selection", LpMaximize)

# Decision variables: 1 if course is selected, 0 otherwise
x1 = LpVariable('EE2213', cat='Binary')
x2 = LpVariable('MA2101', cat='Binary')
x3 = LpVariable('CDE2212', cat='Binary')
x4 = LpVariable('CE3201', cat='Binary')
x5 = LpVariable('EE3801', cat='Binary')

# Objective function: maximize total benefit
prob += 9*x1 + 7*x2 + 8*x3 + 5*x4 + 6*x5 # Total benefit

# Constraints
prob += 4*x1 + 4*x2 + 4*x3 + 2*x4 + 3*x5 <= 20   # Credit_Limit"
prob += 10*x1 + 9*x2 + 11*x3 + 5*x4 + 7*x5 <= 40 # Workload_Limit
prob += x1 + x2 <= 1 # Time conflict: EE2213 and MA2101 have the same time slot
prob += x1 + x2 + x3 + x4 + x5 <= 3 # Limit on total number of courses

# Solve it
prob.solve()
```

LpProblem(name, sense): creates a new LP problem.
- name: just a label (optional, for readability)
- sense: *LpMaximize* (to maximize) or *LpMinimize* (to minimize)

LpVariable(name, lowBound, upBound, cat): defines a variable for LP
- name: variable name (string)
- lowBound: lower limit (e.g., 0 for non-negativity)
- cat: category of variable (e.g., Continuous, Integer). Default is Continuous.

"**+=**" is used in PuLP to add an expression to the LP model
Here, we are adding the objective function and constraints to 'prob'

.solve(): runs the built-in solver to find the optimal solution (usually using the Simplex algorithm)

Q5 Method 3: Using PuLP library (cont.)

```python
# Output results

# Check and print solver status
status_code = prob.status               # Numeric code (e.g., 1 for Optimal)
status_text = LpStatus[status_code]     # Convert to readable text (e.g., 'Optimal')

print("Solver Status:", status_text)    # Display the outcome

# Show selected courses if solution is optimal
if status_text == "Optimal":
    print("Selected courses:")
    for var in prob.variables():
        if var.varValue == 1:
            print(var.name, "is selected")
    print("Total benefit:", value(prob.objective))
elif status_text == "Unbounded":
    print("The problem is unbounded. Add more constraints.")
elif status_text == "Infeasible":
    print("The problem has no feasible solution.")
else:
    print("Solver returned an undefined status. Please check your model.")
```

.status: returns a numeric code indicating solver's outcome (.e.g., 0, 1, -1, -2, -3)

LpStatus[status_code]: turns that number into a readable message like "Optimal" or "Infeasible".

LpStatus is a built-in Python dictionary that maps status codes to their meanings:

```
{
  0: 'Not Solved',
  1: 'Optimal',
 -1: 'Infeasible',
 -2: 'Unbounded',
 -3: 'Undefined'
}
```

