Underactuated Robots
Lecture 1: Optimization

Nicola Scianca

December 2024

info

- these slides were made by **Nicola Scianca**, researcher at Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza University of Rome, Italy
- email: scianca@diag.uniroma1.it
- the slides and the examples are available at this **repository**: `https://github.com/DIAG-Robotics-Lab/underactuated`

why optimization

- with complex systems, such as underactuated robots with many degrees of freedom, we often can't explicitly write a control law $u = f(x)$ as a closed-form expression

- what we can do instead is to write an **optimization problem** that encodes our control objectives, and its solution will implicitly define our control law

- these lectures will be mostly focused on optimization-based techniques, which we collectively call **trajectory optimization**

outline

- introduction to optimization
- trajectory optimization overview
- dynamic programming
- differential dynamic programming
- model predictive control
- models of legged robots
- whole-body control
- model predictive control of legged robots

introduction to optimization

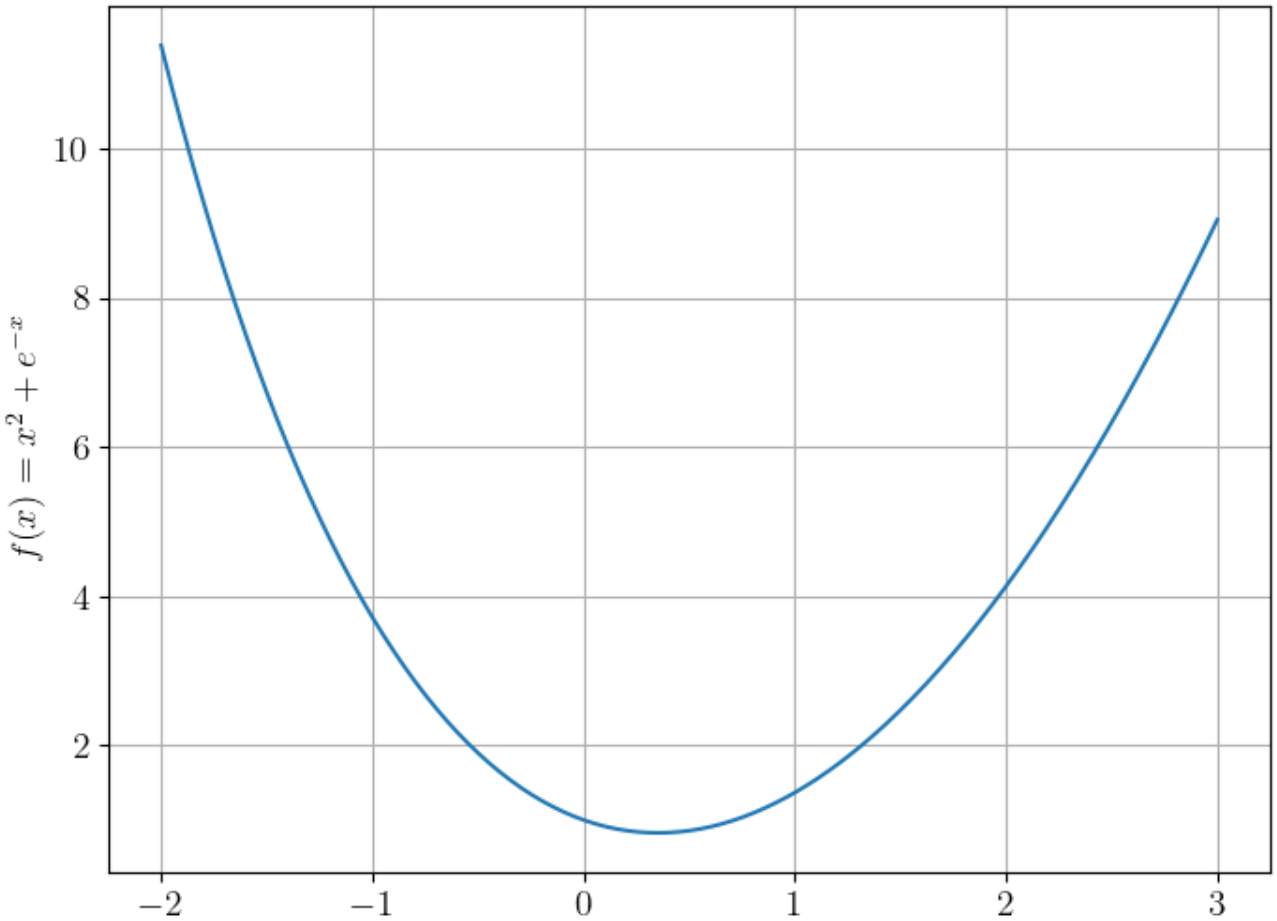
- optimization is the process of finding the **minimum** (or the maximum) of some **function**, possibly in the presence of **constraints**

- key concepts:
  - constrained / unconstrained optimization
  - convexity
  - gradient descent
  - newton's method

unconstrained optimization

- let's say we want to find the minimum of the function

$$f(x) = x^2 + e^{-x}$$



gradient descent

- we learned in calculus class that we can compute the derivative and set it equal to zero

$$\frac{df}{dx} = 2x - e^{-x} = 0 \quad \Longrightarrow \quad 2x = e^{-x}$$

  but solving this equation is not straightforward: we must zero it **numerically**

