CS498: Algorithmic Engineering

Lecture 12: Penalty Methods, Lagrangian Duality & Constrained Optimization

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 06 – 02/26/2026

Outline

Where We Left Off

Last lecture: **projected gradient descent** for constrained optimization.

$$x_{k+1} = \Pi_C(x_k - \alpha \nabla f(x_k))$$

Where We Left Off

Last lecture: **projected gradient descent** for constrained optimization.

$$x_{k+1} = \Pi_C(x_k - \alpha \nabla f(x_k))$$

We worked through **box projections**:

- Constraint: $x \in [\mathbf{l}, \mathbf{u}]$.
- Projection: coordinate-wise clamp $\Pi(z)_i = \min(u_i, \max(l_i, z_i))$.
- Cost: $O(n)$. In PyTorch: `x.clamp_(min=l, max=u)`.

Where We Left Off

Last lecture: **projected gradient descent** for constrained optimization.

$$x_{k+1} = \Pi_C(x_k - \alpha \nabla f(x_k))$$

We worked through **box projections**:

- Constraint: $x \in [\mathbf{l}, \mathbf{u}]$.
- Projection: coordinate-wise clamp $\Pi(z)_i = \min(u_i, \max(l_i, z_i))$.
- Cost: $O(n)$. In PyTorch: `x.clamp_(min=l, max=u)`.

Let's do one more easy projection, then see what happens when things get hard.

Projection onto a Ball

Constraint: $\|x - c\| \leq r$ (Euclidean ball centered at c, radius r).

Projection onto a Ball

Constraint: $\|x - c\| \leq r$ (Euclidean ball centered at c, radius r).

Question: given a point z, what is the closest point in the ball?

Projection onto a Ball

Constraint: $\|x - c\| \le r$ (Euclidean ball centered at c, radius r).

Question: given a point z, what is the closest point in the ball?

- If z is already inside: $\Pi(z) = z$. Done.

Projection onto a Ball

Constraint: $\|x - c\| \leq r$ (Euclidean ball centered at c, radius r).

Question: given a point z, what is the closest point in the ball?

- If z is already inside: $\Pi(z) = z$. Done.
- If z is outside: the closest point lies on the **boundary**, along the line from c to z.

Projection onto a Ball

Constraint: $\|x - c\| \leq r$ (Euclidean ball centered at c, radius r).

Question: given a point z, what is the closest point in the ball?

- If z is already inside: $\Pi(z) = z$. Done.
- If z is outside: the closest point lies on the **boundary**, along the line from c to z.

$$\Pi(z) = \begin{cases} z & \text{if } \|z - c\| \leq r, \\ c + r \cdot \dfrac{z - c}{\|z - c\|} & \text{if } \|z - c\| > r. \end{cases}$$

Projection onto a Ball

Constraint: $\|x - c\| \leq r$ (Euclidean ball centered at c, radius r).

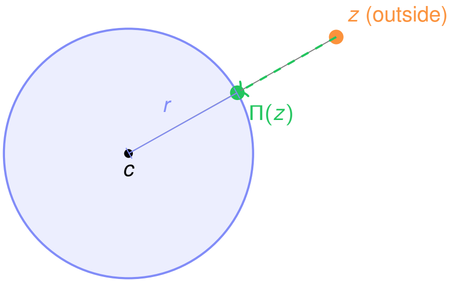
Question: given a point z, what is the closest point in the ball?

- If z is already inside: $\Pi(z) = z$. Done.
- If z is outside: the closest point lies on the **boundary**, along the line from c to z.

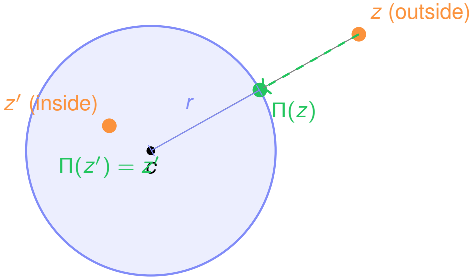
$$\Pi(z) = \begin{cases} z & \text{if } \|z - c\| \leq r, \\ c + r \cdot \dfrac{z - c}{\|z - c\|} & \text{if } \|z - c\| > r. \end{cases}$$

Intuition: walk from the center toward z, stop at the surface. Cost: $O(n)$.

Ball Projection: Picture



Ball Projection: Picture



Example: Minimizing over a Ball

$$\min_{\|x\| \le 1} f(x) = (x_1 - 3)^2 + (x_2 - 4)^2.$$

Example: Minimizing over a Ball

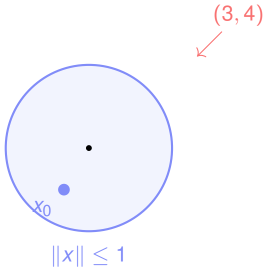
$$\min_{\|x\| \leq 1} \; f(x) = (x_1 - 3)^2 + (x_2 - 4)^2.$$

Unconstrained optimum: $(3, 4)$ — far outside the unit ball.

Example: Minimizing over a Ball

$$\min_{\|x\| \leq 1} \; f(x) = (x_1 - 3)^2 + (x_2 - 4)^2.$$

Unconstrained optimum: $(3, 4)$ — far outside the unit ball.
Projected GD: step toward $(3, 4)$, then rescale to $\|x\| \leq 1$ whenever we leave.

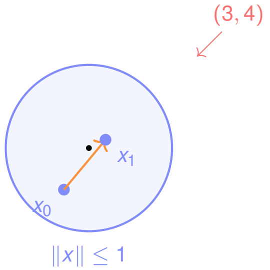


Constrained optimum: $(3/5, 4/5) = (0.6, 0.8)$.

Example: Minimizing over a Ball

$$\min_{\|x\|\le 1} \; f(x) = (x_1 - 3)^2 + (x_2 - 4)^2.$$

Unconstrained optimum: $(3, 4)$ — far outside the unit ball.
Projected GD: step toward $(3, 4)$, then rescale to $\|x\| \le 1$ whenever we leave.

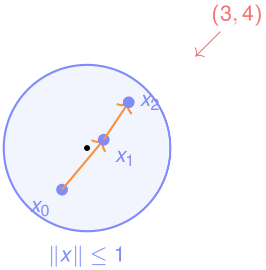


Constrained optimum: $(3/5, 4/5) = (0.6, 0.8)$.

Example: Minimizing over a Ball

$$\min_{\|x\|\le 1} f(x) = (x_1 - 3)^2 + (x_2 - 4)^2.$$

Unconstrained optimum: $(3,4)$ — far outside the unit ball.
Projected GD: step toward $(3,4)$, then rescale to $\|x\| \le 1$ whenever we leave.

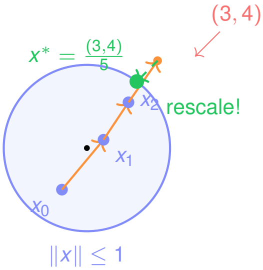


Constrained optimum: $(3/5, 4/5) = (0.6, 0.8)$.

Example: Minimizing over a Ball

$$\min_{\|x\|\leq 1} \; f(x) = (x_1 - 3)^2 + (x_2 - 4)^2.$$

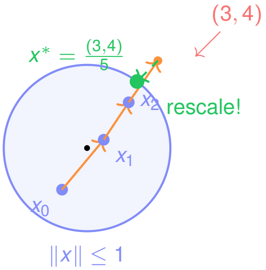
Unconstrained optimum: $(3, 4)$ — far outside the unit ball.
Projected GD: step toward $(3, 4)$, then rescale to $\|x\| \leq 1$ whenever we leave.



