EE613 - Machine Learning for Engineers

https://moodle.epfl.ch/course/view.php?id=16819

LINEAR REGRESSION

Sylvain Calinon

Robot Learning and Interaction Group

Idiap Research Institute

Nov 2, 2023

EE613 schedule

Thu. 21.09.2023	(C) 1. ML introduction
Thu. 28.09.2023	(C) 2. Bayesian 1 (C) 3. Bayesian 2
Thu. 12.10.2023	(C) 4. Hidden Markov Models
Thu. 19.10.2023	(C) 5. Dimensionality reduction
Thu. 26.10.2023	(C) 6. Decision trees
Thu. 02.11.2023	(C) 7. Linear regression
Thu. 09.11.2023	(C) 8. Nonlinear regression
Thu. 16.11.2023	(C) 9. Kernel Methods - SVM
Thu. 23.11.2023	(C) 10. Tensor factorization
Thu. 30.11.2023	(C) 11. Deep learning 1
Thu. 07.12.2023	(C) 12. Deep learning 2
Thu. 14.12.2023	(C) 13. Deep learning 3
Thu. 21.12.2023	(C) 14. Deep learning 4

Regression

Nonlinear regression:

Approximation function
(predictor function)
(forecasting function)

$$\boldsymbol{Y} = \boldsymbol{f}(\boldsymbol{X})$$

Output
(response variable)
(outcome variable)

Input
(explanatory variable)
(feature variable)

Linear regression:

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{A}$$

Tensor regression:

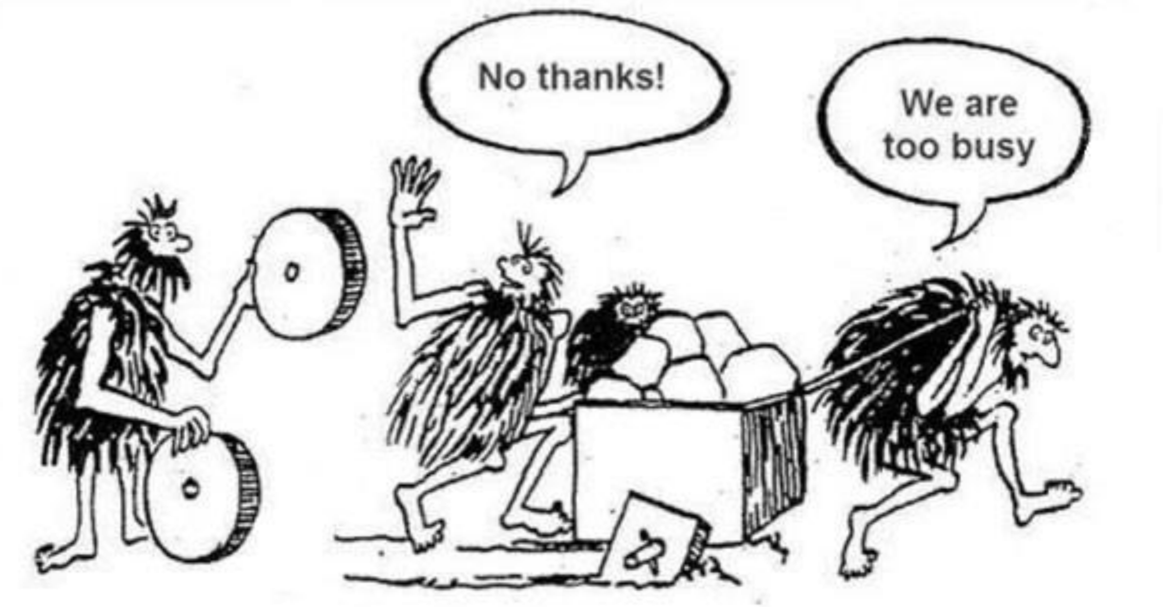
Similar to $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{A}$, but with $\boldsymbol{X}$ a tensor instead of a matrix

Nonlinear regression on x reformulated as linear regression on f:

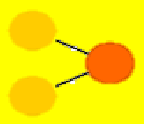
$$\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{f}_1(\boldsymbol{x}) \\ \boldsymbol{f}_2(\boldsymbol{x}) \\ \vdots \end{bmatrix} \boldsymbol{A}$$

LEAST SQUARES

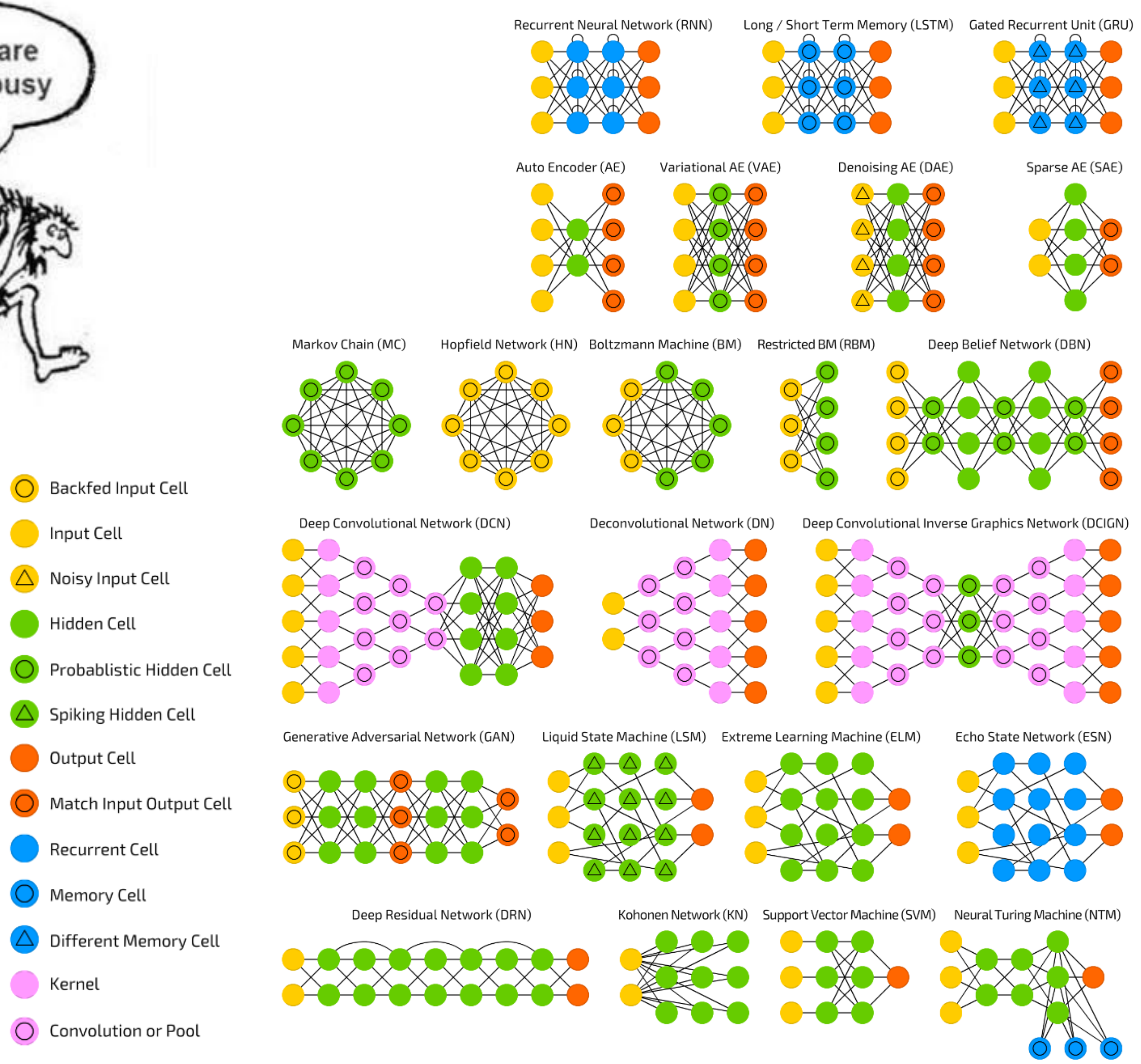
circa 1795



The **pinv-net!** ☺
single layer,
single node,
linear activation!

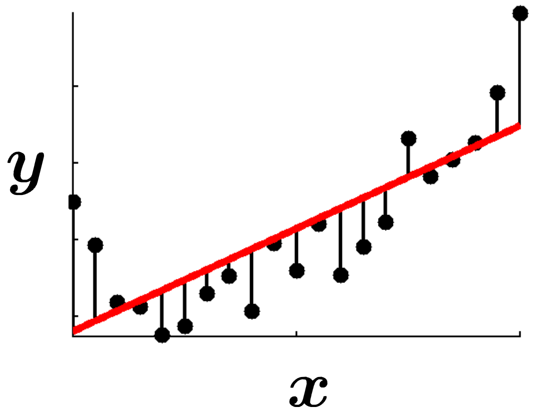


$$\hat{A} = X^{\dagger}\, Y$$



Least squares: a ubiquitous tool

$$\hat{A} = X^{\dagger}\, Y$$



Weighted least squares?

Regularized least squares?

L1-norm instead of L2-norm?

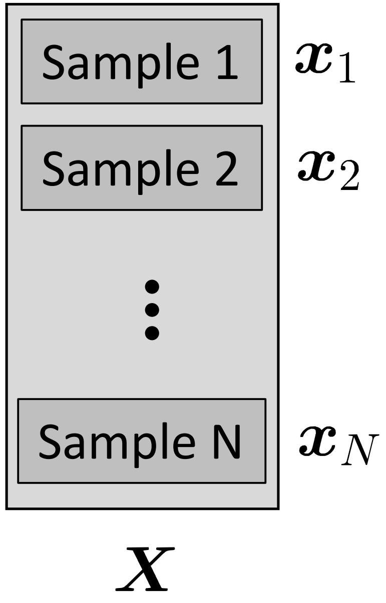
Solution with a secondary objective?

Fast and robust implementation?

Recursive computation?

Multivariate linear regression

By describing the input data as $\boldsymbol{X} \in \mathbb{R}^{N \times D^{\mathcal{I}}}$ and the output data as $\boldsymbol{y} \in \mathbb{R}^{N}$, we want to find $\boldsymbol{a} \in \mathbb{R}^{D^{\mathcal{I}}}$ to have $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{a}$.



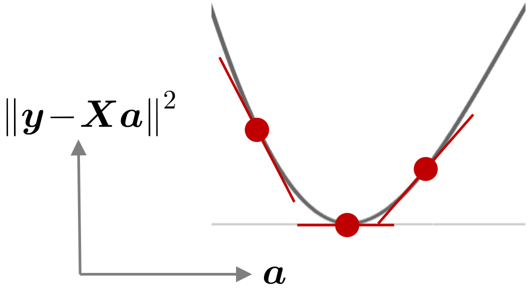
A solution can be found by minimizing the ℓ_2 norm

$$
\begin{aligned}
\hat{\boldsymbol{a}} &= \arg\min_{\boldsymbol{a}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{a}\|^2 \\
&= \arg\min_{\boldsymbol{a}} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{a})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{a}) \\
&= \arg\min_{\boldsymbol{a}} \boldsymbol{y}^\top \boldsymbol{y} - 2\boldsymbol{a}^\top \boldsymbol{X}^\top \boldsymbol{y} + \boldsymbol{a}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{a}.
\end{aligned}
$$

By differentiating with respect to $\boldsymbol{a}$ and equating to zero

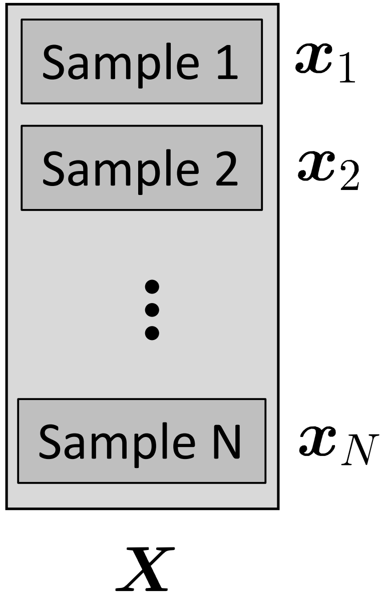
$$
-2\boldsymbol{X}^\top \boldsymbol{y} + 2\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{a} = \mathbf{0} \quad \Longleftrightarrow \quad \hat{\boldsymbol{a}} = \boxed{(\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top} \boldsymbol{y}
$$

Moore-Penrose pseudoinverse $\boldsymbol{X}^\dagger$



Multiple multivariate linear regression

By describing the input data as $\boldsymbol{X} \in \mathbb{R}^{N \times D^{\mathcal{I}}}$ and the output data as $\boldsymbol{Y} \in \mathbb{R}^{N \times D^{\mathcal{O}}}$, we want to find $\boldsymbol{A} \in \mathbb{R}^{D^{\mathcal{I}} \times D^{\mathcal{O}}}$ to have $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{A}$.



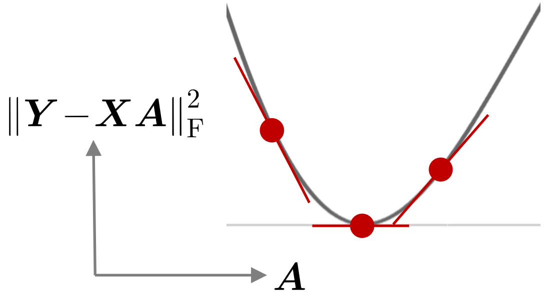
A solution can be found by minimizing the Frobenius norm

$$\begin{aligned}
\hat{\boldsymbol{A}} &= \arg\min_{\boldsymbol{A}} \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A}\|_{\mathrm{F}}^2 \\
&= \arg\min_{\boldsymbol{A}} \operatorname{tr}\Big((\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A})^\top (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A})\Big) \\
&= \arg\min_{\boldsymbol{A}} \operatorname{tr}(\boldsymbol{Y}^\top \boldsymbol{Y} - 2\boldsymbol{A}^\top \boldsymbol{X}^\top \boldsymbol{Y} + \boldsymbol{A}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{A}).
\end{aligned}$$

By differentiating with respect to $\boldsymbol{A}$ and equating to zero

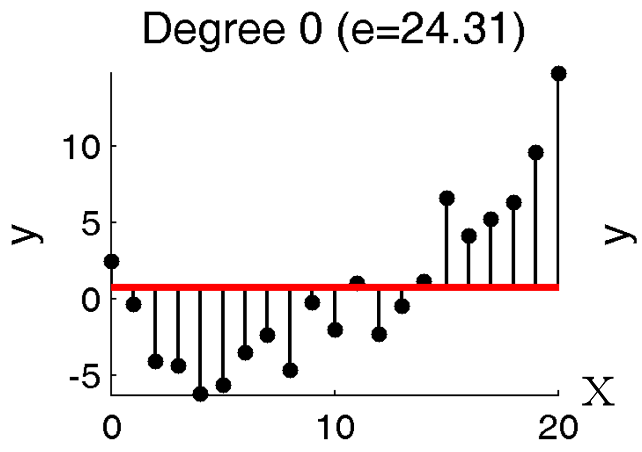
$$-2\boldsymbol{X}^\top \boldsymbol{Y} + 2\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{A} = \boldsymbol{0} \quad \iff \quad \hat{\boldsymbol{A}} = \underbrace{\boxed{(\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top}}_{\boldsymbol{X}^\dagger}\boldsymbol{Y}$$

Moore-Penrose pseudoinverse $\boldsymbol{X}^\dagger$

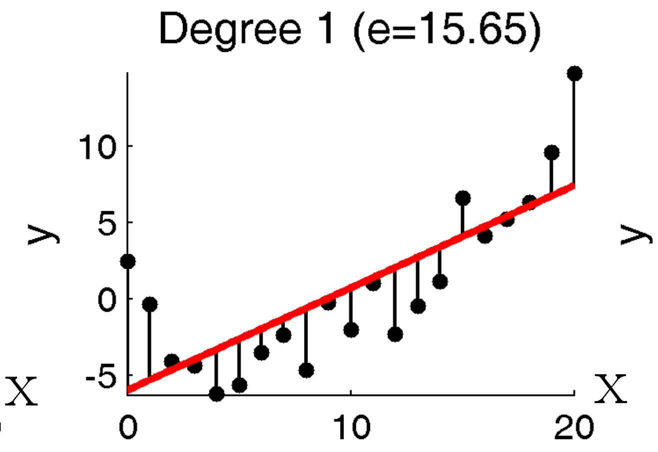


Polynomial fitting with least squares

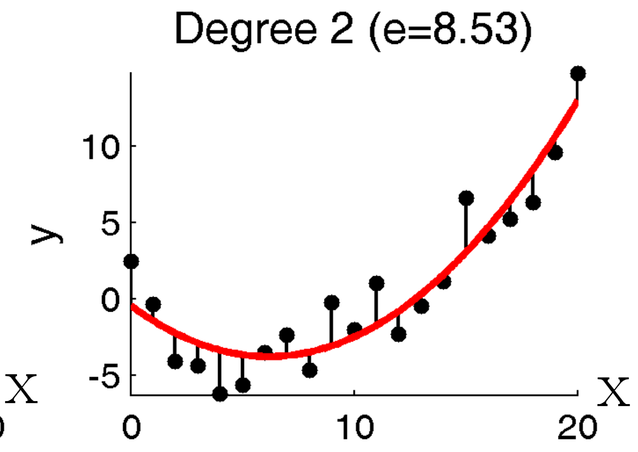
$$\hat{A} = X^{\dagger} Y$$



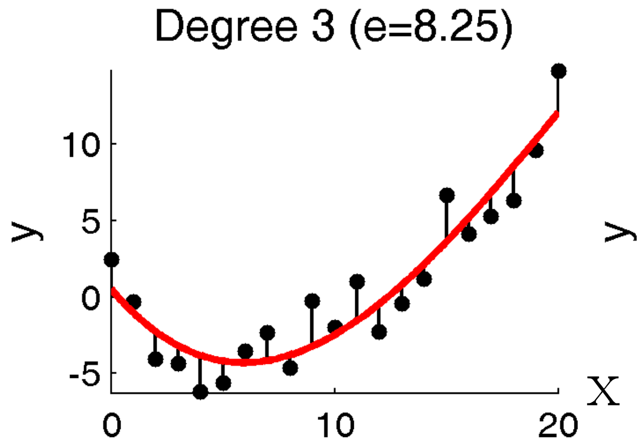
$$x = 1$$



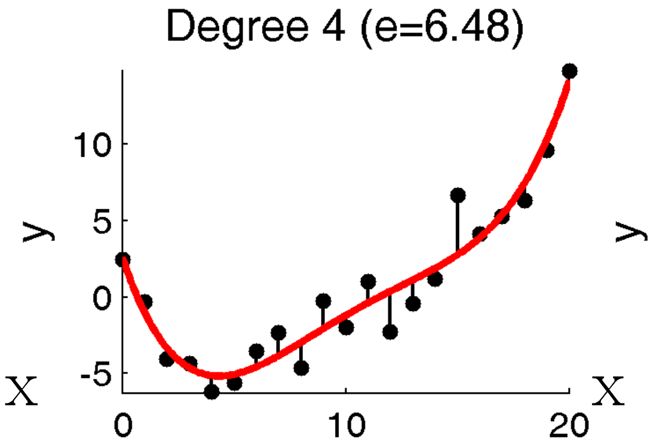
$$\boldsymbol{x} = [1, \mathrm{x}]$$



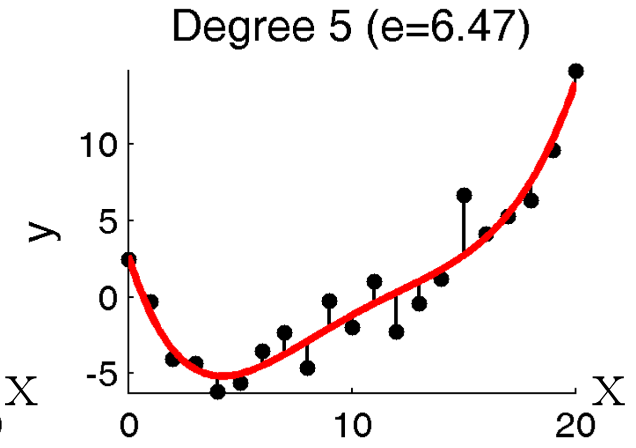
$$\boldsymbol{x} = [1, \mathrm{x}, \mathrm{x}^2]$$



$$\boldsymbol{x} = [1, \mathrm{x}, \mathrm{x}^2, \mathrm{x}^3]$$



$$\boldsymbol{x} = [1, \mathrm{x}, \mathrm{x}^2, \mathrm{x}^3, \mathrm{x}^4]$$



$$\boldsymbol{x} = [1, \mathrm{x}, \mathrm{x}^2, \mathrm{x}^3, \mathrm{x}^4, \mathrm{x}^5]$$