- the basic idea of **gradient descent**: pick some point and check the derivative
  - if **positive**, the function decreases to the left
  - if **negative**, the function decreases to the right

- if we just keep going in the direction in which the function **decreases**, we should, at some point, find a (local) minimum

gradient descent

- in gradient descent we are computing a **linear** approximation of the function around our guess $\bar{x}$

$$f(\bar{x} + \Delta x) = f(\bar{x}) + \nabla f|_{\bar{x}} \cdot \Delta x$$

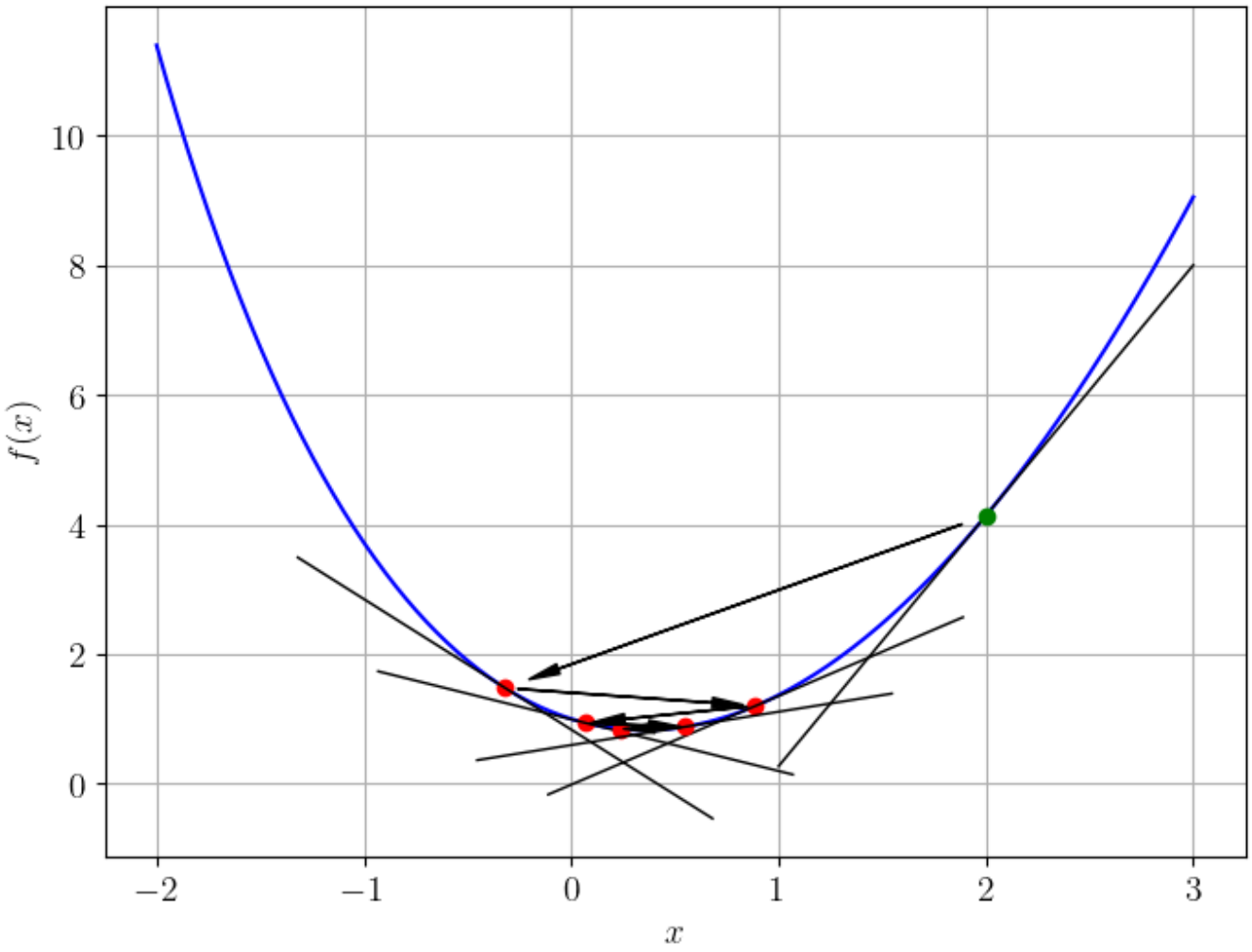
- then we take a step in the direction in which this linear approximation decreases, which means, the negative of its gradient

$$\Delta x = -\alpha \nabla f|_{\bar{x}}$$

- α is the **step size**, if too small we converge very slowly, if too big we risk overshooting the minimum

