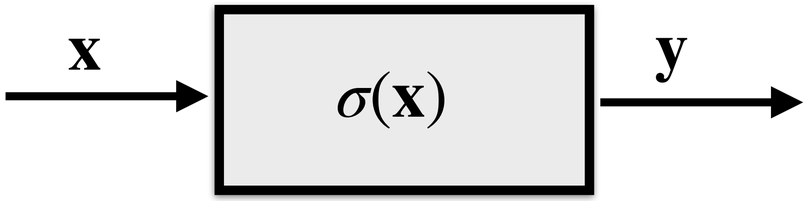
Implicit layers

Backpropagation through root finder, constrained and unconstrained optimization and ODE solvers

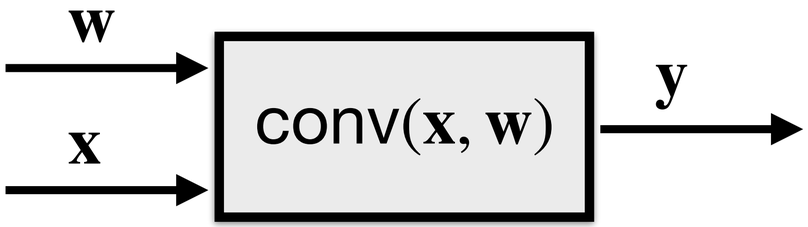
Karel Zimmermann

Explicit layers

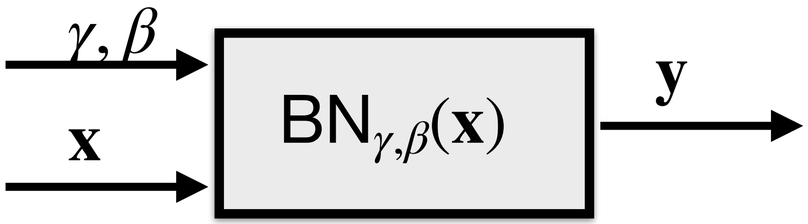
Sigmoid:



Convolution:



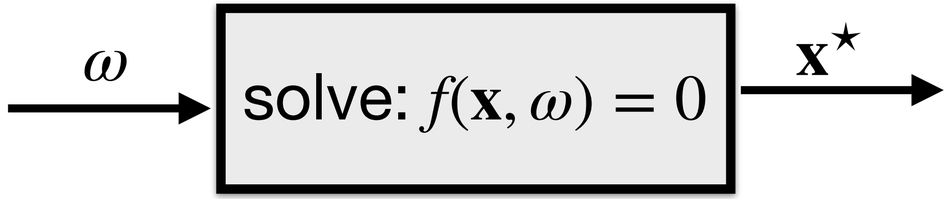
Batch-norm:



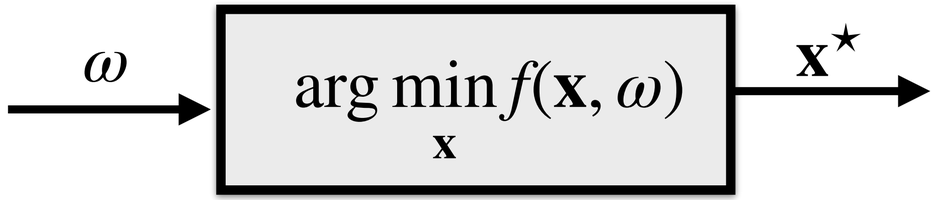
Can I use autodiff on the problem solver?

Implicit layers

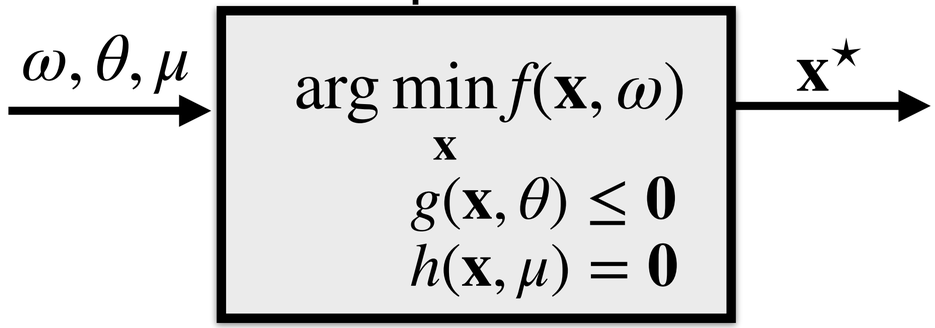
Root finder:



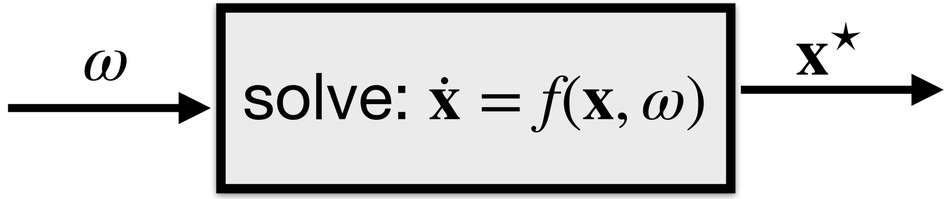
Unconstrained optimizer:



Constrained optimizer:



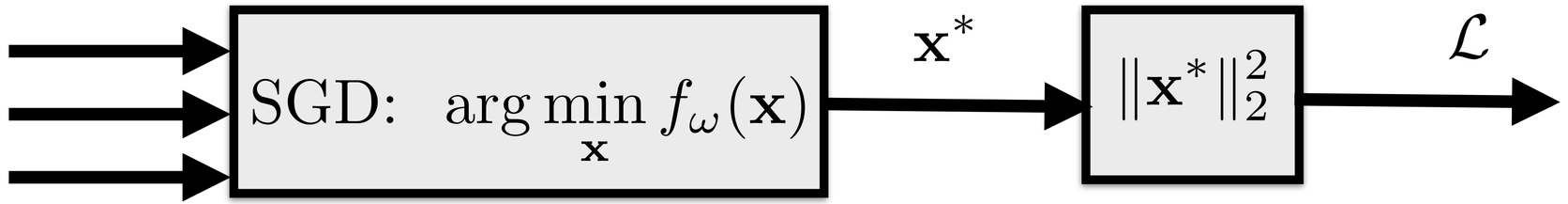
ODE solver:

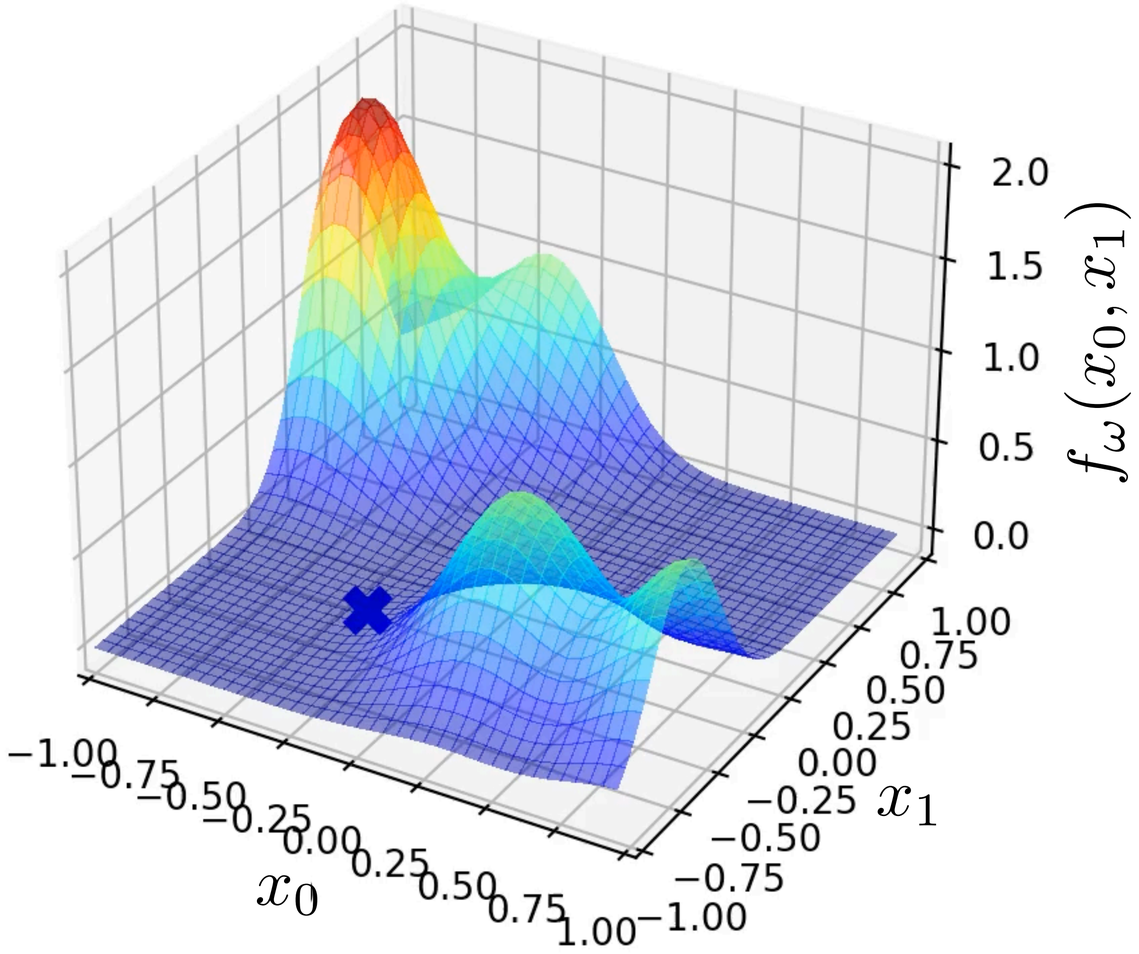


Backpropagation through the solver of an optimization problem

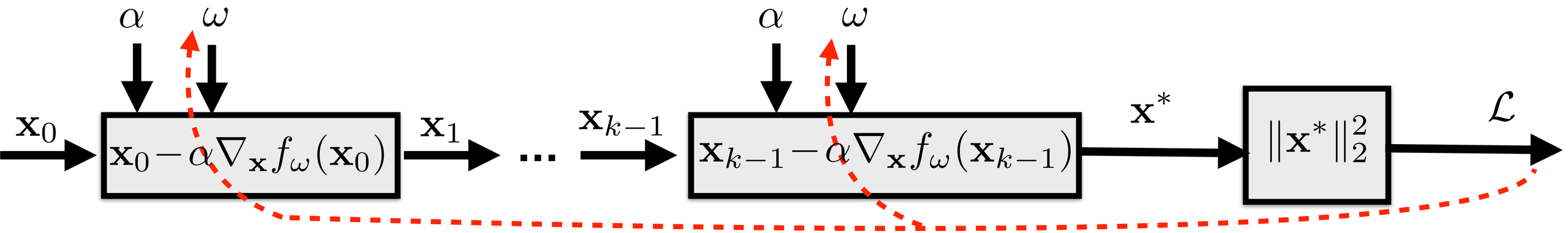
$$\omega^* = \arg\min_{\omega} \|\text{SGD:} \quad \arg\min_{\mathbf{x}} f_\omega(\mathbf{x})\|^2$$

$\mathbf{x}_0$… initial point
ω… criterion params
α, β… optimizer params



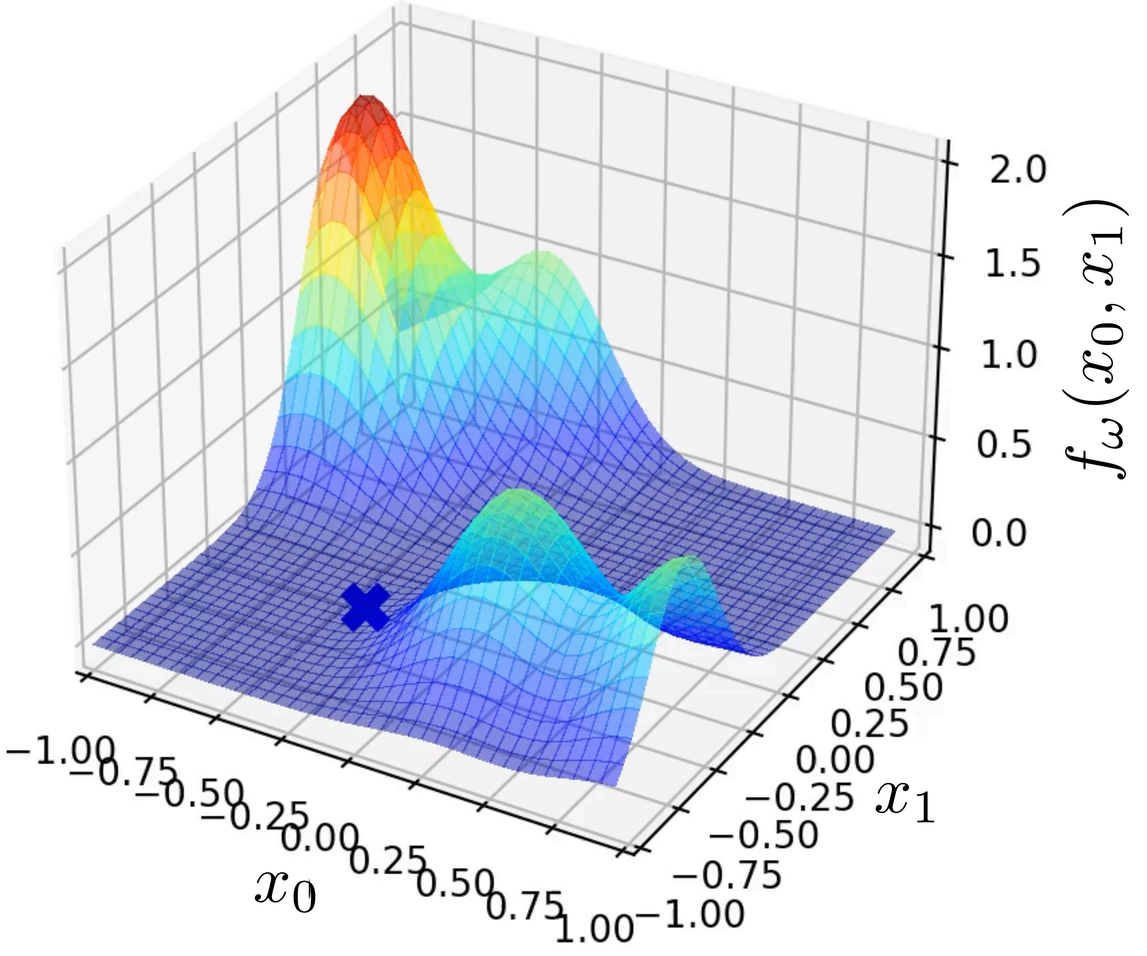


Backpropagation through the solver of an optimization problem



PyTorch:

```
for i in range(iter):
  x = x - torch.autograd.grad(f, x, …
                  create_graph=True,…
                  retain_graph=True)
loss = x.norm()
torch.autograd.grad(loss, omega)
```



Naive straightforward solution

- **Feedforward** pass through the **implicit layer** (solver) is a **code**
  => Its computational graph can be **backpropagated via autograd**.
- **Disadvantage**:
  - Problems of autodiff in complex solvers (“if” has often zero or undefined grad)

```
def solver(x, w):
    if x>w:
        y = 2*x
    else:
        y = sin(x)
    return y
```

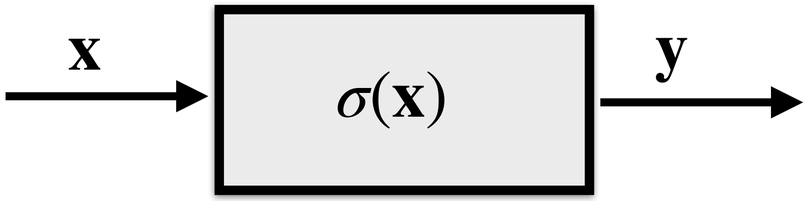
$$\frac{\partial \text{solver}(\mathbf{x}, \omega)}{\partial \omega} = \begin{cases} \mathbf{0} & \mathbf{x} \neq \omega \\ \text{nan} & \mathbf{x} = \omega \end{cases}$$

  - Memory and computationaly inefficient (e.g. iterations of simplex method)

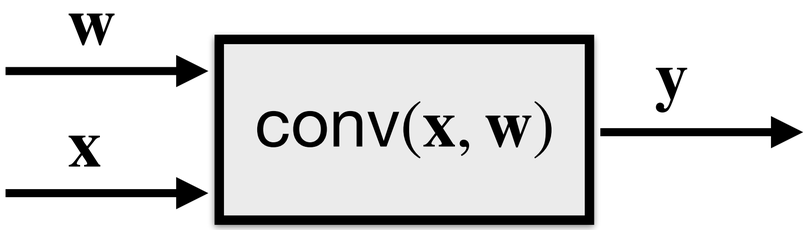
- **Advantage**:
  - It can backpropagate towards internal parameters of the solver and optimize the optimization itself.

Explicit layers

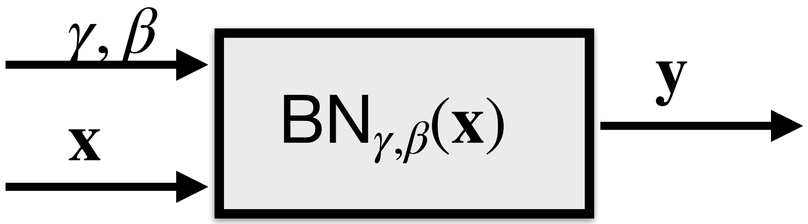
Sigmoid:



Convolution:

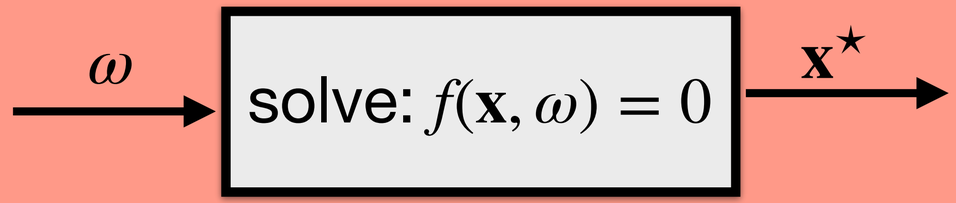


Batch-norm:

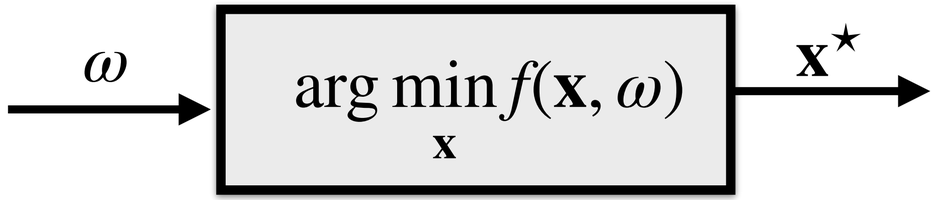


Implicit layers

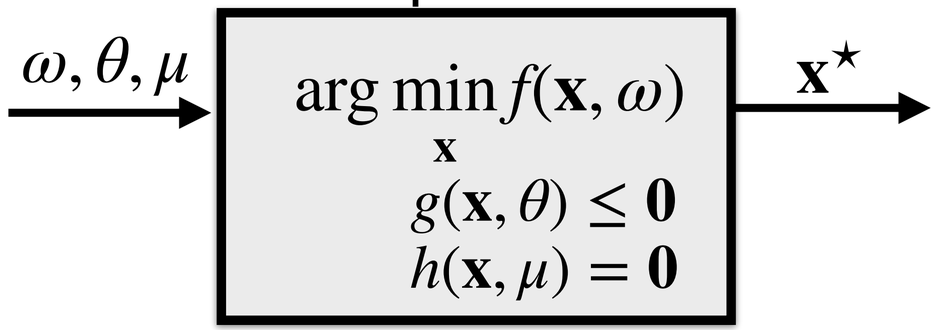
Root finder:



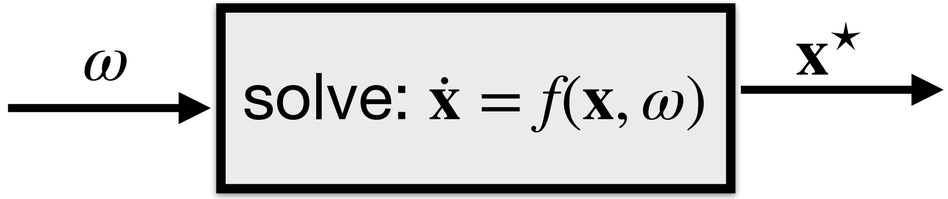
Unconstrained optimizer:



Constrained optimizer:



ODE solver:



Root finder

- Let's assume that root finder is a **compiled binary** w/o the access to its source
- Given parameters ω, the code delivers solution $\mathbf{x}^\star$ such that $f(\mathbf{x}^\star, \omega) = 0$
- This **solution depends on parameters**, we refer it $\mathbf{x}^\star(\omega) : \mathbb{R}^n \to \mathbb{R}^m$
- We want to find $\dfrac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = ?$ can we do better than a numerical differentiations?