Least squares computed with SVD

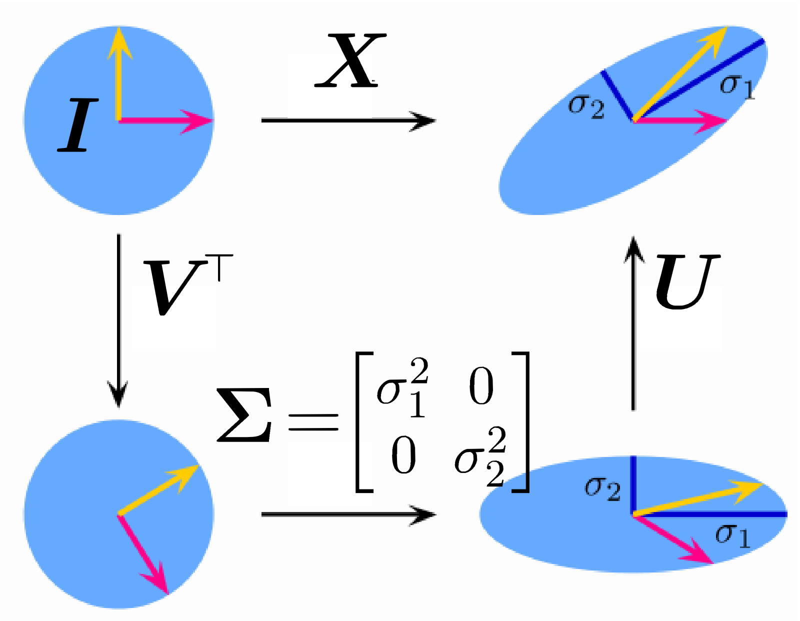
Singular value decomposition (SVD)

$$\overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \end{bmatrix}}^{\boldsymbol{X} \in \mathbb{R}^{N \times D^{\mathcal{I}}}} = \overbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix}}^{\boldsymbol{U} \in \mathbb{R}^{N \times N}} \overbrace{\begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}^{\boldsymbol{\Sigma} \in \mathbb{R}^{N \times D^{\mathcal{I}}}} \overbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \sqrt{0.2} & 0 & 0 & 0 & \sqrt{0.8} \\ 0 & 0 & 0 & 1 & 0 \\ -\sqrt{0.8} & 0 & 0 & 0 & \sqrt{0.2} \end{bmatrix}}^{\boldsymbol{V}^\top \in \mathbb{R}^{D^{\mathcal{I}} \times D^{\mathcal{I}}}}$$

Unitary matrix (orthogonal)

Unitary matrix (orthogonal)

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$$



Least squares computed with SVD

$\boldsymbol{X} \in \mathbb{R}^{N \times D^{\mathcal{I}}}$

$$\hat{\boldsymbol{A}} = \overbrace{(\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top}^{\boldsymbol{X}^\dagger} \boldsymbol{Y}$$

$\boldsymbol{X}$ can be decomposed with the **singular value decomposition**

$$\boldsymbol{X} = \boldsymbol{U \Sigma V}^\top$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are $N \times N$ and $D^{\mathcal{I}} \times D^{\mathcal{I}}$ orthogonal matrices, and $\boldsymbol{\Sigma}$ is an $N \times D^{\mathcal{I}}$ matrix with all its elements outside of the main diagonal equal to 0.

$$\boldsymbol{\Sigma} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

With this decomposition, a solution to the least squares problem is given by

$$\hat{\boldsymbol{A}} = \boldsymbol{V \Sigma}^\dagger \boldsymbol{U}^\top \boldsymbol{Y}$$

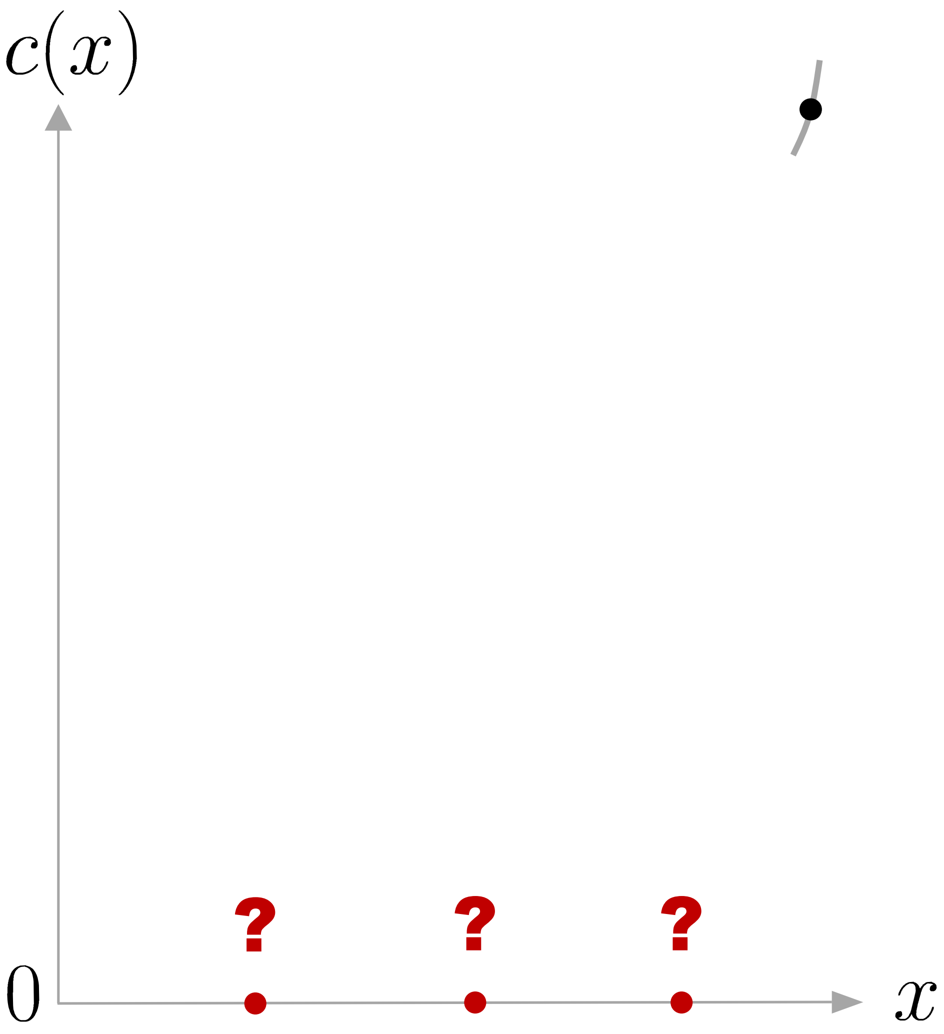
where the pseudoinverse of $\boldsymbol{\Sigma}$ can be easily obtained by inverting the non-zero diagonal elements and transposing the resulting matrix.

$$\boldsymbol{\Sigma}^\dagger = \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Newton's method

(least squares problem solved at each iteration of the optimization process)

Newton's method for minimization



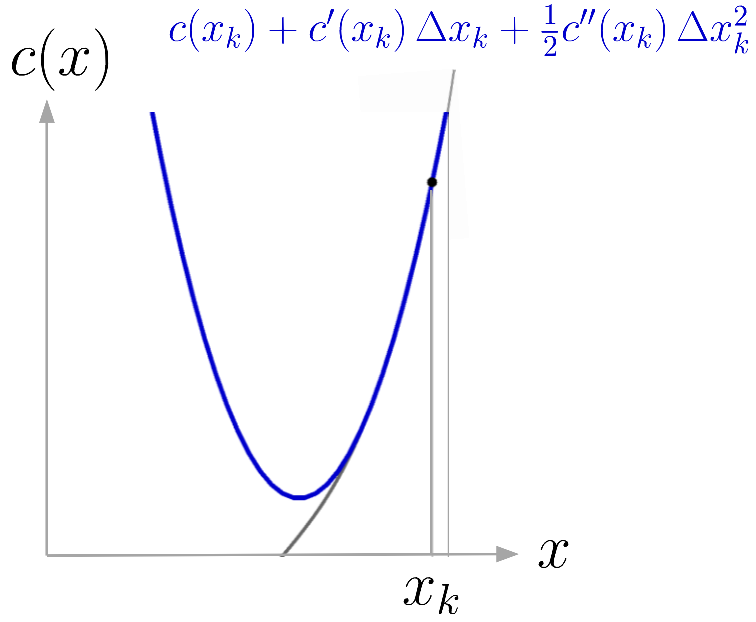
Newton's method for minimization

Newton's method attempts to solve $\min_x c(x)$ or $\max_x c(x)$ from an initial guess x_1 by using a sequence of second-order Taylor approximations of $c(x)$ around the iterates.

The second-order Taylor expansion of $c(x)$ around x_k is

$$c(x_k + \Delta x_k) \approx c(x_k) + c'(x_k)\,\Delta x_k + \frac{1}{2} c''(x_k)\,\Delta x_k^2.$$

The next iterate $x_{k+1} = x_k + \Delta x_k$ is defined so as to minimize this quadratic approximation in Δx_k.



Newton's method for minimization

If the second derivative is positive, the quadratic approximation is a convex function of Δx_k, and its minimum can be found by setting the derivative to zero.

Since

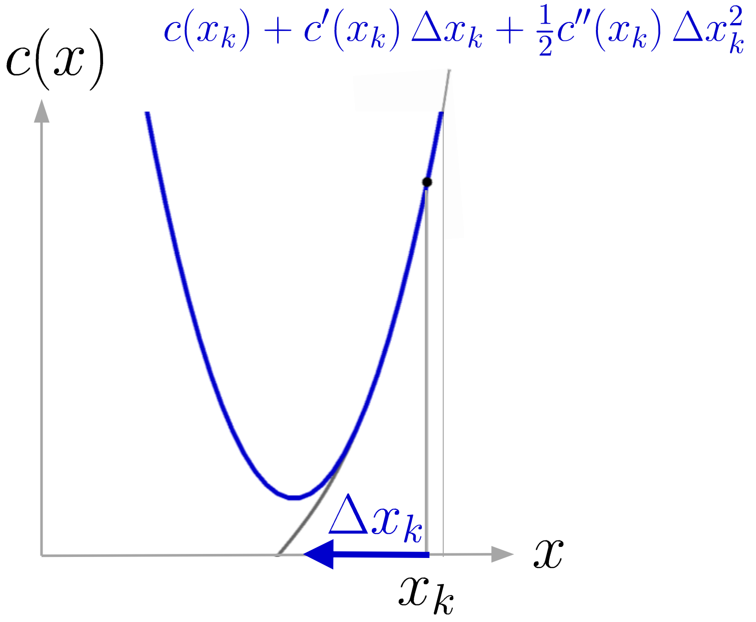
$$\frac{\mathrm{d}}{\mathrm{d\Delta x_k}}\left(c(x_k) + c'(x_k)\,\Delta x_k + \frac{1}{2}c''(x_k)\,\Delta x_k^2\right) = c'(x_k) + c''(x_k)\,\Delta x_k = 0,$$

the minimum is achieved for

$$\Delta x_k = -\frac{c'(x_k)}{c''(x_k)}.$$

Newton's method thus performs the iteration

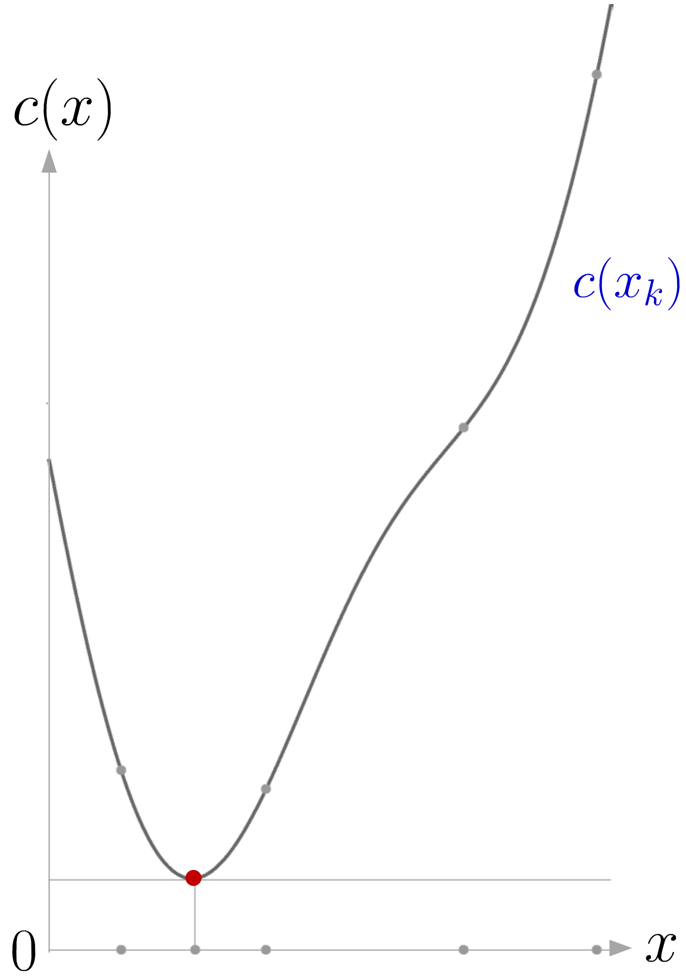
$$\boxed{x_{k+1} = x_k - \frac{c'(x_k)}{c''(x_k)}.}$$



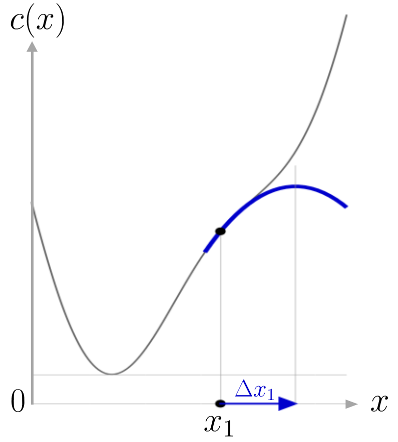
Newton's method for minimization

$$x_{k+1} = x_k - \frac{c'(x_k)}{c''(x_k)}$$

The geometric interpretation of Newton's method is that at each iteration, the goal is to **fit a paraboloid** to the surface of c(x) at x_k, having the same slope and curvature as the surface at that point, and then move to the maximum or minimum of that paraboloid.



$$c(x_k) + c'(x_k)\,\Delta x_k + \tfrac{1}{2}c''(x_k)\,\Delta x_k^2$$



$c''(x)$ must be positive to find a minimum.

If f happens to be a quadratic function, then the exact extremum is found in one step.

Newton's method is often modified to include a small step size (either constant or estimated).

Newton's method for minimization

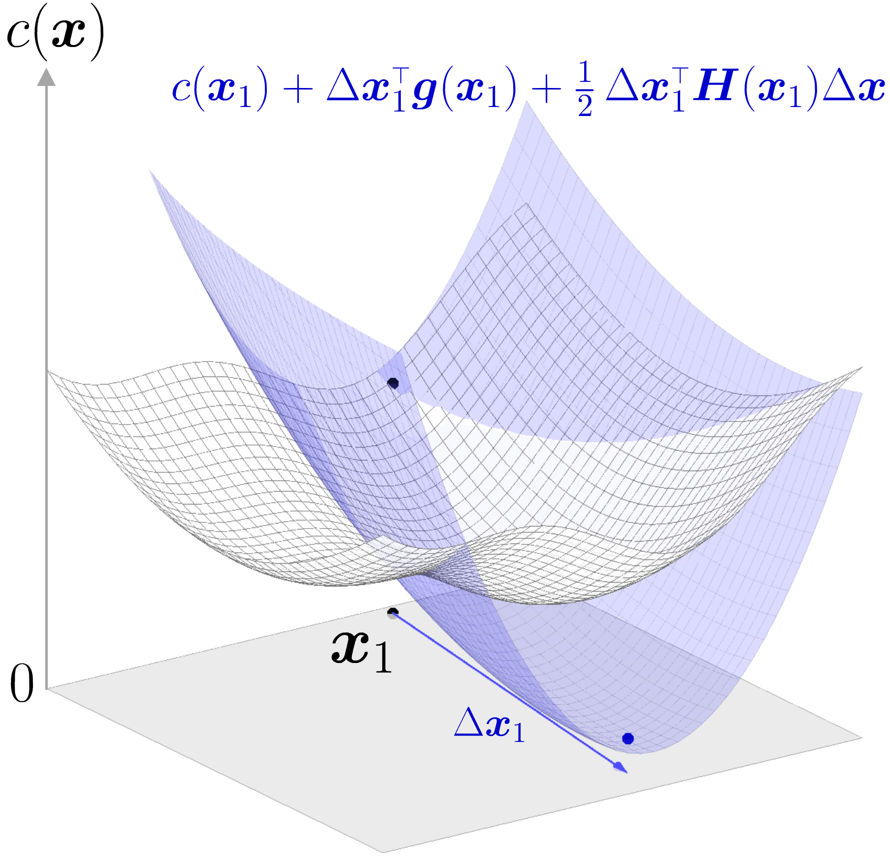
(1D case)

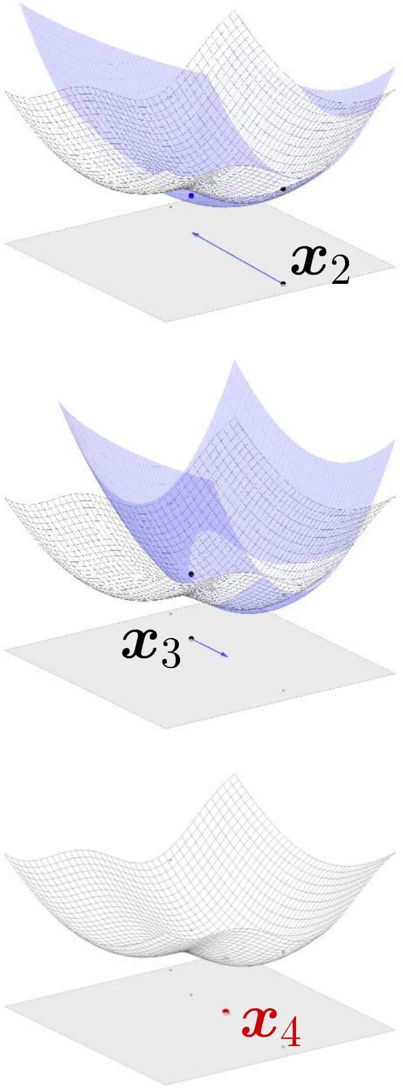
$$x_{k+1} = x_k - \frac{c'(x_k)}{c''(x_k)}$$

The multidimensional case similarly provides

$$\boxed{\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{H}(\boldsymbol{x}_k)^{-1}\boldsymbol{g}(\boldsymbol{x}_k),}$$

with $\boldsymbol{g}$ and $\boldsymbol{H}$ the **gradient** and **Hessian** matrix of $\boldsymbol{f}$.