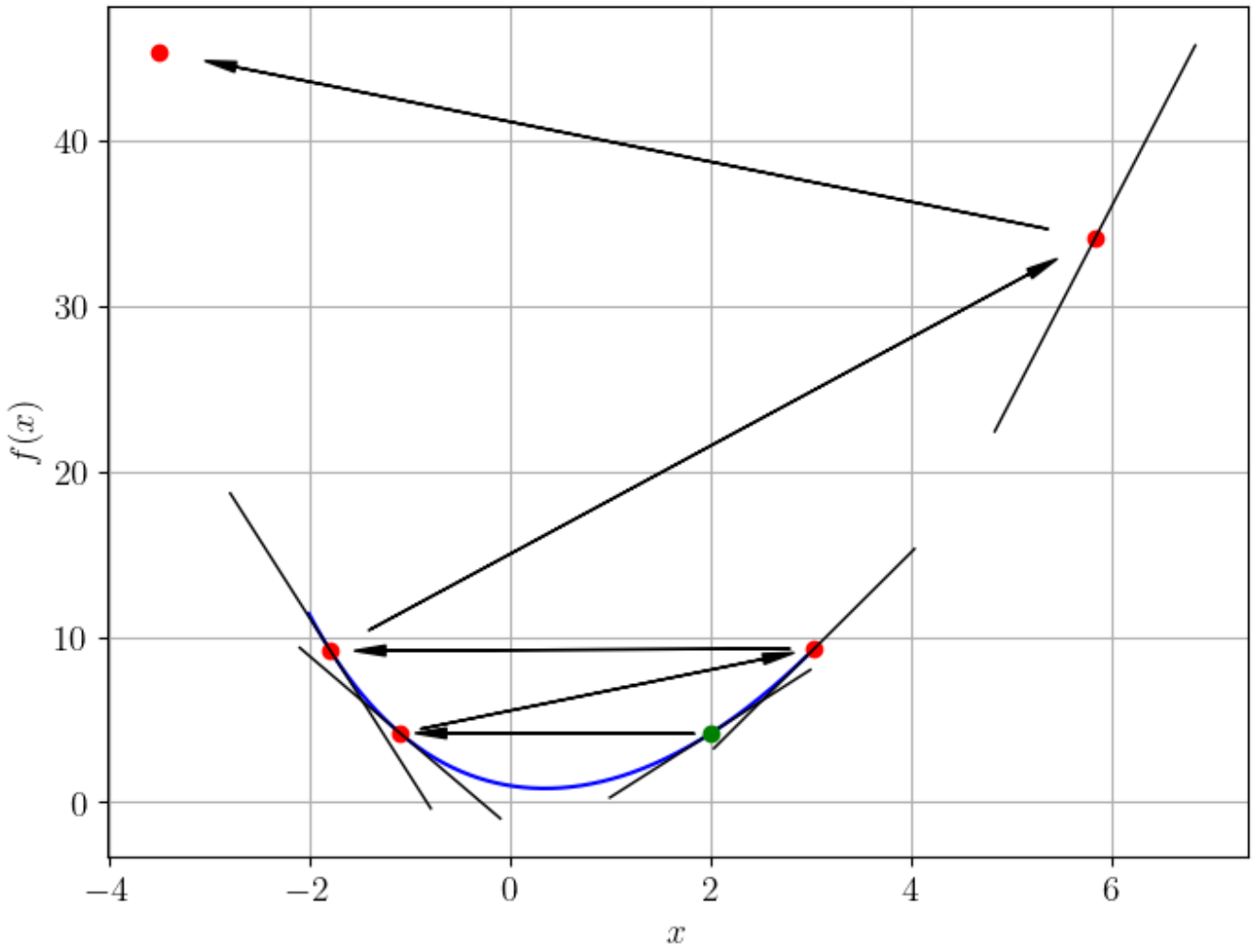
gradient descent

- for our example function, if we start from the point $x = 2$ and iterate with step size $\alpha = 0.6$, we converge quickly