```
Solver Status: Optimal
Selected courses:
CDE2212 is selected
EE2213 is selected
EE3801 is selected
Total benefit: 23.0
```

Want to explore more?

Python practice with examples

CVXPY Examples: https://www.cvxpy.org/examples/index.html
PuLP Case Studies: https://coin-or.github.io/pulp/CaseStudies/

LeetCode #322: Coin Change
https://leetcode.com/problems/coin-change/description/

Set-ExecutionPolicy – ExecutionPolicy Remote Signed
– Scope Process
.\venv\Scripts\activate

EE2213 Introduction to Artificial Intelligence

Tutorial 5 (Lecture 6)

Dr. Shaojing Fan
fanshaojing@nus.edu.sg

Q1

Which of the following functions are convex? Select all that apply.

A: $f(x) = 5 - (x-2)^2, x \in \boldsymbol{\mathcal{R}}$ concave

B: $f(x) = 7 - 3x, \ x \in \boldsymbol{\mathcal{R}}$ yes linear

C: $f(x) = \log(x), x \in (0, +\infty)$ no

D: $f(x) = e^x, \ x \in \boldsymbol{\mathcal{R}}$ yes

E: $f(x) = |2x + 2|, x \in \boldsymbol{\mathcal{R}}$ yes

Why this question?

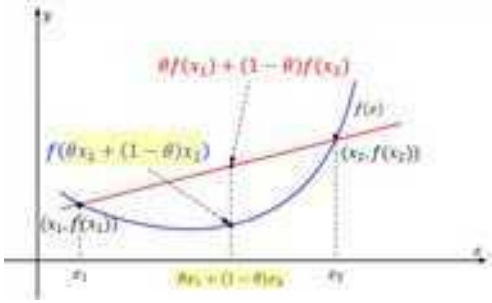
It's useful to know if a function is convex. Later on, especially in areas like machine learning, you'll need to design (objective) functions that can be minimized reliably. If the function is convex, you can be sure there's only one best solution, and your optimization method (e.g., gradient descent) will be able to find it.

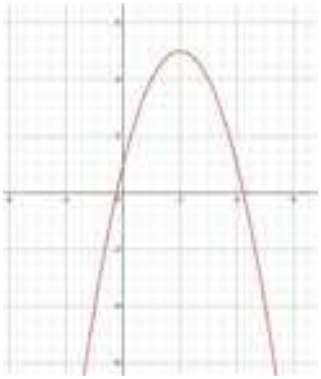
Q1 Method 1: Use the definition

Let C be a convex set. A function $f: C \to \mathbb{R}$ is convex in C if $\forall x_1, x_2 \in C$, $\forall \theta \in [0,1]$,

$$f(\theta x_1 + (1-\theta)x_2) \leq \theta f(x_1) + (1-\theta) f(x_2)$$

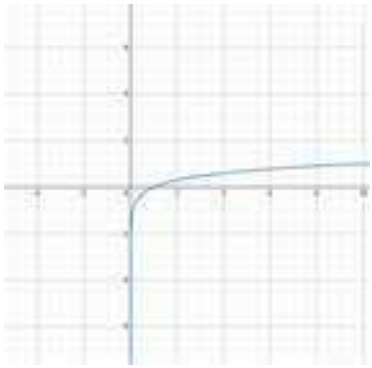
If $C = \mathbb{R}^n$, we simply say f is a convex function.





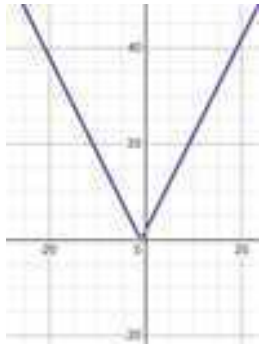
$f(x) = 5 - (x-2)^2$

$f(x) = 7 - 3x$



$f(x) = \log(x)$

$f(x) = e^x$



$f(x) = |2x + 2|$

Q1

Which of the following functions are convex? Select all that apply.

A: $f(x) = 5 - (x-2)^2, x \in \boldsymbol{\mathcal{R}}$

B: $f(x) = 7 - 3x,\ x \in \boldsymbol{\mathcal{R}}$ ✓

C: $f(x) = \log(x), x \in (0, +\infty)$

D: $f(x) = e^x,\ x \in \boldsymbol{\mathcal{R}}$ ✓

E: $f(x) = |2x + 2|, x \in \boldsymbol{\mathcal{R}}$ ✓

Q1 Method 2: The second derivative test (for single-variable functions)

If the second derivative, $f''(x)$ is **always greater than or equal to 0** ($f''(x) \geq 0$) *throughout the function's domain*, then the function is **convex**, if $f''(x) \leq 0$, then the function is concave.