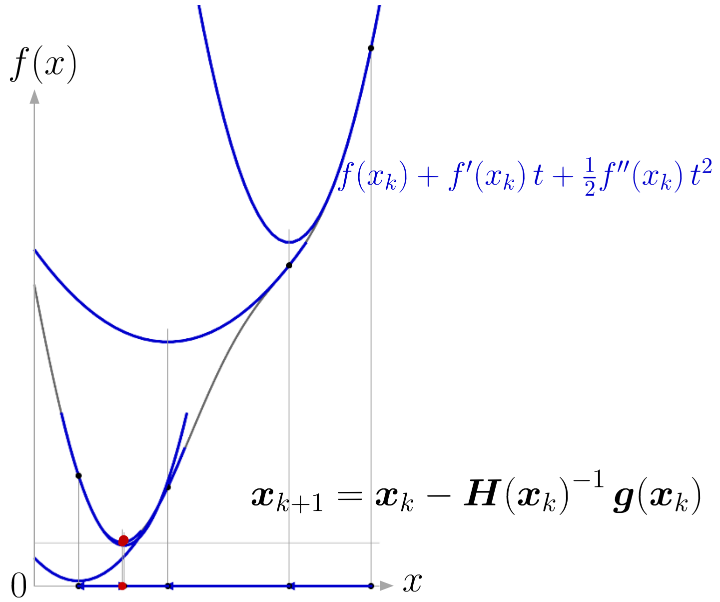
Example: Robot inverse kinematics (IK)

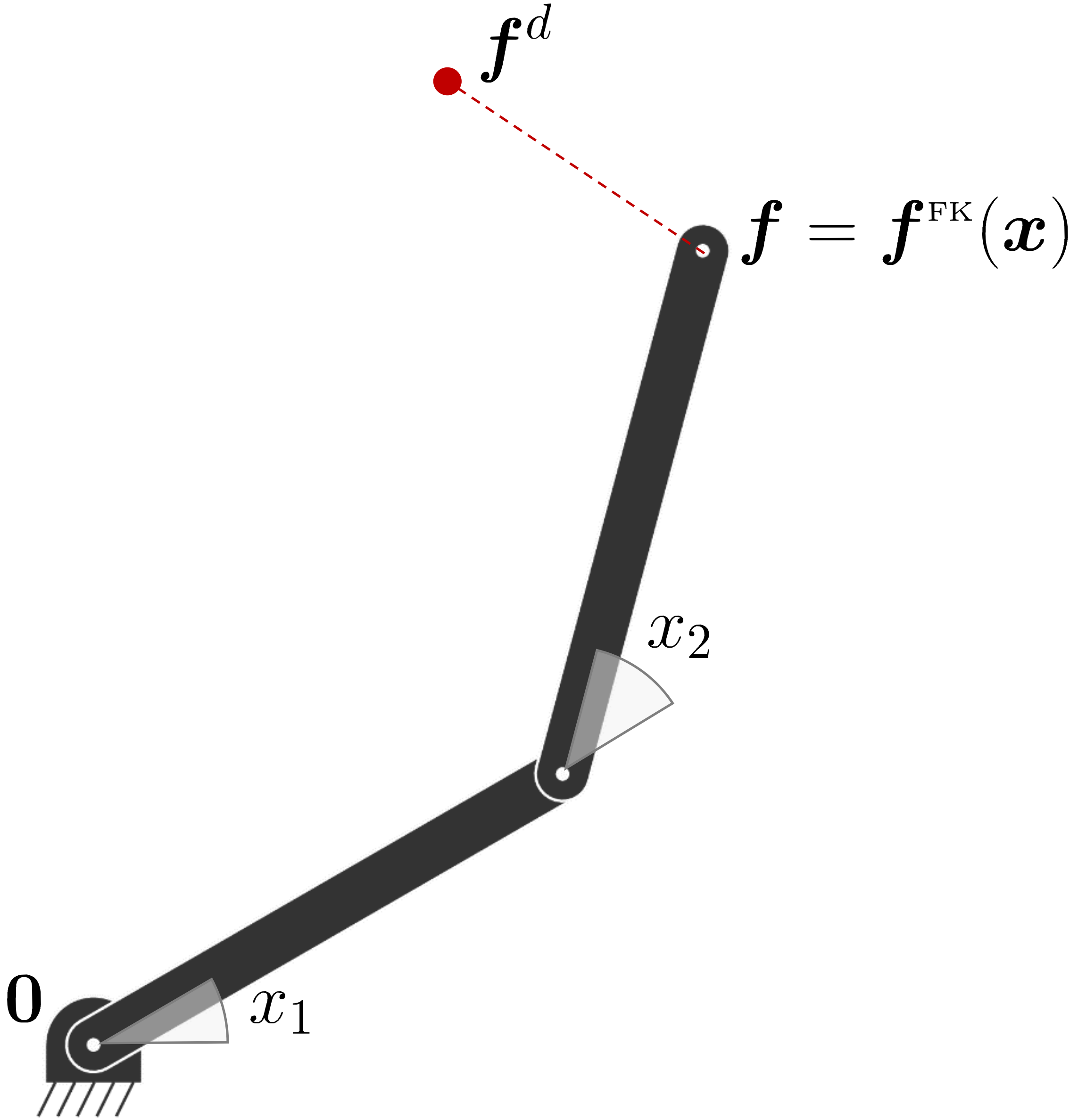
Forward kinematics (FK): $\boldsymbol{f} = \boldsymbol{f}^{\text{FK}}(\boldsymbol{x})$

Inverse kinematics (IK): $\boldsymbol{x} = \boldsymbol{f}^{\text{FK}-1}(\boldsymbol{f})$?

IK in practice: Minimizing $\left\| \boldsymbol{f}^d - \boldsymbol{f}^{\text{FK}}(\boldsymbol{x}) \right\|^2$

→ **Newton's method**





Newton's method applied to IK: Gauss-Newton algorithm

The **Gauss-Newton algorithm** is a special case of Newton's method in which the cost is quadratic (sum of squared function values), with $c(\boldsymbol{x}) = \sum_{i=1}^{D_{\boldsymbol{f}}} r_i^2(\boldsymbol{x}) = \boldsymbol{r}^\top(\boldsymbol{x})\,\boldsymbol{r}(\boldsymbol{x})$, and by ignoring the second-order derivative terms of the Hessian.

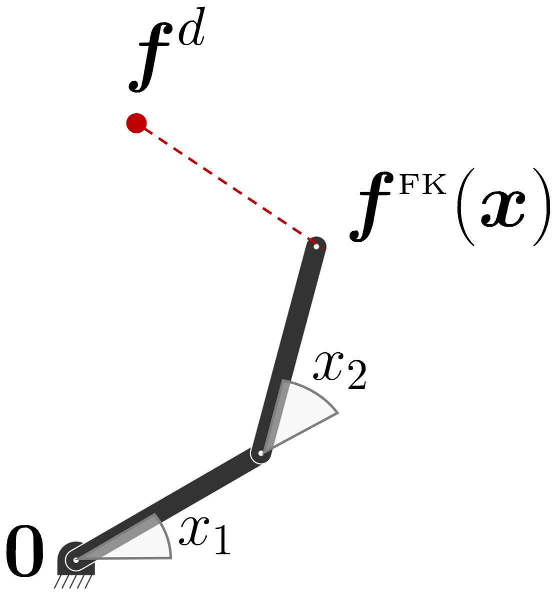
The gradient and Hessian can in this case be computed with

$$\boldsymbol{g}(\boldsymbol{x}) = 2\,\boldsymbol{J}^\top(\boldsymbol{x})\,\boldsymbol{r}(\boldsymbol{x}), \qquad \boldsymbol{H}(\boldsymbol{x}) \approx 2\,\boldsymbol{J}^\top(\boldsymbol{x})\,\boldsymbol{J}(\boldsymbol{x}),$$

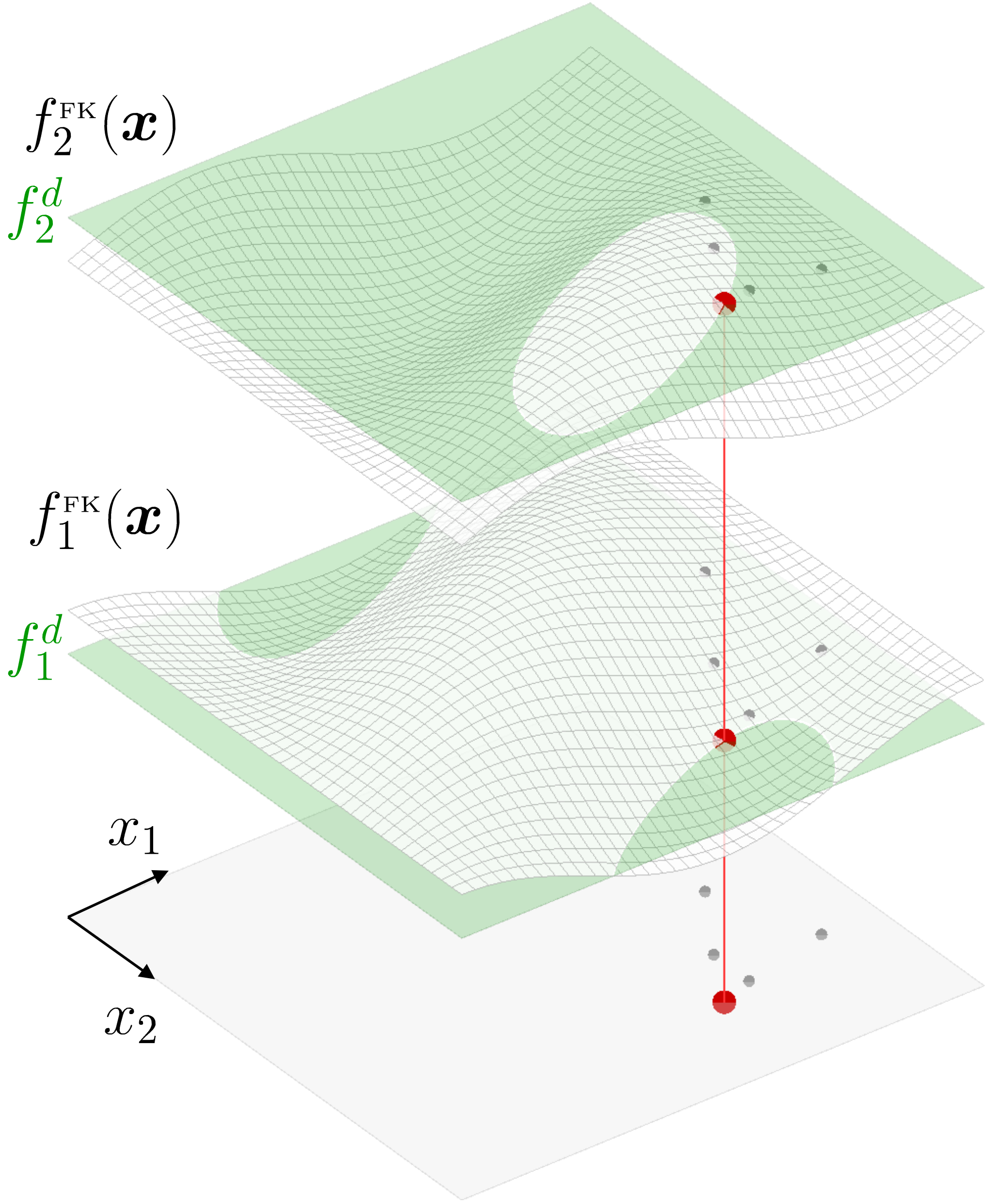
with $\boldsymbol{J}(\boldsymbol{x}) = \frac{\partial \boldsymbol{r}(\boldsymbol{x})}{\partial \boldsymbol{x}} \in \mathbb{R}^{D_{\boldsymbol{f}} \times D_{\boldsymbol{x}}}$ the Jacobian matrix of $\boldsymbol{r}(\boldsymbol{x}) \in \mathbb{R}^{D_{\boldsymbol{f}}}$.

By considering $\boldsymbol{r}(\boldsymbol{x}) = \boldsymbol{f}^{\text{FK}}(\boldsymbol{x}) - \boldsymbol{f}^d$ for our IK problem, the update at each iteration k then becomes

$$\begin{aligned}
\boldsymbol{x}_{k+1} &= \boldsymbol{x}_k - \boldsymbol{H}(\boldsymbol{x}_k)^{-1}\,\boldsymbol{g}(\boldsymbol{x}_k)\\
&= \boldsymbol{x}_k - \underbrace{\left(\boldsymbol{J}^\top(\boldsymbol{x}_k)\boldsymbol{J}(\boldsymbol{x}_k)\right)^{-1}\boldsymbol{J}^\top(\boldsymbol{x}_k)}_{\boldsymbol{J}^\dagger(\boldsymbol{x}_k)}\,\boldsymbol{r}(\boldsymbol{x}_k)\\
&= \boldsymbol{x}_k + \boldsymbol{J}^\dagger(\boldsymbol{x}_k)\left(\boldsymbol{f}^d - \boldsymbol{f}^{\text{FK}}(\boldsymbol{x})\right).
\end{aligned}$$

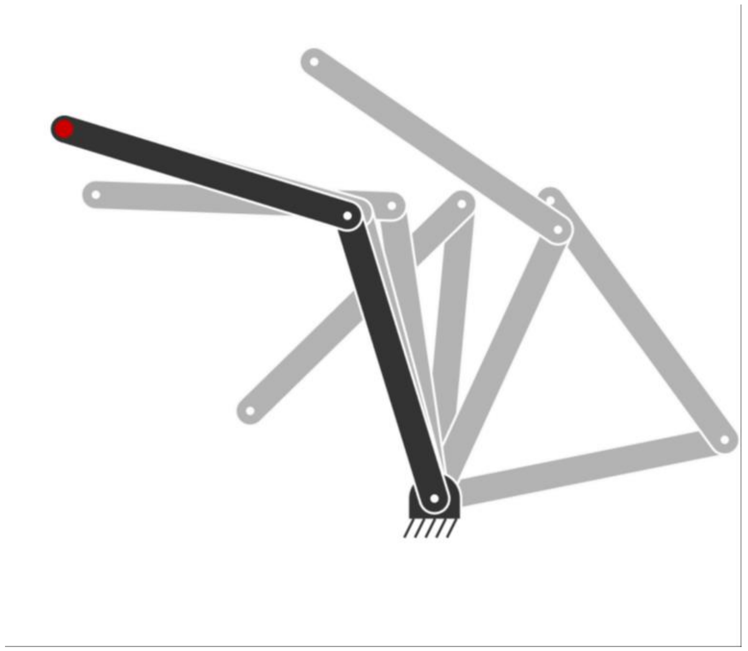


Example: Robot inverse kinematics (IK)



$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{J}^{\dagger}(\boldsymbol{x}_k) \overbrace{\left(\boldsymbol{f}^d - \boldsymbol{f}^{\text{FK}}(\boldsymbol{x})\right)}^{\boldsymbol{r}(\boldsymbol{x})}$$

$$\boldsymbol{J}(\boldsymbol{x}_k) = \begin{bmatrix} \frac{\partial r_1(\boldsymbol{x}_k)}{\partial f_{1,k}} & \frac{\partial r_1(\boldsymbol{x}_k)}{\partial f_{2,k}} \\ \frac{\partial r_2(\boldsymbol{x}_k)}{\partial f_{1,k}} & \frac{\partial r_2(\boldsymbol{x}_k)}{\partial f_{2,k}} \end{bmatrix} \in \mathbb{R}^{2\times 2}$$

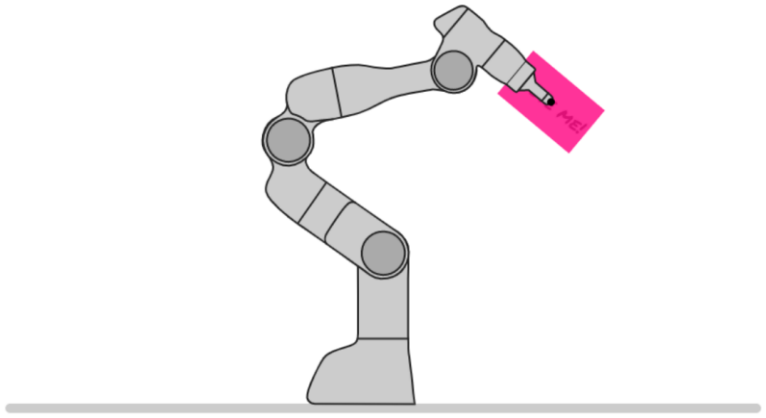


Example: Robot inverse kinematics (IK)

IK | Damped IK | Weighted IK | Prioritized IK

```python
x = [-np.pi/4, np.pi/2, np.pi/4] # Initial robot state
def controlCommand(x, param):
  f = fkin(x, param)
  J = Jkin(x, param)
  u = np.linalg.pinv(J) @ logmap(param.Mu, f) # Position & orientation tracking
  #u = J.T @ logmap(param.Mu, f) * 1E-4 # Gradient-based tracking
  #u = np.linalg.pinv(J[:2,:]) @ (param.Mu[:2] - f[:2]) # Position tracking
  #u = np.linalg.pinv(J[2:,:]) @ (param.Mu[2:] - f[2:]) # Orientation tracking
  #u = np.zeros(param.nbVarX) # Zero control commands
  return u
```

Object orientation



https://robotics-codes-from-scratch.github.io/

Nullspace projection (kernels in least squares)

Python notebook:
demo_LS_polFit.ipynb

Matlab code:
demo_LS_polFit_nullspace01.m

Nullspace projection

The pseudoinverse provides a single least norm solution, but we can sometimes obtain other solutions by employing a **nullspace projection matrix $\boldsymbol{N}$**

$$\hat{\boldsymbol{A}} = \boldsymbol{X}^{\dagger}\boldsymbol{Y} + \overbrace{(\boldsymbol{I} - \boldsymbol{X}^{\dagger}\boldsymbol{X})}^{\boldsymbol{N}}\boldsymbol{V}.$$

$\boldsymbol{V}$ can be any vector/matrix (typically, a gradient minimizing a secondary objective function).