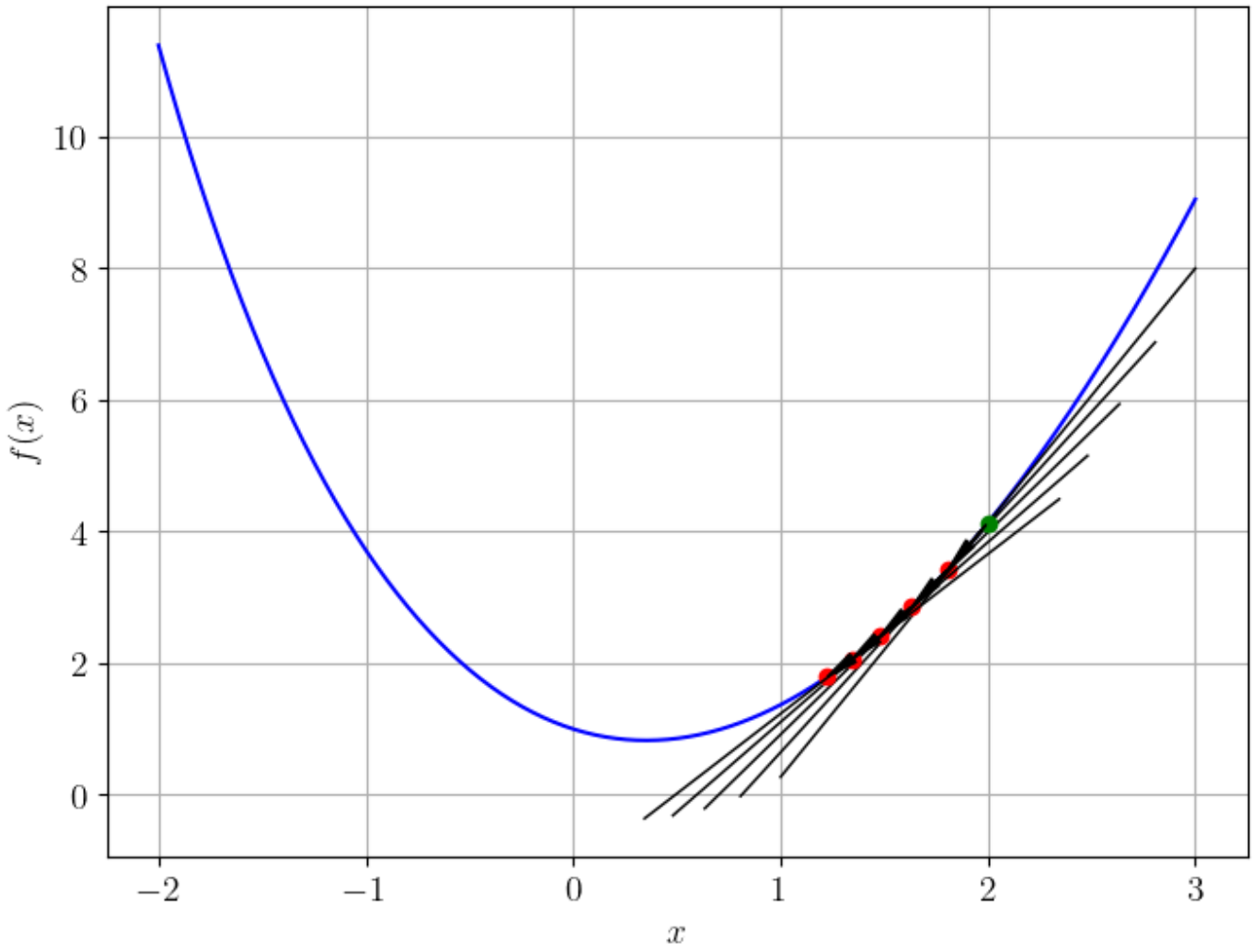
gradient descent

- however, if we choose a slightly larger step size ($\alpha = 0.8$) we actually diverge from the minimum!



gradient descent

- if we play it safe and choose a small step size ($\alpha = 0.05$) then the convergence can be very slow!



line search

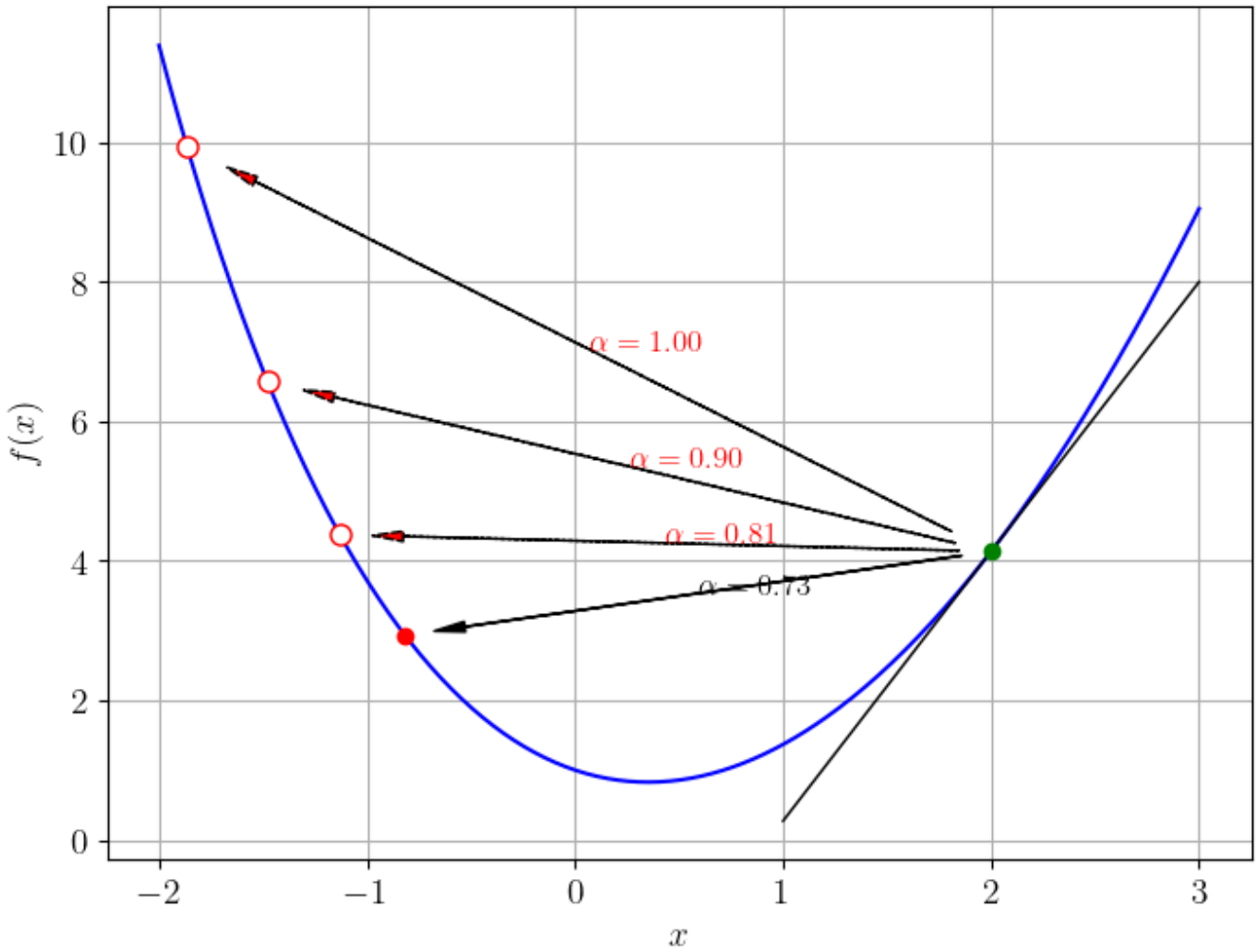
- one way to avoid these problems is to do a **line search**
- the basic idea is that we want to move **along the descent direction** (the negative gradient), but we want to choose the step size small enough to have the function decrease
- the simplest way: starting from an initial step size, say $\alpha = 1$ we check if our function decreases with this step size