Example 1: For the function $f(x) = (x-2)^2$,
$f''(x) = 2$, which is always ≥ 0, so the function is convex.

Example 2: For the function $f(x) = \log(x),\ x \in (0, +\infty)$
$f'(x) = \frac{1}{x}$, $f''(x) = -\frac{1}{x^2}$, which is always < 0, so the function is concave.

Q2 True or false: For a convex optimization problem with a differentiable objective function, gradient descent always converges to the global minimum if the learning rate is chosen appropriately.

A. True ✓

B. False

Q3

Why this question? **Gradient descent is one of the most common topics in machine learning interviews and a key concept behind many algorithms!**

Which of the following statements about gradient descent are correct? Select all that apply.

A: Gradient descent moves in the direction of steepest descent, which is the negative gradient of the function.

B: For convex functions with a properly chosen learning rate, gradient descent will converge to the global minimum.

C: If the learning rate is too large, gradient descent may overshoot and fail to converge.

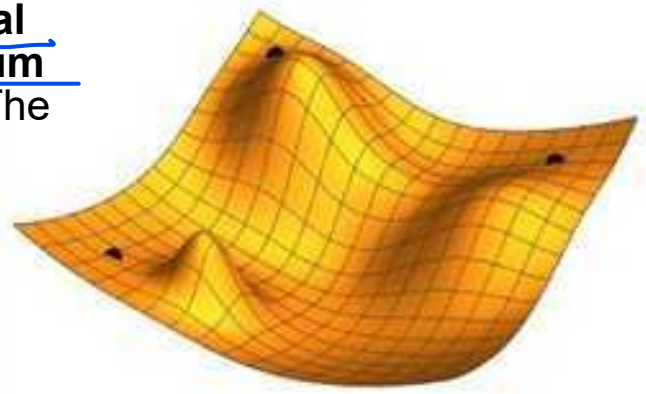
D: Gradient descent is guaranteed to find the global minimum even for non-convex functions.

E: Gradient descent only works for functions that are differentiable and strictly convex.

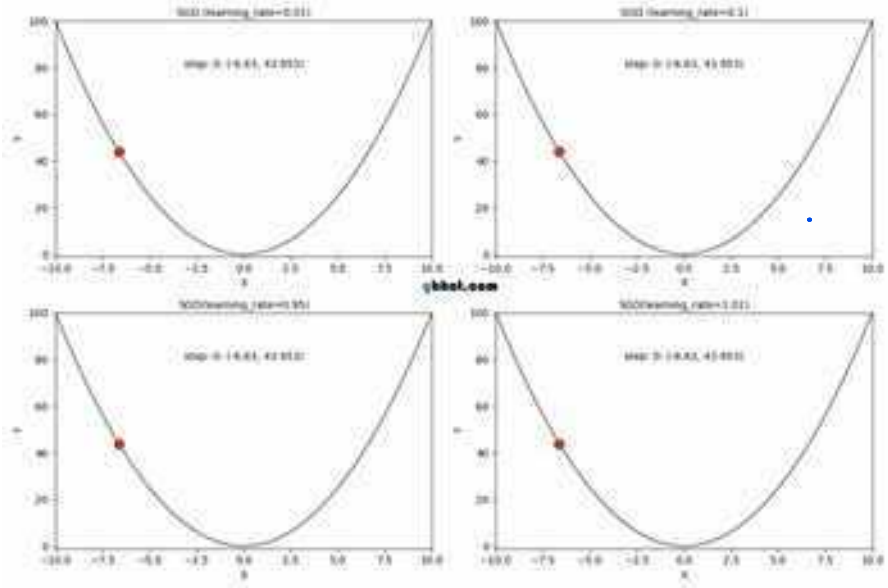
References: https://github.com/Devinterview-io/**gradient-descent-interview-questions**

Limitation of Gradient Descent

•**Gets stuck in local minima**: For *non-convex functions*, gradient descent can get stuck in a **local minimum**, and may fail to find the **global minimum** (the lowest possible point of the entire function). The starting point for the algorithm can significantly influence which minimum it converges to.



•**Initialization & learning rate sensitivity**: The outcome depends heavily on the starting point (initialization) and the learning rate (step size). A poor choice of either can lead to suboptimal results.



Source: https://commons.wikimedia.org/wiki/File:Gradient_descent.gif
https://gbhat.com/machine_learning/gradient_descent_learning_rates.html

Q3

Which of the following statements about gradient descent are correct? Select all that apply.

- ✓ A: Gradient descent moves in the direction of steepest descent, which is the negative gradient of the function.