The nullspace projection guarantees that $\|\boldsymbol{Y} - \boldsymbol{X}\hat{\boldsymbol{A}}\|_{\mathrm{F}}^2$ is still minimized.

Nullspace projection computed with SVD

$$\hat{\boldsymbol{A}} = \boldsymbol{X}^{\dagger}\boldsymbol{Y} + \overbrace{\left(\boldsymbol{I} - \boldsymbol{X}^{\dagger}\boldsymbol{X}\right)}^{\boldsymbol{N}}\boldsymbol{V}$$

An alternative way of computing the nullspace projection matrix is to exploit the singular value decomposition

$$\boldsymbol{X}^{\dagger} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\top}$$

to compute

$$\boldsymbol{N} = \tilde{\boldsymbol{U}}\tilde{\boldsymbol{U}}^{\top}$$

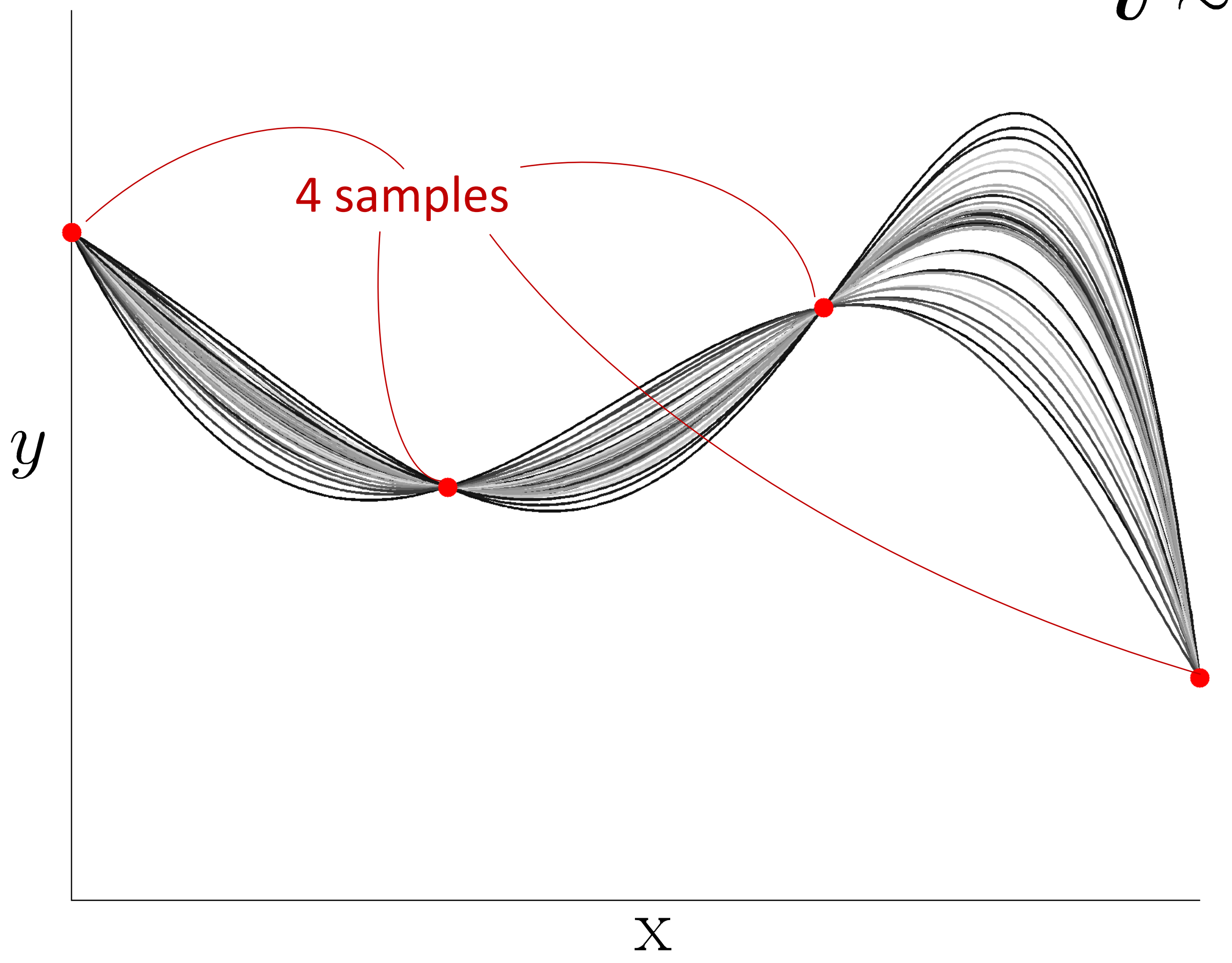
where $\tilde{\boldsymbol{U}}$ is a matrix formed by the columns of $\boldsymbol{U}$ that span for the corresponding zero rows in $\boldsymbol{\Sigma}$.

$$\overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \end{bmatrix}}^{\boldsymbol{X}^{\dagger} \in \mathbb{R}^{D^{\mathcal{I}} \times N}} = \overbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix}}^{\boldsymbol{U} \in \mathbb{R}^{D^{\mathcal{I}} \times D^{\mathcal{I}}}} \overbrace{\begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}^{\boldsymbol{\Sigma} \in \mathbb{R}^{D^{\mathcal{I}} \times N}} \overbrace{\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \sqrt{0.2} & 0 & 0 & 0 & \sqrt{0.8} \\ 0 & 0 & 0 & 1 & 0 \\ -\sqrt{0.8} & 0 & 0 & 0 & \sqrt{0.2} \end{bmatrix}}^{\boldsymbol{V}^{\top} \in \mathbb{R}^{N \times N}}$$

$\tilde{U}$

Example: Polynomial fitting

$$\hat{a} = \boldsymbol{X}^{\dagger}\boldsymbol{y} + \boldsymbol{N}\boldsymbol{v} \quad \text{with} \quad \boldsymbol{x} = [1, \mathrm{x}, \mathrm{x}^2, \ldots, \mathrm{x}^6]$$
$$\boldsymbol{v} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$$



$\hat{\boldsymbol{a}} \in \mathbb{R}^7$

$\boldsymbol{X} \in \mathbb{R}^{4\times 7}$

$\boldsymbol{y} \in \mathbb{R}^4$

$\boldsymbol{N} \in \mathbb{R}^{7\times 7}$

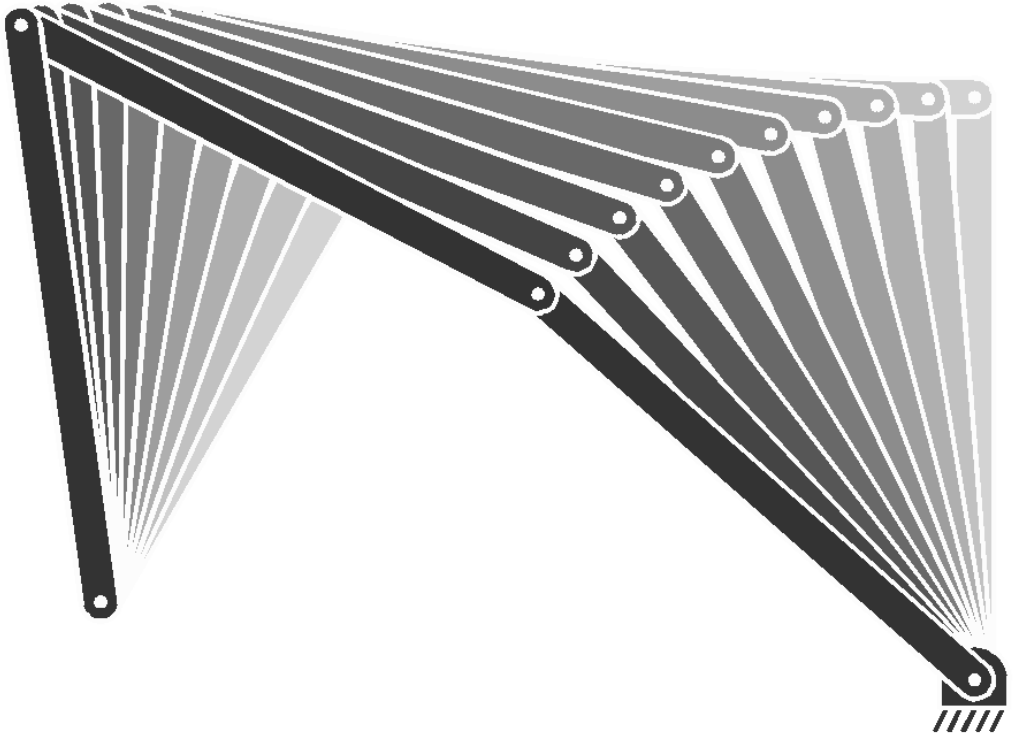
$\boldsymbol{v} \in \mathbb{R}^7$

Example: Robot inverse kinematics

$$\Delta \boldsymbol{x} = \boldsymbol{J}^{\dagger}(\boldsymbol{x})\, \Delta \boldsymbol{f} \;\; + \boldsymbol{N}(\boldsymbol{x})\, \Delta \boldsymbol{x}^{\text{secondary}}$$

$$= \boldsymbol{J}^{\dagger}(\boldsymbol{x}) \begin{bmatrix} 0 \\ 0 \end{bmatrix} \;\; + \boldsymbol{N}(\boldsymbol{x}) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Primary objective:
keeping the tip of the robot still



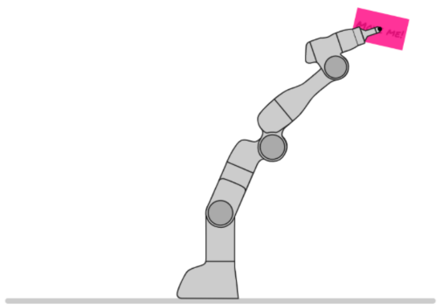
Secondary objective:
moving the first joint

Example: Robot inverse kinematics (single and dual arms)

IK | Damped IK | Weighted IK | Prioritized IK

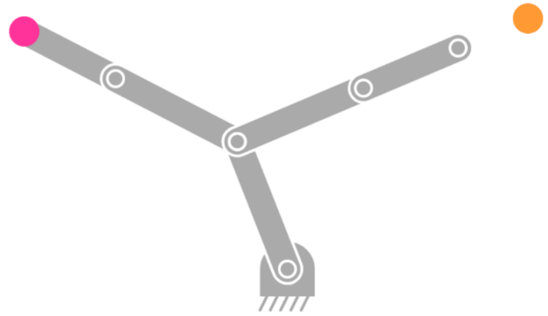
```
x = [-np.pi/4, np.pi/2, np.pi/4] # Initial robot state
def controlCommand(x, param):
  f = fkin(x, param)
  J = Jkin(x, param)

  #N = np.eye(param.nbVarX) - np.linalg.pinv(J[:2,:]) @ J[:2,:] # Nullspace projecti
  #u = u + N @ [1, 0, 0] # Prioritized tracking

  # Prioritized control (position tracking prioritized over orientation tracking)
  dfp = (param.Mu[:2] - f[:2]) * 10 # Position correction
  dfo = (param.Mu[2:] - f[2:]) * 10 # Orientation correction
  Jp = J[:2,:] # Jacobian for position
  Jo = J[2:,:] # Jacobian for orientation
  pinvJp = np.linalg.inv(Jp.T @ Jp + np.eye(param.nbVarX) * 1e-2) @ Jp.T # Damped ps
  Np = np.eye(param.nbVarX) - pinvJp @ Jp # Nullspace projection operator
  up = pinvJp @ dfp # Command for position tracking
  JoNp = Jo @ Np
  pinvJoNp = JoNp.T @ np.linalg.inv(JoNp @ JoNp.T + np.eye(1) * 1e1) #
  uo = pinvJoNp @ (dfo - Jo @ up) # Command for orientation tracking (
  u = up + Np @ uo # Control commands

  return u
```

Object orientation



IK | Prioritized IK

```
def controlCommand(x, param):
  f = fkin(x, param)
  J = Jkin(x, param)
  # Prioritized control (left tracking as main objective)
  dfl = (param.Mu[:2] - f[:2,0]) * 10 # Left hand correction
  dfr = (param.Mu[2:] - f[2:,0]) * 10 # Right hand correction
  Jl = J[:2,:] # Jacobian for left hand
  Jr = J[2:,:] # Jacobian for right hand
  pinvJl = np.linalg.inv(Jl.T @ Jl + np.eye(param.nbVarX) * 1e1) @ Jl.T # Damped pse
  Nl = np.eye(param.nbVarX) - pinvJl @ Jl # Nullspace projection operator
  ul = pinvJl @ dfl # Command for position tracking
  JrNl = Jr @ Nl
  pinvJrNl = JrNl.T @ np.linalg.inv(JrNl @ JrNl.T + np.eye(2) * 1e4) # Damped pseud
  ur = pinvJrNl @ (dfr - Jr @ ul) # Command for right hand tracking (with left hand
  u = ul + Nl @ ur # Control commands
  return u
```



https://robotics-codes-from-scratch.github.io/

Ridge regression
(robust regression, Tikhonov regularization, penalized least squares)

Python notebook:
demo_LS_polFit.ipynb

Matlab example:
demo_LS_polFit02.m

Ridge regression

The least squares objective can be modified to give preference to a particular solution with

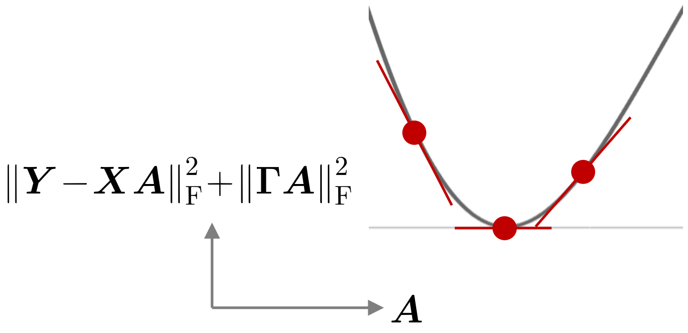
$$\hat{\boldsymbol{A}} = \arg\min_{\boldsymbol{A}} \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A}\|_{\mathrm{F}}^2 + \|\boldsymbol{\Gamma}\boldsymbol{A}\|_{\mathrm{F}}^2$$
$$= \arg\min_{\boldsymbol{A}} \operatorname{tr}\Big((\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A})^\top(\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A})\Big) + \operatorname{tr}\Big((\boldsymbol{\Gamma}\boldsymbol{A})^\top\boldsymbol{\Gamma}\boldsymbol{A}\Big)$$

By differentiating with respect to $\boldsymbol{A}$ and equating to zero, we can see that

$$-2\boldsymbol{X}^\top\boldsymbol{Y} + 2\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{A} + 2\boldsymbol{\Gamma}^\top\boldsymbol{\Gamma}\boldsymbol{A} = \boldsymbol{0}$$

yielding

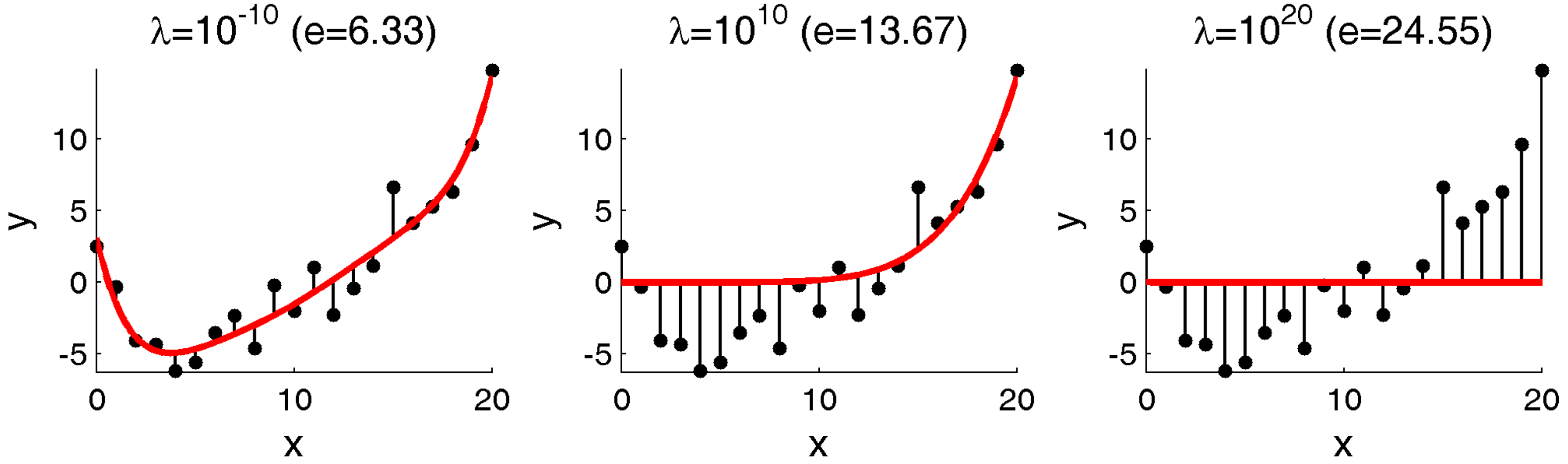
$$\hat{\boldsymbol{A}} = (\boldsymbol{X}^\top\boldsymbol{X} + \boldsymbol{\Gamma}^\top\boldsymbol{\Gamma})^{-1}\boldsymbol{X}^\top\boldsymbol{Y}$$



If $\boldsymbol{\Gamma} = \lambda\boldsymbol{I}$ with $0 < \lambda \ll 1$ (i.e., giving preference to solutions with smaller norms), the process is known as $\boldsymbol{\ell_2}$ **regularization**.