$$f(\bar{x} - \alpha \nabla f|_{\bar{x}}) \leq f(\bar{x})$$

- if it does, then we **accept** the step, if it doesn't, we reduce α and try again

gradient descent

- in this example we reduce α by a factor of 0.9 each time (usually it is reduced by 0.5, this is just for illustration)



Newton's method

- Netwon's method is another technique for numerically finding the minimum of a function
- instead of just looking at the descent direction, at each iteration we approximate the function $f(x)$ as a **quadratic function**, then jump to the minimum of the approximation
- it uses more information (the second derivative), but usually performs much better

Newton's method

- first we have to compute both the first and second derivative

$$\frac{df}{dx} = 2x - e^{-x}, \quad \frac{d^2 f}{dx^2} = 2 + e^{-x}$$

- this gives us a **quadratic** approximation of our function (second order Taylor expansion)

$$f(\bar{x} + \Delta x) = f(\bar{x}) + \left.\frac{df}{dx}\right|_{\bar{x}} \Delta x + \frac{1}{2} \left.\frac{d^2 f}{dx^2}\right|_{\bar{x}} \Delta x^2$$

- the minimum of this approximation is easy to find

$$\left.\frac{df}{dx}\right|_{\bar{x}} + \left.\frac{d^2 f}{dx^2}\right|_{\bar{x}} \Delta x = 0 \quad \Longrightarrow \quad \Delta x = -\left(\left.\frac{d^2 f}{dx^2}\right|_{\bar{x}}\right)^{-1} \left.\frac{df}{dx}\right|_{\bar{x}}$$