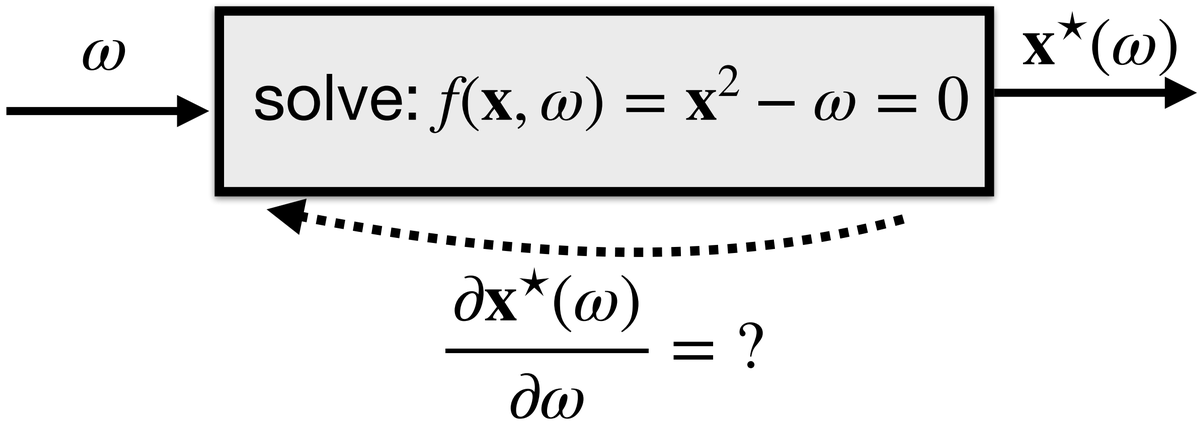
$$f\Big(\mathbf{x}^\star(\omega), \omega\Big) = 0$$

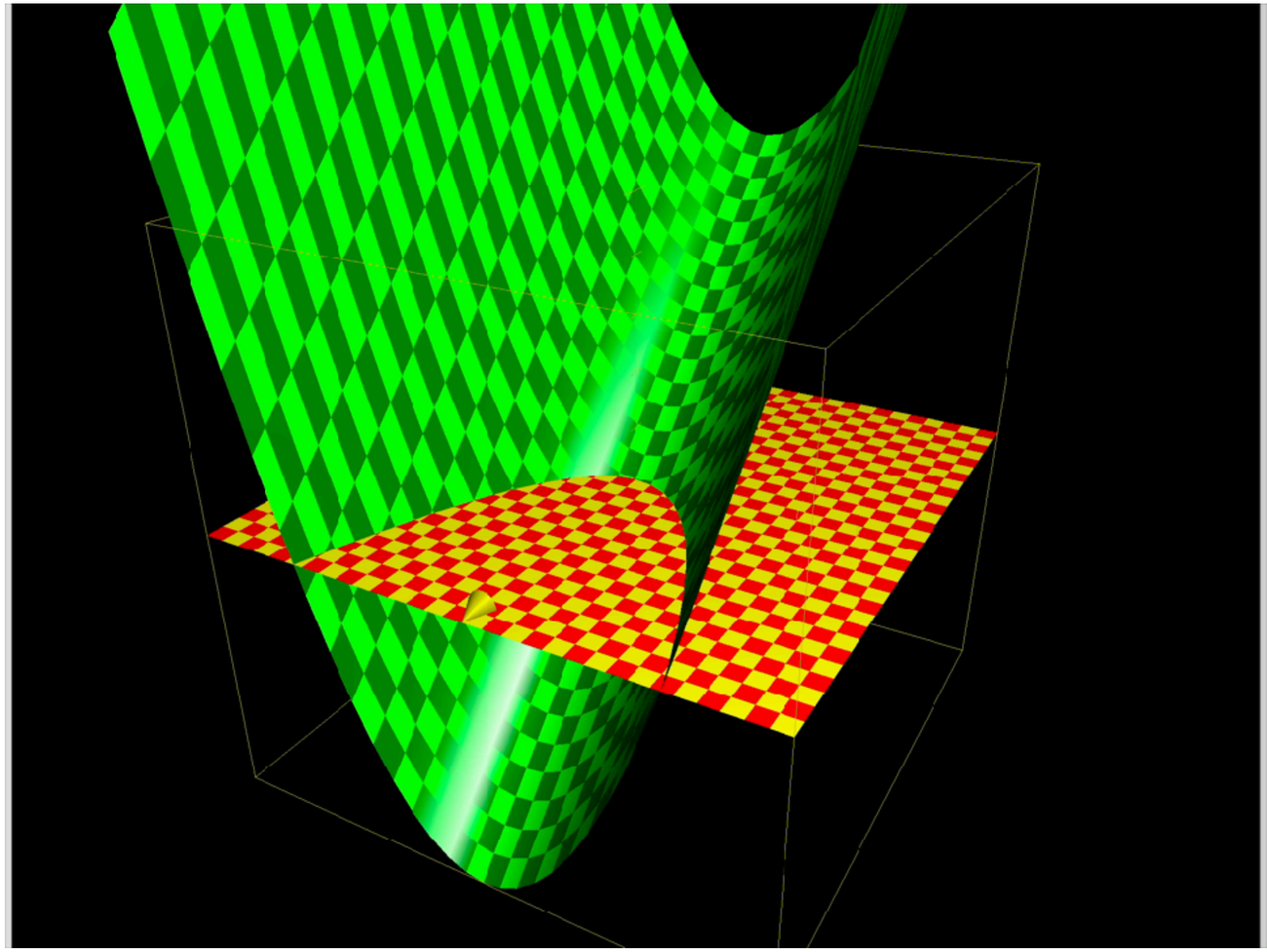
Given ω, the root function $\mathbf{x}^\star(\omega)$ satisfy this equation.

$$\frac{\partial f\Big(\mathbf{x}^\star(\omega), \omega\Big)}{\partial \omega} = 0$$

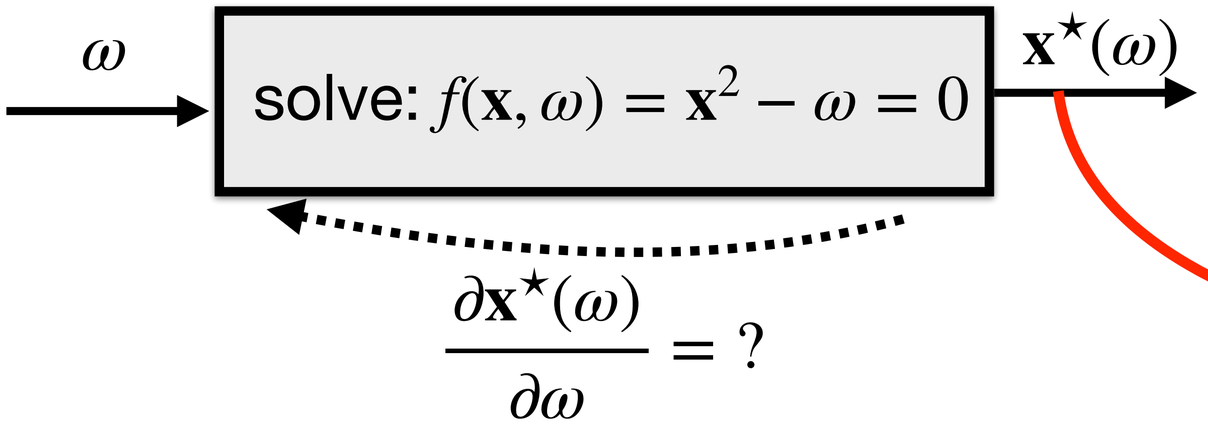
ω is allowed to **change only in directions** which **does not change** the value of $f\Big(\mathbf{x}^\star(\omega), \omega\Big)$ in order to stay within the solution manifold

Example:


ω
solve: $f(\mathbf{x}, \omega) = \mathbf{x}^2 - \omega = 0$
$\mathbf{x}^\star(\omega)$
$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = ?$

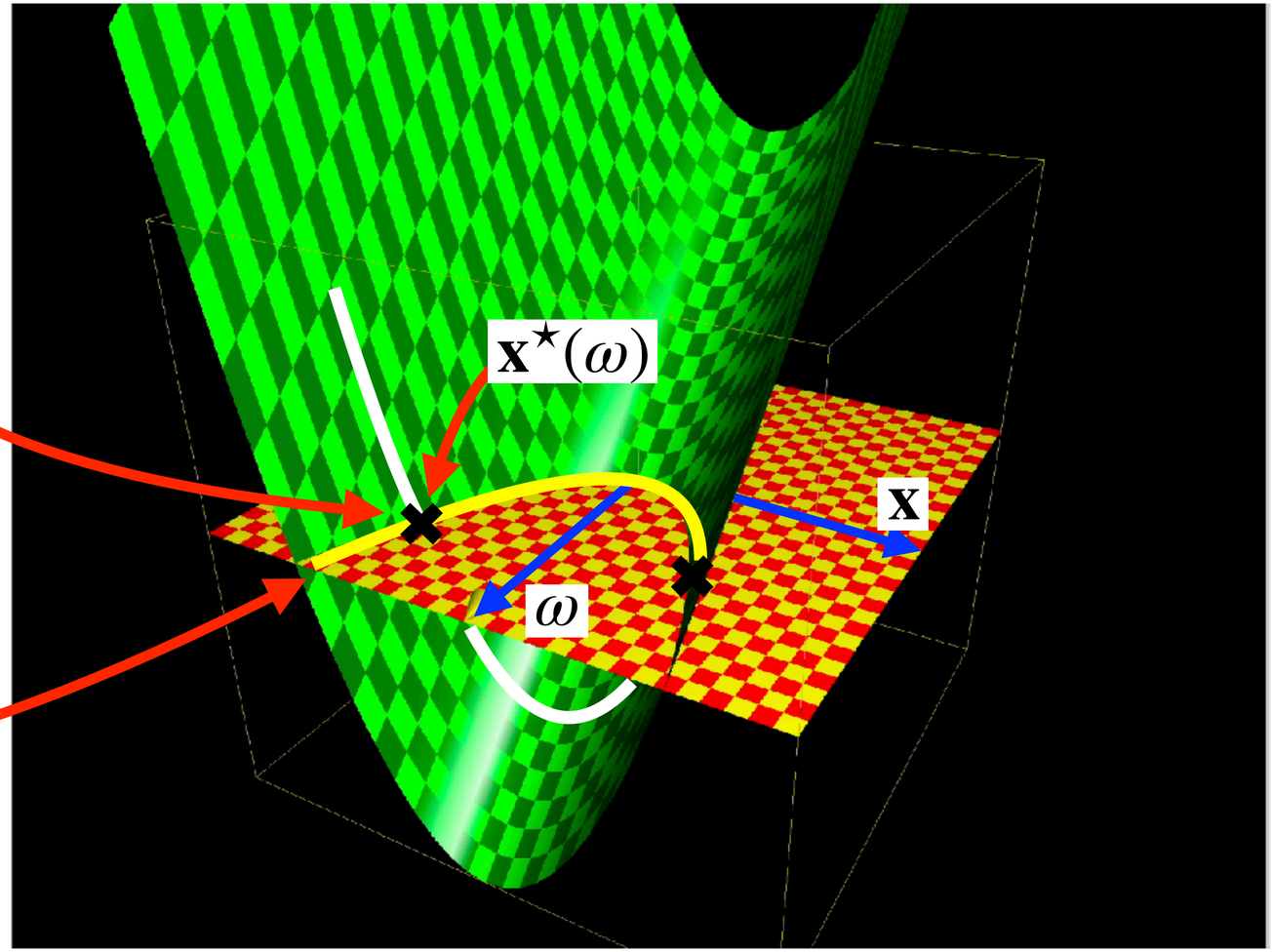


Example:

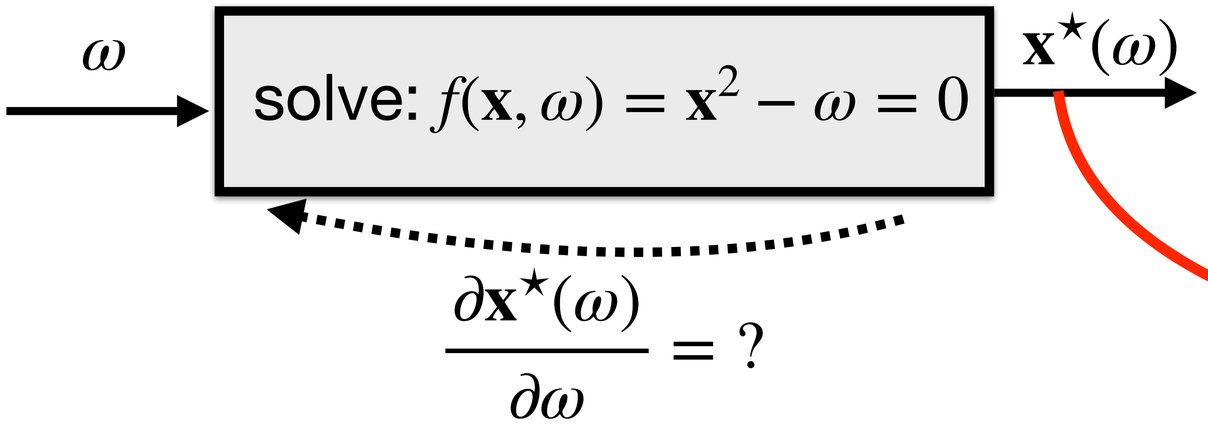


While changing ω, root function $\mathbf{x}^\star(\omega)$ follows the yellow curve

$$f\left(\mathbf{x}^\star(\omega), \omega\right) = \mathbf{x}^\star(\omega)^2 - \omega = 0$$

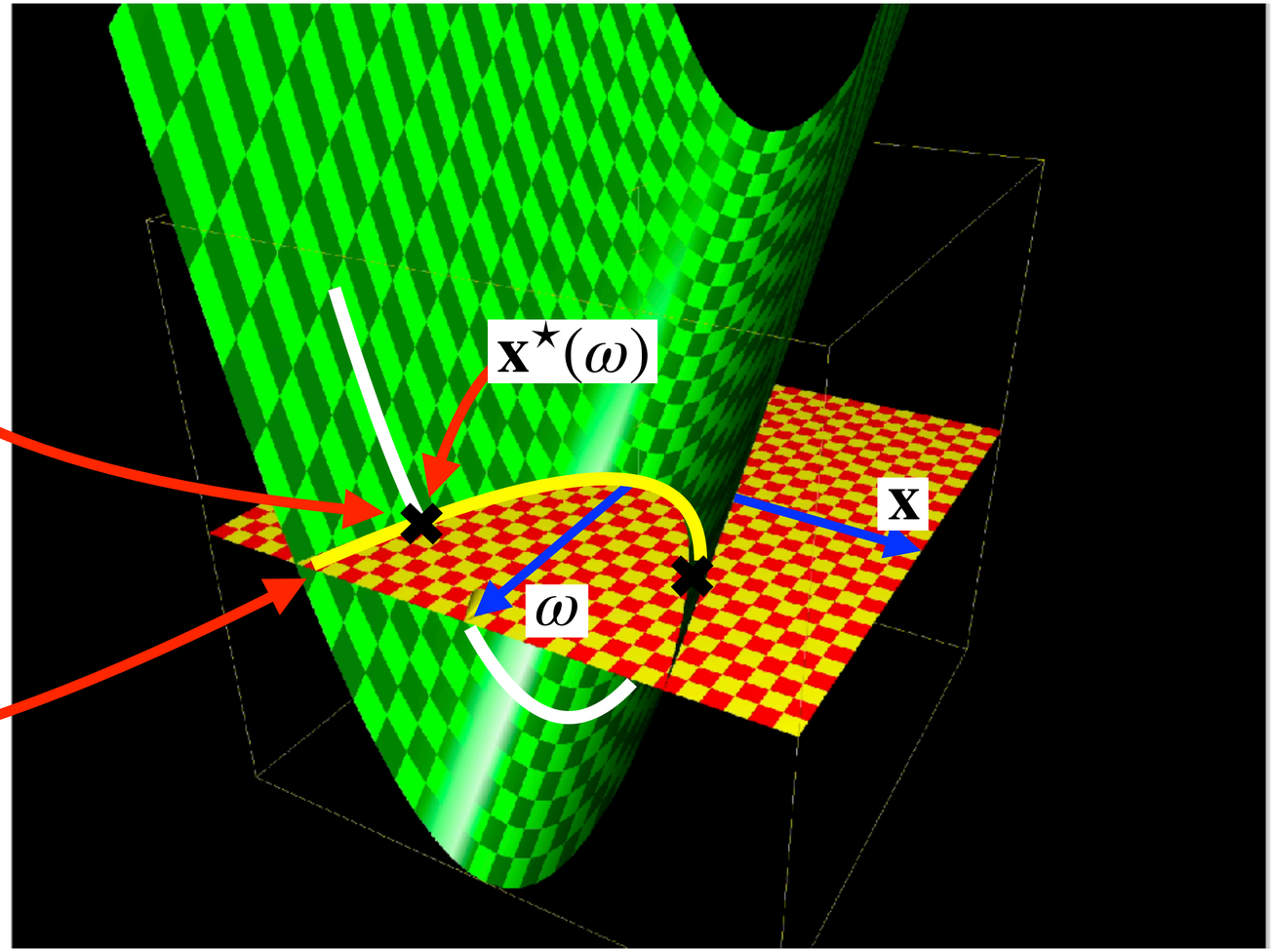


Example:

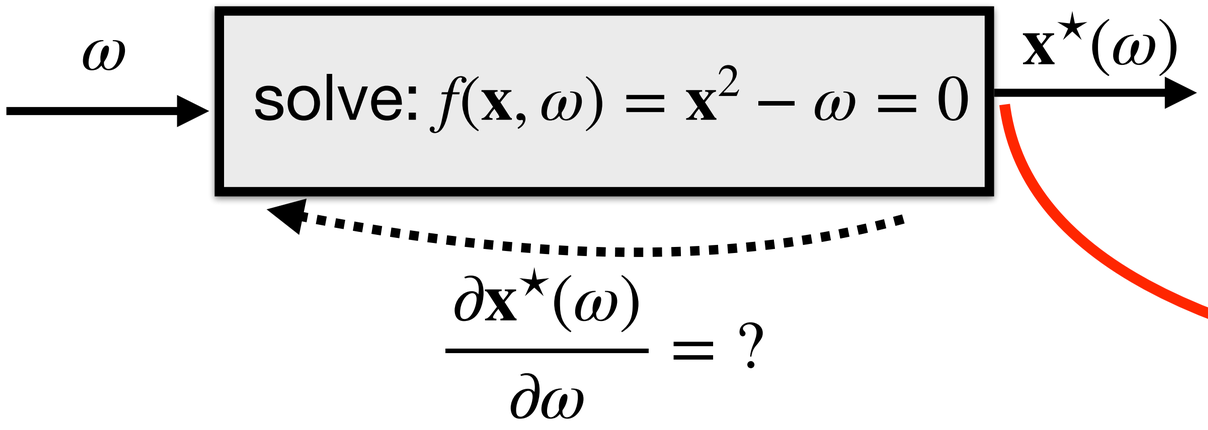


While changing ω, root function $\mathbf{x}^\star(\omega)$ follows the yellow curve

$$f\left(\mathbf{x}^\star(\omega), \omega\right) = \mathbf{x}^\star(\omega)^2 - \omega = 0$$

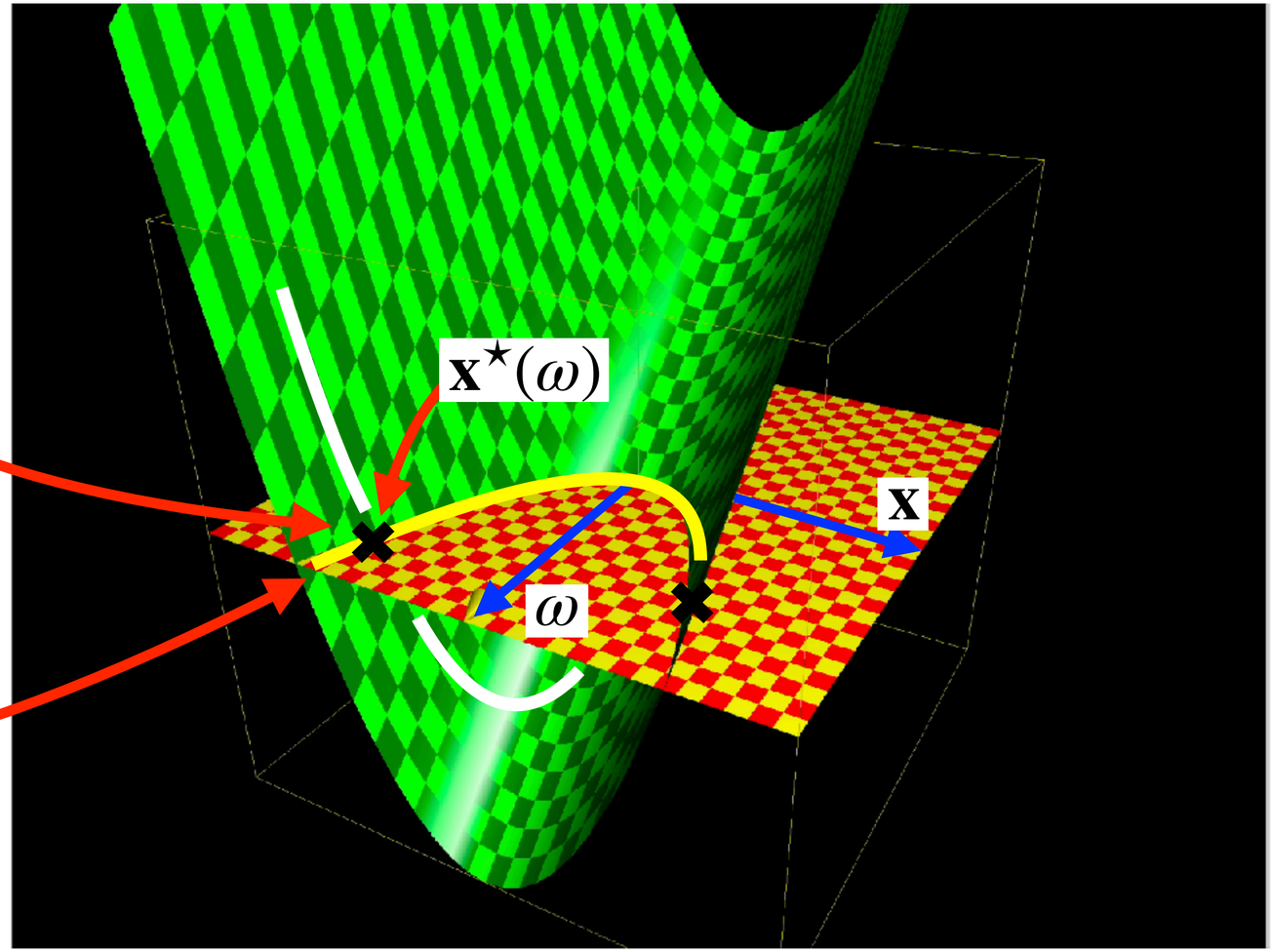


Example:

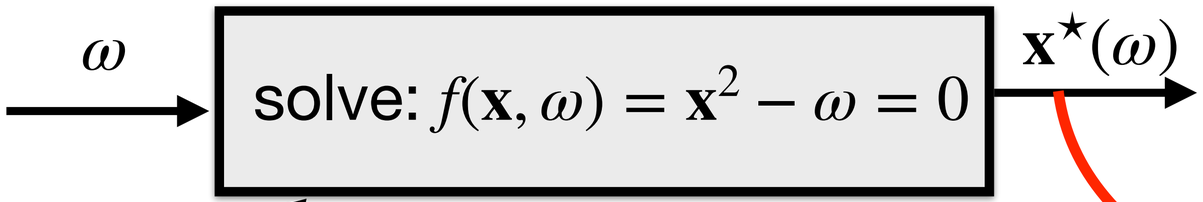


While changing ω, root function $\mathbf{x}^\star(\omega)$ follows the yellow curve