- ✓ B: For convex functions with a properly chosen learning rate, gradient descent will converge to the global minimum.
- ✓ C: If the learning rate is too large, gradient descent may overshoot and fail to converge.

D: Gradient descent is guaranteed to find the global minimum even for non-convex functions.

E: Gradient descent only works for functions that are differentiable and strictly convex.

Q4

Suppose we are minimizing $f(x) = x^6$ with respect to x. We initialize $x = 1$, and perform gradient descent with learning rate $\eta = 0.01$.

(i) Calculate the updated value of x after the second iteration. Round your answer to 2 decimal places.

(ii) Explain why the updates in gradient descent become very small as x approaches 0. How does the shape of the function $f(x) = x^6$ influence the speed of convergence compared to a function like $f(x) = x^2$?

Q4 Common derivatives

[DERIVATION TABLES] --- Algebric and Logarithmes functions

$\frac{d}{dx}(c) = 0$	$\frac{d}{dx}(cx) = c$	$\frac{d}{dx}(x^c) = cx^{c-1}$
$\frac{d}{dx}(c^x) = c^x \ln(c) \quad c > 0$	$\frac{d}{dx}(x^x) = x^x(1 + \ln x)$	$\frac{d}{dx}(e^x) = e^x$
$\frac{d}{dx}\left(\frac{1}{x}\right) = \frac{-1}{x^2}$	$\frac{d}{dx}\left(\frac{1}{x^2}\right) = \frac{-2}{x^3}$	$\frac{d}{dx}\left(\frac{1}{x^n}\right) = \frac{-n}{x^{n+1}}$
$\frac{d}{dx}(\sqrt{x}) = \frac{1}{2\sqrt{x}} \quad x > 0$	$\frac{d}{dx}(\sqrt[3]{x}) = \frac{1}{3 \cdot \sqrt[3]{x^2}}$	$\frac{d}{dx}(\sqrt[n]{x}) = \frac{1}{n \cdot \sqrt[n]{x^{n-1}}}$
$\frac{d}{dx}\left(\frac{1}{\sqrt{x}}\right) = \frac{-1}{2\sqrt{x^3}}$	$\frac{d}{dx}\left(\frac{1}{\sqrt[3]{x}}\right) = \frac{-1}{3 \cdot \sqrt[3]{x^4}}$	$\frac{d}{dx}\left(\frac{1}{\sqrt[n]{x}}\right) = \frac{-1}{n \cdot \sqrt[n]{x^{n+1}}}$
$\frac{d}{dx}(\ln x) = \frac{1}{x} \quad x > 0$	$\frac{d}{dx}(x \cdot \ln x) = \ln x + 1$	$\frac{d}{dx}(\log_c x) = \frac{1}{x \ln c} \quad c > 0 \quad c \neq 1$
$\frac{d}{dx}\left(\frac{1}{\ln x}\right) = \frac{-1}{x(\ln x)^2}$	$\frac{d}{dx}\left(\frac{1}{x \cdot \ln x}\right) = \frac{-(\ln x + 1)}{(x \cdot \ln x)^2}$	$\frac{d}{dx}\left(\frac{1}{\log_c x}\right) = \frac{-1}{x \cdot \ln c \cdot (\log_c x)^2}$
$\frac{d}{dx}\left(\frac{1}{x+1}\right) = \frac{-1}{(x+1)^2}$	$\frac{d}{dx}\left(\frac{1}{(x+1)^2}\right) = \frac{-2}{(x+1)^3}$	$\frac{d}{dx}\left(\frac{1}{(x+1)^n}\right) = \frac{-n}{(x+1)^{n+1}}$