Example: Polynomial fitting

$$D^{\mathcal{I}} = 7 \text{ (polynomial of degree 6)}$$



Ridge regression computed with SVD

For the singular value decomposition

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$$

with σ_i the singular values in the diagonal of $\boldsymbol{\Sigma}$, a solution to the ridge regression problem is given by

$$\hat{\boldsymbol{A}} = \boldsymbol{V}\tilde{\boldsymbol{\Sigma}}\boldsymbol{U}^\top \boldsymbol{Y}$$

where $\tilde{\boldsymbol{\Sigma}}$ has diagonal values

$$\tilde{\sigma}_i = \frac{\sigma_i}{\sigma_i^2 + \lambda}$$

and has zeros elsewhere.

$$\boldsymbol{\Sigma} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\tilde{\boldsymbol{\Sigma}} = \begin{bmatrix} 0.2498 & 0 & 0 & 0 \\ 0 & 0.4988 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(for $\lambda = 0.01$)

Weighted least squares
(Generalized least squares)

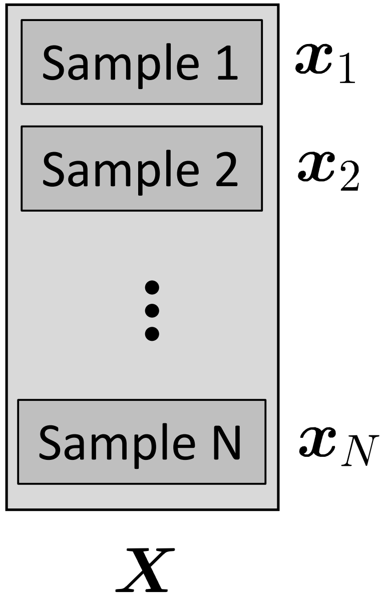
Python notebook:
demo_LS_weighted.ipynb

Matlab example:
demo_LS_weighted01.m

Weighted least squares

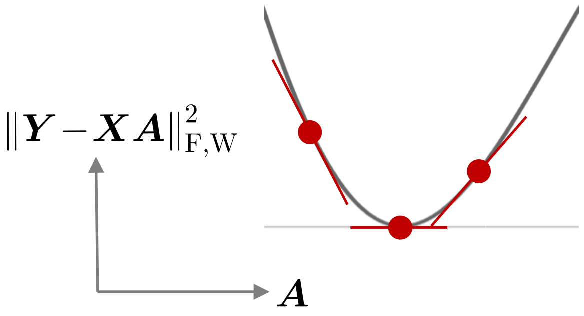
By describing the input data as $\boldsymbol{X} \in \mathbb{R}^{N \times D^{\mathcal{I}}}$ and the output data as $\boldsymbol{Y} \in \mathbb{R}^{N \times D^{\mathcal{O}}}$, with a weight matrix $\boldsymbol{W} \in \mathbb{R}^{N \times N}$, we want to minimize

$$\begin{aligned}
\hat{\boldsymbol{A}} &= \arg\min_{\boldsymbol{A}} \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A}\|^2_{\mathrm{F},\boldsymbol{W}} \\
&= \arg\min_{\boldsymbol{A}} \operatorname{tr}\Big((\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A})^\top \boldsymbol{W} (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A})\Big) \\
&= \arg\min_{\boldsymbol{A}} \operatorname{tr}(\boldsymbol{Y}^\top \boldsymbol{W}\boldsymbol{Y} - 2\boldsymbol{A}^\top \boldsymbol{X}^\top \boldsymbol{W}\boldsymbol{Y} + \boldsymbol{A}^\top \boldsymbol{X}^\top \boldsymbol{W}\boldsymbol{X}\boldsymbol{A}).
\end{aligned}$$



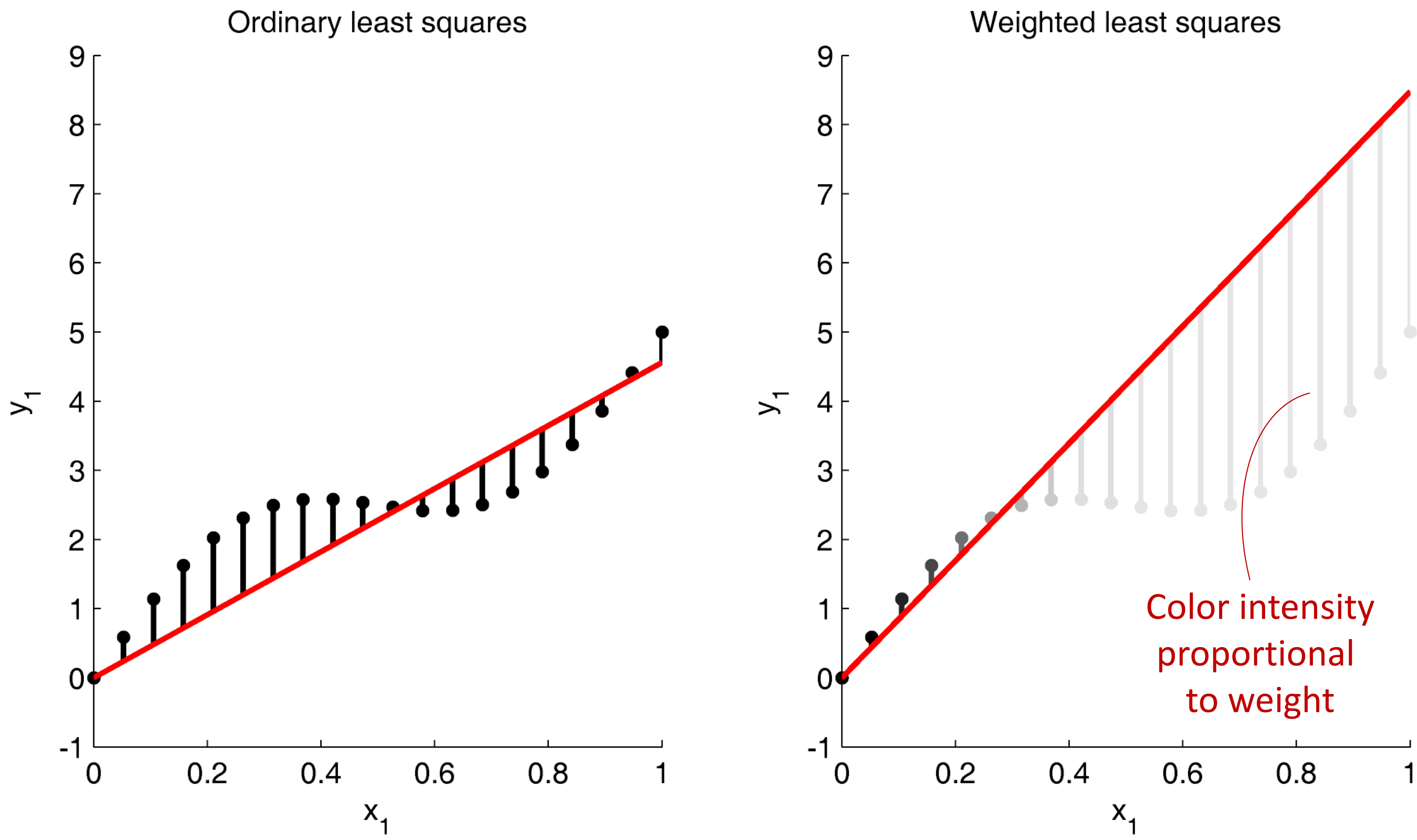
By differentiating with respect to $\boldsymbol{A}$ and equating to zero

$$-2\boldsymbol{X}^\top \boldsymbol{W}\boldsymbol{Y} + 2\boldsymbol{X}^\top \boldsymbol{W}\boldsymbol{X}\boldsymbol{A} = \boldsymbol{0} \iff \hat{\boldsymbol{A}} = \underbrace{\boxed{(\boldsymbol{X}^\top \boldsymbol{W}\boldsymbol{X})^{-1}\boldsymbol{X}^\top \boldsymbol{W}}}_{\boldsymbol{X}^\dagger_{\boldsymbol{W}}}\boldsymbol{Y}$$



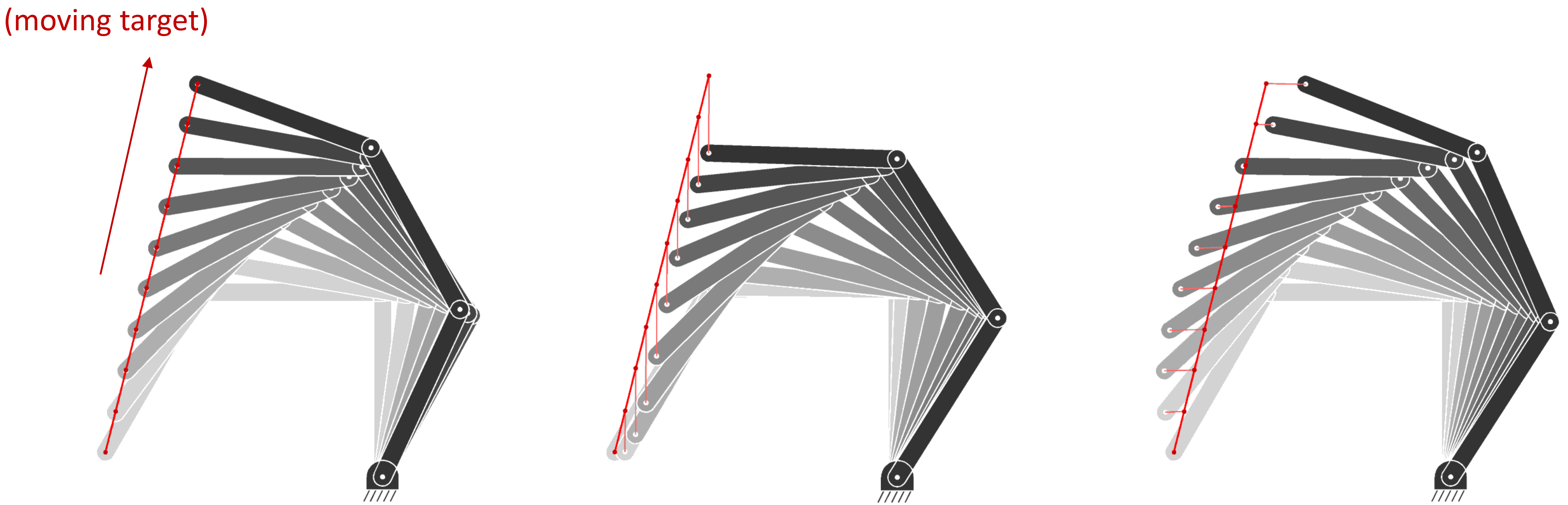
Weighted least squares

$$\hat{A} = (X^\top W X)^{-1} X^\top W Y$$



Example: Robot inverse kinematics (weights in task space)

$$\Delta\boldsymbol{x} = (\boldsymbol{J}^\top \boldsymbol{W}^{\mathcal{F}} \boldsymbol{J} + \lambda \boldsymbol{I}_x)^{-1} \boldsymbol{J}^\top \boldsymbol{W}^{\mathcal{F}} \Delta\boldsymbol{f}$$



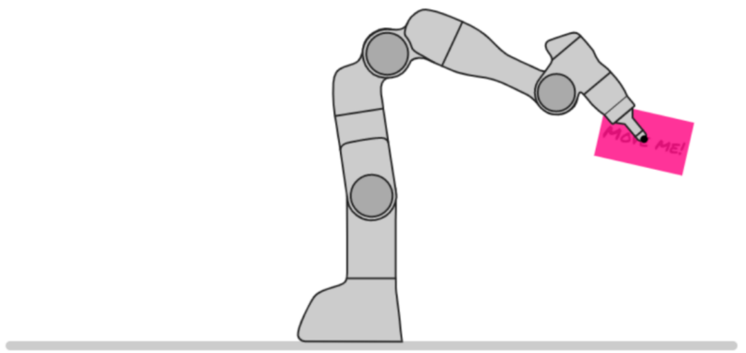
$$\boldsymbol{W}^{\mathcal{F}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \boldsymbol{W}^{\mathcal{F}} = \begin{bmatrix} 1 & 0 \\ 0 & .01 \end{bmatrix} \qquad \boldsymbol{W}^{\mathcal{F}} = \begin{bmatrix} .01 & 0 \\ 0 & 1 \end{bmatrix}$$

Example: Robot inverse kinematics (weights in task space)

IK | Damped IK | Weighted IK | Prioritized IK

```
x = [-np.pi/4, np.pi/2, np.pi/4] # Initial robot state
def controlCommand(x, param):
  f = fkin(x, param)
  J = Jkin(x, param)

  # Weights in task space
  Wf = np.diag([1, 1, 0])
  pinvWJ = np.linalg.inv(J.T @ Wf @ J + np.eye(param.nbVarX) * 1E-2) @ J.T @ Wf # We
  u = pinvWJ @ logmap(param.Mu, f) # Position & orientation tracking

#  # Weights in configuration space
#  Wx = np.diag([0.01, 1, 1])
#  pinvWJ =  Wx @ J[:2,:].T @ np.linalg.inv(J[:2,:] @ Wx @ J[:2,:].T + np.eye(2) * 1
#  u = pinvWJ @ (param.Mu[:2] - f[:2]) # Position tracking

  return u
```

Object orientation



https://robotics-codes-from-scratch.github.io/

Recursive least squares

Python notebook:
demo_LS_recursive.ipynb

Matlab code:
demo_LS_recursive01.m

Recursive least squares

$$\hat{\boldsymbol{A}} = \overbrace{\left(\boldsymbol{X}^\top \boldsymbol{X}\right)^{-1} \boldsymbol{X}^\top}^{\boldsymbol{X}^\dagger} \boldsymbol{Y}$$

$$\boldsymbol{X} \in \mathbb{R}^{N \times D^{\mathcal{I}}}$$

$$\Downarrow$$

$$\boldsymbol{X}_{\text{new}} \in \mathbb{R}^{N_{\text{new}} \times D^{\mathcal{I}}}$$

Sherman-Morrison-Woodbury relation:

$$\left(\boldsymbol{B} + \boldsymbol{U}\boldsymbol{V}\right)^{-1} = \boldsymbol{B}^{-1} - \overbrace{\boldsymbol{B}^{-1}\boldsymbol{U}\left(\boldsymbol{I} + \boldsymbol{V}\boldsymbol{B}^{-1}\boldsymbol{U}\right)^{-1}\boldsymbol{V}\boldsymbol{B}^{-1}}^{\boldsymbol{E}}$$

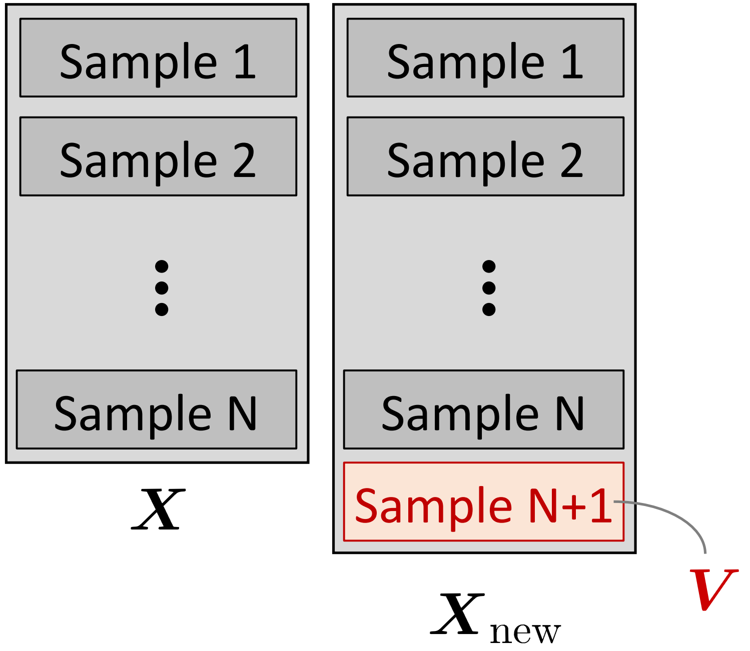
with $\boldsymbol{U} \in \mathbb{R}^{n \times m}$ and $\boldsymbol{V} \in \mathbb{R}^{m \times n}$.

When $m \ll n$, the correction term $\boldsymbol{E}$ can be computed more efficiently than inverting $\boldsymbol{B} + \boldsymbol{U}\boldsymbol{V}$.

By defining $\boldsymbol{B} = \boldsymbol{X}^\top \boldsymbol{X}$, the above relation can be exploited to update a least squares solution when new datapoints are available.

Recursive least squares

$$\hat{\boldsymbol{A}}_{\text{new}} = \overbrace{\left(\boldsymbol{X}_{\text{new}}^\top \boldsymbol{X}_{\text{new}}\right)^{-1} \boldsymbol{X}_{\text{new}}^\top}^{\boldsymbol{X}_{\text{new}}^\dagger} \boldsymbol{Y}_{\text{new}}$$