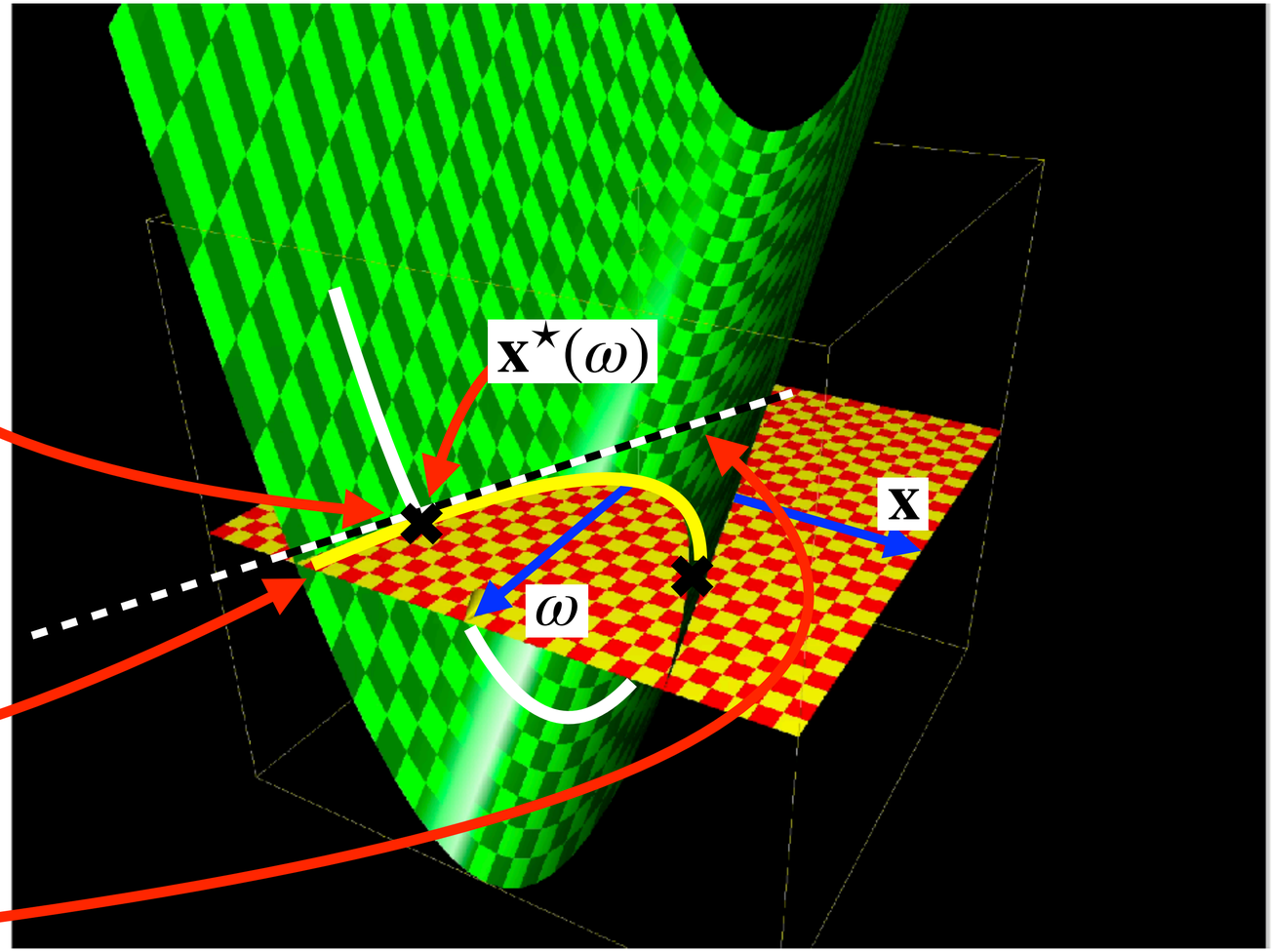
$$f\left(\mathbf{x}^\star(\omega), \omega\right) = \mathbf{x}^\star(\omega)^2 - \omega = 0$$



Example:



$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = ?$$



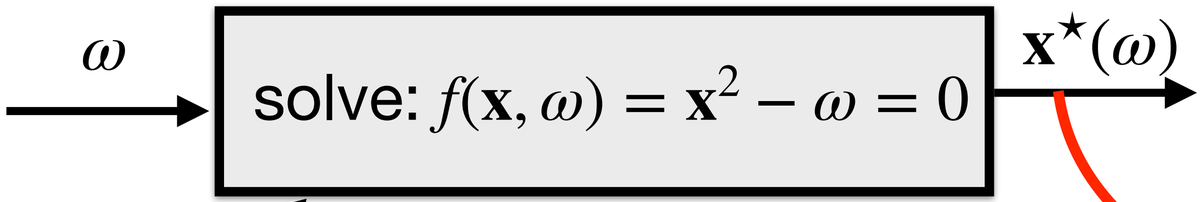
While changing ω, root function $\mathbf{x}^\star(\omega)$ follows the yellow curve

$$f\left(\mathbf{x}^\star(\omega), \omega\right) = \mathbf{x}^\star(\omega)^2 - \omega = 0$$

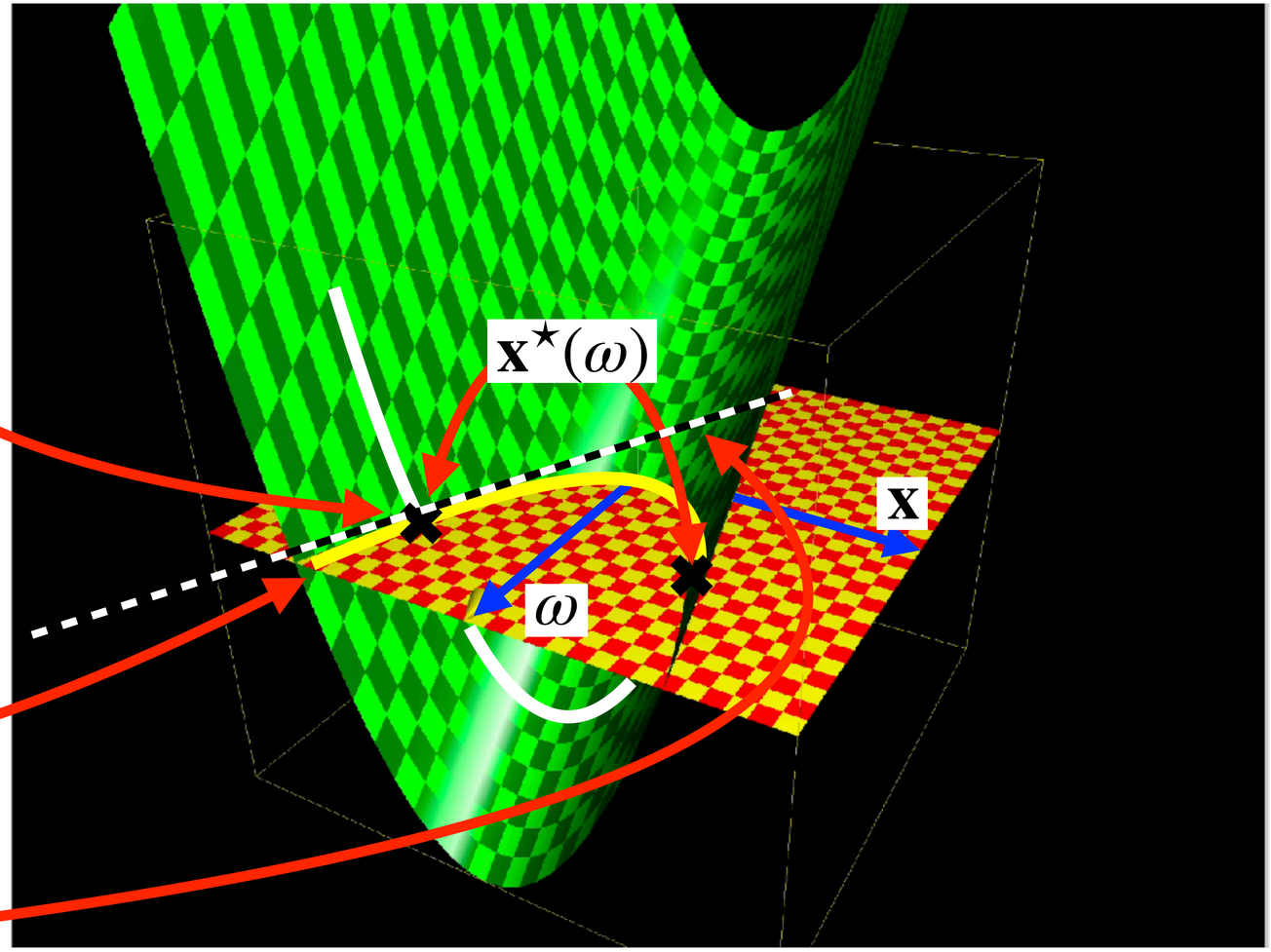
$$\frac{\partial f\left(\mathbf{x}^\star(\omega), \omega\right)}{\partial \omega} = 0$$

Yellow space is locally defined as the direction in ω-domain which locally preserves zero value of $f\left(\mathbf{x}^\star(\omega), \omega\right)$ function

Example:



$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = ?$$



While changing ω, root function $\mathbf{x}^\star(\omega)$ follows the yellow curve

$$f\left(\mathbf{x}^\star(\omega),\omega\right)=\mathbf{x}^\star(\omega)^2-\omega=0$$

$$\frac{\partial f\left(\mathbf{x}^\star(\omega),\omega\right)}{\partial \omega}=0$$

Yellow space is locally defined as the direction in ω-domain which locally preserves zero value of $f\left(\mathbf{x}^\star(\omega),\omega\right)$ function

$$2\mathbf{x}^\star(\omega)\cdot\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega}-1=0 \quad\Rightarrow\quad \boxed{\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega}=\frac{1}{2\mathbf{x}^\star(\omega)}}$$

Can you compute gradient ???

How is it for $\omega=0$???

How is it for other root ???

Root finder

$$\frac{\partial f\left(\mathbf{x}^{\star}(\omega),\omega\right)}{\partial \omega}=0$$

ω is allowed to **change only in directions** which **does not change** the value of $f\left(\mathbf{x}^{\star}(\omega),\omega\right)$ in order to stay within the solution manifold

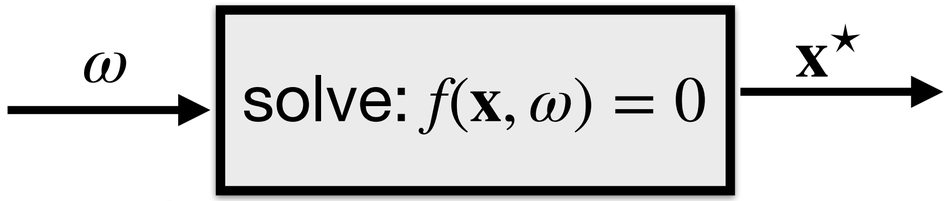
Introduce notation: $\frac{\partial f(a,b)}{\partial a}=\partial_1 f(a,b),\ \ \frac{\partial f(a,b)}{\partial b}=\partial_2 f(a,b)$

Differentiation of compound function yields manifold eq. for searched $\frac{\partial \mathbf{x}^{\star}(\omega)}{\partial \omega}$

$$\frac{\partial f\left(\mathbf{x}^{\star}(\omega),\omega\right)}{\partial \omega}=\partial_1 f\left(\mathbf{x}^{\star}(\omega),\omega\right)\frac{\partial \mathbf{x}^{\star}(\omega)}{\partial \omega}+\partial_2 f\left(\mathbf{x}^{\star}(\omega),\omega\right)=0$$

$$\frac{\partial \mathbf{x}^{\star}(\omega)}{\partial \omega}=-\left[\partial_1 f\left(\mathbf{x}^{\star}(\omega),\omega\right)\right]^{-1}\partial_2 f\left(\mathbf{x}^{\star}(\omega),\omega\right)$$

Root finder



$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = -\left[\partial_1 f\left(\mathbf{x}^\star(\omega), \omega\right)\right]^{-1} \partial_2 f\left(\mathbf{x}^\star(\omega), \omega\right)$$

Implicit gradient (terms can be obtained by autograd)

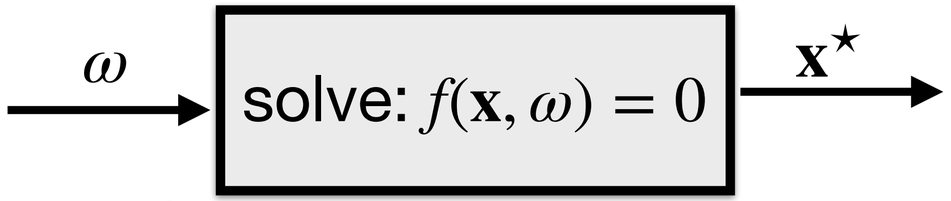
The implicit function theorem. *Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p$, $z_0 \in \mathbb{R}^n$ be such that*

1. $f(a_0, z_0) = 0$, *and*
2. f *is continuously differentiable with non-singular Jacobian* $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset R^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, *and*
3. z^* *is differentiable on* S_{a_0}.

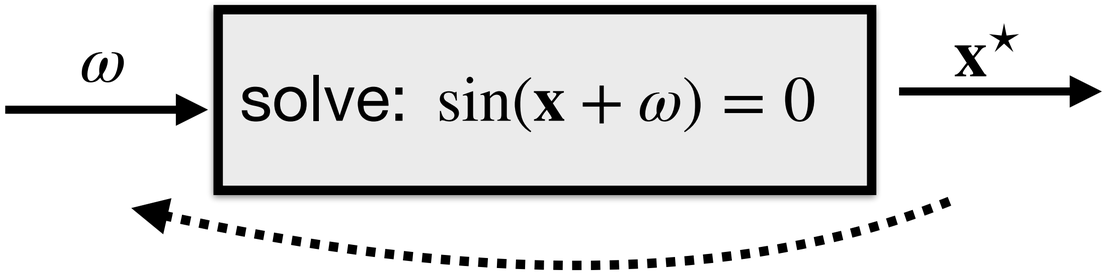
Root finder

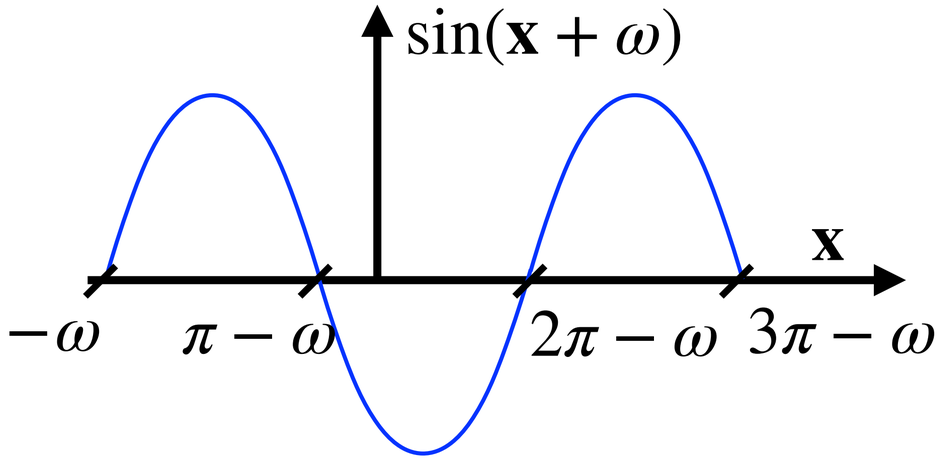


$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = -\left[\partial_1 f\left(\mathbf{x}^\star(\omega), \omega\right)\right]^{-1} \partial_2 f\left(\mathbf{x}^\star(\omega), \omega\right)$$

Implicit gradient (terms can be obtained by autograd)

Example: $f(\mathbf{x}, \omega) = \sin(\mathbf{x} + \omega)$

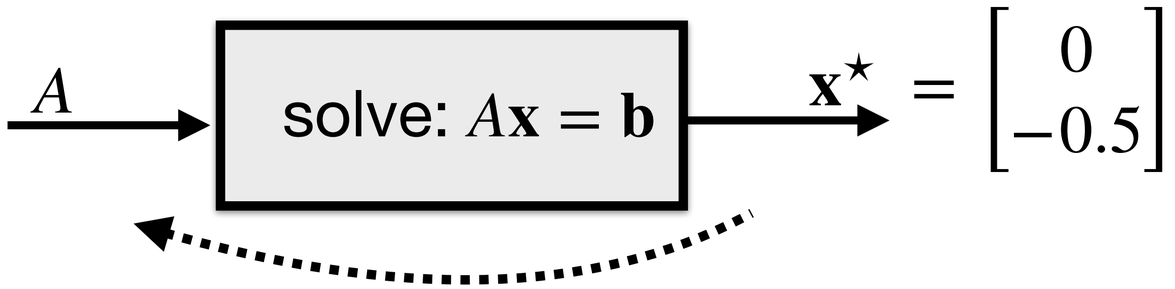




$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = -\left[\partial_0 f\left(\mathbf{x}^\star(\omega), \omega\right)\right]^{-1} \partial_1 f\left(\mathbf{x}^\star(\omega), \omega\right) = -\frac{\cos(\mathbf{x}^\star + \omega)}{\cos(\mathbf{x}^\star + \omega)} = -1$$

Root finder (backprop through linear system / inverse matrix)

Example2: $f(\mathbf{x}, A) = A\mathbf{x} - \mathbf{b}$ $\quad A = \begin{bmatrix} 2 & -2 \\ -2 & 0 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$



$$\frac{\partial \mathbf{x}^\star(A)}{\partial A} = -\left[\partial_1 f\left(\mathbf{x}^\star(A), A\right)\right]^{-1} \partial_2 f\left(\mathbf{x}^\star(A), A\right) = -A^{-1} \cdot \begin{bmatrix} \mathbf{x}^{\star\top}, & \mathbf{0}^\top & \dots & \mathbf{0}^\top \\ \mathbf{0}^\top, & \mathbf{x}^{\star\top} & \dots & \mathbf{0}^\top \\ \vdots & \vdots & & \vdots \\ \mathbf{0}^\top, & \mathbf{0}^\top & \dots & \mathbf{x}^{\star\top} \end{bmatrix}$$

You can learn to predict equations the solution of which is equal to something

$$= -\begin{bmatrix} 2 & -2 \\ -2 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} x_1^\star & x_2^\star & 0 & 0 \\ 0 & 0 & x_1^\star & x_2^\star \end{bmatrix} = -\begin{bmatrix} 0 & 0 & 0 & 0.25 \\ 0 & 0.25 & 0 & 0.25 \end{bmatrix}$$

```
x_star = torch.linalg.inv(A) @ b

torch.autograd.grad(x_star[0], A)[0].reshape(4)
> tensor([-0.0000, -0.0000, -0.0000, -0.2500])
torch.autograd.grad(x_star[1], A)[0].reshape(4)
> tensor([-0.0000, -0.2500, -0.0000, -0.2500])
```

Do not have to preserve computational graph of the matrix inverse computation !!!

Root finder: backprop through set of equations typical application

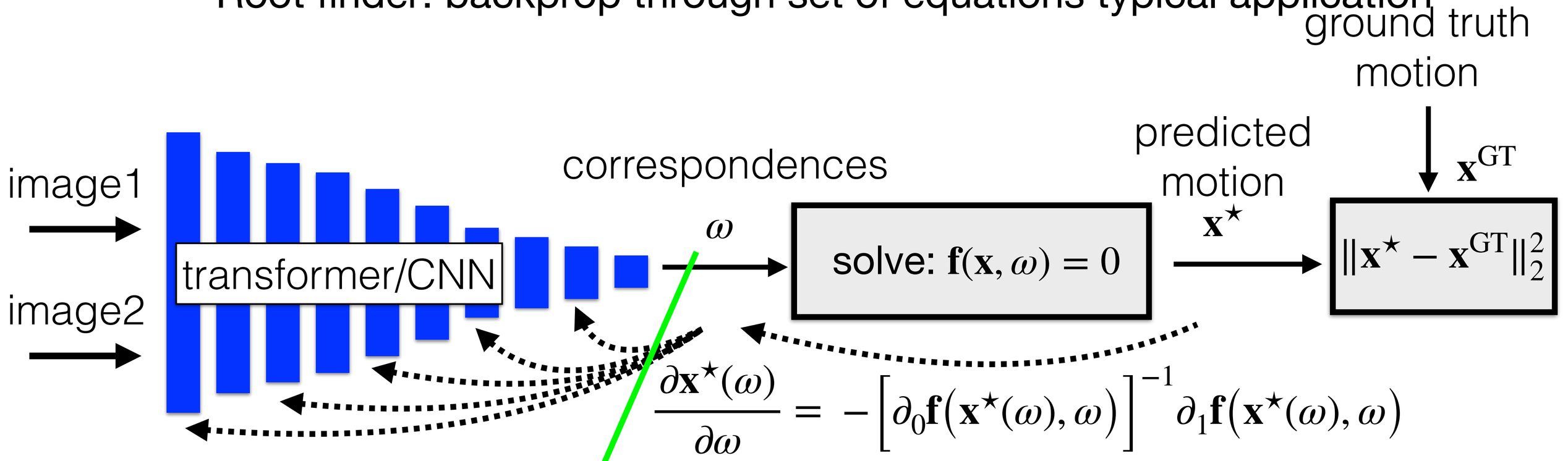


image1

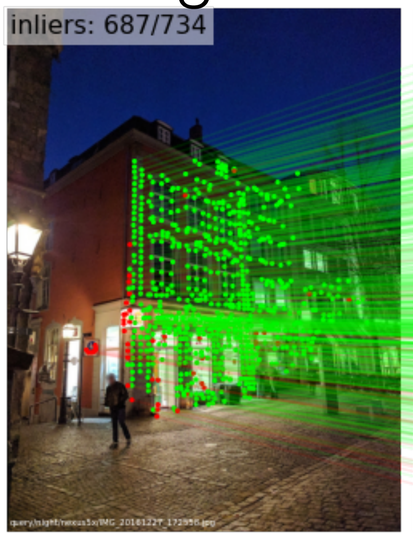
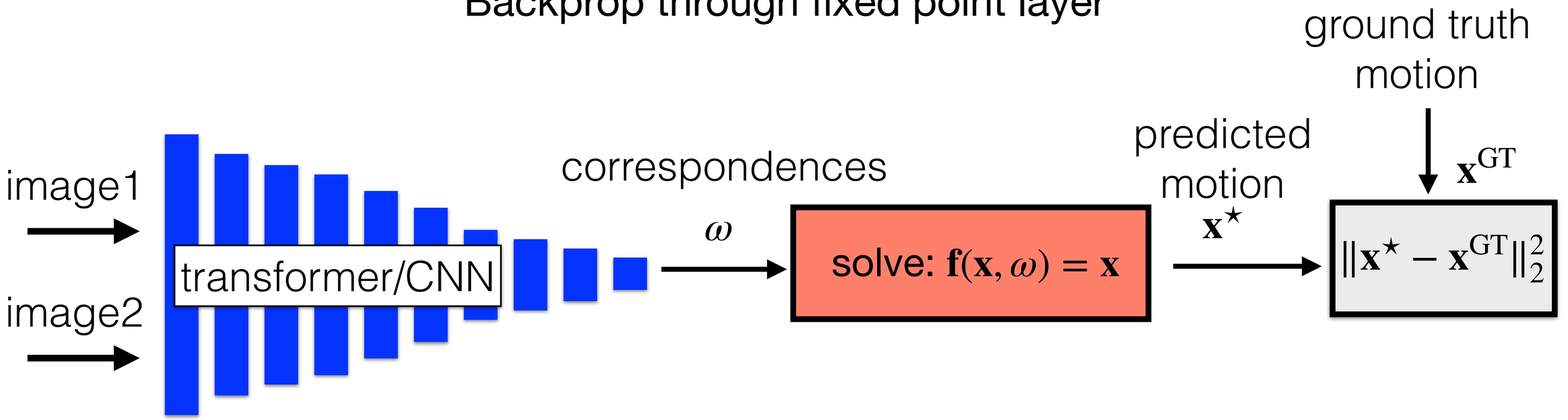


image2

$\mathbf{f}(\mathbf{x}, \omega) = 0$ is set of non-linear equations

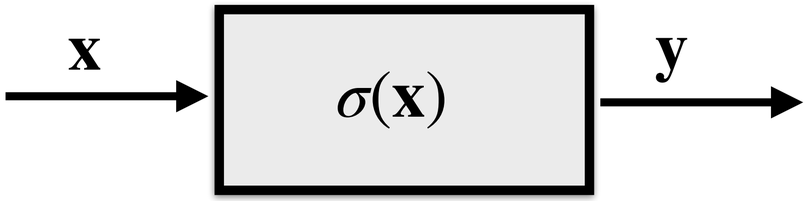
Backprop through fixed point layer



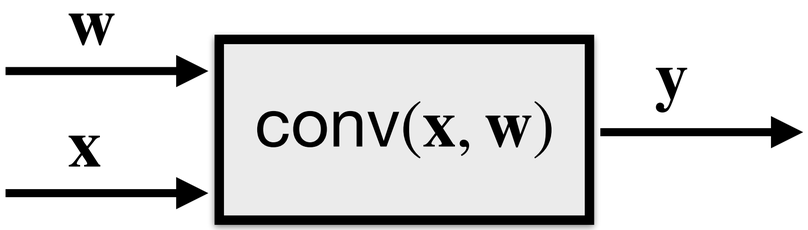
- Infinitely deep residual network layer
- Q-learning as a feed-forward pass
- Convergence assured for some function (contractivity or monotocinity)