$\frac{d}{dx}\left(\frac{1}{\sqrt{x+1}}\right) = \frac{-1}{2 \cdot \sqrt{(x+1)^3}}$	$\frac{d}{dx}\left(\frac{1}{\sqrt[3]{x+1}}\right) = \frac{-1}{3 \cdot \sqrt[3]{(x+1)^4}}$	$\frac{d}{dx}\left(\frac{1}{\sqrt[n]{x+1}}\right) = \frac{-1}{n \cdot \sqrt[n]{(x+1)^{n+1}}}$

Q4

Suppose we are minimizing $f(x) = x^6$ with respect to x. We initialize $x = 1$, and perform gradient descent with learning rate $\eta = 0.01$.

(i) Calculate the updated value of x after the second iteration.

1. Compute the gradient: $\nabla_x f(x) = 6x^5$
2. At each iteration, $x_{k+1} \leftarrow x_k - \eta \nabla_x f(x_k)$

Gradient Descent:

Initialize $\mathbf{x}_0$ and learning rate η

while true **do**

  Compute $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta \nabla_x f(\mathbf{x}_k)$

  if converge **then**

    return $\mathbf{x}_{k+1}$

  end

end

Iteration 1

- $x_0 = 1$
- $\nabla_x f(x_0) = 6{x_0}^5 = 6$
- $x_1 = x_0 - 0.01 \nabla_x f(x_0) = 0.94$

Iteration 2

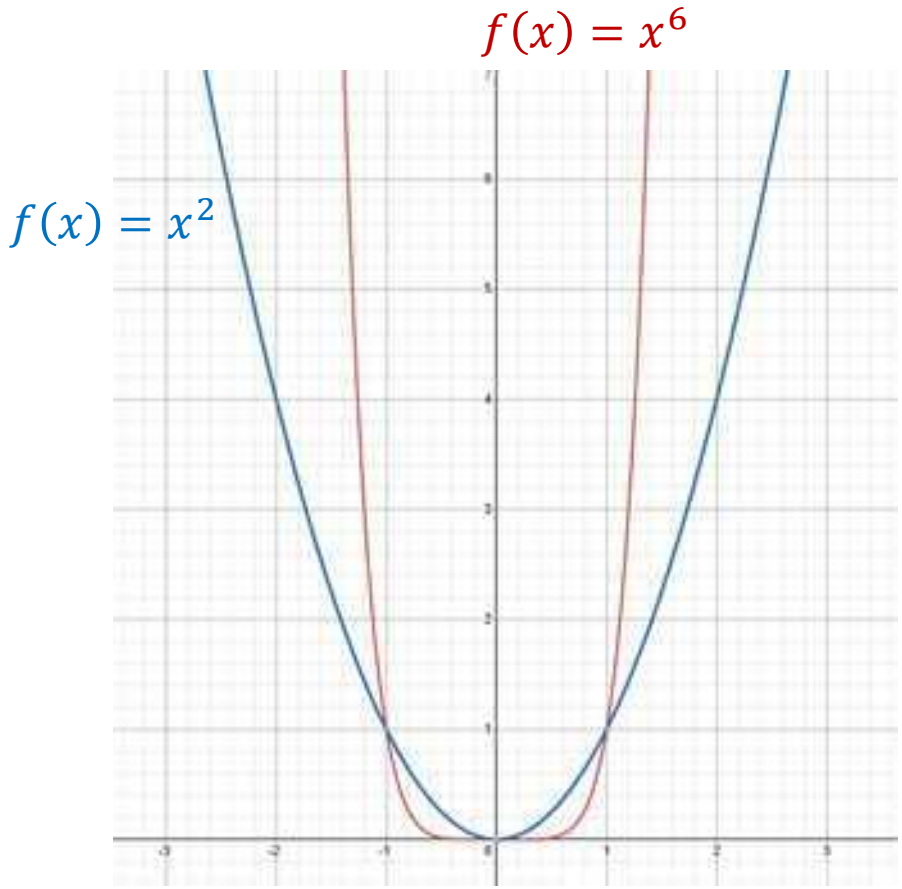
- $x_1 = 0.84$
- $\nabla_x f(x_1) = 6{x_1}^5 = 4.4034$
- $x_2 = x_1 - 0.01 \nabla_x f(x_1) = 0.90$ (round to 2 decimal places)

Q4

(ii) Explain why the updates in gradient descent become very small as x approaches 0. How does the shape of the function $f(x) = x^6$ influence the speed of convergence compared to a function like $f(x) = x^2$?

As x approaches 0, the gradient $f'(x) = 6x^5$ becomes very small, so the gradient descent updates shrink and progress slows down.

The shape of $f(x) = x^6$ is much flatter near $x = 0$, than $f(x) = x^2$, so convergence takes many more iterations.



Q5

Suppose we have a function that depends on a vector of numbers $\mathbf{w}$:

$$C(\mathbf{w}) = \sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i)^2$$

where:

$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ are given vectors,

$y_1, y_2, \ldots, y_m$ are given numbers (scalars).

Why this question?
Bridging to machine learning!

We want to minimize $C(\mathbf{w})$ using gradient descent.

(i) Check if the function $C(\mathbf{w})$ is convex. Explain your reasoning.

(ii) Compute the gradient of $C(\mathbf{w})$ with respect to $\mathbf{w}$ using $\sum$ notation.

(iii) Write the gradient descent update rule for $\mathbf{w}$ with learning rate η.

(iv) If $\mathbf{x}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $\mathbf{x}_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix}$, $\eta = 0.01$, we initialize $\mathbf{w} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, calculate the updated value of $\mathbf{w}$ after the first iteration.