Example: Minimizing over a Ball

$$\min_{\|x\| \le 1} f(x) = (x_1 - 3)^2 + (x_2 - 4)^2.$$

Unconstrained optimum: $(3, 4)$ — far outside the unit ball.
Projected GD: step toward $(3, 4)$, then rescale to $\|x\| \le 1$ whenever we leave.



Constrained optimum: $(3/5, 4/5) = (0.6, 0.8)$.

Ball Projection in PyTorch (and with torch.no_grad())

```
import torch

x = torch.tensor([0.1, 0.2], requires_grad=True)
r = 1.0
alpha = 0.1

for k in range(100):
    loss = (x[0] - 3)**2 + (x[1] - 4)**2
    loss.backward()

    #x.data -= alpha * x.grad
```

Ball Projection in PyTorch (and with torch.no_grad())

```python
import torch

x = torch.tensor([0.1, 0.2], requires_grad=True)
r = 1.0
alpha = 0.1

for k in range(100):
    loss = (x[0] - 3)**2 + (x[1] - 4)**2
    loss.backward()

    #x.data -= alpha * x.grad
    with torch.no_grad():
        x -= alpha * x.grad           # gradient step
        norm = torch.norm(x)
        if norm > r:
            x *= r / norm              # project: rescale to boundary
        x.grad.zero_()

print(x)   # tensor([0.6000, 0.8000], requires_grad=True)
```

Ball Projection in PyTorch (and with torch.no_grad())

```python
import torch

x = torch.tensor([0.1, 0.2], requires_grad=True)
r = 1.0
alpha = 0.1

for k in range(100):
    loss = (x[0] - 3)**2 + (x[1] - 4)**2
    loss.backward()

    #x.data -= alpha * x.grad
    with torch.no_grad():
        x -= alpha * x.grad           # gradient step
        norm = torch.norm(x)
        if norm > r:
            x *= r / norm              # project: rescale to boundary
        x.grad.zero_()

print(x)   # tensor([0.6000, 0.8000], requires_grad=True)
```

Same pattern as boxes: gradient step, then project. Two extra lines for the projection.

When Projection is Hard

Boxes and balls have cheap, closed-form projections. Great!

When Projection is Hard

Boxes and balls have cheap, closed-form projections. Great!

But what about **general linear constraints**?

$$C = \{x : Ax \leq b,\ Cx = d,\ x \geq 0\}$$

When Projection is Hard

Boxes and balls have cheap, closed-form projections. Great!

But what about **general linear constraints**?

$$C = \{x : Ax \leq b,\ Cx = d,\ x \geq 0\}$$

To project z onto C, we must solve:

$$\Pi_C(z) = \arg \min_{Ax \leq b,\, Cx = d,\, x \geq 0} \|x - z\|^2.$$

When Projection is Hard

Boxes and balls have cheap, closed-form projections. Great!

But what about **general linear constraints**?

$$C = \{x : Ax \leq b,\ Cx = d,\ x \geq 0\}$$

To project z onto C, we must solve:

$$\Pi_C(z) = \arg \min_{Ax \leq b,\, Cx = d,\, x \geq 0} \|x - z\|^2.$$

That's a **quadratic program**. At *every single iteration* of gradient descent.

When Projection is Hard

Boxes and balls have cheap, closed-form projections. Great!

But what about **general linear constraints**?

$$C = \{x : Ax \leq b,\ Cx = d,\ x \geq 0\}$$

To project z onto C, we must solve:

$$\Pi_C(z) = \arg \min_{Ax \leq b,\, Cx = d,\, x \geq 0} \|x - z\|^2.$$

That's a **quadratic program**. At *every single iteration* of gradient descent.

For **nonlinear constraints** $g(x) \leq 0$: even defining the projection may be intractable.

When Projection is Hard

Boxes and balls have cheap, closed-form projections. Great!

But what about **general linear constraints**?

$$C = \{x : Ax \leq b,\ Cx = d,\ x \geq 0\}$$

To project z onto C, we must solve:

$$\Pi_C(z) = \arg \min_{Ax \leq b,\, Cx = d,\, x \geq 0} \|x - z\|^2.$$

That's a **quadratic program**. At *every single iteration* of gradient descent.

For **nonlinear constraints** $g(x) \leq 0$: even defining the projection may be intractable.

Bottom line

Projected GD is beautiful when projection is "cheap". But for complex feasible sets, we need something else entirely.

The Idea: Don't Project. Penalize

New idea: forget about the feasible set. Just run *unconstrained* gradient descent, but make it **painful to violate constraints**.

The Idea: Don't Project. Penalize

New idea: forget about the feasible set. Just run *unconstrained* gradient descent, but make it **painful to violate constraints**.

Original problem:

$$\min_x f(x) \quad \text{subject to} \quad g(x) \leq 0.$$

The Idea: Don't Project. Penalize

New idea: forget about the feasible set. Just run *unconstrained* gradient descent, but make it **painful to violate constraints**.

Original problem:

$$\min_x f(x) \quad \text{subject to} \quad g(x) \le 0.$$

Penalty version:

$$\min_x f(x) + \rho \cdot \left[\max(0,\ g(x))\right]^2, \qquad \rho > 0.$$

The Idea: Don't Project. Penalize

New idea: forget about the feasible set. Just run *unconstrained* gradient descent, but make it **painful to violate constraints**.

Original problem:

$$\min_x f(x) \quad \text{subject to} \quad g(x) \leq 0.$$

Penalty version:

$$\min_x f(x) + \rho \cdot \left[\max(0,\ g(x))\right]^2, \qquad \rho > 0.$$

- If $g(x) \leq 0$ (feasible): penalty term $= 0$. No effect.

The Idea: Don't Project. Penalize

New idea: forget about the feasible set. Just run *unconstrained* gradient descent, but make it **painful to violate constraints**.

Original problem:

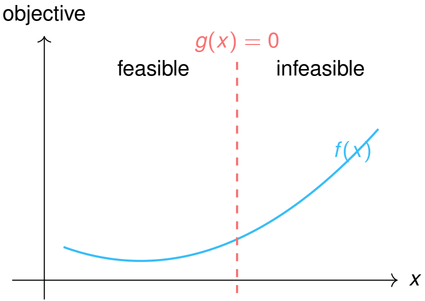
$$\min_x f(x) \quad \text{subject to} \quad g(x) \leq 0.$$

Penalty version:

$$\min_x f(x) + \rho \cdot \left[\max(0,\ g(x))\right]^2, \qquad \rho > 0.$$

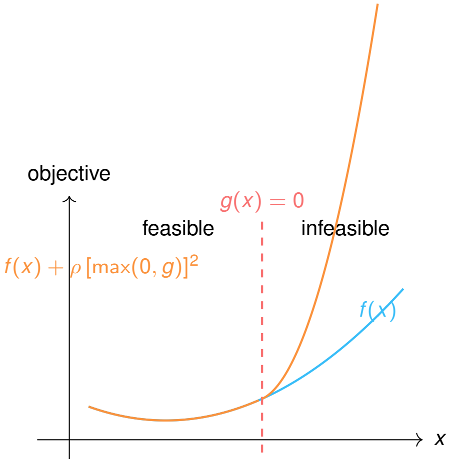
- If $g(x) \leq 0$ (feasible): penalty term $= 0$. No effect.
- If $g(x) > 0$ (violated): $f(x) + \rho g(x)^2$. GD pushes you back.