Explicit layers

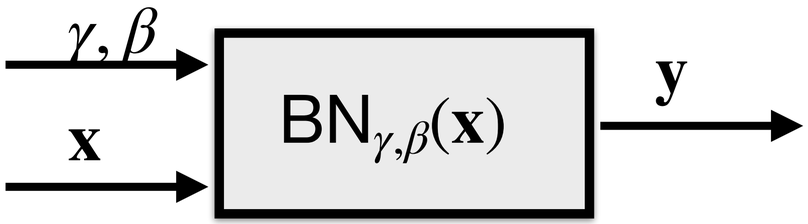
Sigmoid:



Convolution:

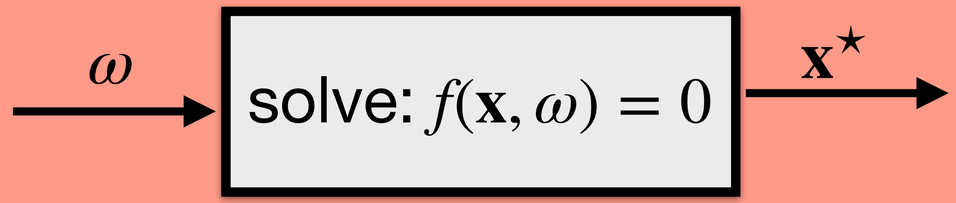


Batch-norm:

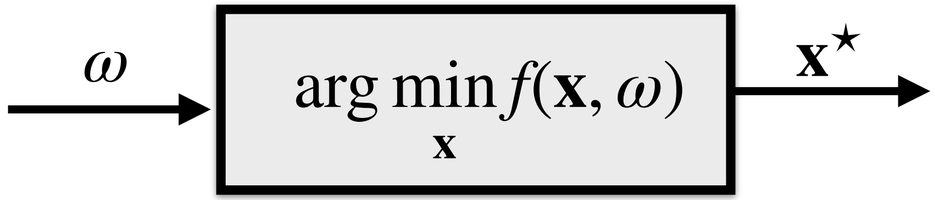


Implicit layers

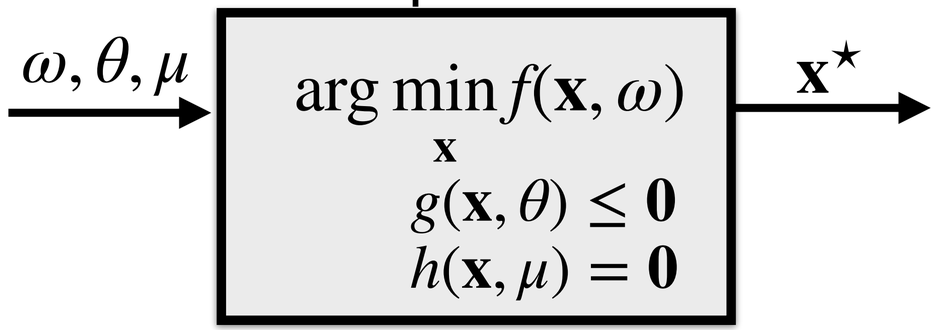
Root finder:



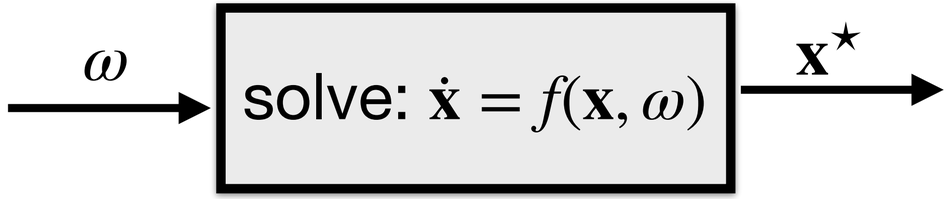
Unconstrained optimizer:



Constrained optimizer:



ODE solver:

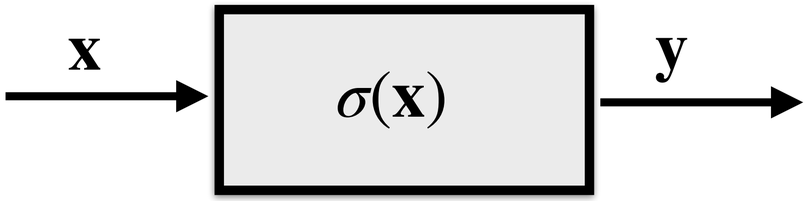


Explicit layers

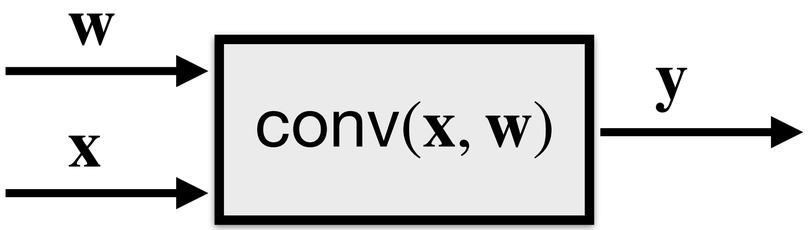
The rest will not be in the exam test

Implicit layers

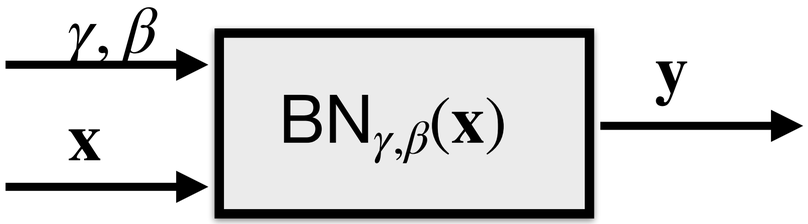
Sigmoid:



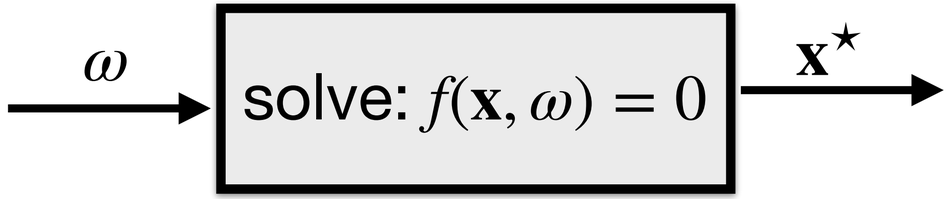
Convolution:



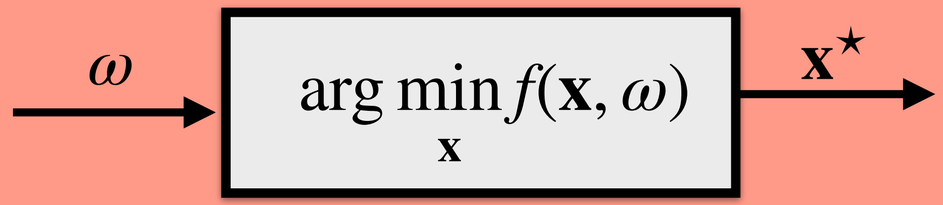
Batch-norm:



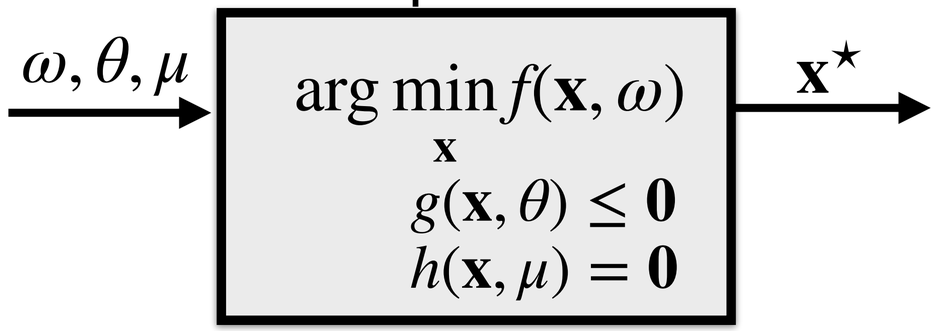
Root finder:



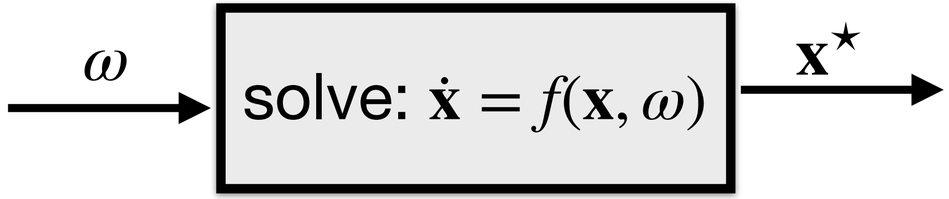
Unconstrained optimizer:



Constrained optimizer:



ODE solver:



Unconstrained optimizer

- Unconstrained local optimizer provides solution $\mathbf{x}^\star$ such that $\dfrac{\partial f\left(\mathbf{x}^\star, \omega\right)}{\partial \mathbf{x}^\star} = 0$
- Let's denote gradient $\dfrac{\partial f\left(\mathbf{x}^\star, \omega\right)}{\partial \mathbf{x}^\star} = g(\mathbf{x}^\star, \omega) = 0$
- This solution depends on parameters, we refer it $\mathbf{x}^\star(\omega) : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- We treat unconstrained optimizer as **gradient root finder** => the rest is the same

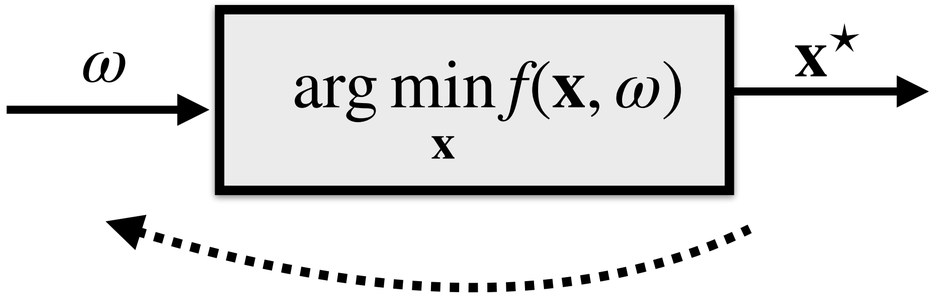
$$\frac{\partial g\left(\mathbf{x}^\star(\omega), \omega\right)}{\partial \omega} = 0$$

ω is allowed to change only in directions which does not change the gradient $g\left(\mathbf{x}^\star(\omega), \omega\right)$ in order to stay within the solution manifold

$$\frac{\partial g\left(\mathbf{x}^\star(\omega), \omega\right)}{\partial \omega} = \partial_1 g\left(\mathbf{x}^\star(\omega), \omega\right) \frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} + \partial_2 g\left(\mathbf{x}^\star(\omega), \omega\right) = 0$$

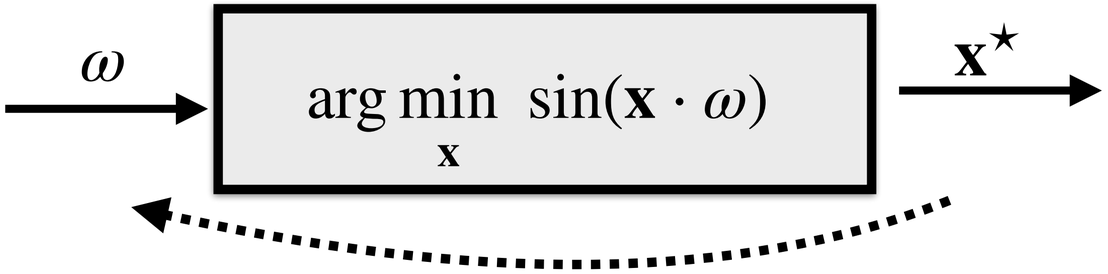
$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = -\left[\partial_1 g\left(\mathbf{x}^\star(\omega), \omega\right)\right]^{-1} \partial_2 g\left(\mathbf{x}^\star(\omega), \omega\right)$$

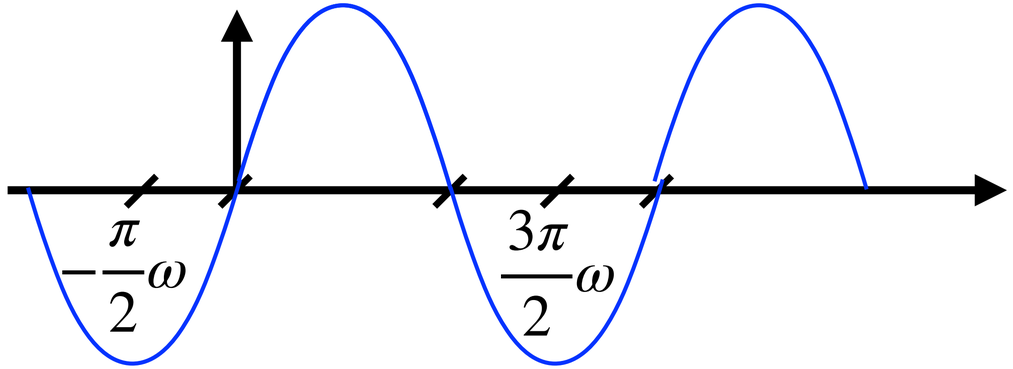
Unconstrained optimizer



$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = -\left[\partial_1 g\left(\mathbf{x}^\star(\omega), \omega\right)\right]^{-1} \partial_2 g\left(\mathbf{x}^\star(\omega), \omega\right) \quad \text{where} \quad g(\mathbf{x}^\star, \omega) = \frac{\partial f\left(\mathbf{x}^\star, \omega\right)}{\partial \mathbf{x}^\star}$$

Example: $f(\mathbf{x}, \omega) = \sin(\mathbf{x} \cdot \omega) \quad \Rightarrow \quad g(\mathbf{x}, \omega) = \omega \cdot \cos(\mathbf{x} \cdot \omega)$

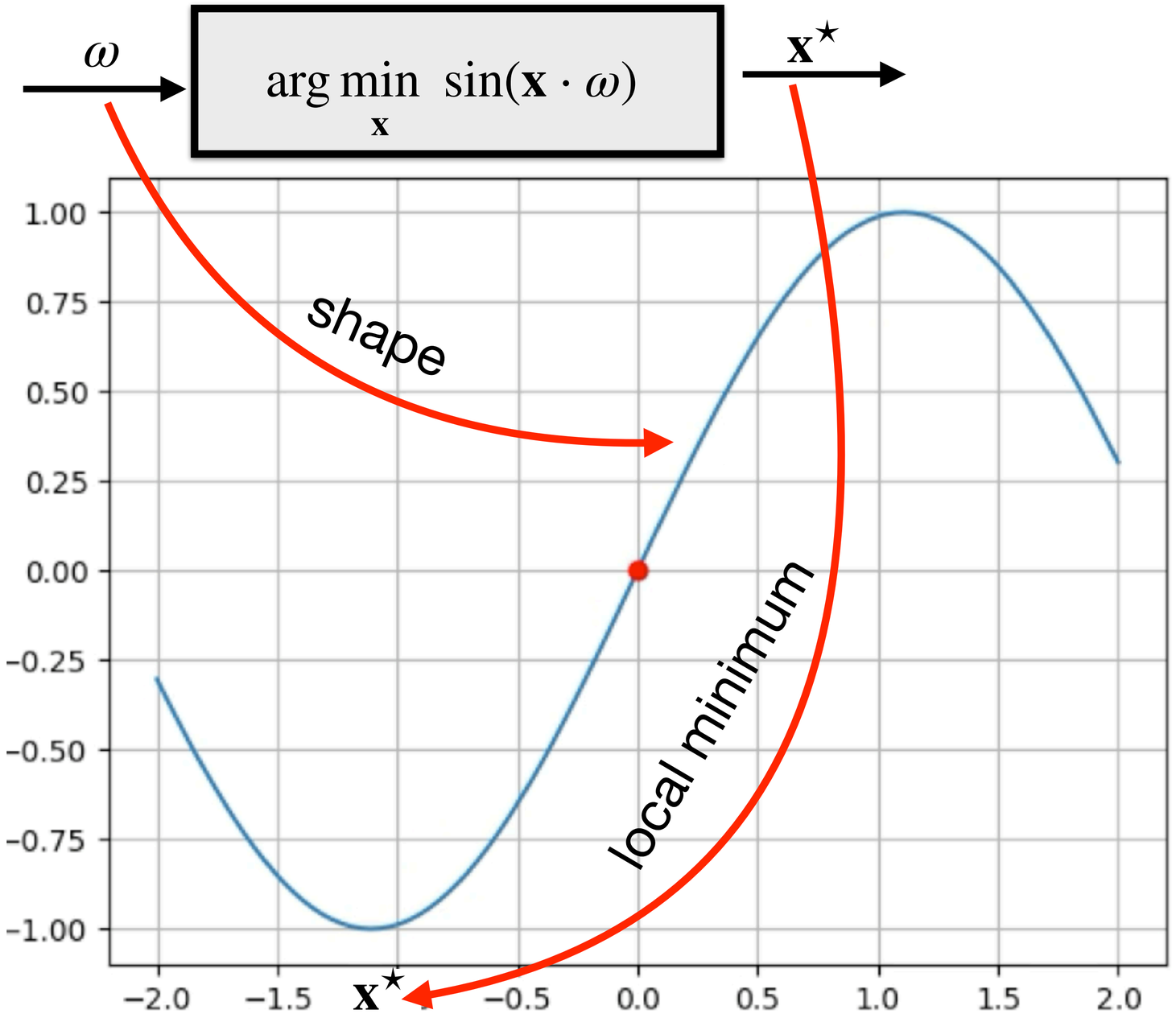




$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = -\Big[\underbrace{-\omega^2 \sin(\omega \mathbf{x})}_{\frac{\partial}{\partial \mathbf{x}}}\Big]^{-1} \cdot \Big(\underbrace{-\omega \mathbf{x} \sin(\omega \mathbf{x}) + \cos(\omega \mathbf{x})}_{\frac{\partial}{\partial \omega}}\Big)$$

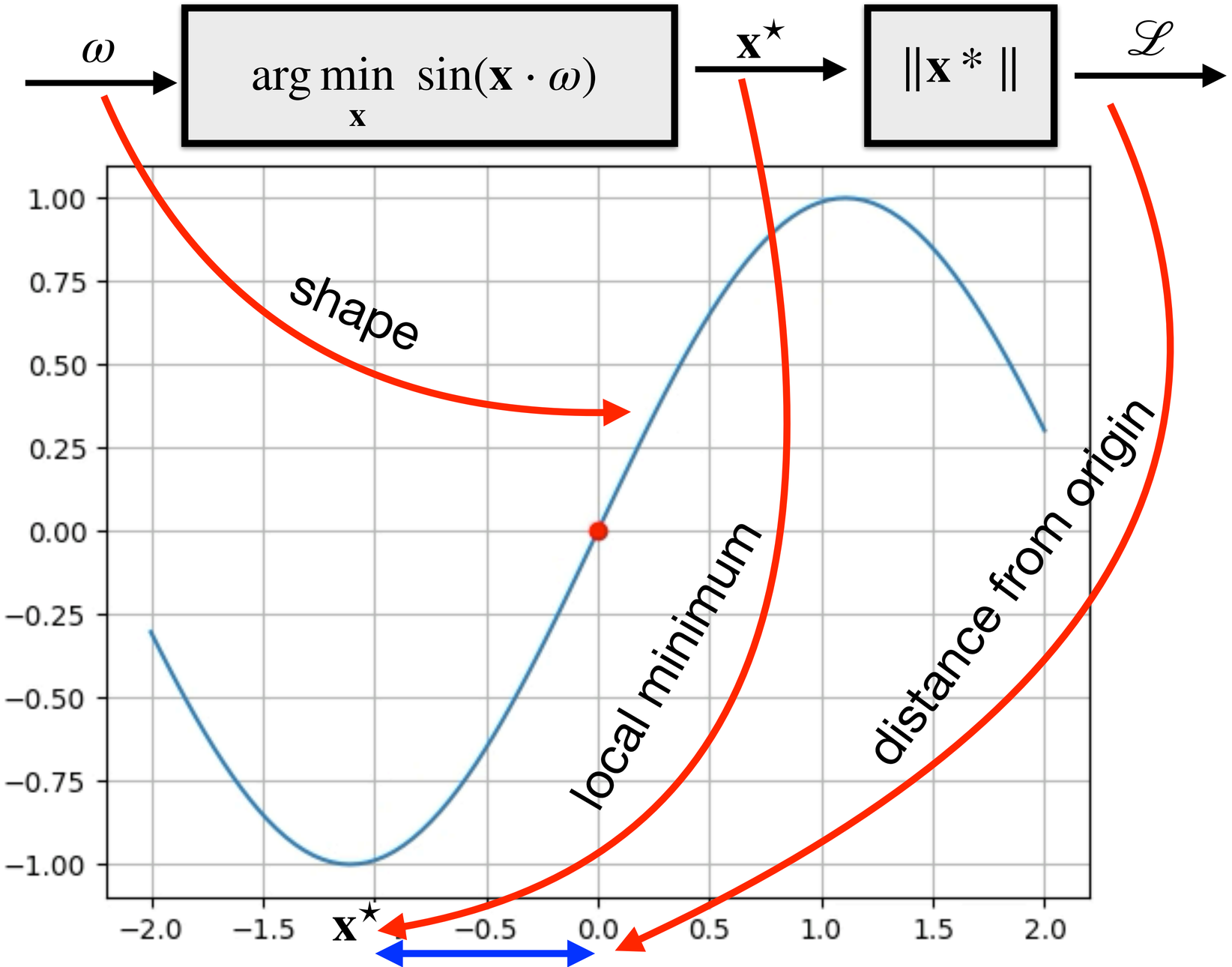
Unconstrained optimizer

Example:



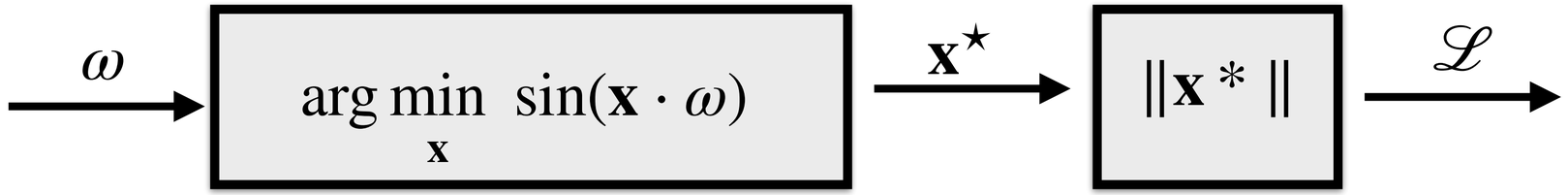
Unconstrained optimizer

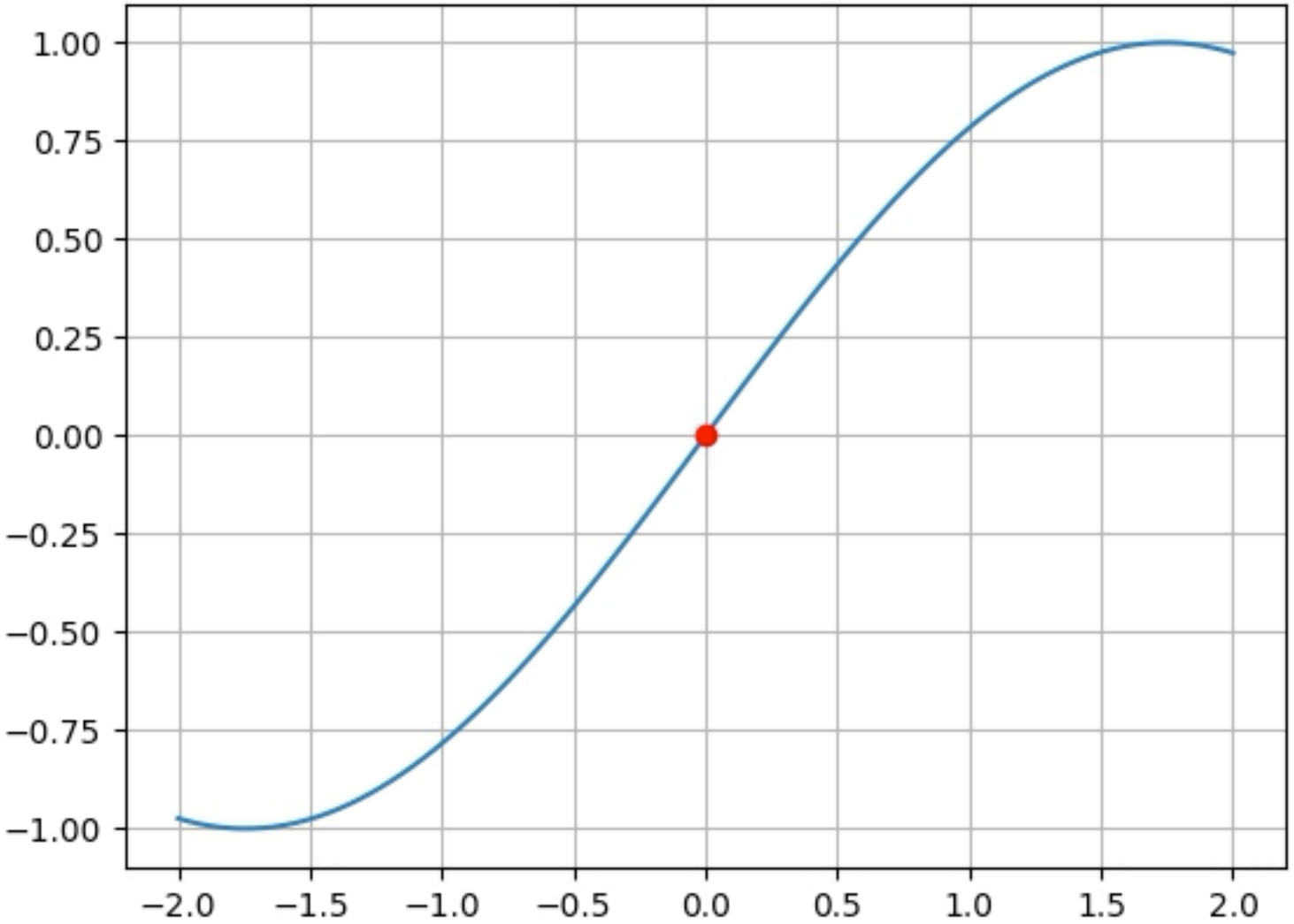
Example:



Unconstrained optimizer

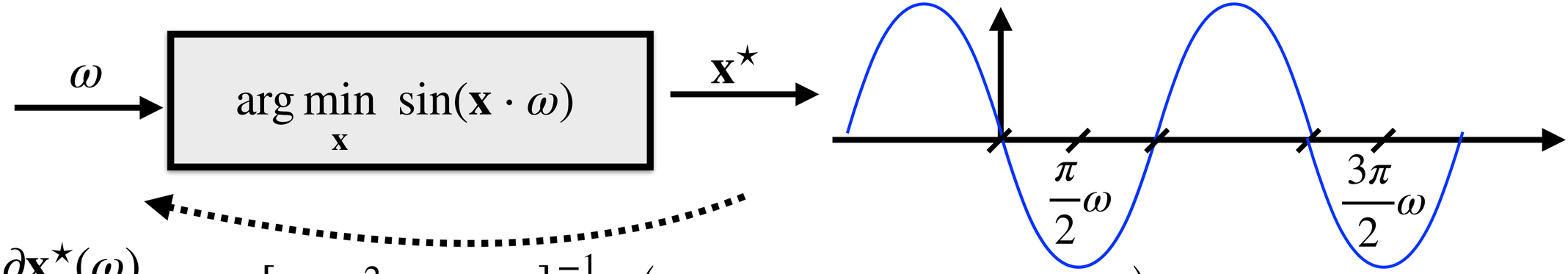
Example:





Unconstrained optimizer

Example:



$$\frac{\partial \mathbf{x}^\star(\omega)}{\partial \omega} = -\left[-\omega^2 \sin(\omega \mathbf{x})\right]^{-1} \cdot \left(-\omega \mathbf{x} \sin(\omega \mathbf{x}) + \cos(\omega \mathbf{x})\right)$$

```python
def f(x, omega):
    return torch.sin(x*omega)

omega = torch.tensor(2.0, requires_grad=True)
x = torch.tensor(0.0, requires_grad=True)
for j in range(25):
    x = x - 0.1*torch.autograd.grad(f(x, omega), x, retain_graph=True, create_graph=True)[0]
```

explicit gradient

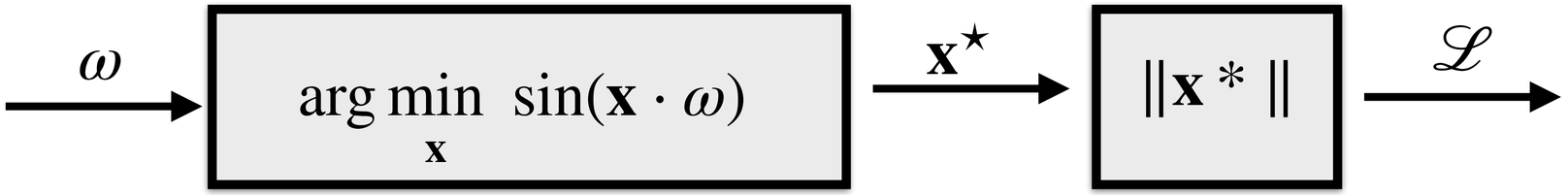
```python
torch.autograd.grad(x, omega)[0]
> tensor(0.3927)
```

implicit gradient

```python
-(-torch.sin(omega*x)*omega*x + torch.cos(omega*x))/(-torch.sin(omega*x)*(omega**2)
> tensor(0.3927)
```

Unconstrained optimizer

Example:



What happens to **explicit gradient** if number of **iterations are decreased**?

```
for j in range(25):
    x = x - 0.1*torch.autograd.grad(f(x, omega), x, retain_graph=True, create_graph=True)[0]

torch.autograd.grad(x.norm(), omega)[0]
> tensor(-0.3927)
```

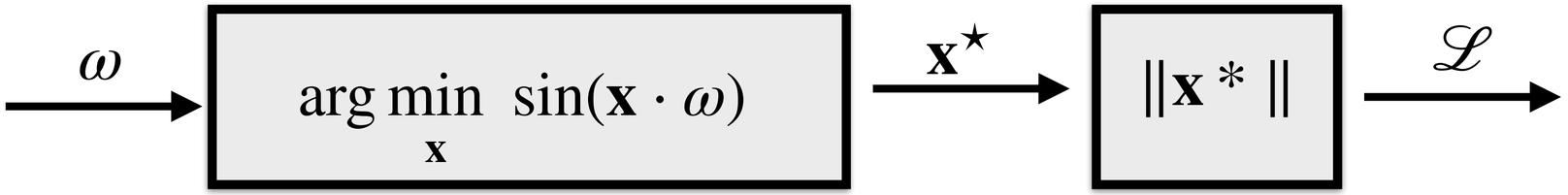
```
for j in range(5):
    x = x - 0.1*torch.autograd.grad(f(x, omega), x, retain_graph=True, create_graph=True)[0]

torch.autograd.grad(x.norm(), omega)[0]
> tensor(0.4527)
```

Is it correct?

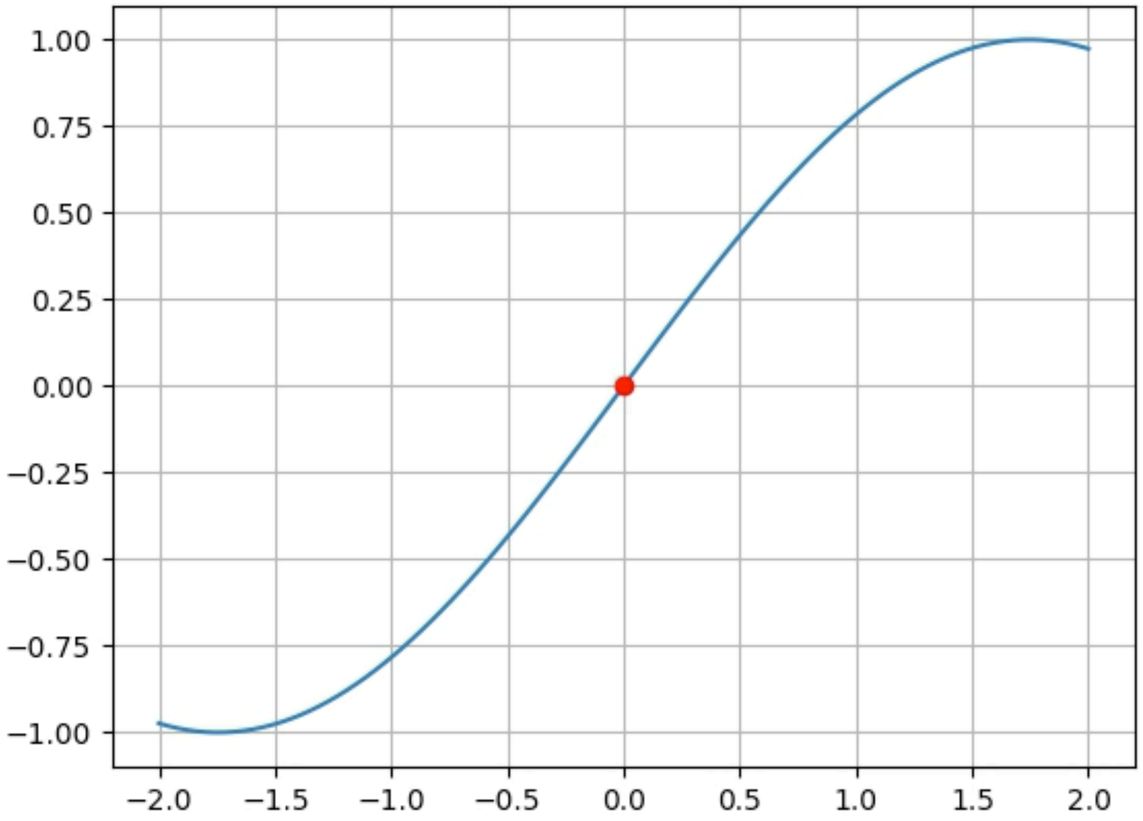
Unconstrained optimizer

Example:

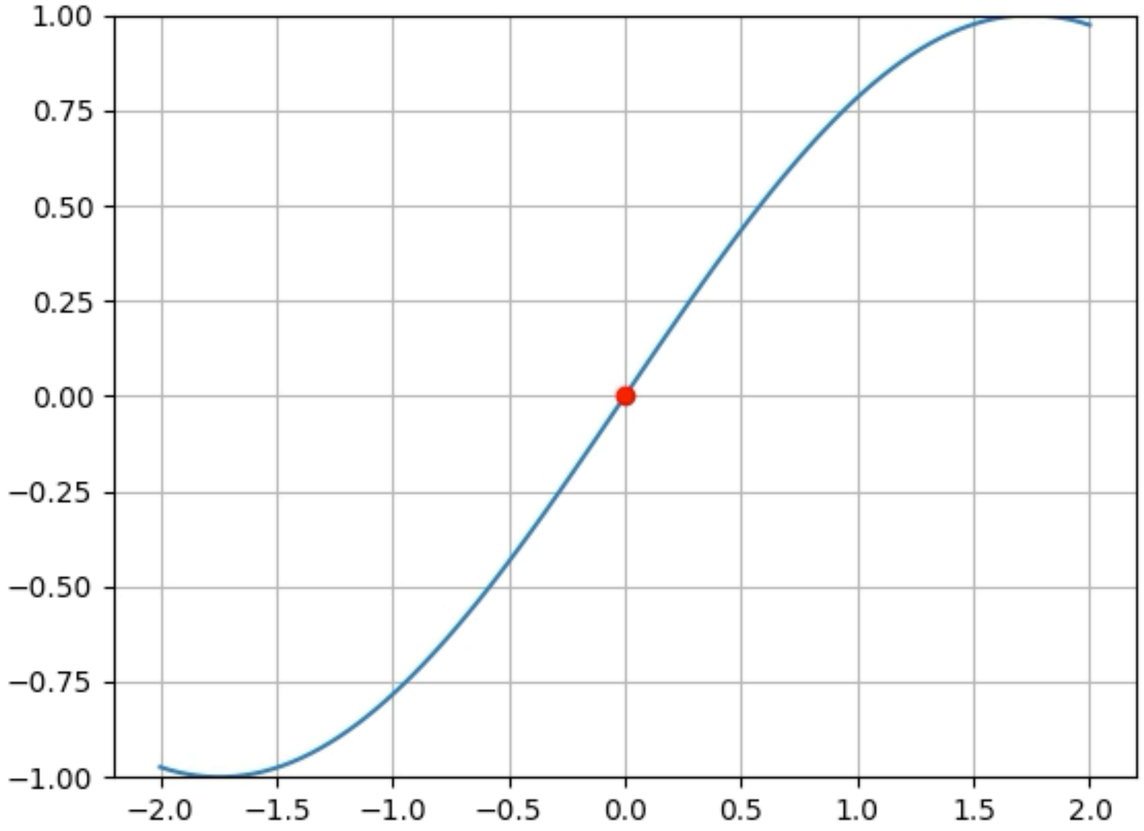


What happens to explicit-gradient if number of iterations are decreased?

```
for j in range(25):
```

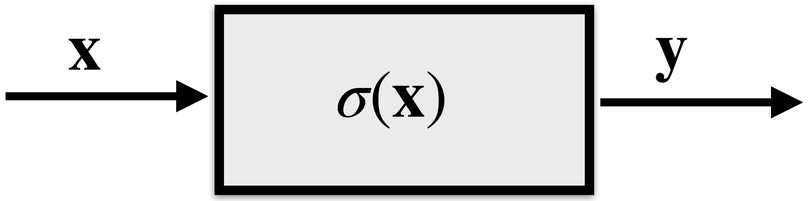



```
for j in range(5):
```

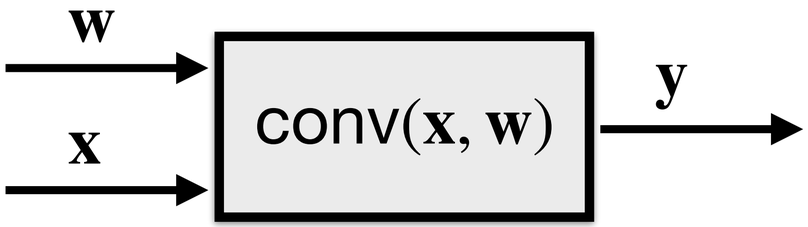


Explicit layers

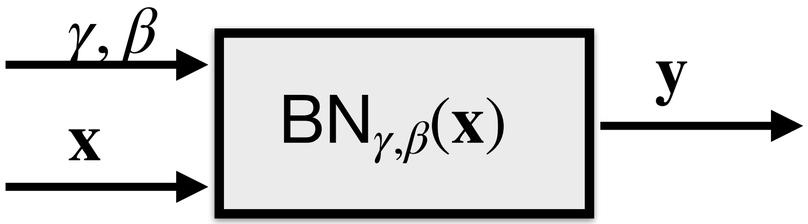
Sigmoid:



Convolution:

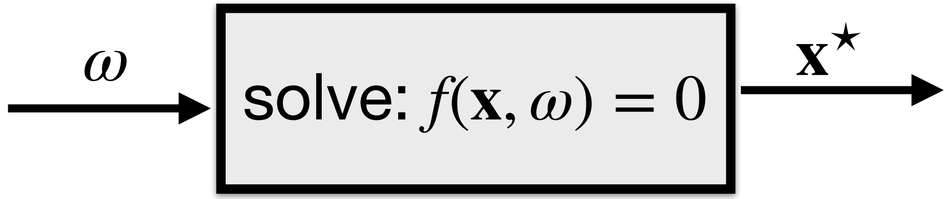


Batch-norm:

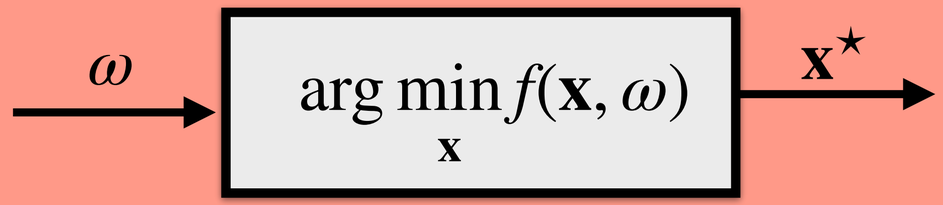


Implicit layers

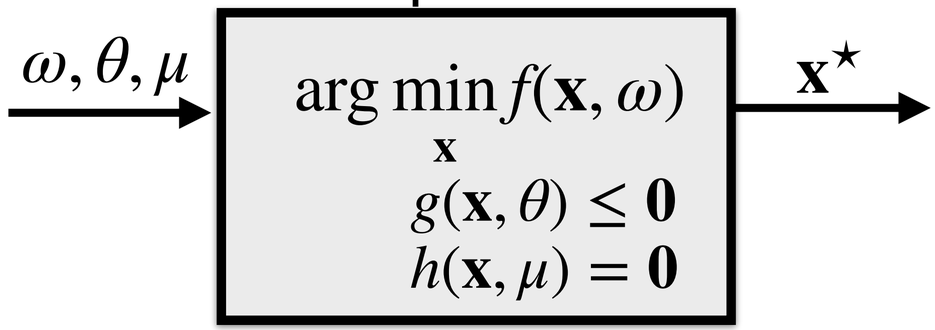
Root finder:



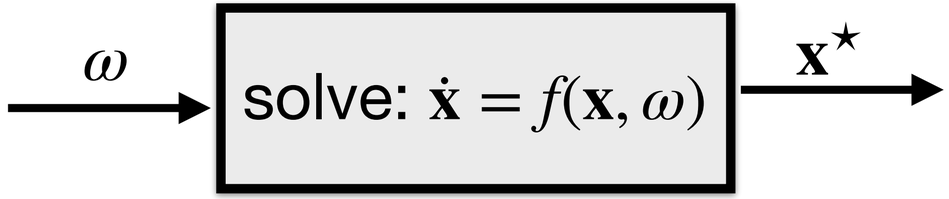
Unconstrained optimizer:



Constrained optimizer:

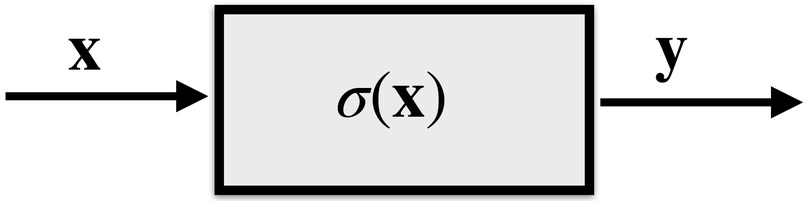


ODE solver:

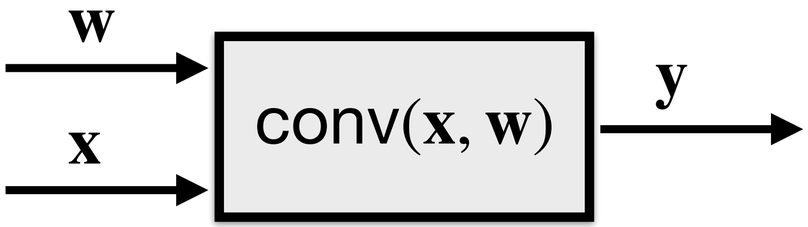


Explicit layers

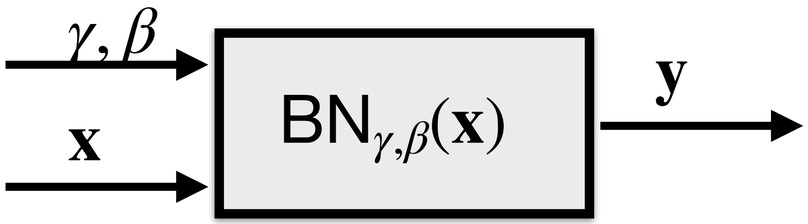
Sigmoid:



Convolution:

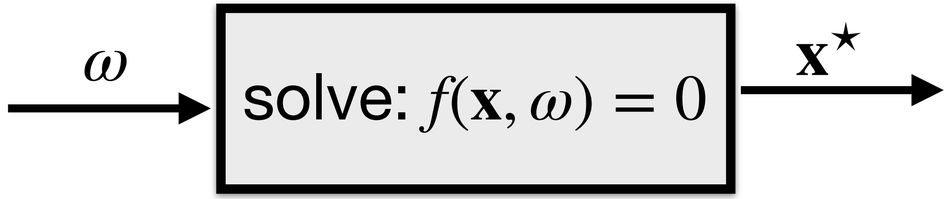


Batch-norm:

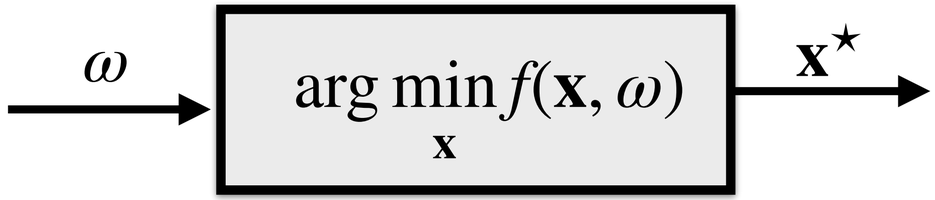


Implicit layers

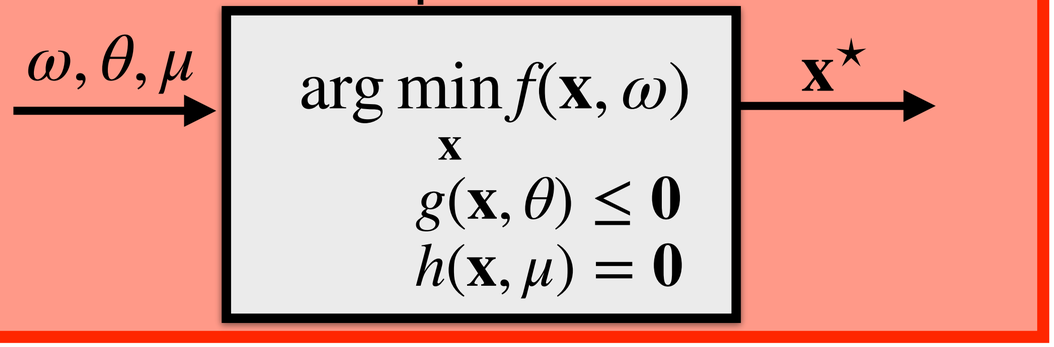
Root finder:



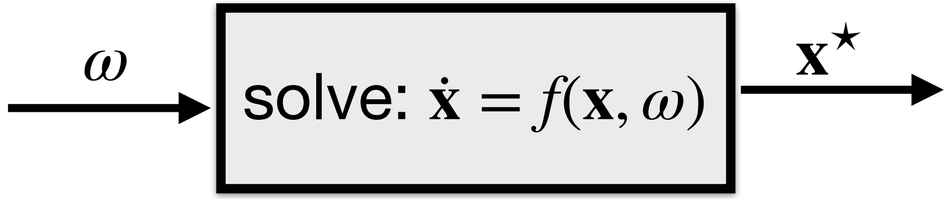
Unconstrained optimizer:



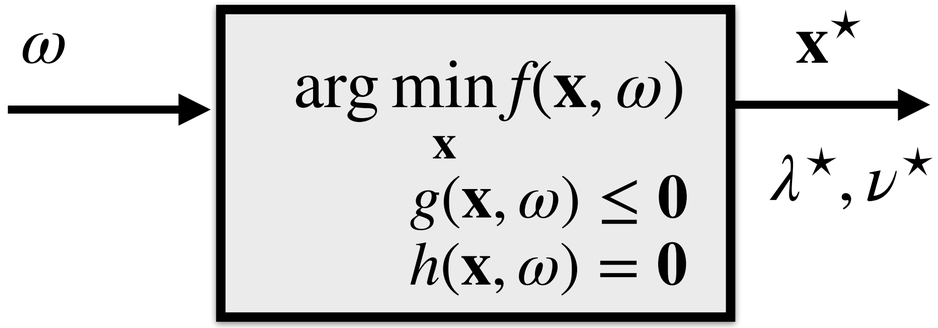
Constrained optimizer:



ODE solver:



Constrained optimizer



- Given parameters ω, typical optimizer provides primal-dual solution $\mathbf{x}^\star, \lambda^\star, \nu^\star$ that satisfy necessary and sufficient optimality conditions (KKT):

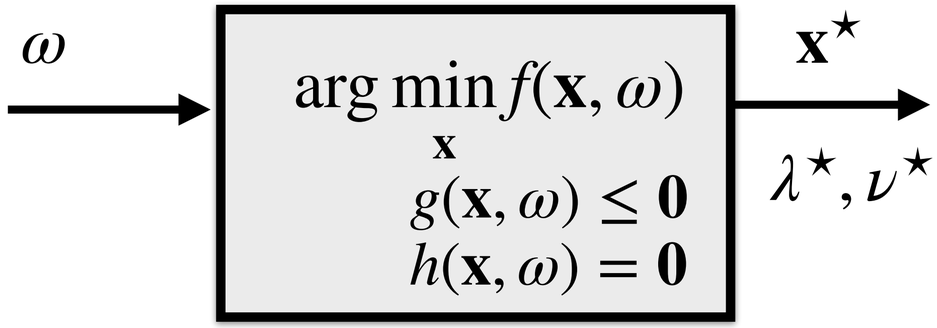
Lagrangian: $$L(\mathbf{x}, \lambda, \nu) = f(\mathbf{x}, \omega) + \sum_i \lambda_i g_i(\mathbf{x}, \omega) + \sum_i \nu_i h_i(\mathbf{x}, \omega)$$

Stationarity: $$\frac{\partial L(\mathbf{x}^\star, \lambda^\star, \nu^\star)}{\partial \mathbf{x}^\star} = \frac{\partial f(\mathbf{x}^\star, \omega)}{\partial \mathbf{x}^\star} + \sum_i \lambda_i \frac{\partial g_i(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} + \sum_i \nu_i \frac{\partial h_i(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} = \mathbf{0}$$

Complementarity: $$\lambda^\star \circ g(\mathbf{x}^\star, \omega) = \mathbf{0}$$

Primal feasibility: $$g(\mathbf{x}^\star, \omega) \leq \mathbf{0}$$
$$h(\mathbf{x}^\star, \omega) = \mathbf{0}$$

Constrained optimizer



- Given parameters ω, typical optimizer provides primal-dual solution $\mathbf{x}^\star, \lambda^\star, \nu^\star$ that satisfy necessary and sufficient optimality conditions (KKT):

Lagrangian: $$L(\mathbf{x}, \lambda, \nu) = f(\mathbf{x}, \omega) + \sum_i \lambda_i g_i(\mathbf{x}, \omega) + \sum_i \nu_i h_i(\mathbf{x}, \omega)$$

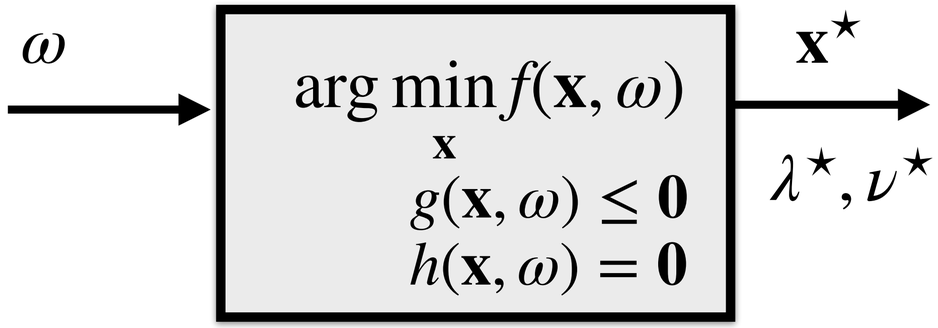
Stationarity: $$\frac{\partial L(\mathbf{x}^\star, \lambda^\star, \nu^\star)}{\partial \mathbf{x}^\star} = \frac{\partial f(\mathbf{x}^\star, \omega)}{\partial \mathbf{x}^\star} + \sum_i \lambda_i \frac{\partial g_i(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} + \sum_i \nu_i \frac{\partial h_i(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} = \mathbf{0}$$

Complementarity: $$\lambda^\star \circ g(\mathbf{x}^\star, \omega) = \mathbf{0}$$

Primal feasibility: $$g(\mathbf{x}^\star, \omega) \leq \mathbf{0}$$
$$h(\mathbf{x}^\star, \omega) = \mathbf{0}$$

Dual feasibility: $$\lambda^\star \geq \mathbf{0}$$

Constrained optimizer



- Given parameters ω, typical optimizer provides primal-dual solution $\mathbf{x}^\star, \lambda^\star, \nu^\star$ that satisfy necessary and sufficient optimality conditions (KKT):

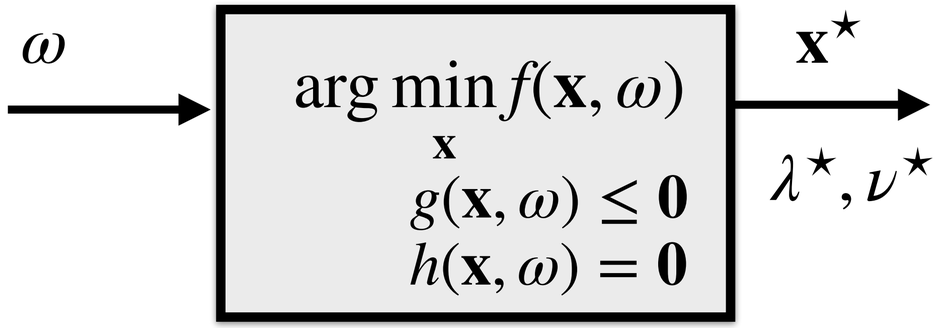
$$G(\mathbf{x}^\star, \lambda^\star, \nu^\star, \omega) = \begin{bmatrix} \dfrac{\partial f(\mathbf{x}^\star, \omega)}{\partial \mathbf{x}^\star} + \sum_i \lambda_i^\star \dfrac{\partial g_i(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} + \sum_i \nu_i^\star \dfrac{\partial h_i(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} \\ \lambda^\star \circ g(\mathbf{x}^\star, \omega) \\ h(\mathbf{x}^\star, \omega) \end{bmatrix} = \mathbf{0}$$

$$g(\mathbf{x}^\star, \omega) \leq \mathbf{0}$$

$$\lambda^\star \geq \mathbf{0}$$

Inequalities can be ignored due to complementarity condition
(i.e. c.s. implies that one of them have to be zero,
if $g(\mathbf{x}^\star, \omega) = \mathbf{0}$ and $\lambda > \mathbf{0}$ then small change of λ will
preserve positivity of λ and the other way around)

Constrained optimizer



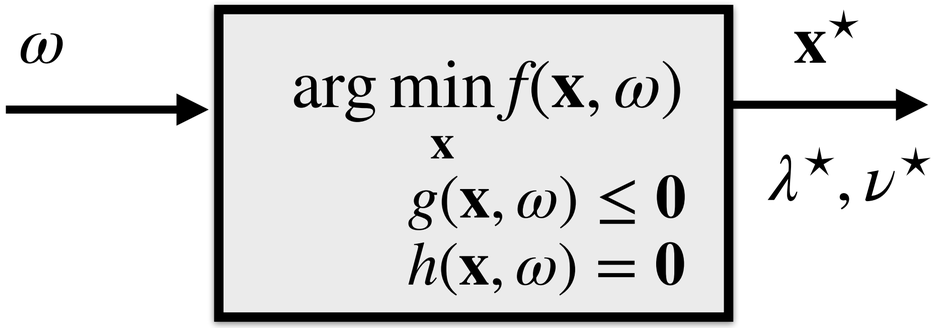
- Given parameters ω, typical optimizer provides primal-dual solution $\mathbf{x}^\star, \lambda^\star, \nu^\star$ that satisfy necessary and sufficient optimality conditions (KKT):

$$G(\mathbf{x}^\star, \lambda^\star, \nu^\star, \omega) = \begin{bmatrix} \dfrac{\partial f(\mathbf{x}^\star, \omega)}{\partial \mathbf{x}^\star} + \sum_i \lambda_i^\star \dfrac{\partial g_i(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} + \sum_i \nu_i^\star \dfrac{\partial h_i(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} \\ \lambda^\star \circ g(\mathbf{x}^\star, \omega) \\ h(\mathbf{x}^\star, \omega) \end{bmatrix} = \mathbf{0}$$

- Input params ω can change output solution $\left(\mathbf{x}^\star, \lambda^\star, \nu^\star\right)(\omega) = [\mathbf{x}^\star(\omega), \lambda^\star(\omega), \nu^\star(\omega)]$ only in direction that preserves these conditions, therefore:

$$\frac{\partial G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \nu^\star(\omega), \omega\right)}{\partial \omega} = \mathbf{0}$$

Constrained optimizer



$$\frac{\partial G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \nu^\star(\omega), \omega\right)}{\partial \omega} = \mathbf{0}$$

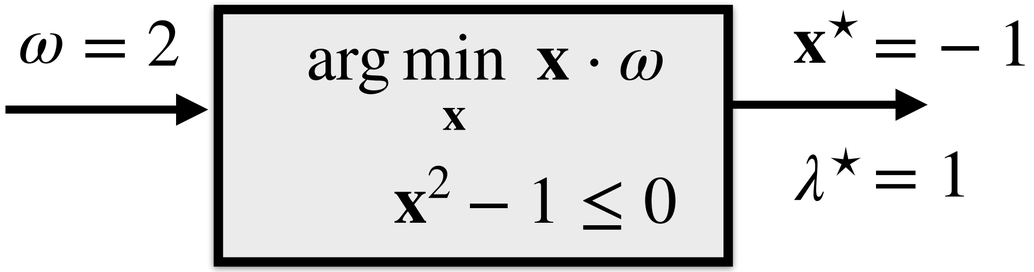
Computing derivative of the compound function yields desired equation for the gradient

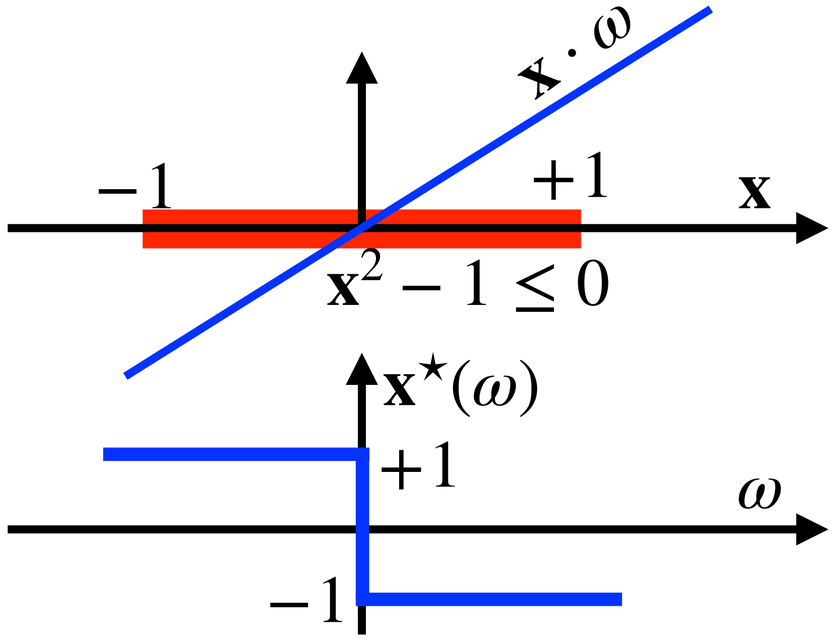
$$\partial_{0,1,2} G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \nu^\star(\omega), \omega\right) \frac{\partial\left(\mathbf{x}^\star, \lambda^\star, \nu^\star\right)(\omega)}{\partial \omega} + \partial_3 G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \nu^\star(\omega), \omega\right) = \mathbf{0}$$

$$\frac{\partial\left(\mathbf{x}^\star, \lambda^\star, \nu^\star\right)(\omega)}{\partial \omega} = -\left[\partial_{0,1,2} G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \nu^\star(\omega), \omega\right)\right]^{-1} \partial_3 G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \nu^\star(\omega), \omega\right)$$

Constrained optimizer

Example:





$$G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \omega\right) = \begin{bmatrix} \frac{\partial f(\mathbf{x}^\star, \omega)}{\partial \mathbf{x}^\star} + \lambda \frac{\partial g(\mathbf{x}, \omega)}{\partial \mathbf{x}^\star} \\ \lambda g(\mathbf{x}^\star, \omega) \end{bmatrix} = \begin{bmatrix} \omega + 2\lambda \mathbf{x} \\ \lambda(\mathbf{x}^2 - 1) \end{bmatrix}$$

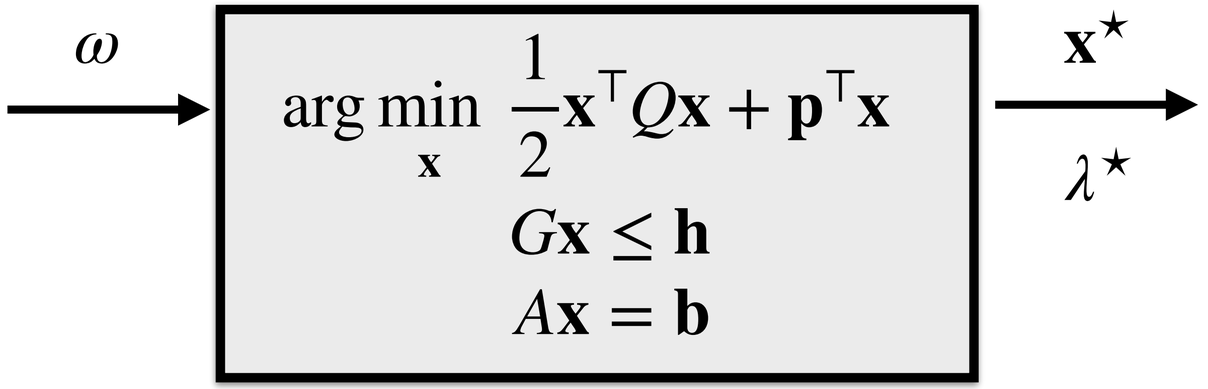
$$\frac{\partial\left(\mathbf{x}^\star, \lambda^\star\right)(\omega)}{\partial \omega} = -\left[\partial_{0,1} G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \omega\right)\right]^{-1} \partial_2 G\left(\mathbf{x}^\star(\omega), \lambda^\star(\omega), \omega\right)$$

$$= -\begin{bmatrix} 2\lambda^\star & 2\mathbf{x}^\star \\ 2\lambda^\star \mathbf{x}^\star & (\mathbf{x}^\star)^2 - 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = -\begin{bmatrix} 2 & -2 \\ -2 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1/2 \end{bmatrix}$$

-inv([2 -2 ; -2 0])* [1;0]

Constrained optimizer

Example2:

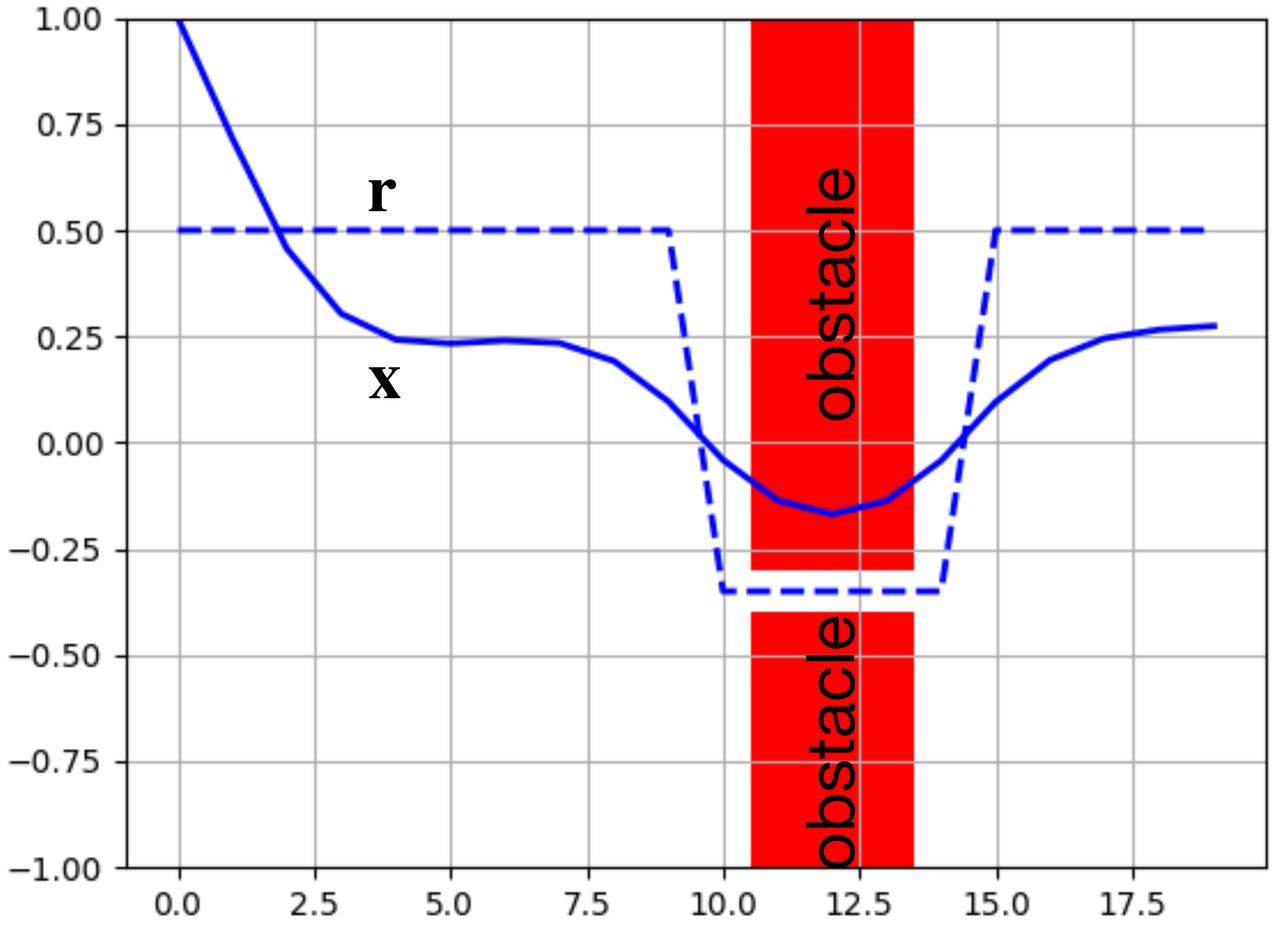


$$\partial_{0,1,2} G\left(\mathbf{x}^\star, \lambda^\star, \nu^\star, \omega\right) = \begin{bmatrix} Q & G^\top & A^\top \\ \text{diag}(\lambda^\star) G & \text{diag}(G\mathbf{x}^\star - \mathbf{h}) & 0 \\ A & 0 & 0 \end{bmatrix}$$

$$\frac{\partial\left(\mathbf{x}^\star, \lambda^\star, \nu^\star\right)(\omega)}{\partial\omega} = -\left[\partial_{0,1,2} G\left(\mathbf{x}^\star, \lambda^\star, \nu^\star, \omega\right)\right]^{-1} \partial_3 G\left(\mathbf{x}^\star, \lambda^\star, \nu^\star, \omega\right)$$

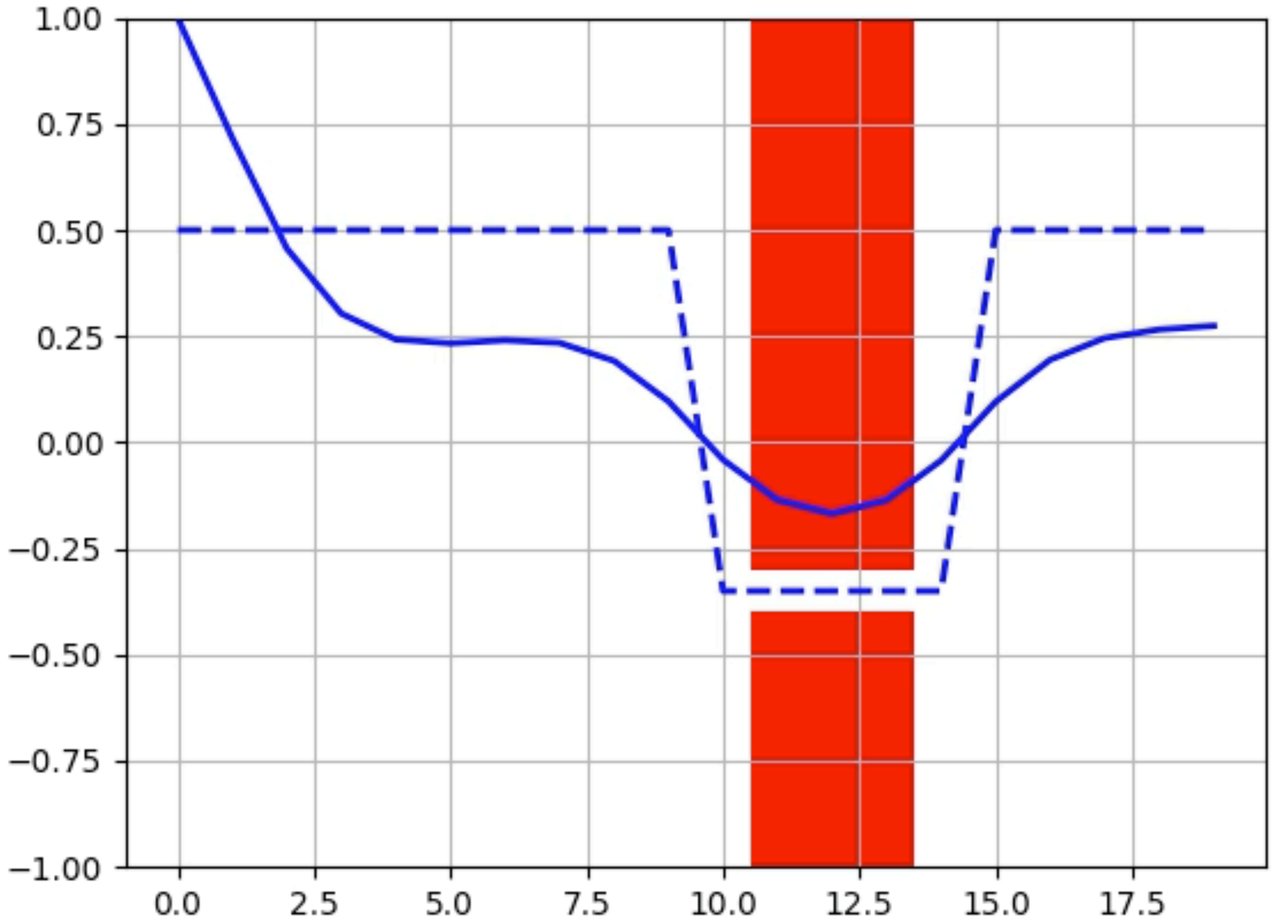
Application: MPC with non-linear system

img → conv network → $\omega = \mathbf{r}$ → $\arg\min_{\mathbf{u}} (\mathbf{x}-\mathbf{r})^\top Q(\mathbf{x}-\mathbf{r}) + \mathbf{u}^\top R\mathbf{u}$ subject to $\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_{k+1}$ → $\mathbf{u}^\star, \mathbf{x}^\star$ → $\text{relu}\big(\text{obs\_dist}(\mathbf{x}^\star)\big)$