(v) (Optional): Following the setup in (iv), solve the convex optimization problem using a solver (e.g., CVXPY).

Q5

Suppose we have a function that depends on a vector of numbers $\mathbf{w}$:

$$C(\mathbf{w}) = \sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i)^2$$

where:

$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ are given vectors,

$y_1, y_2, \ldots, y_m$ are given numbers (i.e., scalars).

We want to minimize $C(\mathbf{w})$ using gradient descent.

(i) Check if the function $C(\mathbf{w})$ is convex. Explain your reasoning.

Ans: Each term $(\mathbf{w}^T \mathbf{x}_i - y_i)^2$ is a squared linear function of $\mathbf{w}$. Squaring a linear function always gives a convex function. The sum of convex functions is also convex. Therefore, $C(\mathbf{w})$ is convex.

Q5

Suppose we have a function that depends on a vector of numbers $\mathbf{w}$:

$$C(\mathbf{w}) = \sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i)^2$$

where:

$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ are given vectors,

$y_1, y_2, \ldots, y_m$ are given numbers (i.e., scalars).

We want to minimize $C(\mathbf{w})$ using gradient descent.

(ii) Compute the gradient of $C(\mathbf{w})$ with respect to $\mathbf{w}$ using $\sum$ notation.

Ans: For a scalar function $f(\mathbf{w}) = (\mathbf{w}^T \mathbf{x} - y)^2$, the gradient with respect to $\mathbf{w}$ is:

$$\nabla_{\mathbf{w}} (\mathbf{w}^T \mathbf{x} - y)^2 = 2\ (\mathbf{w}^T \mathbf{x} - y)\mathbf{x}$$

The gradient of a sum is just the sum of the gradients:

$$\nabla_{\mathbf{w}} C(\mathbf{w}) = \nabla_{\mathbf{w}} \left[\sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i)^2 \right] = \sum_{i=1}^{m} \nabla_{\mathbf{w}} [(\mathbf{w}^T \mathbf{x} - y)^2] = 2 \sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i)\mathbf{x}_i$$

Q5

Suppose we have a function that depends on a vector of numbers $\mathbf{w}$:

$$C(\mathbf{w}) = \sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i)^2$$

where:

$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ are given vectors,

$y_1, y_2, \ldots, y_m$ are given numbers (i.e., scalars).

We want to minimize $C(\mathbf{w})$ using gradient descent.

(iii) Write the gradient descent update rule for $\mathbf{w}$ with learning rate η.

$$\nabla_{\mathbf{w}} C(\mathbf{w}) = 2 \sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i) \mathbf{x}_i$$