Penalty Methods: The Picture



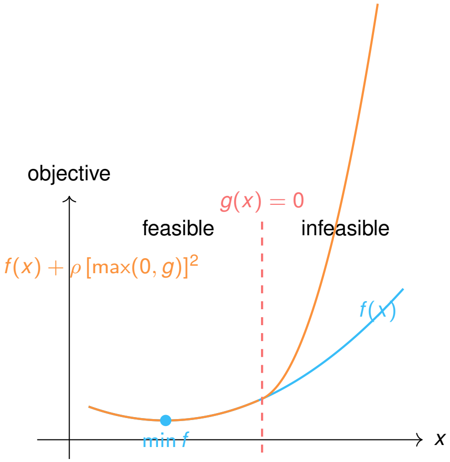
In the feasible region, the orange curve = the blue curve. In the infeasible region, the penalty kicks in and the orange curve shoots up.

Penalty Methods: The Picture

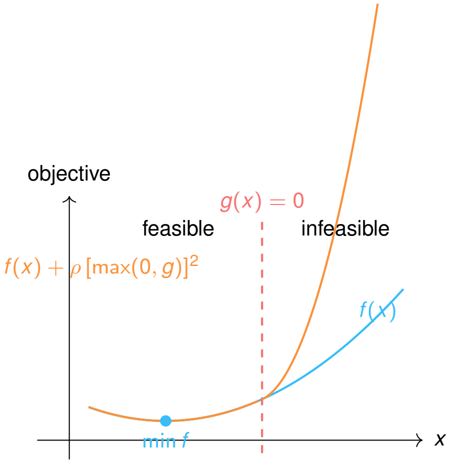


In the feasible region, the orange curve = the blue curve. In the infeasible region, the penalty kicks in and the orange curve shoots up.

Penalty Methods: The Picture



Penalty Methods: The Picture



In the feasible region, the orange curve = the blue curve. In the infeasible region, the penalty kicks in and the orange curve shoots up.

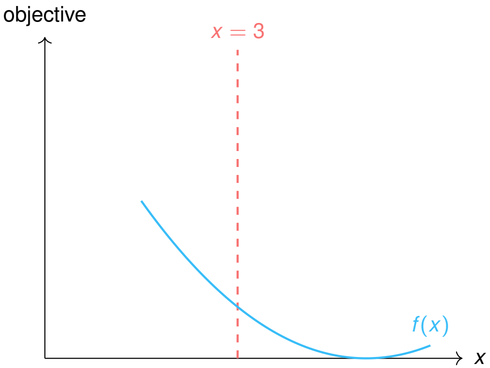
A Concrete Example

$$\min_x \, (x-5)^2 \quad \text{subject to} \quad x \le 3.$$

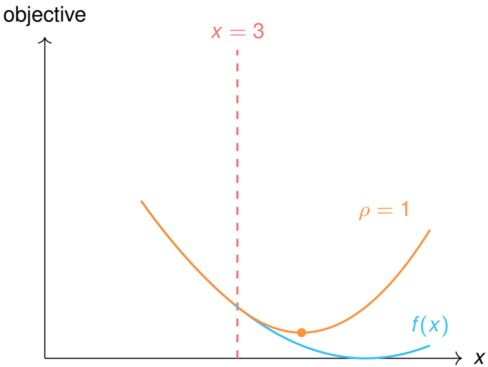
Penalty version with $g(x) = x - 3$:

$$\min_x \, (x-5)^2 + \rho\,[\max(0,\ x-3)]^2.$$

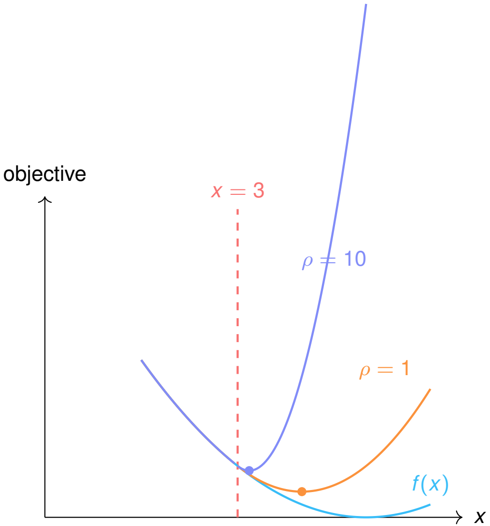
Penalty Methods: Visualizing ρ



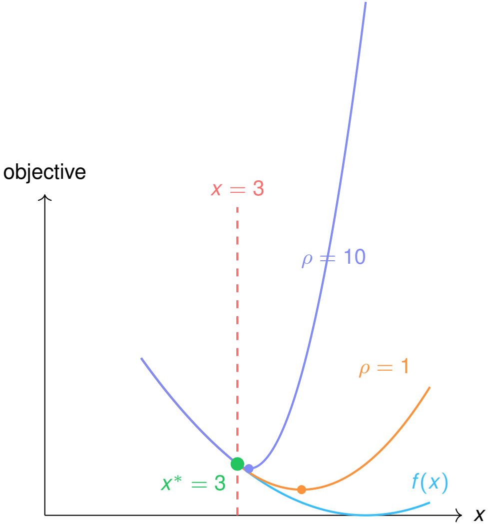
Penalty Methods: Visualizing ρ



Penalty Methods: Visualizing ρ



Penalty Methods: Visualizing ρ



Penalty Method in PyTorch

```
x = torch.tensor([0.0, 0.0], requires_grad=True)
rho = 500.0
alpha = 0.005

for k in range(2000):
    obj = (x[0]-5)**2 + (x[1]-5)**2

    # single constraint: x1 + x2 <= 6  (i.e., g(x) = x1+x2-6 <= 0)
    # g(x) = x1 + x2 - 6 <= 0
    g = x[0] + x[1] - 6
    penalty = torch.maximum(torch.tensor(0.0), g)**2

    loss = obj + rho * penalty
    loss.backward()

    with torch.no_grad():
        x -= alpha * x.grad
        x.grad.zero_()

print(f"x = [{x[0].item():.3f}, {x[1].item():.3f}]")  # [3.002, 3.002]
```

Penalty Method in PyTorch

```
x = torch.tensor([0.0, 0.0], requires_grad=True)
rho = 500.0
alpha = 0.005

for k in range(2000):
    obj = (x[0]-5)**2 + (x[1]-5)**2

    # single constraint: x1 + x2 <= 6  (i.e., g(x) = x1+x2-6 <= 0)
    # g(x) = x1 + x2 - 6 <= 0
    g = x[0] + x[1] - 6
    penalty = torch.maximum(torch.tensor(0.0), g)**2

    loss = obj + rho * penalty
    loss.backward()

    with torch.no_grad():
        x -= alpha * x.grad
        x.grad.zero_()

print(f"x = [{x[0].item():.3f}, {x[1].item():.3f}]")  # [3.002, 3.002]
```

Notice: **no projection step at all**. The penalty does all the work. But we get 3.002, not 3.0. We'd need $\rho \to \infty$ for exactness.