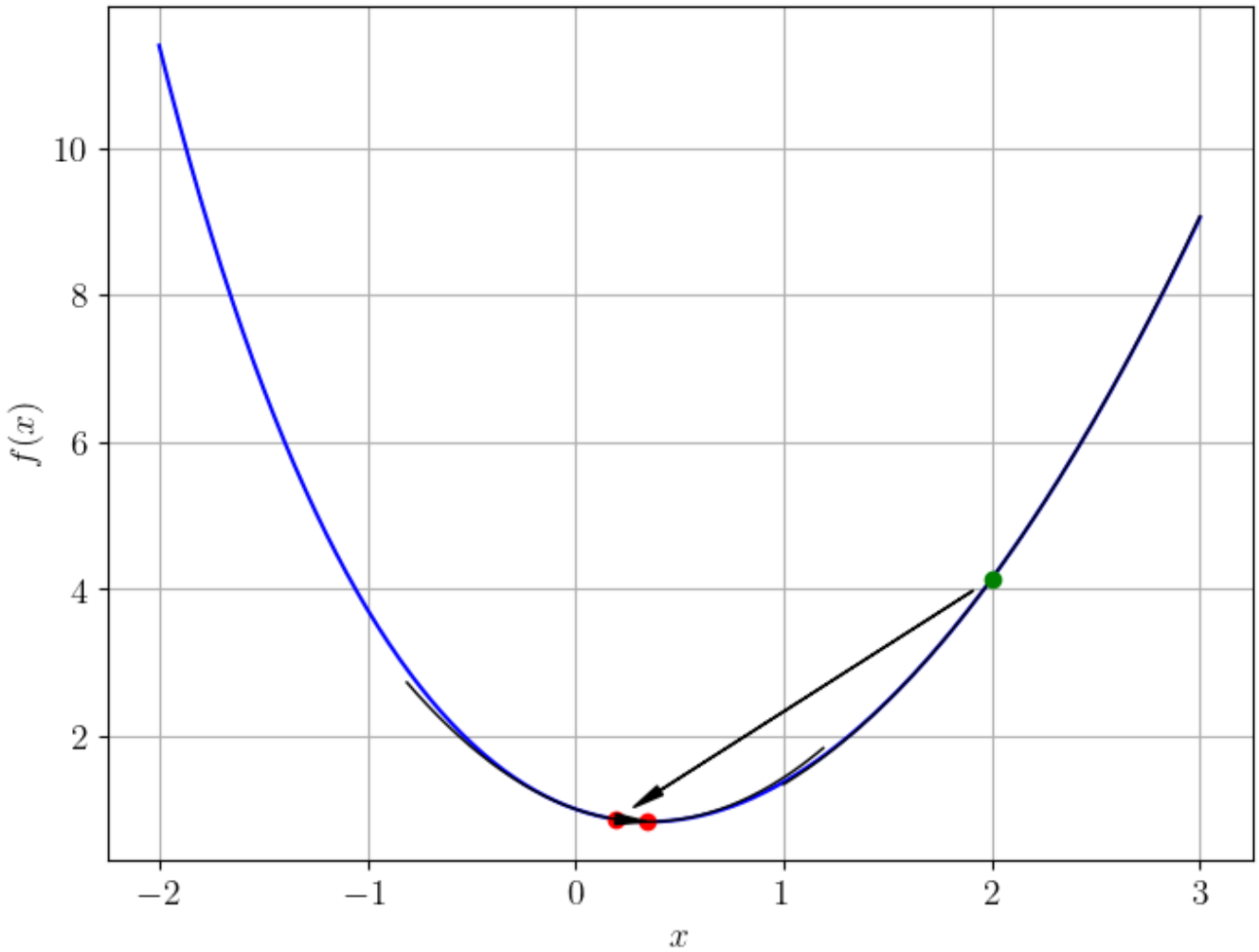
Newton's method

$$\Delta x = -\left(\nabla^2 f|_{\bar{x}}\right)^{-1} \nabla f|_{\bar{x}}$$

- note that, if $\left(\nabla^2 f|_{\bar{x}}\right)^{-1}$ is positive, Newton's method also follows the direction of the negative gradient, but it does so in a smarter way
- however, if $\left(\nabla^2 f|_{\bar{x}}\right)^{-1}$ is negative, we actually go up the gradient, and we might end up in a maximum!
- this should make sense because it means that our quadratic approximation is upside down, so Newton's method actually takes us to the maximum