If $\boldsymbol{X}_{\text{new}} = [\boldsymbol{X}^\top, \boldsymbol{V}^\top]^\top$ and $\boldsymbol{Y}_{\text{new}} = [\boldsymbol{Y}^\top, \boldsymbol{C}^\top]^\top$, we then have

$$\begin{aligned} \boldsymbol{B}_{\text{new}} &= \boldsymbol{X}_{\text{new}}^\top \boldsymbol{X}_{\text{new}} \\ &= \boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{V}^\top \boldsymbol{V} \\ &= \boldsymbol{B} + \boldsymbol{V}^\top \boldsymbol{V} \end{aligned}$$

whose inverse can be computed with

$$\boldsymbol{B}_{\text{new}}^{-1} = \boldsymbol{B}^{-1} - \boldsymbol{B}^{-1}\boldsymbol{V}^\top \left(\boldsymbol{I} + \boldsymbol{V}\boldsymbol{B}^{-1}\boldsymbol{V}^\top\right)^{-1} \boldsymbol{V}\boldsymbol{B}^{-1}$$

which is exploited to efficiently compute the lease squares update as

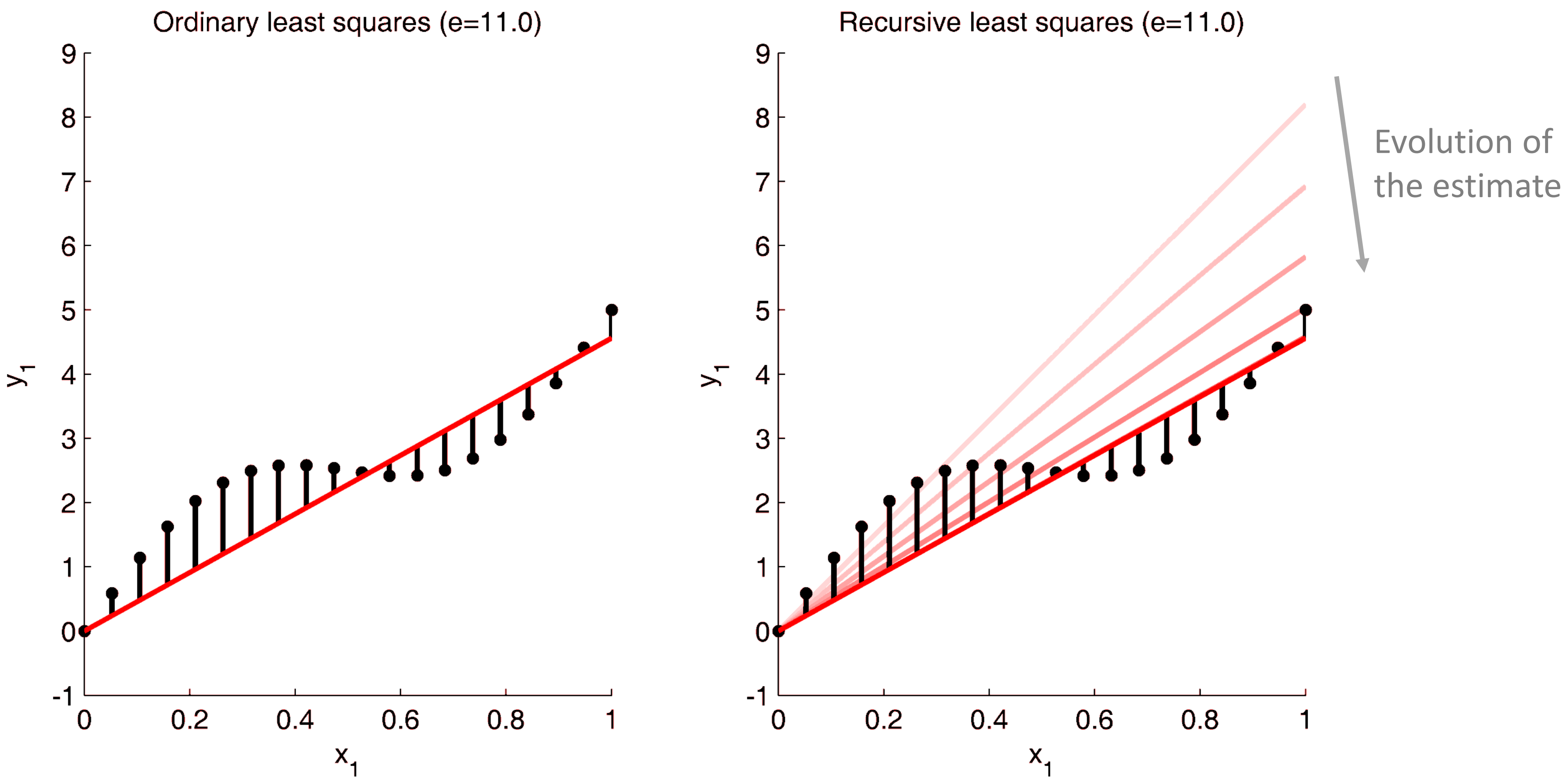
$$\hat{\boldsymbol{A}}_{\text{new}} = \hat{\boldsymbol{A}} + \boldsymbol{K}\left(\boldsymbol{C} - \boldsymbol{V}\hat{\boldsymbol{A}}\right)$$

with Kalman gain $\quad \boldsymbol{K} = \boldsymbol{B}^{-1}\boldsymbol{V}^\top \left(\boldsymbol{I} + \boldsymbol{V}\boldsymbol{B}^{-1}\boldsymbol{V}^\top\right)^{-1}$

$$(\boldsymbol{B} + \boldsymbol{U}\boldsymbol{V})^{-1} =$$

$$\boldsymbol{B}^{-1} - \overbrace{\boldsymbol{B}^{-1}\boldsymbol{U} \left(\boldsymbol{I} + \boldsymbol{V}\boldsymbol{B}^{-1}\boldsymbol{U}\right)^{-1} \boldsymbol{V}\boldsymbol{B}^{-1}}^{\boldsymbol{E}}$$

Recursive least squares



→ the least squares estimate is the same in the two cases

Linear regression:

A more elaborated example

(but still only least squares!)

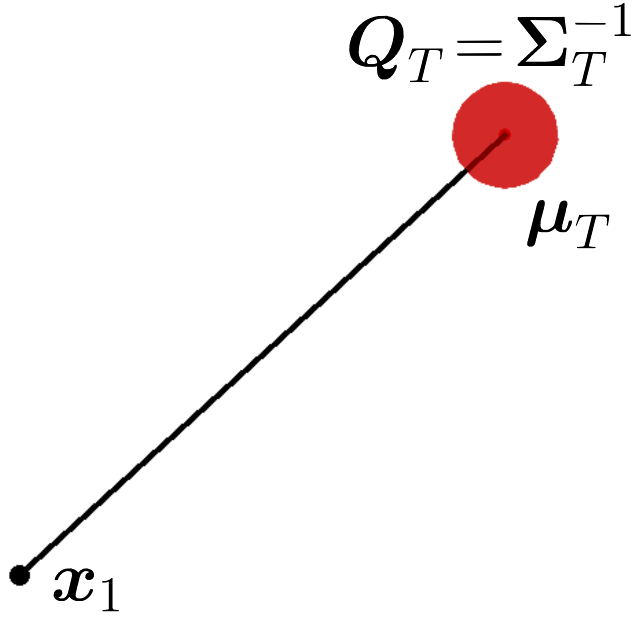
Linear quadratic regulator (LQR)

Track path! Use low control commands!

$$\min_{\boldsymbol{u}} \sum_{t=1}^{T} \left\|\boldsymbol{\mu}_t - \boldsymbol{x}_t\right\|^2_{\boldsymbol{Q}_t} + \left\|\boldsymbol{u}_t\right\|^2_{\boldsymbol{R}_t}$$

$$\text{s.t.} \quad \boldsymbol{x}_{t+1} = \boldsymbol{A}\boldsymbol{x}_t + \boldsymbol{B}\boldsymbol{u}_t$$

System dynamics



$\boldsymbol{x}_t$ state variable (position+velocity)

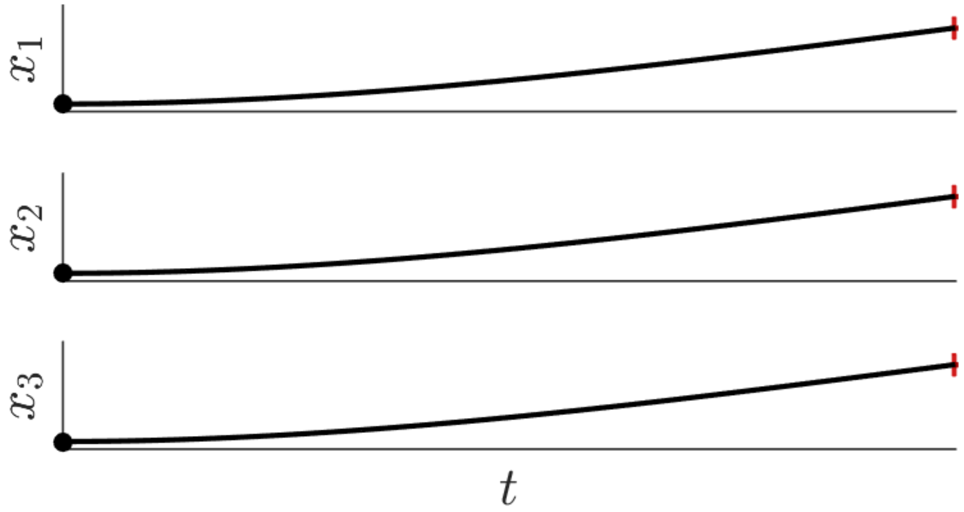
$\boldsymbol{\mu}_t$ desired state

$\boldsymbol{u}_t$ control command (acceleration)

$\boldsymbol{Q}_t$ precision matrix

$\boldsymbol{R}_t$ control weight matrix

$$\boldsymbol{u} = \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \vdots \\ \boldsymbol{u}_T \end{bmatrix}$$



Linear quadratic regulator (LQR)

Track path! Use low control commands!

$$\min_{\boldsymbol{u}} \sum_{t=1}^{T} \left\|\boldsymbol{\mu}_t - \boldsymbol{x}_t\right\|_{\boldsymbol{Q}_t}^2 + \left\|\boldsymbol{u}_t\right\|_{\boldsymbol{R}_t}^2$$

$$\text{s.t.} \quad \boldsymbol{x}_{t+1} = \boldsymbol{A}\boldsymbol{x}_t + \boldsymbol{B}\boldsymbol{u}_t$$

System dynamics

Pontryagin's max. principle, Riccati equation, Hamilton-Jacobi-Bellman

(the Physicist perspective)

Dynamic programming

(the Computer Scientist perspective)

Linear algebra

(the Algebraist perspective)

Linear quadratic regulator (LQR)

$$c = \sum_{t=1}^{T} \Big((\boldsymbol{\mu}_t - \boldsymbol{x}_t)^\top \boldsymbol{Q}_t (\boldsymbol{\mu}_t - \boldsymbol{x}_t) \; + \; \boldsymbol{u}_t^\top \boldsymbol{R}_t \boldsymbol{u}_t \Big)$$

$$= (\boldsymbol{\mu} - \boldsymbol{x})^\top \boldsymbol{Q} (\boldsymbol{\mu} - \boldsymbol{x}) \; + \; \boldsymbol{u}^\top \boldsymbol{R} \boldsymbol{u}$$

$$\boldsymbol{u} = \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \vdots \\ \boldsymbol{u}_T \end{bmatrix}$$

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_1 & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{R}_2 & \cdots & \boldsymbol{0} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{R}_T \end{bmatrix}$$

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{Q}_1 & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_2 & \cdots & \boldsymbol{0} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{Q}_T \end{bmatrix}$$

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \\ \vdots \\ \boldsymbol{\mu}_T \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_T \end{bmatrix}$$

Linear quadratic regulator (LQR)

$$\boldsymbol{x}_{t+1} = \boldsymbol{A}\,\boldsymbol{x}_t + \boldsymbol{B}\,\boldsymbol{u}_t$$

$$\begin{aligned}
\boldsymbol{x}_2 &= \boldsymbol{A}\boldsymbol{x}_1 + \boldsymbol{B}\boldsymbol{u}_1\\
\boldsymbol{x}_3 &= \boldsymbol{A}\boldsymbol{x}_2 + \boldsymbol{B}\boldsymbol{u}_2 = \boldsymbol{A}(\boldsymbol{A}\boldsymbol{x}_1 + \boldsymbol{B}\boldsymbol{u}_1) + \boldsymbol{B}\boldsymbol{u}_2\\
&\vdots\\
\boldsymbol{x}_T &= \boldsymbol{A}^{T-1}\boldsymbol{x}_1 + \boldsymbol{A}^{T-2}\boldsymbol{B}\boldsymbol{u}_1 + \boldsymbol{A}^{T-3}\boldsymbol{B}\boldsymbol{u}_2 + \cdots + \boldsymbol{B}_{T-1}\boldsymbol{u}_{T-1}
\end{aligned}$$

$$\begin{bmatrix}\boldsymbol{x}_1\\ \boldsymbol{x}_2\\ \boldsymbol{x}_3\\ \vdots\\ \boldsymbol{x}_T\end{bmatrix} = \underbrace{\begin{bmatrix}\boldsymbol{I}\\ \boldsymbol{A}\\ \boldsymbol{A}^2\\ \vdots\\ \boldsymbol{A}^{T-1}\end{bmatrix}}_{\boldsymbol{S}^{\boldsymbol{x}}} \boldsymbol{x}_1 + \underbrace{\begin{bmatrix}\boldsymbol{0} & \boldsymbol{0} & \cdots & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{B} & \boldsymbol{0} & \cdots & \boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{A}\boldsymbol{B} & \boldsymbol{B} & \cdots & \boldsymbol{0} & \boldsymbol{0}\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ \boldsymbol{A}^{T-2}\boldsymbol{B} & \boldsymbol{A}^{T-3}\boldsymbol{B} & \cdots & \boldsymbol{B} & \boldsymbol{0}\end{bmatrix}}_{\boldsymbol{S}^{\boldsymbol{u}}} \begin{bmatrix}\boldsymbol{u}_1\\ \boldsymbol{u}_2\\ \vdots\\ \boldsymbol{u}_T\end{bmatrix}$$

$$\boldsymbol{x} = \boldsymbol{S}^{\boldsymbol{x}}\boldsymbol{x}_1 + \boldsymbol{S}^{\boldsymbol{u}}\boldsymbol{u}$$

Linear quadratic regulator (LQR)

The constraint can then be inserted in the cost function:

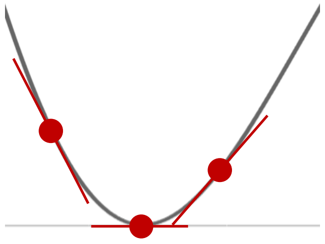
$$\boldsymbol{x} = \boldsymbol{S}^{\boldsymbol{x}}\boldsymbol{x}_1 + \boldsymbol{S}^{\boldsymbol{u}}\boldsymbol{u}$$

$$\begin{aligned} c &= (\boldsymbol{\mu} - \boldsymbol{x})^{\top}\boldsymbol{Q}(\boldsymbol{\mu} - \boldsymbol{x}) \; + \; \boldsymbol{u}^{\top}\boldsymbol{R}\boldsymbol{u} \\ &= (\boldsymbol{\mu} - \boldsymbol{S}^{\boldsymbol{x}}\boldsymbol{x}_1 - \boldsymbol{S}^{\boldsymbol{u}}\boldsymbol{u})^{\top}\boldsymbol{Q}\,(\boldsymbol{\mu} - \boldsymbol{S}^{\boldsymbol{x}}\boldsymbol{x}_1 - \boldsymbol{S}^{\boldsymbol{u}}\boldsymbol{u}) + \boldsymbol{u}^{\top}\boldsymbol{R}\boldsymbol{u} \end{aligned}$$

Solving for *u* is similar to a weighted ridge regression problem, and results in the analytic solution:

$$\hat{\boldsymbol{u}} = \left(\boldsymbol{S}^{\boldsymbol{u}\top}\boldsymbol{Q}\boldsymbol{S}^{\boldsymbol{u}} + \boldsymbol{R}\right)^{-1}\boldsymbol{S}^{\boldsymbol{u}\top}\boldsymbol{Q}\,\left(\boldsymbol{\mu} - \boldsymbol{S}^{\boldsymbol{x}}\boldsymbol{x}_1\right)$$

$$\hat{\boldsymbol{u}} = \begin{bmatrix} \hat{\boldsymbol{u}}_1 \\ \hat{\boldsymbol{u}}_2 \\ \vdots \\ \hat{\boldsymbol{u}}_T \end{bmatrix}$$

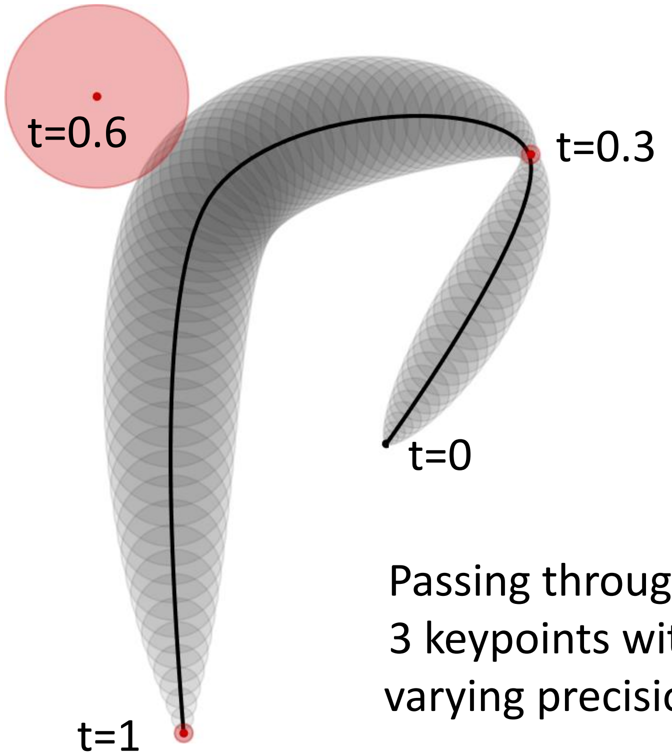


Linear quadratic regulator (LQR)

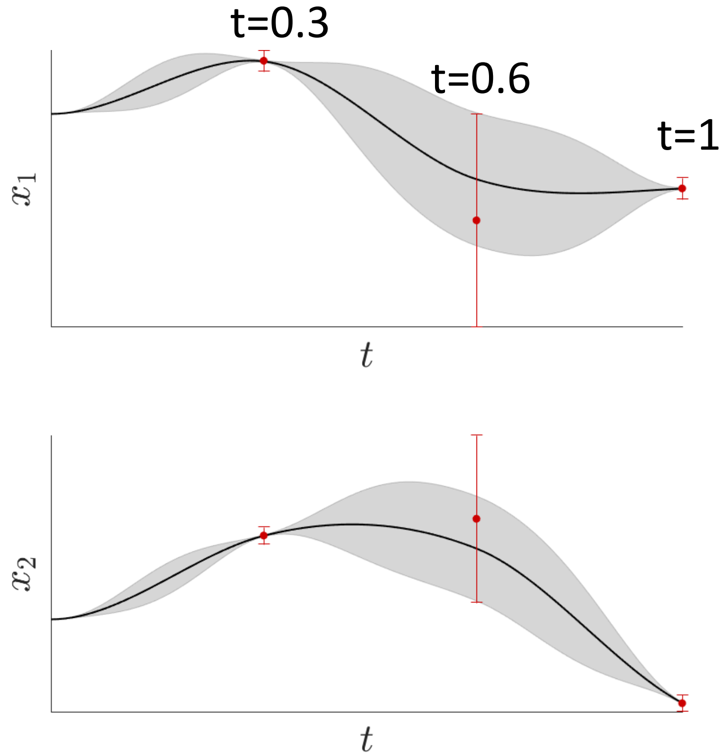
$$\hat{\boldsymbol{u}} = \left(\boldsymbol{S}^{\boldsymbol{u}\top}\boldsymbol{Q}\boldsymbol{S}^{\boldsymbol{u}} + \boldsymbol{R}\right)^{-1}\boldsymbol{S}^{\boldsymbol{u}\top}\boldsymbol{Q}\,\left(\boldsymbol{\mu} - \boldsymbol{S}^{\boldsymbol{x}}\boldsymbol{x}_1\right)$$

$$\hat{\boldsymbol{x}} = \boldsymbol{S}^{\boldsymbol{x}}\boldsymbol{x}_1 + \boldsymbol{S}^{\boldsymbol{u}}\hat{\boldsymbol{u}}$$

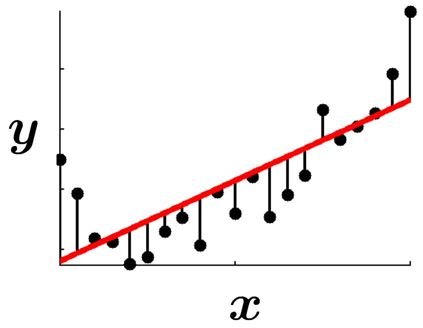
The control trajectories can then be converted to state trajectories