Multiple Constraints? Just Add More Penalties

$$\min_x \, (x_1 - 5)^2 + (x_2 - 5)^2 \quad \text{s.t.} \quad x_1 + x_2 \leq 6, \quad x_1 \geq 0, \quad x_2 \geq 0.$$

Multiple Constraints? Just Add More Penalties

$$\min_{x} (x_1 - 5)^2 + (x_2 - 5)^2 \quad \text{s.t.} \quad x_1 + x_2 \le 6, \quad x_1 \ge 0, \quad x_2 \ge 0.$$

```python
x = torch.tensor([0.0, 0.0], requires_grad=True)
rho = 500.0
alpha = 0.005
for k in range(2000):
    obj = (x[0]-5)**2 + (x[1]-5)**2
    p1 = torch.maximum(torch.zeros(1), x[0] + x[1] - 6)**2
    p2 = torch.maximum(torch.zeros(1), -x[0])**2
    p3 = torch.maximum(torch.zeros(1), -x[1])**2

    loss = obj + rho * (p1 + p2 + p3)
    loss.backward()
    with torch.no_grad():
        x -= alpha * x.grad
        x.grad.zero_()

print(f"x = [{x[0].item():.3f}, {x[1].item():.3f}]")  # ~[3.002, 3.002]
```

The Penalty Method Tradeoff

Pros:

- Turns *any* constrained problem into an unconstrained one.
- Works with gradient descent / PyTorch directly.
- Easy to implement: just add terms to the loss.
- Objective remains convex!

The Penalty Method Tradeoff

Pros:

- Turns *any* constrained problem into an unconstrained one.
- Works with gradient descent / PyTorch directly.
- Easy to implement: just add terms to the loss.
- Objective remains convex!

Cons:

- Need large ρ for accuracy $\Rightarrow$ objective becomes **ill-conditioned**.

The Penalty Method Tradeoff

Pros:

- Turns *any* constrained problem into an unconstrained one.
- Works with gradient descent / PyTorch directly.
- Easy to implement: just add terms to the loss.
- Objective remains convex!

Cons:

- Need large ρ for accuracy $\Rightarrow$ objective becomes **ill-conditioned**.
- Gradient descent slows down: the penalty creates steep cliffs and tiny steps.

The Penalty Method Tradeoff

Pros:

- Turns *any* constrained problem into an unconstrained one.
- Works with gradient descent / PyTorch directly.
- Easy to implement: just add terms to the loss.
- Objective remains convex!

Cons:

- Need large ρ for accuracy $\Rightarrow$ objective becomes **ill-conditioned**.
- Gradient descent slows down: the penalty creates steep cliffs and tiny steps.
- For $\rho = 100$: $x \approx 3.02$. For $\rho = 10000$: $x = nan$.

The Penalty Method Tradeoff

Pros:

- Turns *any* constrained problem into an unconstrained one.
- Works with gradient descent / PyTorch directly.
- Easy to implement: just add terms to the loss.
- Objective remains convex!

Cons:

- Need large ρ for accuracy $\Rightarrow$ objective becomes **ill-conditioned**.
- Gradient descent slows down: the penalty creates steep cliffs and tiny steps.
- For $\rho = 100$: $x \approx 3.02$. For $\rho = 10000$: $x = nan$.

The question

Is there a smarter version of “penalize violations” that doesn’t require $\rho \to \infty$?

From Penalty to Lagrangian: The Key Insight

Penalty method:

$$\min_x \; f(x) + \rho \left[\max(0, g(x))\right]^2.$$

ρ is *fixed by us*. Guess too low $\Rightarrow$ infeasible. Guess too high $\Rightarrow$ ill-conditioned.

From Penalty to Lagrangian: The Key Insight

Penalty method:

$$\min_{x} \; f(x) + \rho \, [\max(0, g(x))]^2.$$

ρ is *fixed by us*. Guess too low $\Rightarrow$ infeasible. Guess too high $\Rightarrow$ ill-conditioned.

Lagrangian idea: what if instead of a fixed ρ, we let the **multiplier learn the right price**?

From Penalty to Lagrangian: The Key Insight

Penalty method:

$$\min_x \; f(x) + \rho \, [\max(0, g(x))]^2.$$

ρ is *fixed by us*. Guess too low $\Rightarrow$ infeasible. Guess too high $\Rightarrow$ ill-conditioned.

Lagrangian idea: what if instead of a fixed ρ, we let the **multiplier learn the right price**?

$$\mathcal{L}(x, \lambda) = f(x) + \lambda \cdot g(x), \qquad \lambda \geq 0.$$

From Penalty to Lagrangian: The Key Insight

Penalty method:

$$\min_x \; f(x) + \rho\,[\max(0, g(x))]^2.$$

ρ is *fixed by us*. Guess too low $\Rightarrow$ infeasible. Guess too high $\Rightarrow$ ill-conditioned.

Lagrangian idea: what if instead of a fixed ρ, we let the **multiplier learn the right price**?

$$\mathcal{L}(x, \lambda) = f(x) + \lambda \cdot g(x), \qquad \lambda \geq 0.$$

- λ is not chosen by us: it's a **variable** that we optimize.

From Penalty to Lagrangian: The Key Insight

Penalty method:

$$\min_x \ f(x) + \rho\,[\max(0, g(x))]^2.$$

ρ is *fixed by us*. Guess too low $\Rightarrow$ infeasible. Guess too high $\Rightarrow$ ill-conditioned.

Lagrangian idea: what if instead of a fixed ρ, we let the **multiplier learn the right price**?

$$\mathcal{L}(x, \lambda) = f(x) + \lambda \cdot g(x), \qquad \lambda \geq 0.$$

- λ is not chosen by us: it's a **variable** that we optimize.
- We **minimize** over x (make the objective small).

From Penalty to Lagrangian: The Key Insight

Penalty method:

$$\min_x \; f(x) + \rho \, [\max(0, g(x))]^2.$$

ρ is *fixed by us*. Guess too low $\Rightarrow$ infeasible. Guess too high $\Rightarrow$ ill-conditioned.

Lagrangian idea: what if instead of a fixed ρ, we let the **multiplier learn the right price**?

$$\mathcal{L}(x, \lambda) = f(x) + \lambda \cdot g(x), \qquad \lambda \geq 0.$$

- λ is not chosen by us: it's a **variable** that we optimize.
- We **minimize** over x (make the objective small).
- We **maximize** over λ (make constraint violations expensive).

What's the trick?

Consider what happens when we maximize over $\lambda \geq 0$:

$$\max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \max_{\lambda \geq 0} \left(f(x) + \lambda\, g(x)\right).$$

What's the trick?

Consider what happens when we maximize over $\lambda \geq 0$:

$$\max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \max_{\lambda \geq 0} (f(x) + \lambda\, g(x)).$$

If $g(x) \leq 0$ (feasible): best $\lambda = 0$, so max $= f(x)$.

What's the trick?

Consider what happens when we maximize over $\lambda \geq 0$:

$$\max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \max_{\lambda \geq 0} \left(f(x) + \lambda\, g(x)\right).$$

If $g(x) \leq 0$ (feasible): best $\lambda = 0$, so max $= f(x)$.
If $g(x) > 0$ (infeasible): can send $\lambda \to \infty$, so max $= +\infty$.

What's the trick?

Consider what happens when we maximize over $\lambda \geq 0$:

$$\max_{\lambda \geq 0} \; \mathcal{L}(x, \lambda) = \max_{\lambda \geq 0} \left(f(x) + \lambda \, g(x) \right).$$

If $g(x) \leq 0$ (feasible): best $\lambda = 0$, so max $= f(x)$.
If $g(x) > 0$ (infeasible): can send $\lambda \to \infty$, so max $= +\infty$.

Therefore:

$$\max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \begin{cases} f(x) & \text{if } x \text{ is feasible} \\ +\infty & \text{if } x \text{ is infeasible} \end{cases}$$

What's the trick?