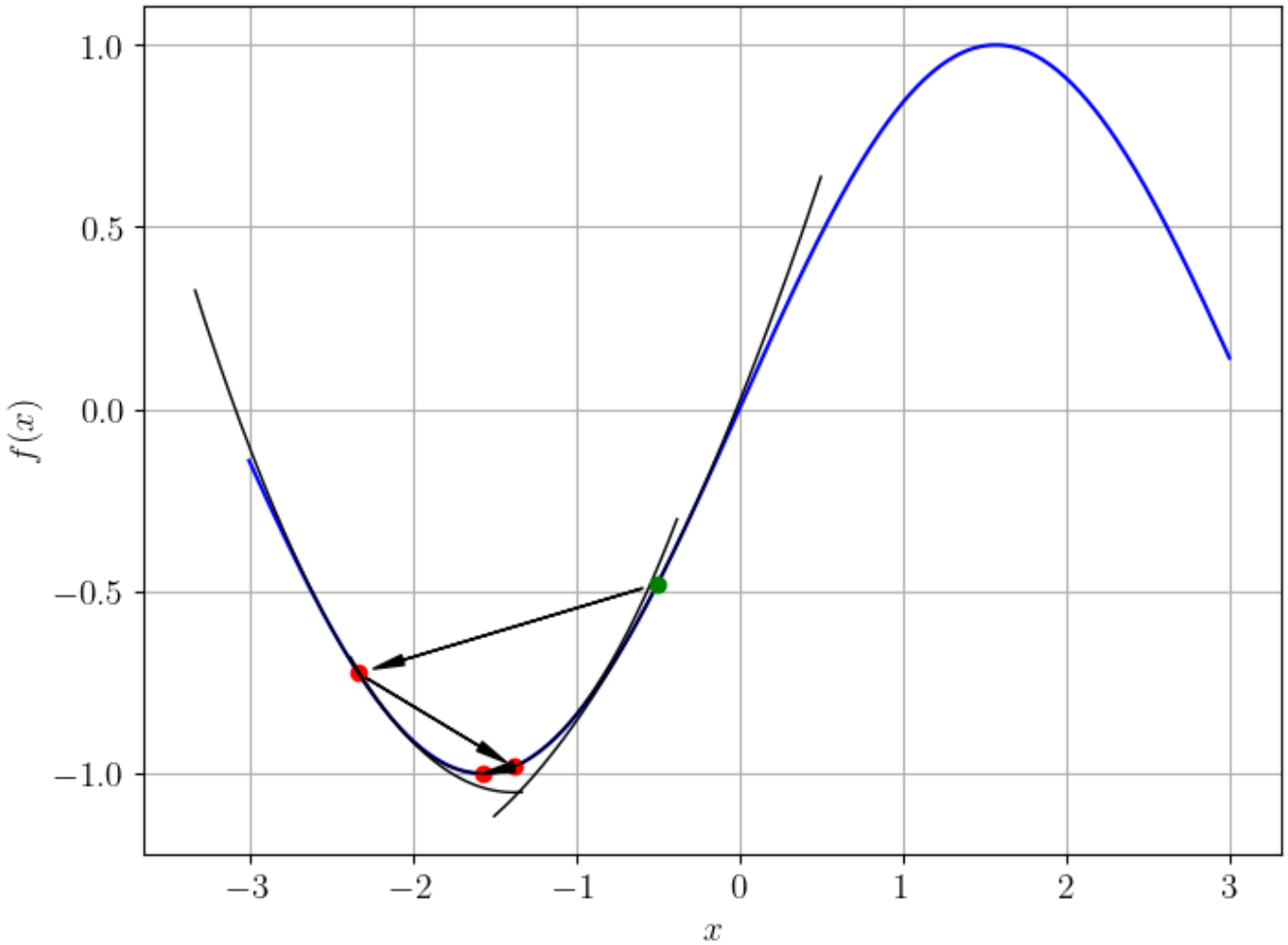
Newton's method

- in our example, Newton's method converges very fast, and we don't even have to choose a step size!



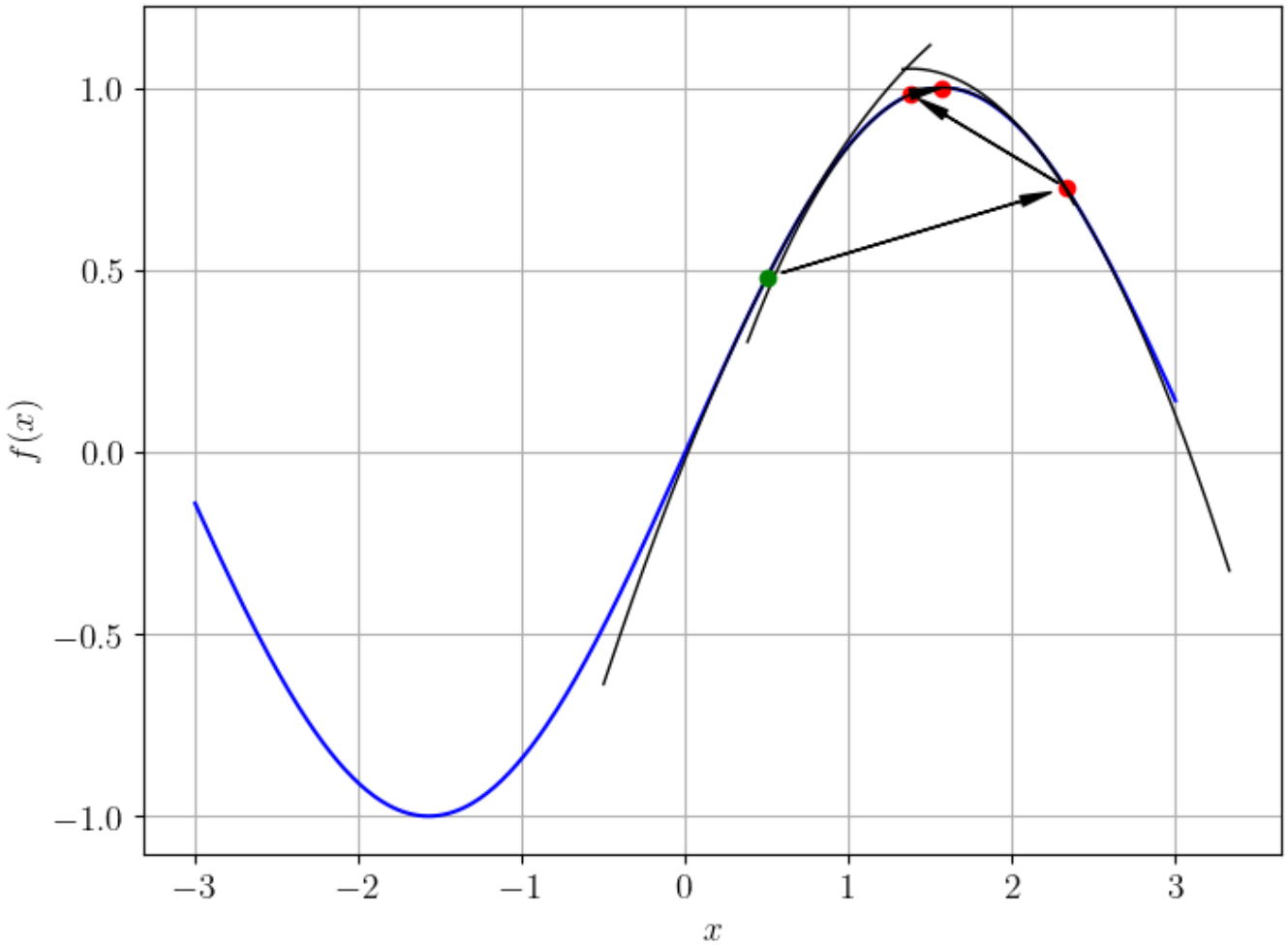
Newton's method

- however, look at what happens if we have a **non-convex** function: if we start close to the minimum, everything is fine