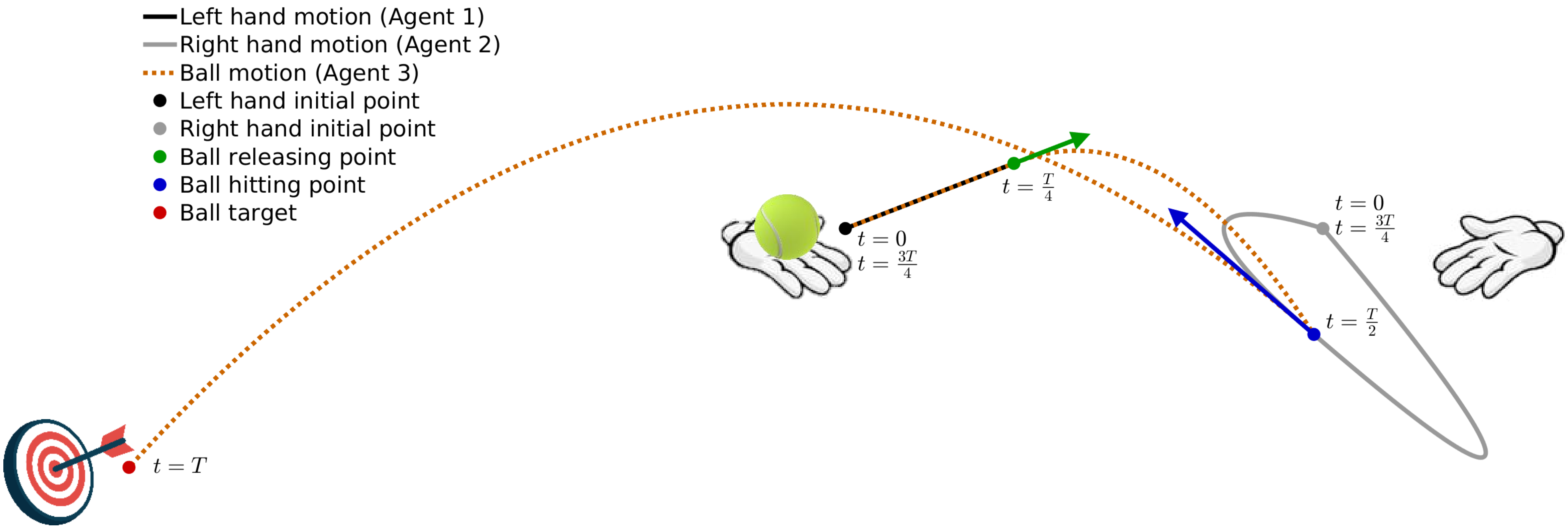


Passing through 3 keypoints with varying precision

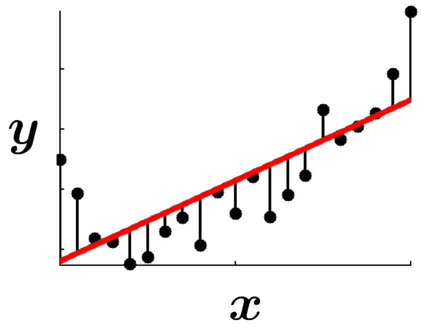


Linear quadratic regulator (LQR) - Example





Linear quadratic regulator (LQR) - Example



For $t \leq \frac{T}{4}$ (left hand holding the ball), we have

$$\underbrace{\begin{bmatrix} \dot{\mathbf{x}}_{1,t} \\ \ddot{\mathbf{x}}_{1,t} \\ \dot{\boldsymbol{f}}_{1,t} \\ \dot{\mathbf{x}}_{2,t} \\ \ddot{\mathbf{x}}_{2,t} \\ \dot{\boldsymbol{f}}_{2,t} \\ \dot{\mathbf{x}}_{3,t} \\ \ddot{\mathbf{x}}_{3,t} \\ \dot{\boldsymbol{f}}_{3,t} \end{bmatrix}}_{\dot{\boldsymbol{x}}_t} = \underbrace{\begin{bmatrix} 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boldsymbol{I} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\boldsymbol{A}_t^c} \underbrace{\begin{bmatrix} \mathbf{x}_{1,t} \\ \dot{\mathbf{x}}_{1,t} \\ \boldsymbol{f}_{1,t} \\ \mathbf{x}_{2,t} \\ \dot{\mathbf{x}}_{2,t} \\ \boldsymbol{f}_{2,t} \\ \mathbf{x}_{3,t} \\ \dot{\mathbf{x}}_{3,t} \\ \boldsymbol{f}_{3,t} \end{bmatrix}}_{\boldsymbol{x}_t} + \underbrace{\begin{bmatrix} 0 & 0 \\ \boldsymbol{I} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \boldsymbol{I} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}}_{\boldsymbol{B}_t^c} \underbrace{\begin{bmatrix} \boldsymbol{u}_{1,t} \\ \boldsymbol{u}_{2,t} \end{bmatrix}}_{\boldsymbol{u}_t}$$

$$\boldsymbol{f}_{i,t} = m_i \boldsymbol{g} \quad \text{with} \quad \boldsymbol{g} = \begin{bmatrix} 0 \\ -9.81 \end{bmatrix}$$

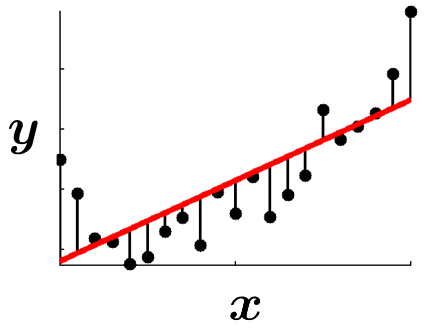
At $t = \frac{T}{2}$ (right hand hitting the ball), we have

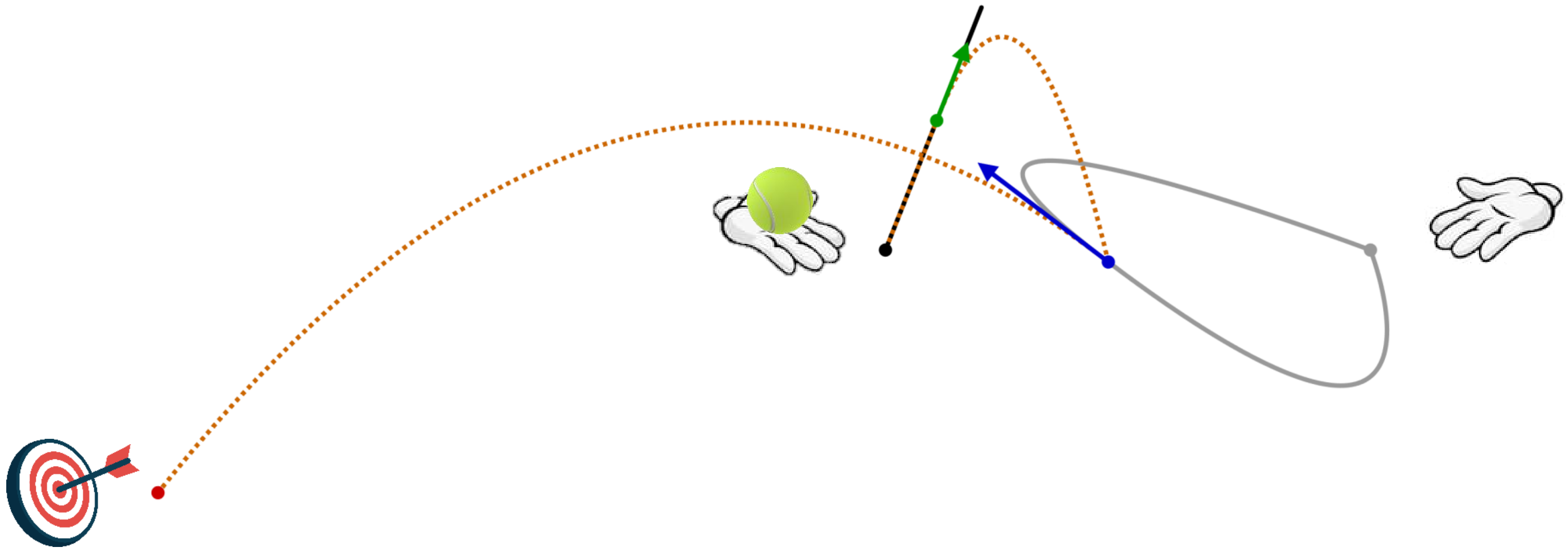
$$\underbrace{\begin{bmatrix} \dot{\mathbf{x}}_{1,t} \\ \ddot{\mathbf{x}}_{1,t} \\ \dot{\boldsymbol{f}}_{1,t} \\ \dot{\mathbf{x}}_{2,t} \\ \ddot{\mathbf{x}}_{2,t} \\ \dot{\boldsymbol{f}}_{2,t} \\ \dot{\mathbf{x}}_{3,t} \\ \ddot{\mathbf{x}}_{3,t} \\ \dot{\boldsymbol{f}}_{3,t} \end{bmatrix}}_{\dot{\boldsymbol{x}}_t} = \underbrace{\begin{bmatrix} 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boldsymbol{I} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\boldsymbol{A}_t^c} \underbrace{\begin{bmatrix} \mathbf{x}_{1,t} \\ \dot{\mathbf{x}}_{1,t} \\ \boldsymbol{f}_{1,t} \\ \mathbf{x}_{2,t} \\ \dot{\mathbf{x}}_{2,t} \\ \boldsymbol{f}_{2,t} \\ \mathbf{x}_{3,t} \\ \dot{\mathbf{x}}_{3,t} \\ \boldsymbol{f}_{3,t} \end{bmatrix}}_{\boldsymbol{x}_t} + \underbrace{\begin{bmatrix} 0 & 0 \\ \boldsymbol{I} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \boldsymbol{I} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}}_{\boldsymbol{B}_t^c} \underbrace{\begin{bmatrix} \boldsymbol{u}_{1,t} \\ \boldsymbol{u}_{2,t} \end{bmatrix}}_{\boldsymbol{u}_t}$$

For $\frac{T}{4} < t < \frac{T}{2}$ and $t > \frac{T}{2}$ (free motion of the ball), we have

$$\underbrace{\begin{bmatrix} \dot{\mathbf{x}}_{1,t} \\ \ddot{\mathbf{x}}_{1,t} \\ \dot{\boldsymbol{f}}_{1,t} \\ \dot{\mathbf{x}}_{2,t} \\ \ddot{\mathbf{x}}_{2,t} \\ \dot{\boldsymbol{f}}_{2,t} \\ \dot{\mathbf{x}}_{3,t} \\ \ddot{\mathbf{x}}_{3,t} \\ \dot{\boldsymbol{f}}_{3,t} \end{bmatrix}}_{\dot{\boldsymbol{x}}_t} = \underbrace{\begin{bmatrix} 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \boldsymbol{I} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boldsymbol{I} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boldsymbol{I} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\boldsymbol{A}_t^c} \underbrace{\begin{bmatrix} \mathbf{x}_{1,t} \\ \dot{\mathbf{x}}_{1,t} \\ \boldsymbol{f}_{1,t} \\ \mathbf{x}_{2,t} \\ \dot{\mathbf{x}}_{2,t} \\ \boldsymbol{f}_{2,t} \\ \mathbf{x}_{3,t} \\ \dot{\mathbf{x}}_{3,t} \\ \boldsymbol{f}_{3,t} \end{bmatrix}}_{\boldsymbol{x}_t} + \underbrace{\begin{bmatrix} 0 & 0 \\ \boldsymbol{I} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \boldsymbol{I} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}}_{\boldsymbol{B}_t^c} \underbrace{\begin{bmatrix} \boldsymbol{u}_{1,t} \\ \boldsymbol{u}_{2,t} \end{bmatrix}}_{\boldsymbol{u}_t}$$

Linear quadratic regulator (LQR) - Example





Recap: Costs functions and associated solutions

Univariate output y:

$$\hat{\boldsymbol{a}} = \arg\min_{\boldsymbol{a}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{a}\|^2 = \overbrace{(\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top}^{\boldsymbol{X}^\dagger}\ \boldsymbol{y}$$

$$\hat{\boldsymbol{a}} = \arg\min_{\boldsymbol{a}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{a}\|^2_{\mathrm{F},\boldsymbol{W}} = (\boldsymbol{X}^\top \boldsymbol{W}\boldsymbol{X})^{-1}\boldsymbol{X}^\top \boldsymbol{W}\,\boldsymbol{y}$$

$$\hat{\boldsymbol{a}} = \arg\min_{\boldsymbol{a}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{a}\|^2_{\mathrm{F}} + \|\boldsymbol{\Gamma}\boldsymbol{a}\|^2_{\mathrm{F}} = (\boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{\Gamma}^\top\boldsymbol{\Gamma})^{-1}\boldsymbol{X}^\top \boldsymbol{y}$$

$$\hat{\boldsymbol{a}} = \arg\min_{\boldsymbol{a}} f_{\boldsymbol{a}}(\boldsymbol{X}, \boldsymbol{y})$$

Multivariate output y:

$$\hat{\boldsymbol{A}} = \arg\min_{\boldsymbol{A}} \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A}\|^2 = \overbrace{(\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top}^{\boldsymbol{X}^\dagger}\ \boldsymbol{Y}$$

$$\hat{\boldsymbol{A}} = \arg\min_{\boldsymbol{A}} \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A}\|^2_{\mathrm{F},\boldsymbol{W}} = (\boldsymbol{X}^\top \boldsymbol{W}\boldsymbol{X})^{-1}\boldsymbol{X}^\top \boldsymbol{W}\,\boldsymbol{Y}$$

$$\hat{\boldsymbol{A}} = \arg\min_{\boldsymbol{A}} \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A}\|^2_{\mathrm{F}} + \|\boldsymbol{\Gamma}\boldsymbol{A}\|^2_{\mathrm{F}} = (\boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{\Gamma}^\top\boldsymbol{\Gamma})^{-1}\boldsymbol{X}^\top \boldsymbol{Y}$$

Iteratively reweighted least squares (IRLS)

Python notebook:
demo_LS_weighted.ipynb

Matlab code:
demo_LS_IRLS01.m

Iteratively reweighted least squares (IRLS)

$$\|\boldsymbol{e}\|_p = \left(\sum_{n=1}^{N} |e_n|^p\right)^{1/p}$$

- IRLS is useful to minimize ℓ_p norms with $\arg\min \|\boldsymbol{e}\|_p = \arg\min \sum_{n=1}^{N} |e_n|^p$

- The strategy of IRLS is that $|e_n|^p$ can be rewritten as $|e_n|^{p-2}\, e_n^2$

- $|e_n|^{p-2}$ can be interpreted as a weight, which is used to minimize e_n^2 with **weighted least squares**.

  → we solve a least squares problem at each iteration of the algorithm

- p=1 corresponds to **least absolute deviation regression**.

Iteratively reweighted least squares (IRLS)

$$|e_n|^p = |e_n|^{p-2} e_n^2$$

transformed as weight **W**

For an ℓ_p norm cost function defined by

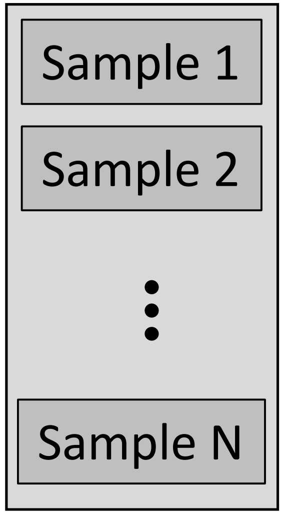
$$\hat{\boldsymbol{A}} = \arg\min_{\boldsymbol{A}} \left\|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A}\right\|_p$$

$\hat{\boldsymbol{A}}$ is estimated by starting from $\boldsymbol{W} = \boldsymbol{I}$ and iteratively computing

$$\hat{\boldsymbol{A}} \leftarrow (\boldsymbol{X}^\top \boldsymbol{W} \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{W} \boldsymbol{Y}$$

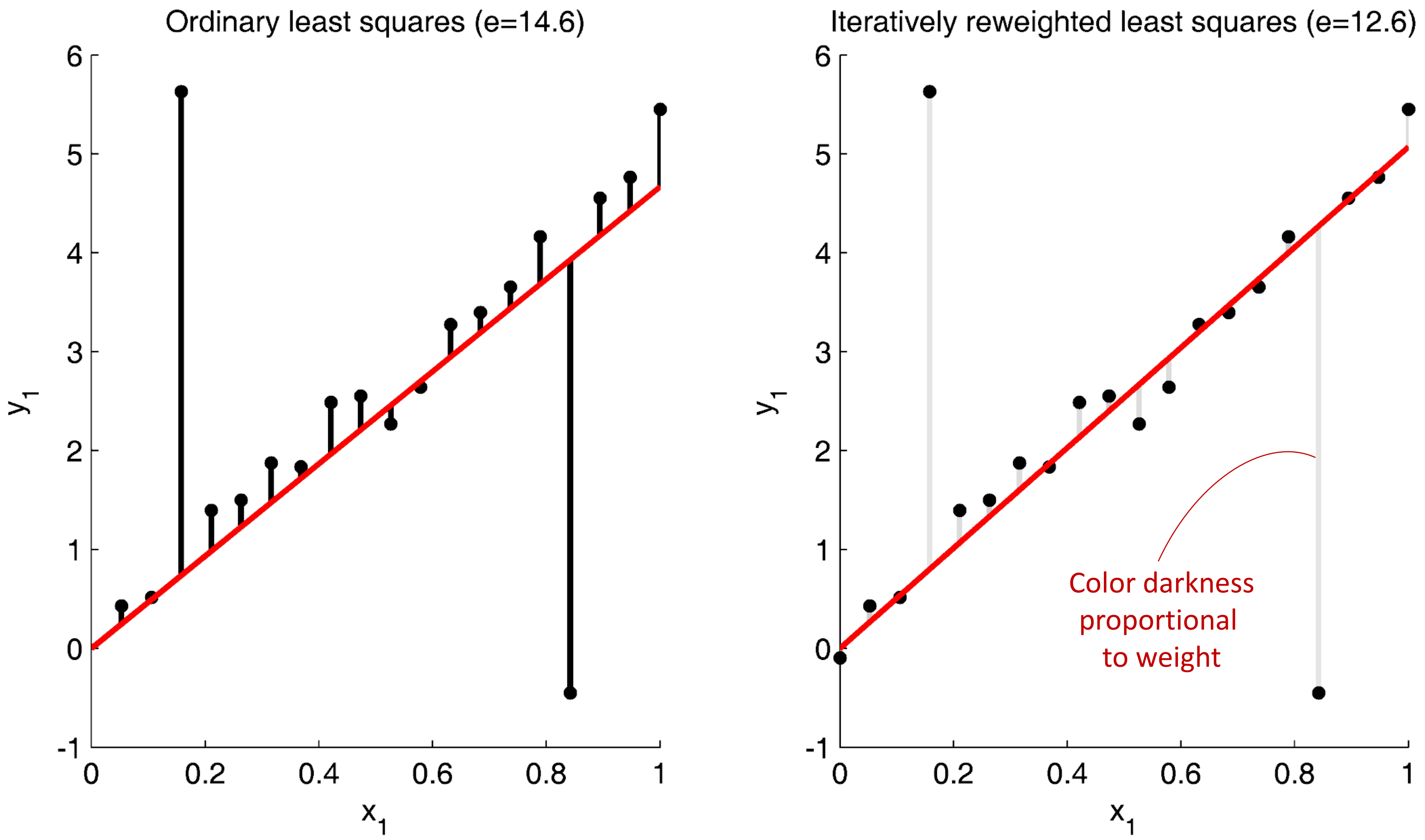
$$\hat{\boldsymbol{A}} = \arg\min_{\boldsymbol{A}} \|\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{A}\|_{\mathrm{F},\boldsymbol{W}}^2$$

$$\boldsymbol{W}_{n,n} \leftarrow |\boldsymbol{Y}_n - \boldsymbol{X}_n \boldsymbol{A}|^{p-2} \quad \forall n \in \{1, \ldots, N\}$$