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla_{\mathbf{w}} C(\mathbf{w}) = \mathbf{w}_k - 2\eta \sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i) \mathbf{x}_i$$

Q5

(iv) Write the gradient descent update rule for $\mathbf{w}$ with learning rate η.

If $\mathbf{x}_1 = \begin{bmatrix}1\\0\end{bmatrix}$, $\mathbf{x}_2 = \begin{bmatrix}0\\1\end{bmatrix}$, $\mathbf{x}_3 = \begin{bmatrix}1\\1\end{bmatrix}$, $\mathbf{y} = \begin{bmatrix}y_1\\y_2\\y_3\end{bmatrix} = \begin{bmatrix}2\\3\\5\end{bmatrix}$, $\eta = 0.01$, we initialize $\mathbf{w} = \begin{bmatrix}0\\0\end{bmatrix}$, calculate the updated value of $\mathbf{w}$ after the first iteration.

1. Compute the gradient: $\nabla_{\mathbf{w}} C(\mathbf{w}) = 2\sum_{i=1}^{3}(\mathbf{w}^T\mathbf{x}_i - y_i)\mathbf{x}_i$
2. At each iteration, $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta\, \nabla_{\mathbf{w}} C(\mathbf{w})$

Method 1:
Solve by hand

Step 1: Compute gradient at $w_0 = \begin{bmatrix}0\\0\end{bmatrix}$

$i = 1, \mathbf{w_0}^T\mathbf{x}_1 - y_1 = 0 - 2 = -2, (\mathbf{w_0}^T\mathbf{x}_1 - y_1)\mathbf{x}_1 = \begin{bmatrix}-2\\0\end{bmatrix}$

$i = 2, \mathbf{w_0}^T\mathbf{x_2} - y_2 = 0 - 3 = -3, (\mathbf{w_0}^T\mathbf{x}_2 - y_2)\mathbf{x}_2 = \begin{bmatrix}0\\-3\end{bmatrix}$

$i = 3, \mathbf{w_0}^T\mathbf{x_3} - y_3 = 0 - 5 = -5, (\mathbf{w_0}^T\mathbf{x}_3 - y_3)\mathbf{x}_3 = \begin{bmatrix}-5\\-5\end{bmatrix}$

Sum:

$$\nabla_{\mathbf{w}} C(\mathbf{w_0}) = 2\sum_{i=1}^{3}(\mathbf{w_0}^T\mathbf{x}_i - y_i)\mathbf{x}_i = 2\left(\begin{bmatrix}-2\\0\end{bmatrix} + \begin{bmatrix}0\\-3\end{bmatrix} + \begin{bmatrix}-5\\-5\end{bmatrix}\right) = \begin{bmatrix}-14\\-16\end{bmatrix}$$

Step 2: Update w

$$\mathbf{w}_1 = \mathbf{w}_0 - \eta\, \nabla_{\mathbf{w}} C(\mathbf{w_0}) = \begin{bmatrix}0\\0\end{bmatrix} - 0.01\begin{bmatrix}-14\\-16\end{bmatrix} = \begin{bmatrix}0.14\\0.16\end{bmatrix}$$

Q5

(iv) Write the gradient descent update rule for $\mathbf{w}$ with learning rate η. If $\mathbf{x}_1 = \begin{bmatrix}1\\0\end{bmatrix}$, $\mathbf{x}_2 = \begin{bmatrix}0\\1\end{bmatrix}$, $\mathbf{x}_3 = \begin{bmatrix}1\\1\end{bmatrix}$, $\mathbf{y} = \begin{bmatrix}y_1\\y_2\\y_3\end{bmatrix} = \begin{bmatrix}2\\3\\5\end{bmatrix}$, $\eta = 0.01$, we initialize $\mathbf{w} = \begin{bmatrix}0\\0\end{bmatrix}$, calculate the updated value of $\mathbf{w}$ after the first iteration.

Method 2: Solve by Python