Consider what happens when we maximize over $\lambda \geq 0$:

$$\max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \max_{\lambda \geq 0} (f(x) + \lambda\, g(x)).$$

If $g(x) \leq 0$ (feasible): best $\lambda = 0$, so max $= f(x)$.
If $g(x) > 0$ (infeasible): can send $\lambda \to \infty$, so max $= +\infty$.

Therefore:

$$\max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \begin{cases} f(x) & \text{if } x \text{ is feasible} \\ +\infty & \text{if } x \text{ is infeasible} \end{cases}$$

So:

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \min_{x \text{ feasible}} f(x) = f^*.$$

What's the trick?

Consider what happens when we maximize over $\lambda \geq 0$:

$$\max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \max_{\lambda \geq 0} \left(f(x) + \lambda\, g(x)\right).$$

If $g(x) \leq 0$ (feasible): best $\lambda = 0$, so max $= f(x)$.
If $g(x) > 0$ (infeasible): can send $\lambda \to \infty$, so max $= +\infty$.

Therefore:

$$\max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \begin{cases} f(x) & \text{if } x \text{ is feasible} \\ +\infty & \text{if } x \text{ is infeasible} \end{cases}$$

So:

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \min_{x \text{ feasible}} f(x) = f^*.$$

The Lagrangian min-max is **exactly** the original constrained problem!

Primal-Dual Gradient Descent

We want to solve:

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \min_{x} \max_{\lambda \geq 0} (f(x) + \lambda\, g(x)).$$

Primal-Dual Gradient Descent

We want to solve:

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \min_{x} \max_{\lambda \geq 0} (f(x) + \lambda\, g(x)).$$

Idea: Start at some (x_0, λ_0), then alternate between two steps every iteration.

Primal-Dual Gradient Descent

We want to solve:

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \min_{x} \max_{\lambda \geq 0} (f(x) + \lambda\, g(x)).$$

Idea: Start at some (x_0, λ_0), then alternate between two steps every iteration.

Step 1 — Primal update (gradient *descent* in x):

$$x_{k+1} = x_k - \alpha\, \nabla_x \mathcal{L}(x_k, \lambda_k) = x_k - \alpha\left(\nabla f(x_k) + \lambda_k \nabla g(x_k)\right).$$

Primal-Dual Gradient Descent

We want to solve:

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \min_{x} \max_{\lambda \geq 0} (f(x) + \lambda\, g(x)).$$

Idea: Start at some (x_0, λ_0), then alternate between two steps every iteration.

Step 1 — Primal update (gradient *descent* in x):

$$x_{k+1} = x_k - \alpha\, \nabla_x \mathcal{L}(x_k, \lambda_k) = x_k - \alpha \left(\nabla f(x_k) + \lambda_k \nabla g(x_k) \right).$$

Step 2 — Dual update (gradient *ascent* in λ): We want

$$\lambda_{k+1} = \lambda_k + \alpha\, \nabla_\lambda \mathcal{L}(x_k, \lambda_k) = \lambda_k + \alpha\, g(x_k) \qquad \times$$

Primal-Dual Gradient Descent

We want to solve:

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = \min_{x} \max_{\lambda \geq 0} (f(x) + \lambda\, g(x)).$$

Idea: Start at some (x_0, λ_0), then alternate between two steps every iteration.

Step 1 — Primal update (gradient *descent* in x):

$$x_{k+1} = x_k - \alpha\, \nabla_x \mathcal{L}(x_k, \lambda_k) = x_k - \alpha \left(\nabla f(x_k) + \lambda_k \nabla g(x_k) \right).$$

Step 2 — Dual update (gradient *ascent* in λ): We want

$$\lambda_{k+1} = \lambda_k + \alpha\, \nabla_\lambda \mathcal{L}(x_k, \lambda_k) = \lambda_k + \alpha\, g(x_k)$$

But project on $\lambda \geq 0$ (clamp!):

$$\lambda_{k+1} = \max\big(0, \lambda_k + \alpha\, g(x_k)\big)$$

Primal–Dual Gradient Method: Update Rules Summary

We solve

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = f(x) + \lambda g(x).$$

Initialize x_0, λ_0.

At iteration k:

Primal–Dual Gradient Method: Update Rules Summary

We solve

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = f(x) + \lambda g(x).$$

Initialize x_0, λ_0.

At iteration k:
Primal step (descent in x):

$$x_{k+1} = x_k - \alpha \left(\nabla f(x_k) + \lambda_k \nabla g(x_k) \right)$$

Primal–Dual Gradient Method: Update Rules Summary

We solve

$$\min_{x} \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = f(x) + \lambda g(x).$$

Initialize x_0, λ_0.

At iteration k:

Primal step (descent in x):

$$x_{k+1} = x_k - \alpha \left(\nabla f(x_k) + \lambda_k \nabla g(x_k) \right)$$

Dual step (ascent in λ):

$$\lambda_{k+1} = \max\big(0, \lambda_k + \alpha\, g(x_k)\big)$$

Interpretation

- $g(x_k) > 0$ (constraint violated) $\Rightarrow \lambda$ increases.
- $g(x_k) < 0$ (constraint slack) $\Rightarrow \lambda$ decreases toward 0.

Concrete Example: Setup

$$\min_x \,(x-5)^2 \quad \text{s.t.} \quad x \le 3.$$

Concrete Example: Setup

$$\min_{x} \; (x-5)^2 \quad \text{s.t.} \quad x \leq 3.$$

Constraint: $g(x) = x - 3 \leq 0$.

Concrete Example: Setup

$$\min_{x} \; (x-5)^2 \quad \text{s.t.} \quad x \le 3.$$

Constraint: $g(x) = x - 3 \le 0$.

Lagrangian:

$$\mathcal{L}(x,\lambda) = (x-5)^2 + \lambda(x-3).$$

Concrete Example: Setup

$$\min_{x} \; (x-5)^2 \quad \text{s.t.} \quad x \le 3.$$

Constraint: $g(x) = x - 3 \le 0$.

Lagrangian:

$$\mathcal{L}(x,\lambda) = (x-5)^2 + \lambda(x-3).$$

Gradients:

$$\frac{\partial \mathcal{L}}{\partial x} = 2(x-5) + \lambda$$

Concrete Example: Setup

$$\min_x \, (x-5)^2 \quad \text{s.t.} \quad x \leq 3.$$

Constraint: $g(x) = x - 3 \leq 0$.

Lagrangian:

$$\mathcal{L}(x,\lambda) = (x-5)^2 + \lambda(x-3).$$

Gradients:

$$\frac{\partial \mathcal{L}}{\partial x} = 2(x-5) + \lambda$$

Updates:

$$x_{k+1} = x_k - \alpha\,[2(x_k - 5) + \lambda_k] \qquad \lambda_{k+1} = \max(0,\ \lambda_k + \alpha\,(x_k - 3))$$

Lagrangian Gradient Descent in PyTorch

```
import torch

x = torch.tensor(0.0, requires_grad=True)
lam = torch.tensor(0.0)
alpha = 0.05

for k in range(500):
    L = (x - 5)**2 + lam * (x-3)             # Lagrangian
    L.backward()                            # computes dL/dx automatically

    with torch.no_grad():
        x_old = x.clone()
        x -= alpha * x.grad                 # primal step: descend in x
        lam = torch.clamp(lam + alpha * (x_old-3), min=0)  # dual step: ascend in lambda
        x.grad.zero_()

print(f"x = {x.item():.4f}, lambda = {lam.item():.4f}")
# x ~ 3.0, lambda ~ 4.0
```

Lagrangian Gradient Descent in PyTorch

```
import torch

x = torch.tensor(0.0, requires_grad=True)
lam = torch.tensor(0.0)
alpha = 0.05

for k in range(500):
    L = (x - 5)**2 + lam * (x-3)              # Lagrangian
    L.backward()                             # computes dL/dx automatically

    with torch.no_grad():
        x_old = x.clone()
        x -= alpha * x.grad                  # primal step: descend in x
        lam = torch.clamp(lam + alpha * (x_old-3), min=0)  # dual step: ascend in lambda
        x.grad.zero_()

print(f"x = {x.item():.4f}, lambda = {lam.item():.4f}")
# x ~ 3.0, lambda ~ 4.0
```

No $\rho \to \infty$ needed. The multiplier λ **finds the right price** by itself.