$\boldsymbol{X}$

Iteratively reweighted least squares (IRLS)



→ regression that can sometimes be more robust to outliers

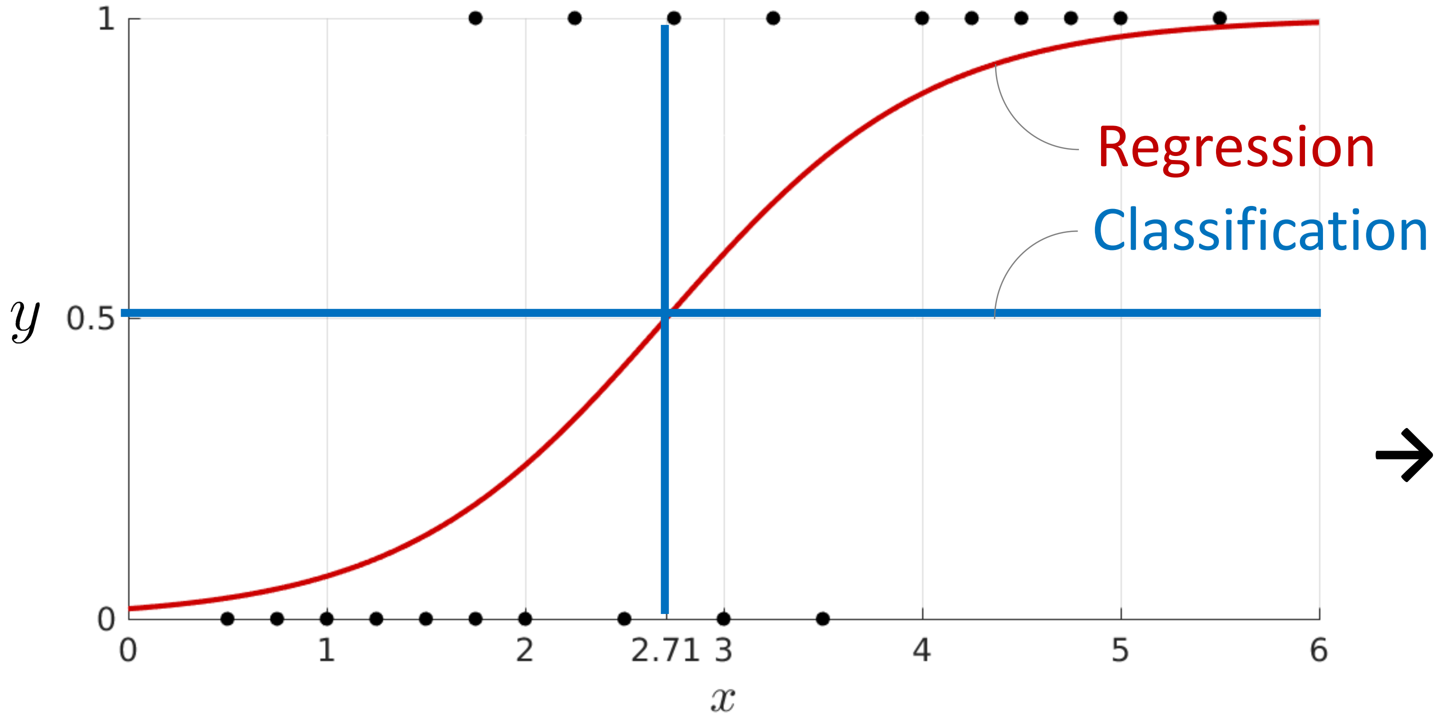
Logistic regression

Python notebook:
demo_LS_IRLS_logRegr.ipynb

Matlab code:
demo_LS_IRLS_logRegr01.m

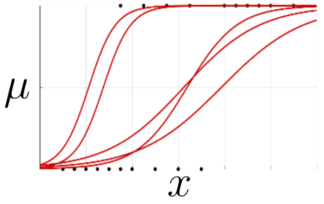
Logistic regression

Example: Pass/fail in function of the time spent to study at an exam:



→ **Regression exploited for classification problem**

Logistic function:



$$\mu_{\boldsymbol{a}}(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{a}^\top \boldsymbol{x}}}$$

$$\mu_{\boldsymbol{a}}(x) = \frac{1}{1+e^{-(a_1+a_2 x)}}$$

Logistic regression

Bernoulli distribution *(for binary variables)*:

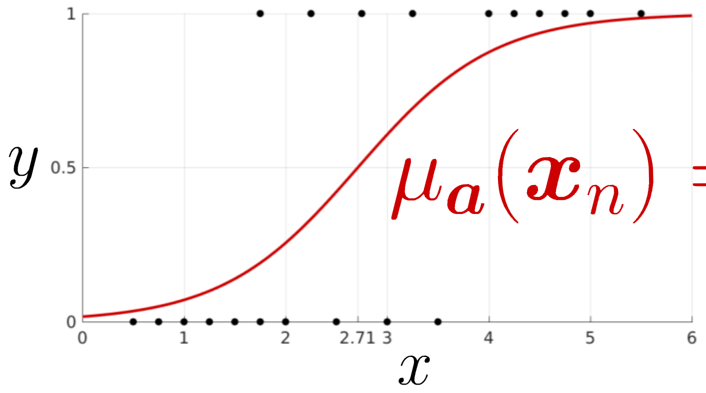
$$\mathcal{L}_n = \begin{cases} \underbrace{\boxed{p}}_{} & \text{if } y_n = 1, \\ \boxed{1-p} & \text{if } y_n = 0. \end{cases}$$

$\mathcal{P}(y_n = 1)$ — p; $\mathcal{P}(y_n = 0)$ — $1-p$

$$= p^{y_n} \, (1-p)^{(1-y_n)}$$

Logistic function:

$y_n \in \{0, 1\}$



$$\mu_{\boldsymbol{a}}(\boldsymbol{x}_n) = \frac{1}{1+e^{-\boldsymbol{a}^\top \boldsymbol{x}_n}}$$

Likelihood of *n*th datapoint:

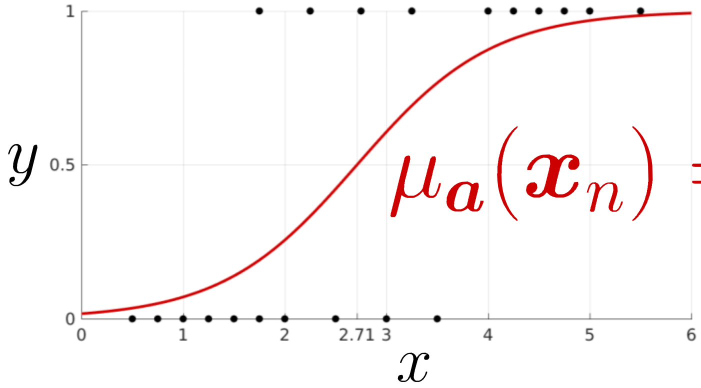
$$\mathcal{L}_n = \mu_{\boldsymbol{a}}(\boldsymbol{x}_n)^{y_n} \left(1 - \mu_{\boldsymbol{a}}(\boldsymbol{x}_n)\right)^{(1-y_n)}$$

Likelihood of N datapoints *(independence assumption)*:

$$\mathcal{L} = \prod_n \mu_{\boldsymbol{a}}(\boldsymbol{x}_n)^{y_n} \left(1 - \mu_{\boldsymbol{a}}(\boldsymbol{x}_n)\right)^{(1-y_n)}$$

Logistic regression

$y_n \in \{0, 1\}$



$$\mu_{\boldsymbol{a}}(\boldsymbol{x}_n) = \frac{1}{1+e^{-\boldsymbol{a}^\top \boldsymbol{x}_n}}$$

Likelihood of N datapoints:

$$\mathcal{L} = \prod_n \mu_{\boldsymbol{a}}(\boldsymbol{x}_n)^{y_n} \left(1 - \mu_{\boldsymbol{a}}(\boldsymbol{x}_n)\right)^{(1-y_n)}$$

Cost function as negative log-likelihood:

$\log(a^b) = b\log(a)$

$\log(ab) = \log(a) + \log(b)$

$$c = -\sum_n y_n \log\left(\mu_{\boldsymbol{a}}(\boldsymbol{x}_n)\right) + (1-y_n)\log\left(1 - \mu_{\boldsymbol{a}}(\boldsymbol{x}_n)\right)$$

$\frac{\partial}{\partial x}\log(x) = \frac{1}{x}$

$$\frac{\partial c}{\partial \boldsymbol{a}} = -\sum_n y_n\, \mu_{\boldsymbol{a}}^{-1} \mu_{\boldsymbol{a}}\left(1-\mu_{\boldsymbol{a}}\right)\boldsymbol{x}_n - (1-y_n)(1-\mu_{\boldsymbol{a}})^{-1}\mu_{\boldsymbol{a}}\left(1-\mu_{\boldsymbol{a}}\right)\boldsymbol{x}_n$$

$$= -\sum_n y_n\left(1-\mu_{\boldsymbol{a}}\right)\boldsymbol{x}_n - (1-y_n)\,\mu_{\boldsymbol{a}}\,\boldsymbol{x}_n$$

$$= \sum_n (\mu_{\boldsymbol{a}} - y_n)\,\boldsymbol{x}_n$$

$$\boxed{\begin{aligned}\mu(x) &= \frac{1}{1+e^{-x}} \\ \frac{\partial \mu}{\partial x} &= \mu(1-\mu)\end{aligned}}$$

Logistic regression

$$\frac{\partial c}{\partial \boldsymbol{a}} = \sum_n (\mu_{\boldsymbol{a}} - y_n)\boldsymbol{x}_n$$

$$\mu_{\boldsymbol{a}}(\boldsymbol{x}_n) = \frac{1}{1+e^{-\boldsymbol{a}^\top \boldsymbol{x}_n}}$$

It can for example be solved with Newton's method, by iterating

$$\boldsymbol{a} \leftarrow \boldsymbol{a} - \boldsymbol{H}^{-1}\boldsymbol{g},$$

with gradient $\boldsymbol{g} = \sum_n (\mu_{\boldsymbol{a}}(\boldsymbol{x}_n) - y_n)\boldsymbol{x}_n = \boldsymbol{X}^\top(\boldsymbol{\mu}_{\boldsymbol{a}} - \boldsymbol{y})$ and Hessian $\boldsymbol{H} = \boldsymbol{X}^\top \boldsymbol{W} \boldsymbol{X}$,
with diagonal matrix $\boldsymbol{W} = \mathrm{diag}\big(\boldsymbol{\mu}_{\boldsymbol{a}} * (\boldsymbol{1} - \boldsymbol{\mu}_{\boldsymbol{a}})\big)$.

Hadamard (elementwise) product

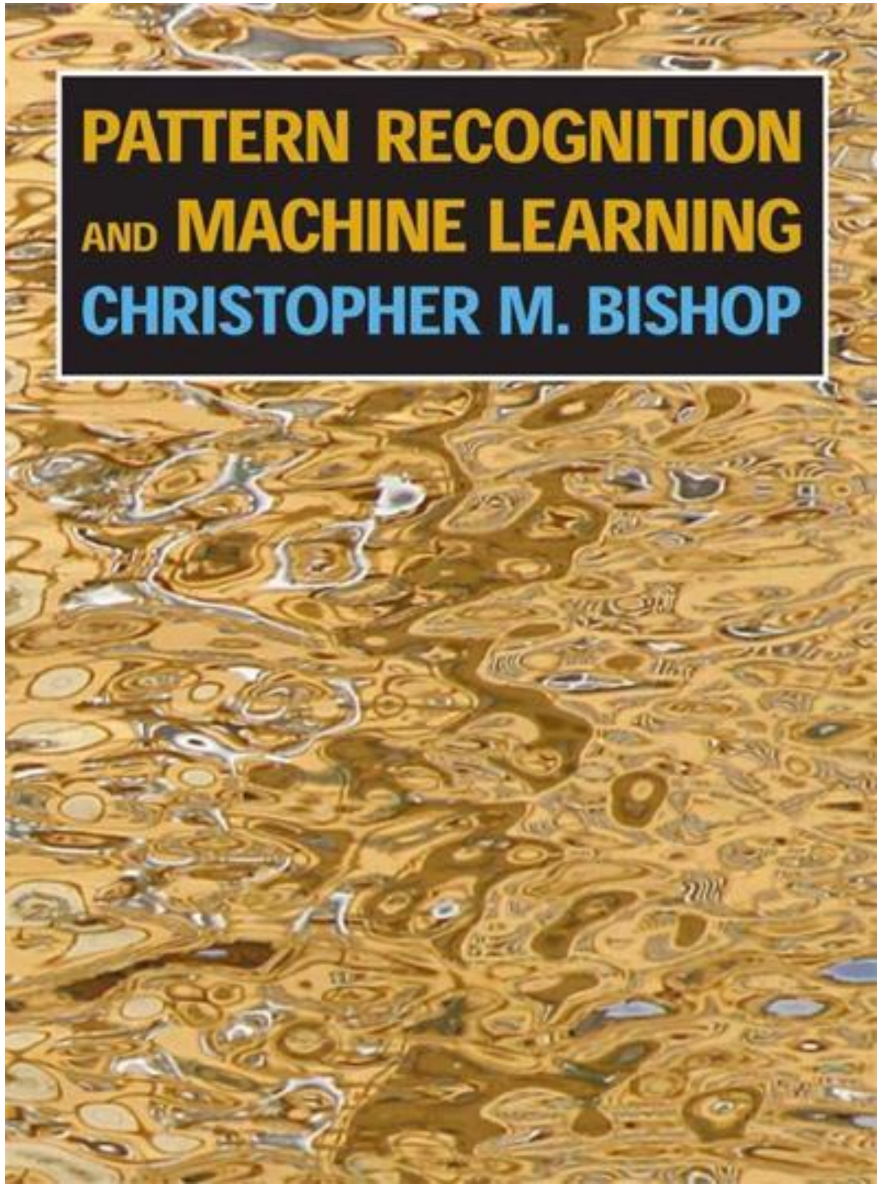
$$\mu(x) = \frac{1}{1+e^{-x}}$$

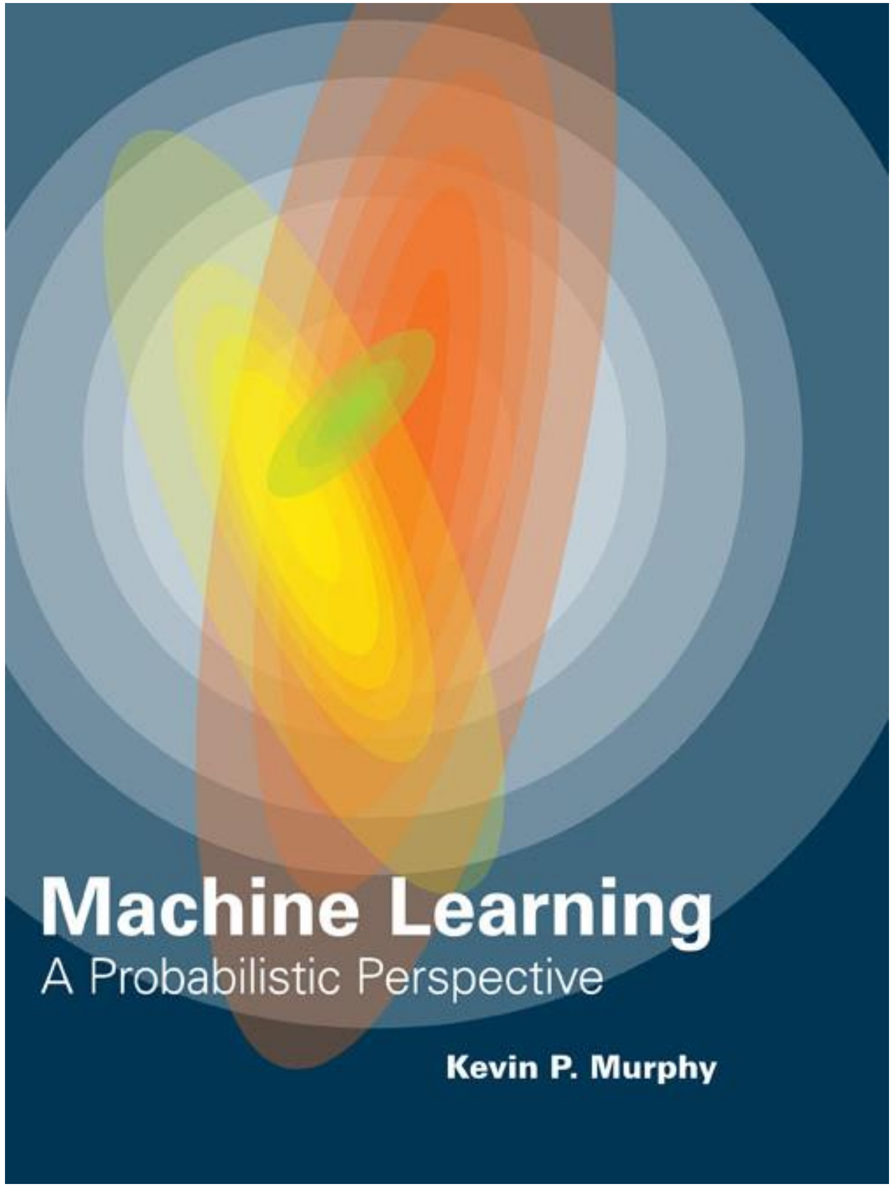
$$\frac{\partial \mu}{\partial x} = \mu(1-\mu)$$

We then obtain

$$\begin{aligned} \boldsymbol{a} &\leftarrow \boldsymbol{a} - \boldsymbol{H}^{-1}\boldsymbol{g} \\ &\leftarrow \boldsymbol{a} - (\boldsymbol{X}^\top \boldsymbol{W} \boldsymbol{X})^{-1}\boldsymbol{X}^\top(\boldsymbol{\mu}_{\boldsymbol{a}} - \boldsymbol{y}) \end{aligned}$$

General references





The Matrix Cookbook

Kaare Brandt Petersen
Michael Syskind Pedersen