```python
import numpy as np

# Given data
X = np.array([[1, 0],
              [0, 1],
              [1, 1]])
y = np.array([[2], [3], [5]])
w = np.array([[0], [0]])  # initial w
eta = 0.01  # learning rate

# Compute the gradient
grad = 2 * X.T @ (X @ w - y)

# Update w
w_new = w - eta * grad

print("\nGradient (∇C(w)):", grad)
print("\nUpdated w after the first iteration:", w_new)
```

Initialize Gradient Descent
- Set X and y values
- Initialize w to be [0, 0]
- Set learning rate to 0.01

$$\nabla_{\mathbf{w}} C(\mathbf{w}) = 2\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}_i - y_i)\mathbf{x}_i$$

$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \eta\nabla_x f(\mathbf{x}_k)$$

Perform Gradient Descent
- Compute gradient
- Update w
  $w_1 \leftarrow w_0$ - learning_rate * gradient.

```
Gradient (∇C(w)): [[-14]
 [-16]]

Updated w after the first iteration: [[0.14]
 [0.16]]
```

Q5 Code highlights (for those new to Python)

import numpy as np

Import the NumPy library in Python, and assign it an alias “np”, so we can refer to it more easily later. NumPy is a powerful library for numerical computing in Python, especially useful for working with arrays, matrices, and mathematical functions.

X @ w

Multiply matrix X with vector w

The @ operator in NumPy performs matrix multiplication, different from element-wise multiplication (∗).

In our example: $\boldsymbol{X} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$, $\mathbf{w} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathrm{X} @ \mathrm{w} = \begin{bmatrix} 1*0+0*0 \\ 0*0+1*0 \\ 1*0+1*0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$

X.T

Transpose of the matrix X, flips the rows and columns of X.

In our example: $\boldsymbol{X} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$, $\boldsymbol{X^T} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$

References: https://numpy.org/doc/stable/reference/index.html#reference

Q5 (Optional): Using CVXPY to solve the convex optimization problem


```python
import cvxpy as cp
import numpy as np

# Given data
X = np.array([[1, 0],
              [0, 1],
              [1, 1]])
y = np.array([[2],
              [3],
              [5]])

# Define variable
w = cp.Variable((2,1))  # w is a 2x1 column vector

# Define the objective function
objective = cp.Minimize(cp.sum_squares(X @ w - y))

# Define problem
problem = cp.Problem(objective)

# Solve
problem.solve()

# Results
print("\nOptimal weights w:")
print(w.value)
print("\nMinimum cost:", problem.value)
```

$$\min_{\mathbf{w}} C(\mathbf{w}) = \min_{\mathbf{w}} \sum_{i=1}^{m} (\mathbf{w}^T \mathbf{x}_i - y_i)^2$$

Prepare data for objective function

Decision variables

Objective function

Create the convex optimization problem (w/o constraint)

Solve (typically non-gradient-based methods)

Get the result

```
Optimal weights w:
[[2.]
 [3.]]

Minimum cost: 4.930380657631324e-32
```

Q5 Takeaway: Gradient descent vs CVXPY

Non-convex problems

- CVXPY works only for convex problems.
- Many real-world ML tasks (e.g., training neural networks) are non-convex, and *gradient descent* (and its variants, e.g., SGD, mini-batch) can still be applied.

Large datasets and high-dimensional models

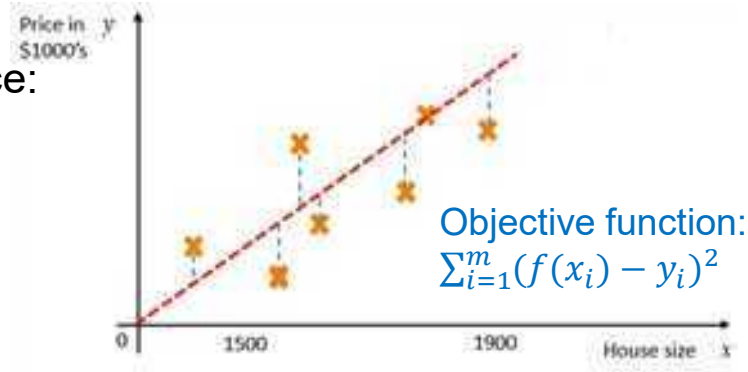
- *Gradient descent* and its variants (e.g., SGD, mini-batch) are scalable to huge datasets and high-dimensional parameter vectors.
- Convex solvers may be too slow or infeasible for large-scale problems.

Foundation for advanced algorithms

- *Gradient descent* provides the basis for understanding modern optimizers such as Adam and momentum methods.

Q5: Bridging to machine learning

1. Goal of machine learning in house price prediction:
Find the best fitting line to map from house size to price:
$y = ax + b$
a: slope, b: intercept



2. Pack parameters into a single vector: $\mathbf{w} = \begin{bmatrix} a \\ b \end{bmatrix}$

3. For each data point (x_i, y_i), define the feature vector to include a 1 for the intercept term:

$$\mathbf{x}_i = \begin{bmatrix} x_i \\ 1 \end{bmatrix}$$

Now the prediction is: $\mathbf{w}^T\mathbf{x}_i = a \cdot x_i + b \cdot 1$

4. We want to minimize the sum of squared error between predicted values ($\mathbf{w}^T\mathbf{x}_i$) and the true value (y_i):

$$C(\mathbf{w}) = \sum_{i=1}^{m} (\mathbf{w}^T\mathbf{x}_i - y_i)^2$$

$\mathbf{w}^T\mathbf{x}_i$: Prediction for sample i, i.e., predicted value
y_i: Actual (true) value
Squared error: (**prediction−truth**)²
$C(\mathbf{w})$ *measures how well the line fits the data.*