The Lagrangian: General Form

Problem:

$$\min_x \; f(x) \quad \text{s.t. } g_i(x) \le 0 \; (i = 1, \ldots, m), \quad h_j(x) = 0 \; (j = 1, \ldots, p).$$

The Lagrangian: General Form

Problem:

$$\min_x \; f(x) \quad \text{s.t. } g_i(x) \leq 0 \; (i = 1, \ldots, m), \quad h_j(x) = 0 \; (j = 1, \ldots, p).$$

Lagrangian:

$$\boxed{\mathcal{L}(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i \, g_i(x) + \sum_{j=1}^{p} \nu_j \, h_j(x)}$$

The Lagrangian: General Form

Problem:

$$\min_x \; f(x) \quad \text{s.t. } g_i(x) \le 0 \; (i = 1, \ldots, m), \quad h_j(x) = 0 \; (j = 1, \ldots, p).$$

Lagrangian:

$$\boxed{\mathcal{L}(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i \, g_i(x) + \sum_{j=1}^{p} \nu_j \, h_j(x)}$$

- $\lambda_i \ge 0$: multiplier ("price") for inequality $g_i(x) \le 0$.
- $\nu_j \in \mathbb{R}$: multiplier for equality $h_j(x) = 0$.

The Lagrangian: General Form

Problem:

$$\min_{x} \; f(x) \quad \text{s.t. } g_i(x) \leq 0 \; (i = 1, \ldots, m), \quad h_j(x) = 0 \; (j = 1, \ldots, p).$$

Lagrangian:

$$\boxed{\mathcal{L}(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i \, g_i(x) + \sum_{j=1}^{p} \nu_j \, h_j(x)}$$

- $\lambda_i \geq 0$: multiplier ("price") for inequality $g_i(x) \leq 0$.
- $\nu_j \in \mathbb{R}$: multiplier for equality $h_j(x) = 0$.

Optimization:

$$\min_{x} \max_{\lambda \geq \mathbf{0},\, \nu} \mathcal{L}(x, \lambda, \nu) = f^*.$$

Primal–Dual Updates: General Case

We solve

$$\min_{x} \max_{\lambda \geq 0,\, \nu} \left(f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \nu_j h_j(x) \right).$$

Primal–Dual Updates: General Case

We solve

$$\min_{x} \max_{\lambda \geq 0,\, \nu} \left(f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \nu_j h_j(x) \right).$$

Primal update (descent in x)

$$x^{(k+1)} = x^{(k)} - \alpha\Big(\nabla f(x^{(k)}) + \sum_{i=1}^{m} \lambda_i^{(k)} \nabla g_i(x^{(k)}) + \sum_{j=1}^{p} \nu_j^{(k)} \nabla h_j(x^{(k)})\Big)$$

Primal–Dual Updates: General Case

We solve

$$\min_{x} \max_{\lambda \geq 0,\, \nu} \left(f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \nu_j h_j(x) \right).$$

Primal update (descent in x)

$$x^{(k+1)} = x^{(k)} - \alpha \Big(\nabla f(x^{(k)}) + \sum_{i=1}^{m} \lambda_i^{(k)} \nabla g_i(x^{(k)}) + \sum_{j=1}^{p} \nu_j^{(k)} \nabla h_j(x^{(k)}) \Big)$$

Dual updates (ascent)

$$\lambda_i^{(k+1)} = \max\Big(0,\; \lambda_i^{(k)} + \alpha\, g_i(x^{(k)})\Big), \qquad i = 1, \ldots, m$$

$$\nu_j^{(k+1)} = \nu_j^{(k)} + \alpha\, h_j(x^{(k)}), \qquad j = 1, \ldots, p$$

Primal–Dual Updates: General Case

We solve

$$\min_{x}\ \max_{\lambda\geq 0,\ \nu}\left(f(x)+\sum_{i=1}^{m}\lambda_i g_i(x)+\sum_{j=1}^{p}\nu_j h_j(x)\right).$$

Primal update (descent in x)

$$x^{(k+1)}=x^{(k)}-\alpha\left(\nabla f(x^{(k)})+\sum_{i=1}^{m}\lambda_i^{(k)}\nabla g_i(x^{(k)})+\sum_{j=1}^{p}\nu_j^{(k)}\nabla h_j(x^{(k)})\right)$$

Dual updates (ascent)

$$\lambda_i^{(k+1)}=\max\left(0,\ \lambda_i^{(k)}+\alpha\, g_i(x^{(k)})\right),\qquad i=1,\ldots,m$$

$$\nu_j^{(k+1)}=\nu_j^{(k)}+\alpha\, h_j(x^{(k)}),\qquad j=1,\ldots,p$$

Convex Example: 2 Inequalities + 1 Equality

$$\min_{x} \, (x_1 - 5)^2 + (x_2 - 5)^2$$

$$\text{s.t.} \quad \underbrace{x_1 + x_2 \le 6}_{g_1(x) \le 0}, \qquad \underbrace{x_1^2 + x_2^2 \le 25}_{g_2(x) \le 0}, \qquad \underbrace{x_1 - x_2 = 0}_{h(x)=0}.$$

Primal–Dual Gradient Method

```
x = torch.tensor([0.0, 0.0], requires_grad=True)
lam = torch.tensor([0.0, 0.0])  # inequality multipliers (>=0)
nu = torch.tensor(0.0)   # equality multiplier
alpha = 0.02

for _ in range(2000):
    L = (x[0]-5)**2 + (x[1]-5)**2 \
            + lam[0]*(x[0]+x[1]-6)  \
            + lam[1]*(x[0]**2 + x[1]**2 - 25) \
            + nu * (x[0] - x[1])
    L.backward()
```

Primal–Dual Gradient Method

```
x = torch.tensor([0.0, 0.0], requires_grad=True)
lam = torch.tensor([0.0, 0.0])  # inequality multipliers (>=0)
nu = torch.tensor(0.0)   # equality multiplier
alpha = 0.02

for _ in range(2000):
    L = (x[0]-5)**2 + (x[1]-5)**2 \
            + lam[0]*(x[0]+x[1]-6)  \
            + lam[1]*(x[0]**2 + x[1]**2 - 25) \
            + nu * (x[0] - x[1])
    L.backward()
    with torch.no_grad():
        x_old = x.clone()
        x -= alpha * x.grad        # descent
```