Application: MPC with non-linear system

img → conv network → $\omega = \mathbf{r}$ → $\arg\min_{\mathbf{u}} (\mathbf{x}-\mathbf{r})^\top Q(\mathbf{x}-\mathbf{r}) + \mathbf{u}^\top R\mathbf{u}$ subject to $\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_{k+1}$ → $\mathbf{u}^\star, \mathbf{x}^\star$ → $\text{relu}\big(\text{obs\_dist}(\mathbf{x}^\star)\big)$

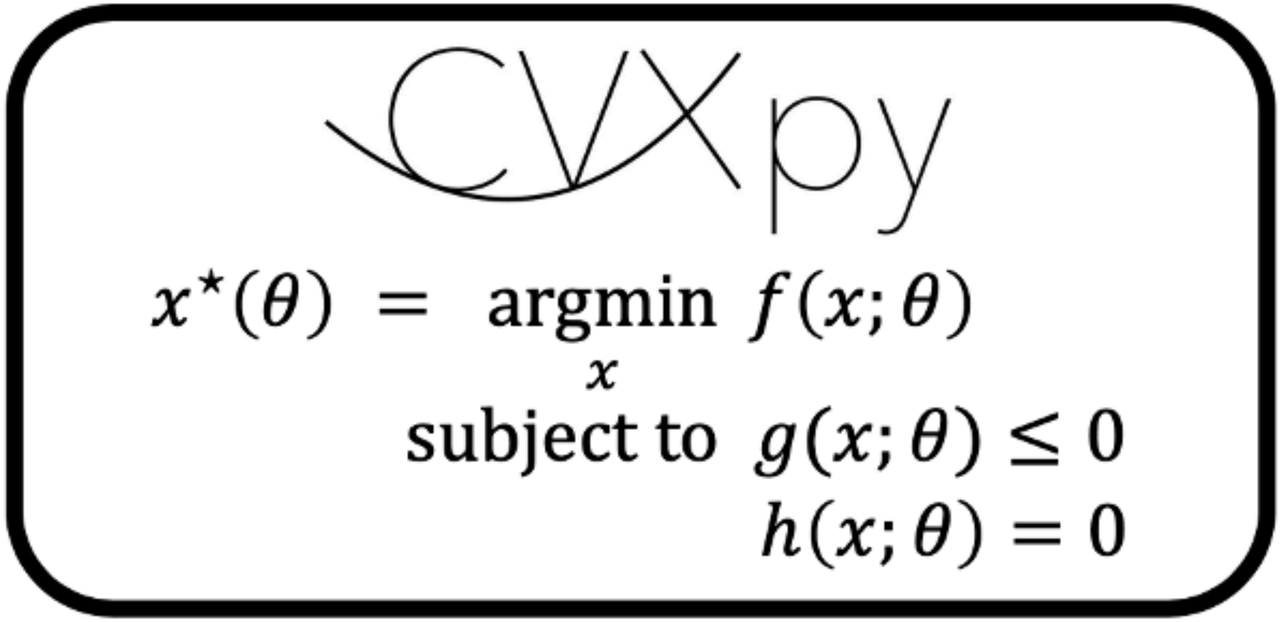


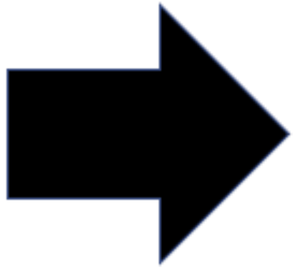
PyTorch embedded modeling language for convex optimization problems

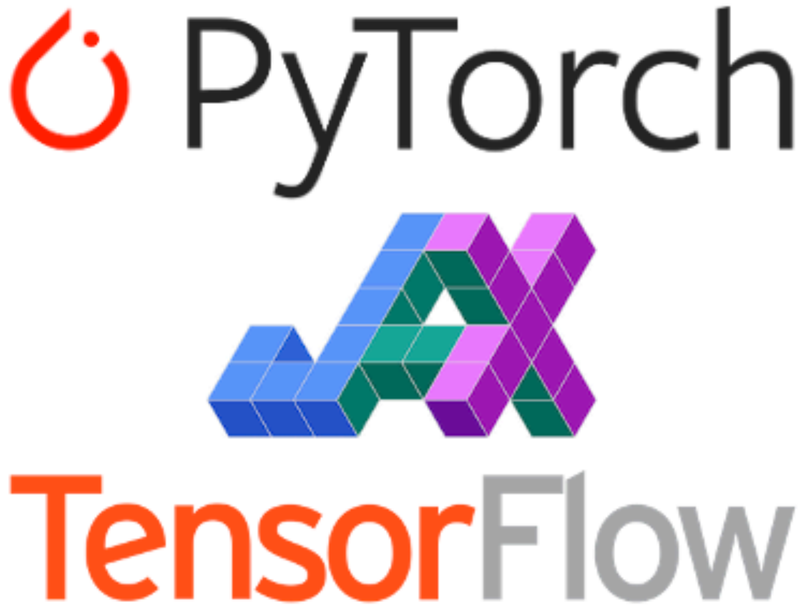
https://www.cvxpy.org/

[Boyd, et al, NIPS, 2019]

```
cvxpylayer = CvxpyLayer(problem, parameters=[A, b], variables=[x])
solution, = cvxpylayer(A, b) # feed-forward pass (solve the problem)
solution.sum().backward()    # backward pass (how A,b influnce solution)
```



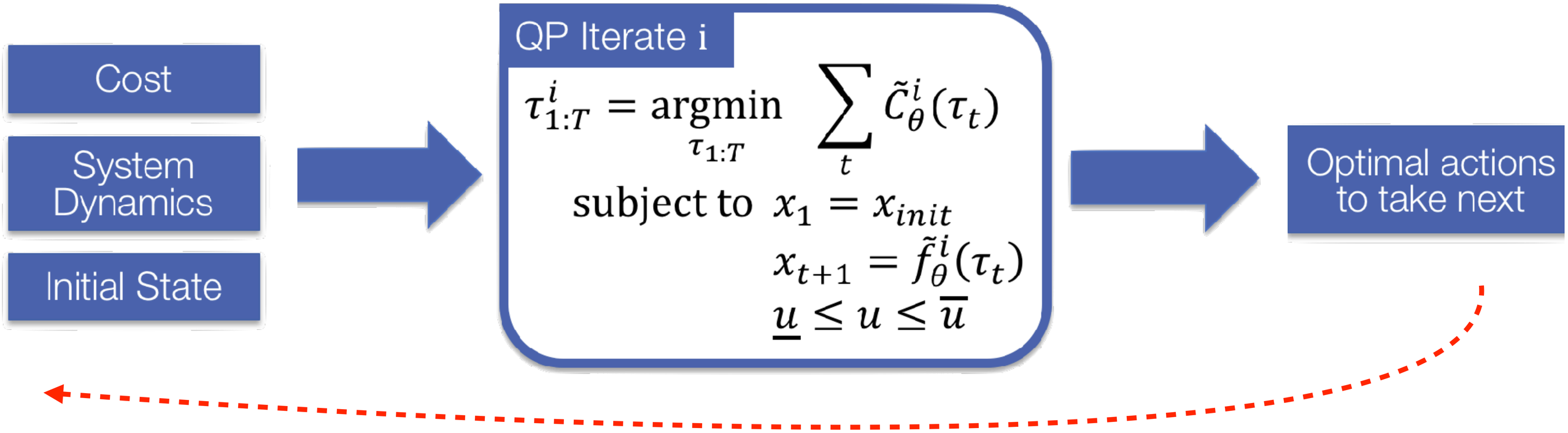




Also optnet: https://github.com/locuslab/optnet

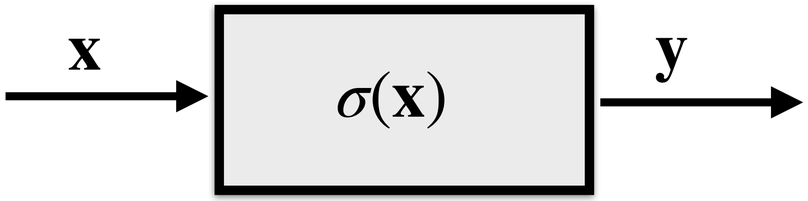
Differentiable MPC control, [Amos et al. NIPS 2018]

https://arxiv.org/abs/1810.13400

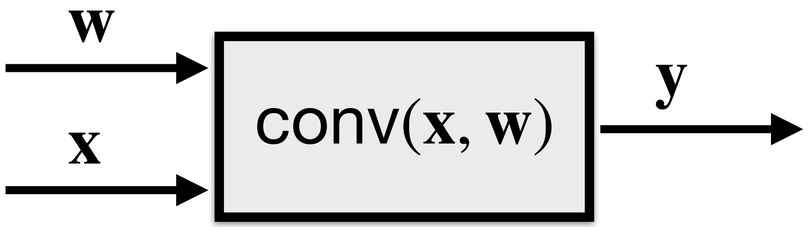


Explicit layers

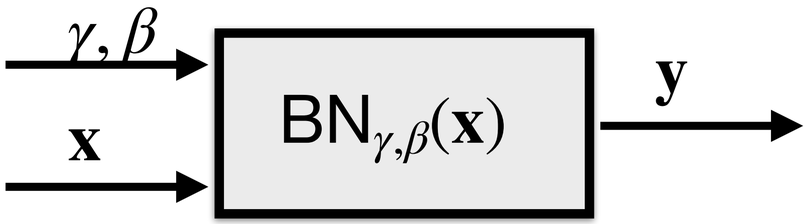
Sigmoid:



Convolution:

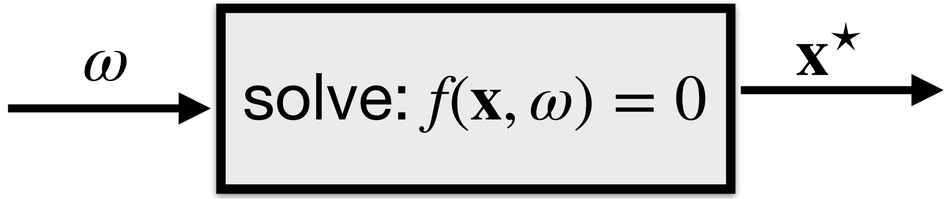


Batch-norm:

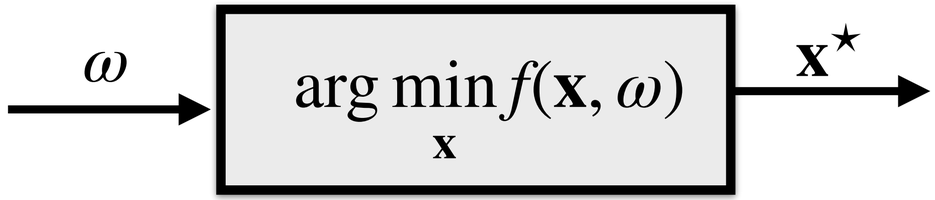


Implicit layers

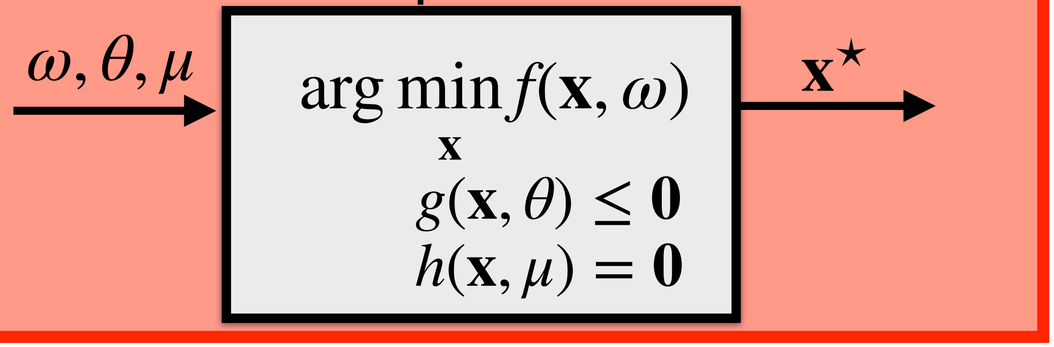
Root finder:



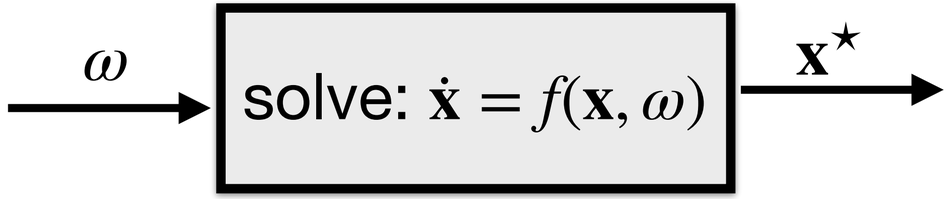
Unconstrained optimizer:



Constrained optimizer:

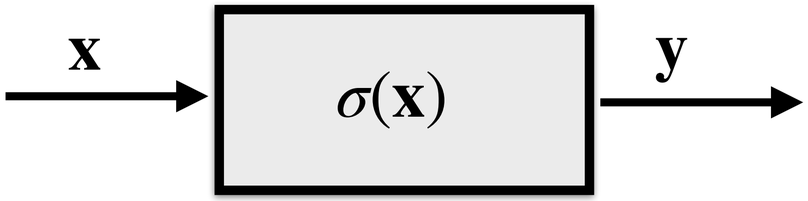


ODE solver:

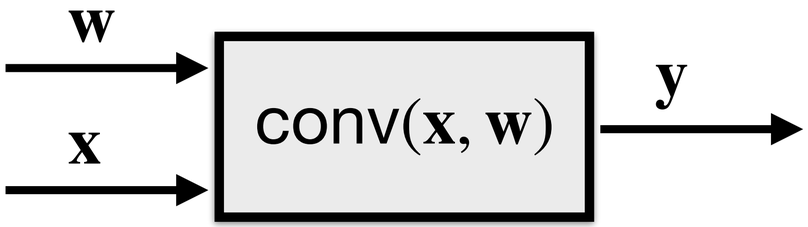


Explicit layers

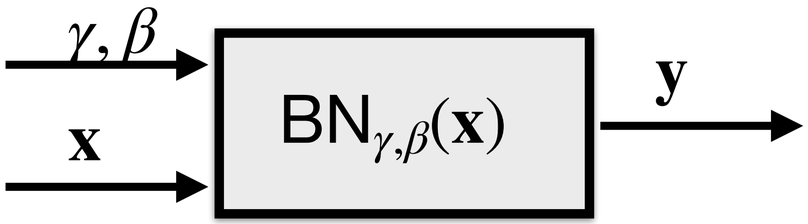
Sigmoid:



Convolution:

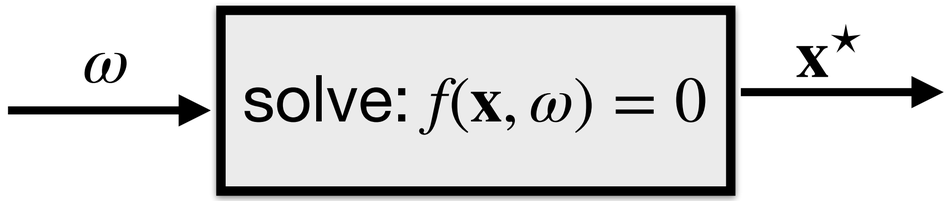


Batch-norm:

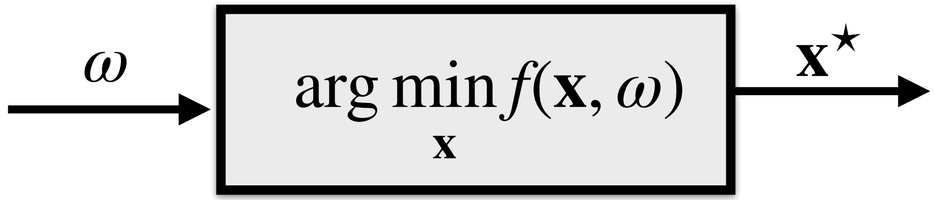


Implicit layers

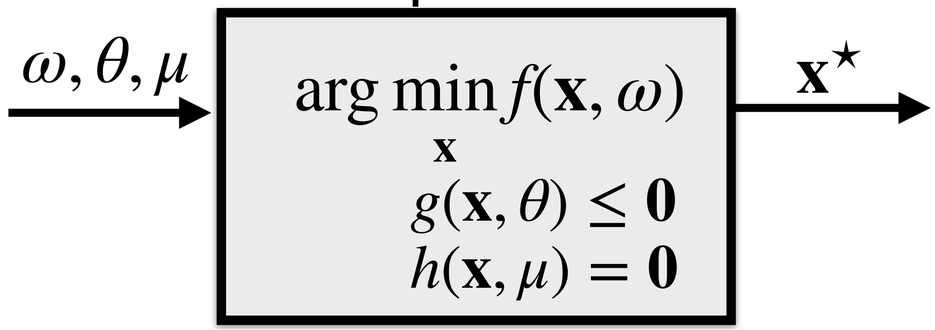
Root finder:



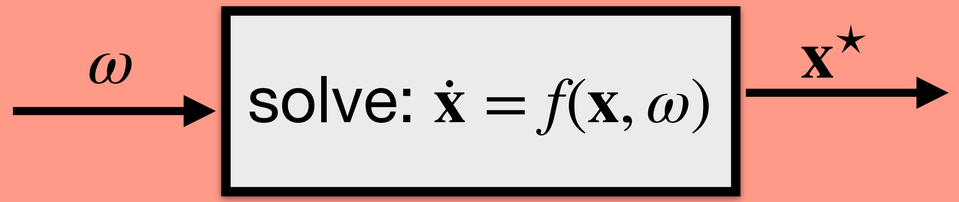
Unconstrained optimizer:



Constrained optimizer:



ODE solver:



ODE solver

$$\dot{\mathbf{x}} = f\left(\mathbf{x}, \omega\right)$$

Any solution $\mathbf{x}^\star(t, \omega)$ (obtained from an arbitrary ODE solver) have to follow:

$$\frac{\partial}{\partial t}\mathbf{x}^\star(t,\omega) - f\left(\mathbf{x}^\star(t,\omega),\omega\right) = 0$$ Given ω, the ODE solution $\mathbf{x}^\star(\omega)$ satisfy this eq.

$$\frac{\partial}{\partial \omega}\frac{\partial}{\partial t}\mathbf{x}^\star(t,\omega) = \frac{\partial f\left(\mathbf{x}^\star(t,\omega),\omega\right)}{\partial \omega}$$ Change of ω generates ODE solution $\mathbf{x}^\star(\omega)$ that satisfy this equation.

$$\frac{\partial}{\partial t}\underbrace{\frac{\partial}{\partial \omega}\mathbf{x}^\star(t,\omega)}_{\mathbf{z}(t,\omega)} = \partial_0 f\left(\mathbf{x}^\star(t,\omega),\omega\right)\underbrace{\frac{\partial \mathbf{x}^\star(t,\omega)}{\partial \omega}}_{\mathbf{z}(t,\omega)} + \partial_1 f\left(\mathbf{x}^\star(t,\omega),\omega\right)$$

$$g\left(\mathbf{z}(t,\omega),\omega\right)$$

$$\frac{\partial}{\partial t}\mathbf{z}(t,\omega) = g\left(\mathbf{z}(t,\omega),\omega,t\right)$$

$$\dot{\mathbf{z}} = g\left(\mathbf{z},\omega,t\right)$$

Since solvers often use adaptive step length, both ODEs solved jointly

ODE solver

Since solvers often use adaptive step length, both ODEs solved jointly

$$\dot{\mathbf{x}} = f(\mathbf{x}, \omega) \qquad \dot{\mathbf{z}} = g(\mathbf{z}, \omega, t)$$

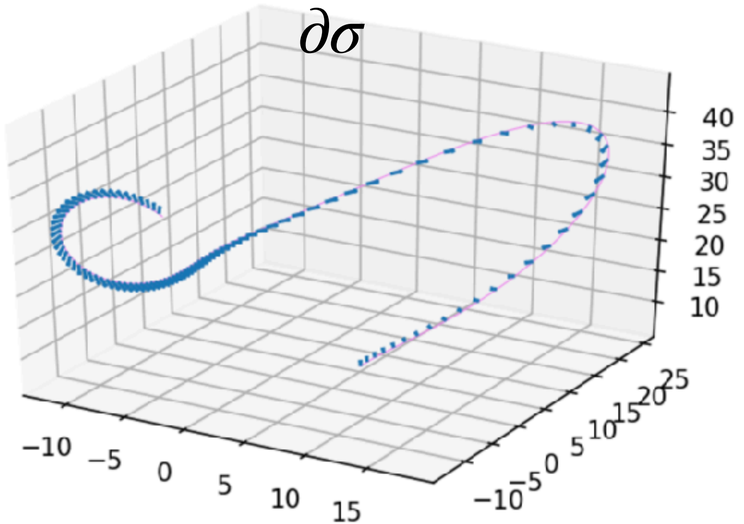
$$\Rightarrow \quad \begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{z}} \end{bmatrix} = \begin{bmatrix} f(\mathbf{x}, \omega) \\ \partial_0 f(\mathbf{x}, \omega)\mathbf{z} + \partial_1 f(\mathbf{x}, \omega) \end{bmatrix} \quad \Rightarrow \quad \mathbf{z}^\star = \frac{\partial \mathbf{x}^\star(t, \omega)}{\partial \omega}$$

Example:

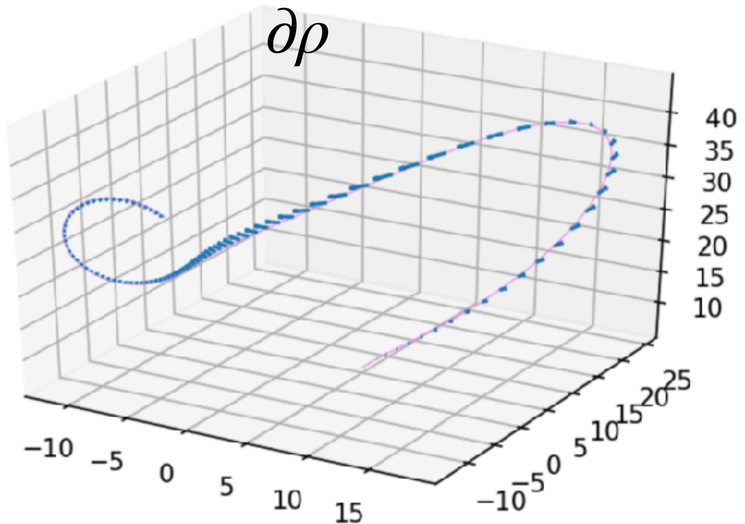
$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = f\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}, \begin{bmatrix} \sigma \\ \rho \\ \beta \end{bmatrix} \right) = \begin{bmatrix} \sigma * (y - x) \\ x(\rho - z) - y \\ xy - \beta z \end{bmatrix}$$

ODE solver: $\mathbf{x}^\star(\sigma, \rho, \beta)$

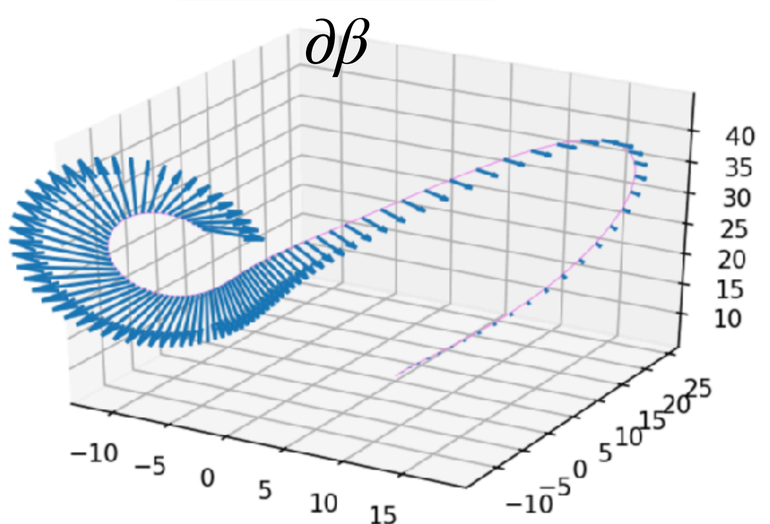
$$\frac{\partial \mathbf{x}^\star(\sigma, \rho, \beta)}{\partial \sigma}$$



$$\frac{\partial \mathbf{x}^\star(\sigma, \rho, \beta)}{\partial \rho}$$

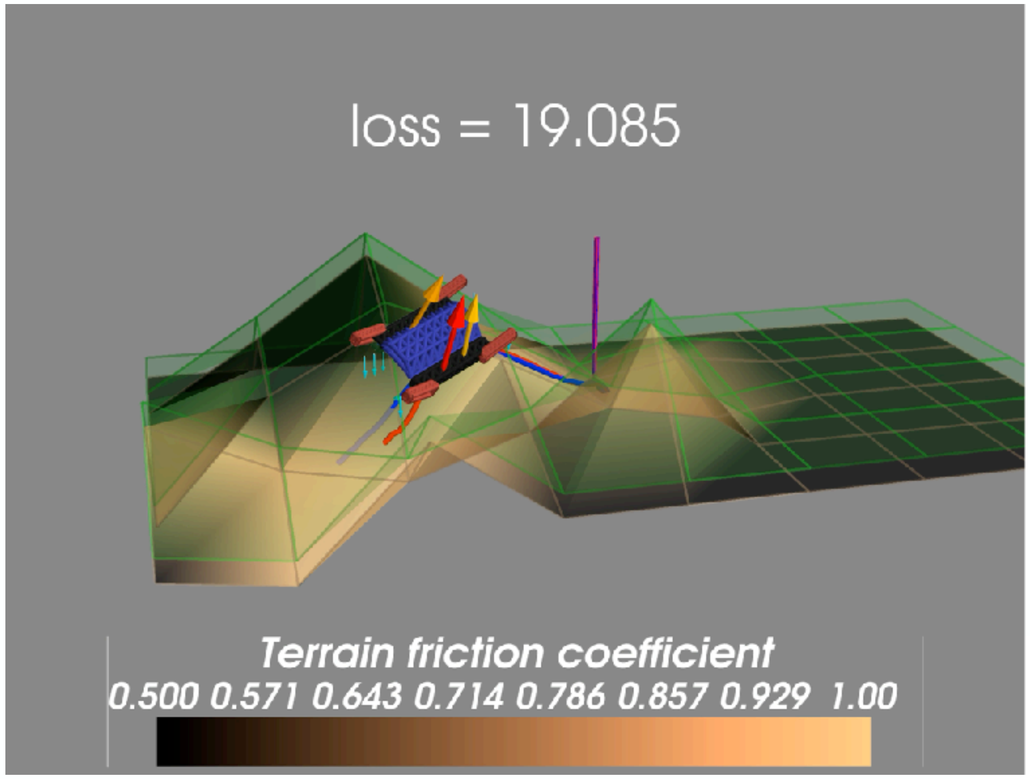


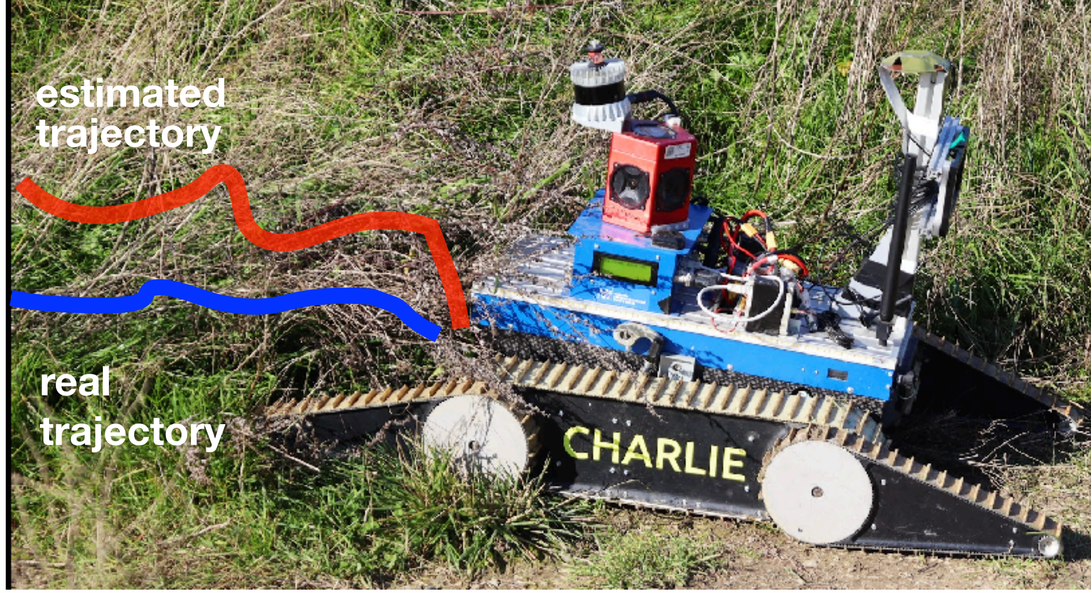
$$\frac{\partial \mathbf{x}^\star(\sigma, \rho, \beta)}{\partial \beta}$$



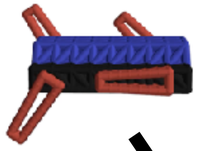
Motivation

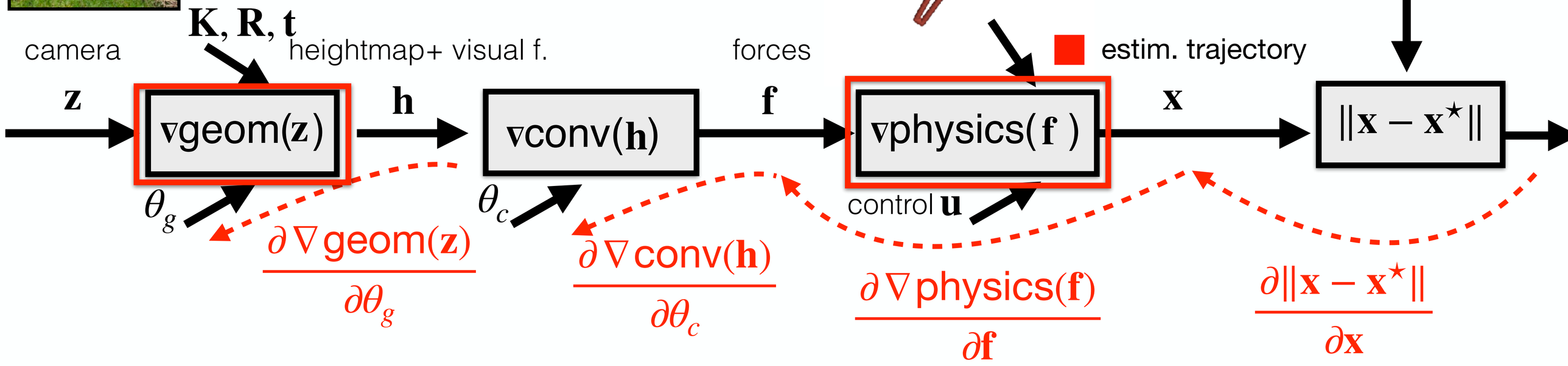
- **Missing piece preventing** reliable **deployment** in wild outdoor environment is a robust **model that predicts robot-terrain behaviour from onboard sensors.**


loss = 19.085
Terrain friction coefficient
0.500 0.571 0.643 0.714 0.786 0.857 0.929 1.00

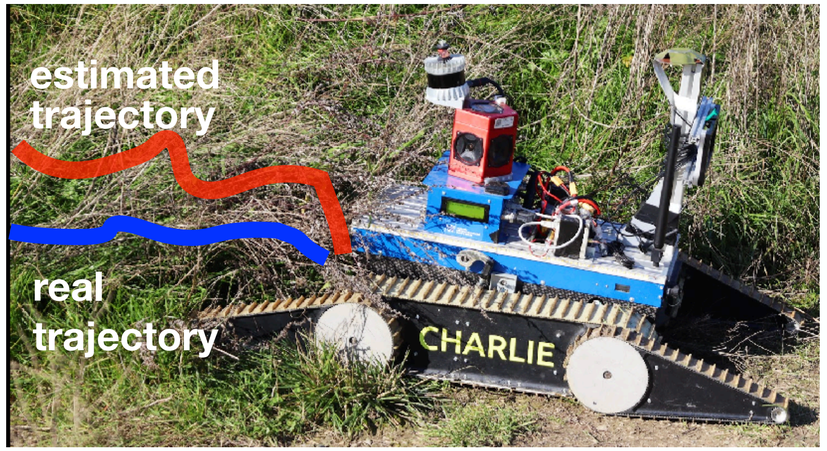

estimated trajectory
real trajectory
CHARLIE


camera




model
real robot trajectory
$\mathbf{x}^\star$
estim. trajectory
$\mathbf{K}, \mathbf{R}, \mathbf{t}$
camera
heightmap+ visual f.
forces
$\mathbf{z}$
$\nabla\text{geom}(\mathbf{z})$
$\mathbf{h}$
$\nabla\text{conv}(\mathbf{h})$
$\mathbf{f}$
$\nabla\text{physics}(\mathbf{f})$
$\mathbf{x}$
$\|\mathbf{x}-\mathbf{x}^\star\|$
θ_g
θ_c
control $\mathbf{u}$
$\frac{\partial \nabla \text{geom}(\mathbf{z})}{\partial \theta_g}$
$\frac{\partial \nabla \text{conv}(\mathbf{h})}{\partial \theta_c}$
$\frac{\partial \nabla \text{physics}(\mathbf{f})}{\partial \mathbf{f}}$
$\frac{\partial \|\mathbf{x}-\mathbf{x}^\star\|}{\partial \mathbf{x}}$

What do we offer to robotics community?



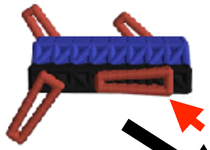
camera recalibration

model identification

parameter learning

control, RL planning

$\mathbf{K}, \mathbf{R}, \mathbf{t}$



$\theta_c \;\; \theta_g$

$\mathbf{u}$

real robot trajectory

$\mathbf{x}^\star$

estim. trajectory

$\mathbf{x}$

$\mathbf{z}$

onboard camera

end-to-end differentiable, $10^5 \times$ real time, image conditioned simulation

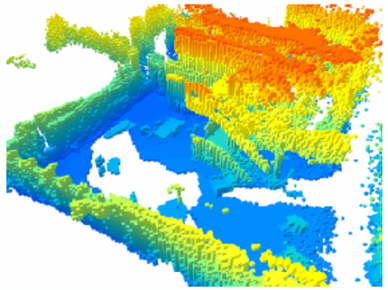
$\|\mathbf{x} - \mathbf{x}^\star\|$

Interpretable Intermediate Outputs: heightmaps, forces, contacts, terrain properties

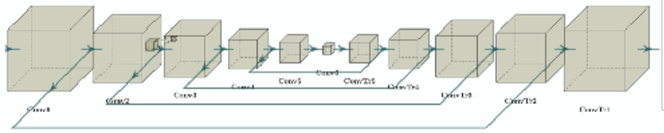
self-supervision

Fixedwing model

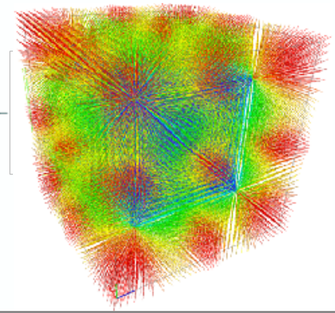
3D gridmap with temperatures



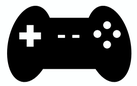
Minkowski U-net



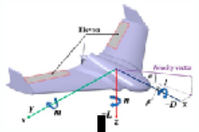
predicted 3D force field f



control

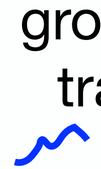


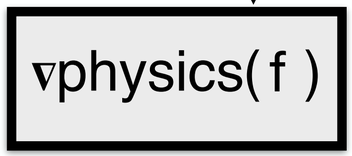
model

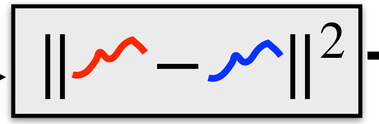


predicted trajectory

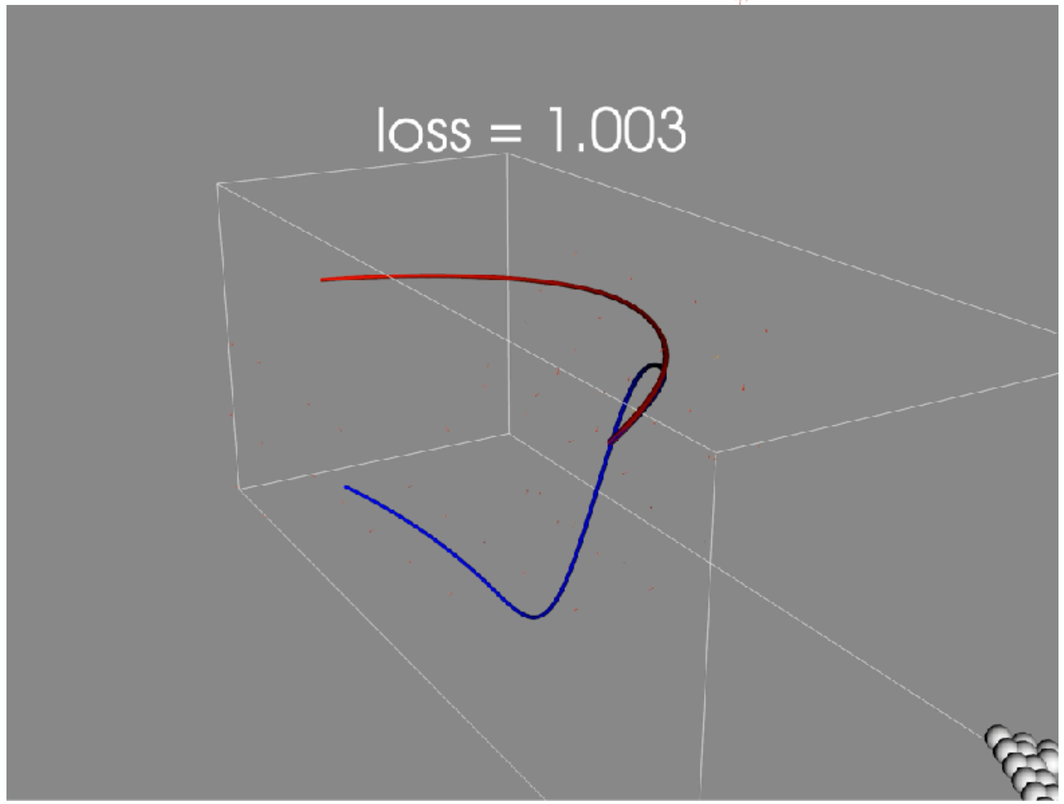
ground truth trajectory







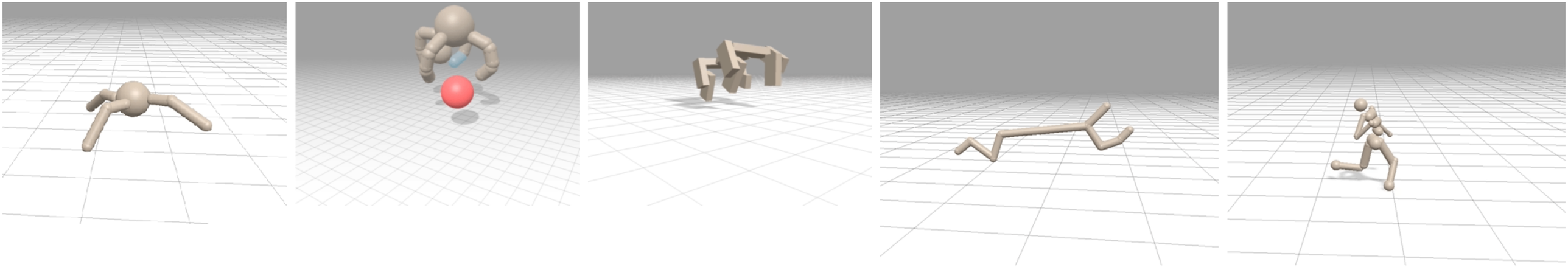
$\mathcal{L}$



- predicted flow have to satisfy Navier-Stokes equations
- add NS-loss: ||NS(f)||

Google’s BRAX/ Nvidia’s WARP - differentiable physics engine

https://github.com/google/brax



Brax simulates these environments at millions of physics steps per second on TPU

BRAX

NVIDIA WARP

https://developer.nvidia.com/warp-python

Cloth simulation

Particle-based simulation

Grad SLAM [Murthy, ICRA, 2021]

https://gradslam.github.io/

SLAM

map:

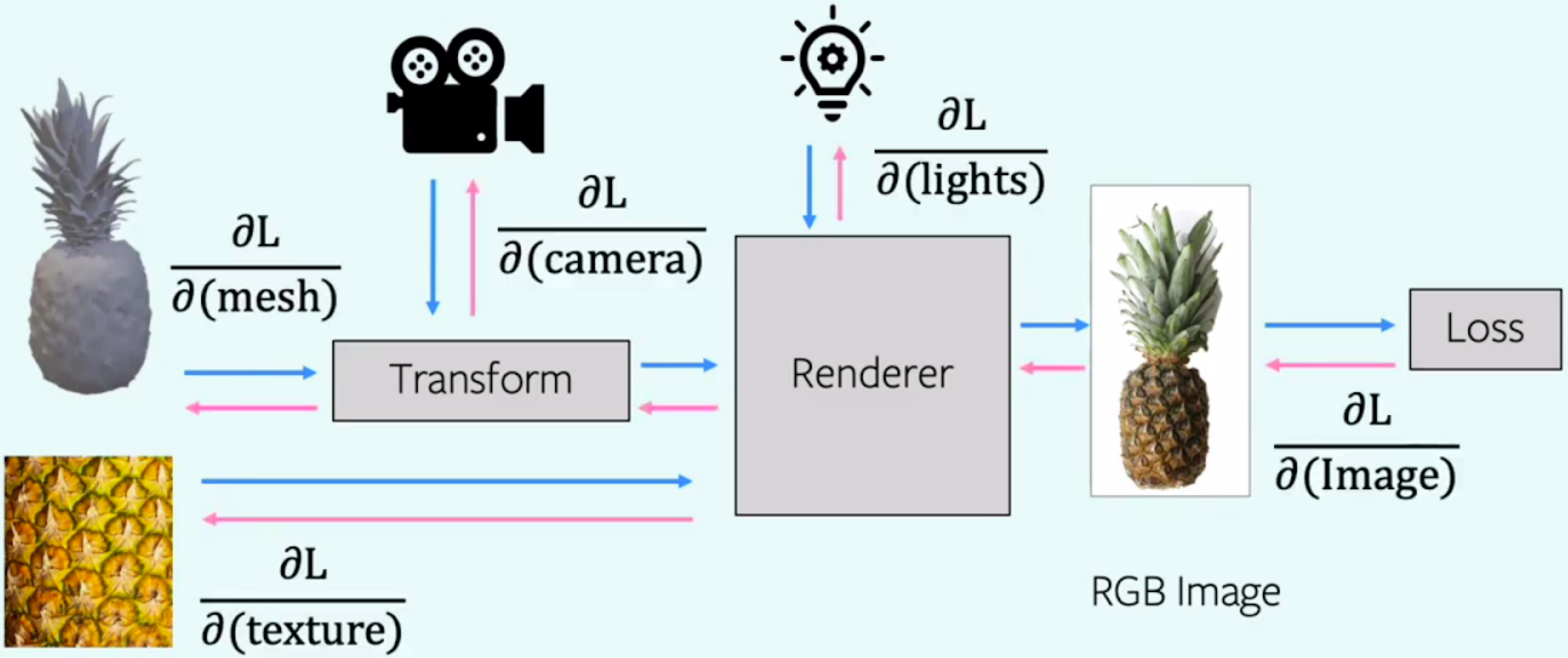
pose: Φ

$$\frac{\partial \mathrm{SLAM}(\mathbf{x}_1, \ldots, \mathbf{x}_t)}{\partial \mathbf{x_1}, \ldots, \mathbf{x_t}} = ?$$

PyTorch3D

https://pytorch3d.org/

DIFFERENTIABLE RENDERING



Summary

- If you know that the mapping is **rendering** use ∇**rendering** layer
- If you know that the mapping is **solution of opt. problem** use ∇**solver** layer
- If you know that the mapping is **ODE solution** use ∇**ODE** layer
- Always use the **right** tool ;-)

References to differentiable physics

1D/2D examples:

https://towardsdatascience.com/physics-informed-neural-networks-pinns-an-intuitive-guide-fff138069563

https://medium.com/@theo.wolf/physics-informed-neural-networks-a-simple-tutorial-with-pytorch-f28a890b874a

https://github.com/benmoseley/harmonic-oscillator-pinn-workshop/blob/main/PINN_intro_workshop.ipynb

Stable simulation:

https://arxiv.org/pdf/2207.05060

https://www.researchgate.net/publication/344464310_Detailed_Rigid_Body_Simulation_using_Extended_Position_Based_Dynamics