Primal–Dual Gradient Method

```
x = torch.tensor([0.0, 0.0], requires_grad=True)
lam = torch.tensor([0.0, 0.0])  # inequality multipliers (>=0)
nu = torch.tensor(0.0)   # equality multiplier
alpha = 0.02

for _ in range(2000):
    L = (x[0]-5)**2 + (x[1]-5)**2 \
            + lam[0]*(x[0]+x[1]-6)  \
            + lam[1]*(x[0]**2 + x[1]**2 - 25) \
            + nu * (x[0] - x[1])
    L.backward()
    with torch.no_grad():
        x_old = x.clone()
        x -= alpha * x.grad        # descent

        lam[0] = torch.clamp(lam[0] + alpha * (x_old[0]+x_old[1]-6), min=0) # ascent & project
```

Primal–Dual Gradient Method

```
x = torch.tensor([0.0, 0.0], requires_grad=True)
lam = torch.tensor([0.0, 0.0])  # inequality multipliers (>=0)
nu = torch.tensor(0.0)   # equality multiplier
alpha = 0.02

for _ in range(2000):
    L = (x[0]-5)**2 + (x[1]-5)**2 \
            + lam[0]*(x[0]+x[1]-6)  \
            + lam[1]*(x[0]**2 + x[1]**2 - 25) \
            + nu * (x[0] - x[1])
    L.backward()
    with torch.no_grad():
        x_old = x.clone()
        x -= alpha * x.grad        # descent
        lam[0] = torch.clamp(lam[0] + alpha * (x_old[0]+x_old[1]-6), min=0) # ascent & project
        lam[1] = torch.clamp(lam[1] + alpha * (x_old[0]**2 + x_old[1]**2 - 25), min=0) # ascent & project
```

Primal–Dual Gradient Method

```python
x = torch.tensor([0.0, 0.0], requires_grad=True)
lam = torch.tensor([0.0, 0.0])  # inequality multipliers (>=0)
nu = torch.tensor(0.0)   # equality multiplier
alpha = 0.02

for _ in range(2000):
    L = (x[0]-5)**2 + (x[1]-5)**2 \
            + lam[0]*(x[0]+x[1]-6)  \
            + lam[1]*(x[0]**2 + x[1]**2 - 25) \
            + nu * (x[0] - x[1])
    L.backward()
    with torch.no_grad():
        x_old = x.clone()
        x -= alpha * x.grad        # descent
        lam[0] = torch.clamp(lam[0] + alpha * (x_old[0]+x_old[1]-6), min=0) # ascent & project
        lam[1] = torch.clamp(lam[1] + alpha * (x_old[0]**2 + x_old[1]**2 - 25), min=0) # ascent & project
        nu += alpha *(x_old[0] - x_old[1])   # ascent (no need to project)
```

Primal–Dual Gradient Method

```
x = torch.tensor([0.0, 0.0], requires_grad=True)
lam = torch.tensor([0.0, 0.0])  # inequality multipliers (>=0)
nu = torch.tensor(0.0)   # equality multiplier
alpha = 0.02

for _ in range(2000):
    L = (x[0]-5)**2 + (x[1]-5)**2 \
            + lam[0]*(x[0]+x[1]-6)  \
            + lam[1]*(x[0]**2 + x[1]**2 - 25) \
            + nu * (x[0] - x[1])
    L.backward()
    with torch.no_grad():
        x_old = x.clone()
        x -= alpha * x.grad        # descent
        lam[0] = torch.clamp(lam[0] + alpha * (x_old[0]+x_old[1]-6), min=0) # ascent & project
        lam[1] = torch.clamp(lam[1] + alpha * (x_old[0]**2 + x_old[1]**2 - 25), min=0) # ascent & project
        nu += alpha *(x_old[0] - x_old[1])   # ascent (no need to project)
        x.grad.zero_()
print(x) #tensor([3.0000, 3.0000], requires_grad=True)
print(lam) #tensor([4.0000, 0.0000])
print(nu) # tensor(0.)
```

KKT: When Are We at the Optimum?

We've been running gradient descent to *find* the solution. The **KKT conditions** tell us how to *check* it.

KKT: When Are We at the Optimum?

We've been running gradient descent to *find* the solution. The **KKT conditions** tell us how to *check* it.

For convex problems, (x^*, λ^*, ν^*) is optimal **if and only if**:

KKT: When Are We at the Optimum?

We've been running gradient descent to *find* the solution. The **KKT conditions** tell us how to *check* it.

For convex problems, (x^*, λ^*, ν^*) is optimal **if and only if**:

1. **Stationarity:** $\nabla_x f(x^*) + \sum_i \lambda_i^* \nabla g_i(x^*) + \sum_j \nu_j^* \nabla h_j(x^*) = 0$

KKT: When Are We at the Optimum?

We've been running gradient descent to *find* the solution. The **KKT conditions** tell us how to *check* it.

For convex problems, (x^*, λ^*, ν^*) is optimal **if and only if**:

1. **Stationarity:** $\nabla_x f(x^*) + \sum_i \lambda_i^* \nabla g_i(x^*) + \sum_j \nu_j^* \nabla h_j(x^*) = 0$
2. **Primal feasibility:** $g_i(x^*) \leq 0, \quad h_j(x^*) = 0$

KKT: When Are We at the Optimum?

We've been running gradient descent to *find* the solution. The **KKT conditions** tell us how to *check* it.

For convex problems, (x^*, λ^*, ν^*) is optimal **if and only if**:

1. **Stationarity:** $\nabla_x f(x^*) + \sum_i \lambda_i^* \nabla g_i(x^*) + \sum_j \nu_j^* \nabla h_j(x^*) = 0$
2. **Primal feasibility:** $g_i(x^*) \leq 0, \quad h_j(x^*) = 0$
3. **Dual feasibility:** $\lambda_i^* \geq 0$

KKT: When Are We at the Optimum?

We've been running gradient descent to *find* the solution. The **KKT conditions** tell us how to *check* it.

For convex problems, (x^*, λ^*, ν^*) is optimal **if and only if**:

1. **Stationarity:** $\nabla_x f(x^*) + \sum_i \lambda_i^* \nabla g_i(x^*) + \sum_j \nu_j^* \nabla h_j(x^*) = 0$
2. **Primal feasibility:** $g_i(x^*) \leq 0, \quad h_j(x^*) = 0$
3. **Dual feasibility:** $\lambda_i^* \geq 0$
4. **Complementary slackness:** $\lambda_i^* g_i(x^*) = 0$

KKT: When Are We at the Optimum?

We've been running gradient descent to *find* the solution. The **KKT conditions** tell us how to *check* it.

For convex problems, (x^*, λ^*, ν^*) is optimal **if and only if**:

1. **Stationarity:** $\nabla_x f(x^*) + \sum_i \lambda_i^* \nabla g_i(x^*) + \sum_j \nu_j^* \nabla h_j(x^*) = 0$
2. **Primal feasibility:** $g_i(x^*) \leq 0, \quad h_j(x^*) = 0$
3. **Dual feasibility:** $\lambda_i^* \geq 0$
4. **Complementary slackness:** $\lambda_i^* g_i(x^*) = 0$

KKT Check: Our Running Example

$\min(x - 5)^2$ s.t. $x \leq 3$. Solution: $x^* = 3$, $\lambda^* = 4$.

KKT Check: Our Running Example

$\min(x-5)^2$ s.t. $x \leq 3$. Solution: $x^* = 3$, $\lambda^* = 4$.

1. **Stationarity:** $\nabla f + \lambda^* \nabla g = 2(3-5) + 4 \cdot 1 = 0$. ✓

KKT Check: Our Running Example

$\min(x-5)^2$ s.t. $x \leq 3$. Solution: $x^* = 3$, $\lambda^* = 4$.

1. **Stationarity:** $\nabla f + \lambda^* \nabla g = 2(3-5) + 4 \cdot 1 = 0$. ✓
2. **Primal feasibility:** $g(x^*) = 3 - 3 = 0 \leq 0$. ✓

KKT Check: Our Running Example

$\min(x-5)^2$ s.t. $x \leq 3$. Solution: $x^* = 3$, $\lambda^* = 4$.

1. **Stationarity:** $\nabla f + \lambda^* \nabla g = 2(3-5) + 4 \cdot 1 = 0$. $\checkmark$
2. **Primal feasibility:** $g(x^*) = 3 - 3 = 0 \leq 0$. $\checkmark$
3. **Dual feasibility:** $\lambda^* = 4 \geq 0$. $\checkmark$

KKT Check: Our Running Example

$\min(x-5)^2$ s.t. $x \leq 3$. Solution: $x^* = 3$, $\lambda^* = 4$.

1. **Stationarity:** $\nabla f + \lambda^* \nabla g = 2(3-5) + 4 \cdot 1 = 0$. ✓
2. **Primal feasibility:** $g(x^*) = 3 - 3 = 0 \leq 0$. ✓
3. **Dual feasibility:** $\lambda^* = 4 \geq 0$. ✓
4. **Complementary slackness:** $\lambda^* \cdot g(x^*) = 4 \cdot 0 = 0$. ✓

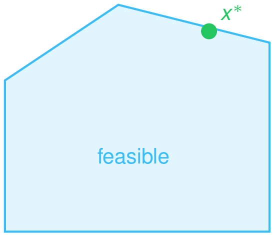
KKT Check: Our Running Example

$\min(x-5)^2$ s.t. $x \leq 3$. Solution: $x^* = 3$, $\lambda^* = 4$.

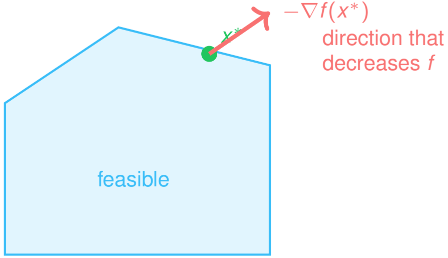
1. **Stationarity:** $\nabla f + \lambda^* \nabla g = 2(3-5) + 4 \cdot 1 = 0$. ✓
2. **Primal feasibility:** $g(x^*) = 3 - 3 = 0 \leq 0$. ✓
3. **Dual feasibility:** $\lambda^* = 4 \geq 0$. ✓
4. **Complementary slackness:** $\lambda^* \cdot g(x^*) = 4 \cdot 0 = 0$. ✓

All four conditions hold $\Rightarrow$ $(3, 4)$ is **certifiably optimal**.

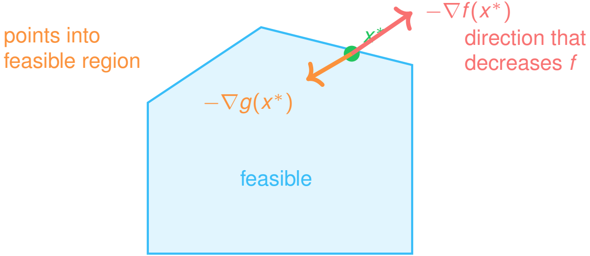
KKT: Geometric Picture



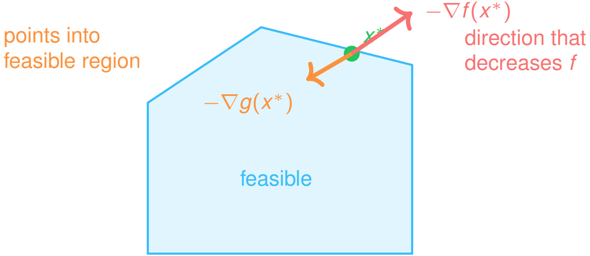
KKT: Geometric Picture



KKT: Geometric Picture



KKT: Geometric Picture



Stationarity: $\nabla f + \lambda^* \nabla g = 0$

Every improving direction violates a constraint. Stuck.

Looking Ahead: Semidefinite Programming

Prof. **Chandra** will spend two lectures on **semidefinite programming (SDP)**.

Looking Ahead: Semidefinite Programming

Prof. **Chandra** will spend two lectures on **semidefinite programming (SDP)**.

SDPs are a powerful class of convex optimization problems:

- Generalize linear programs.

Looking Ahead: Semidefinite Programming

Prof. **Chandra** will spend two lectures on **semidefinite programming (SDP)**.

SDPs are a powerful class of convex optimization problems:

- Generalize linear programs.
- The variable is a **matrix** $X \succeq 0$ (positive semidefinite).

Looking Ahead: Semidefinite Programming

Prof. **Chandra** will spend two lectures on **semidefinite programming (SDP)**.

SDPs are a powerful class of convex optimization problems:

- Generalize linear programs.
- The variable is a **matrix** $X \succeq 0$ (positive semidefinite).
- Applications: Max-Cut, graph partitioning, relaxations of combinatorial problems.

Looking Ahead: Semidefinite Programming

Prof. **Chandra** will spend two lectures on **semidefinite programming (SDP)**.

SDPs are a powerful class of convex optimization problems:

- Generalize linear programs.
- The variable is a **matrix** $X \succeq 0$ (positive semidefinite).
- Applications: Max-Cut, graph partitioning, relaxations of combinatorial problems.

In Homework 4, you solved it using row generation and linear programming. In practice, convex optimization solvers are used instead because they are faster.

Looking Ahead: Semidefinite Programming

Prof. **Chandra** will spend two lectures on **semidefinite programming (SDP)**.

SDPs are a powerful class of convex optimization problems:

- Generalize linear programs.
- The variable is a **matrix** $X \succeq 0$ (positive semidefinite).
- Applications: Max-Cut, graph partitioning, relaxations of combinatorial problems.

In Homework 4, you solved it using row generation and linear programming. In practice, convex optimization solvers are used instead because they are faster.

To solve SDPs in practice, we use **CVXPY**: a Python library for convex optimization. Let me show you the bare minimum you'll need.

CVXPY for SDPs

An SDP optimizes over **positive semidefinite matrices**:

$$\min \langle C, X \rangle \quad \text{s.t.} \quad \langle A_i, X \rangle = b_i, \quad X \succeq 0.$$

CVXPY for SDPs

An SDP optimizes over **positive semidefinite matrices**:

$$\min \langle C, X \rangle \quad \text{s.t.} \quad \langle A_i, X \rangle = b_i, \quad X \succeq 0.$$

```
import cvxpy as cp
import numpy as np
np.random.seed(0)
n = 5
C = np.random.randn(n, n)
C = (C + C.T) / 2
A1, A2 = np.eye(n), np.ones((n, n))
b1, b2 = 1.0, 0.0

X = cp.Variable((n, n), symmetric=True)
constraints = [
    X >> 0, #positive semidefinite constraint
    cp.trace(A1 @ X) == b1, #Trace(A1 @ X) = <A1, X>
    cp.trace(A2 @ X) == b2 #Trace(A2 @ X) = <A2, X>
]
objective = cp.Minimize(cp.trace(C @ X))  #Trace(C @ X) = <C, X>
prob = cp.Problem(objective, constraints)
prob.solve(solver=cp.SCS, eps=1e-4, max_iters=300000)
```