Newton's method

- but if we start at a point where the function is not **convex**, we actually end up in a maximum!



convexity

- **convexity**: if we pick any two points x_a and x_b, and draw a line in between them, the function $f(x)$ is **below** this line

$$\forall x_a, x_b \in \mathbb{R}, \alpha \in (0,1)$$
$$f(\alpha x_a + (1-\alpha)x_b) < \alpha f(x_a) + (1-\alpha) f(x_b)$$

- convex functions have a **single minimum**, thus if we find a solution we can always be sure that it's the **global optimum**

- also, convex functions have a Hessian $\nabla^2 f$ that is always positive, which means that we never run into trouble with Newton's method

Hessian regularization

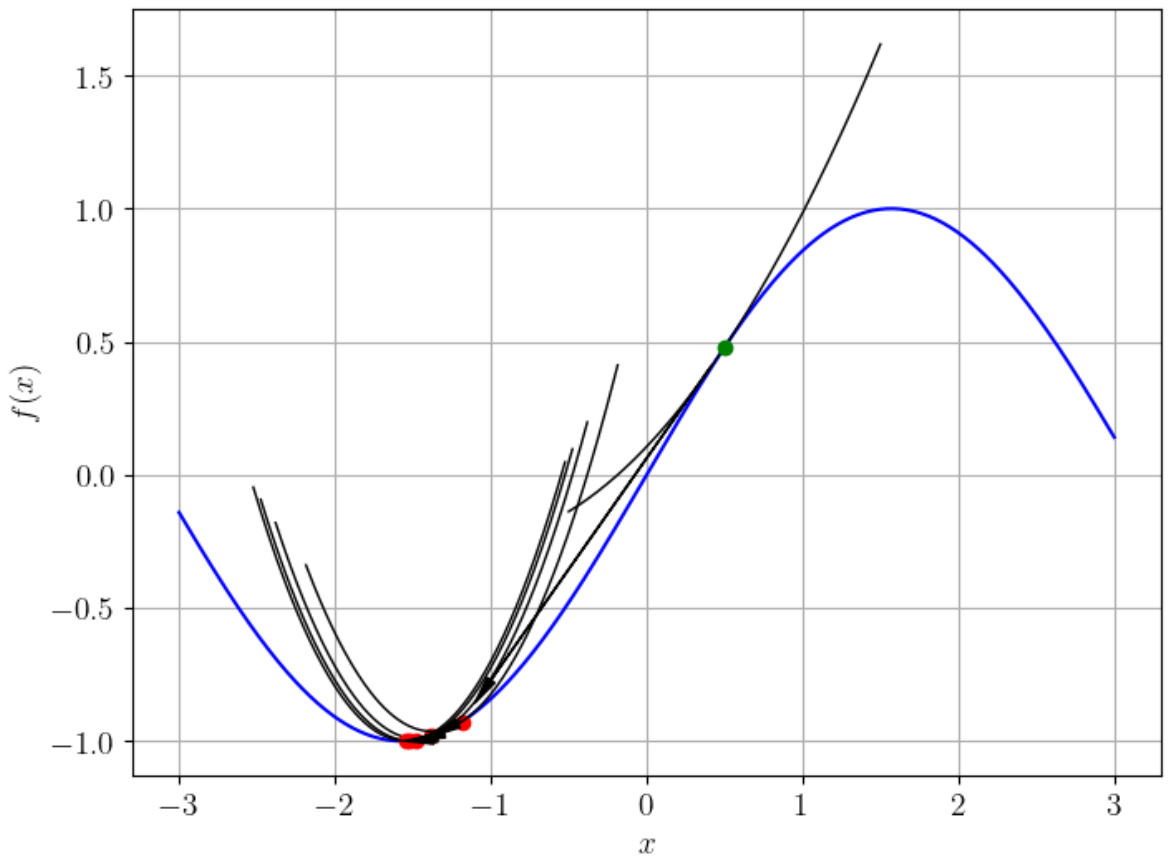
- due to the issue with convexity, it is not always the best thing to use the true Hessian of the function
- we can regularize the Hessian by adding a **damping term** ϵI

$$H_{\text{reg}} = \nabla^2 f|_{\bar{x}} + \epsilon I$$

- this takes care of spots where the function is **slightly** non-convex, but H_{reg} can still become negative

Hessian regularization

- if we regularize the Hessian with $\epsilon = 1$, we converge to the minimum even when starting from a non-convex spot
- note that the approximations are narrower than the actual function, because we are "artificially" increasing the convexity



constraints

- in constrained optimization, we want to minimize a function, but the solution should satisfy some constraints

- usually, we write these constraints in the form of some function that has to be $= 0$ or ≤ 0
  - **equality constraints**: $g(x) = 0$
  - **inequality constraints**: $h(x) \leq 0$

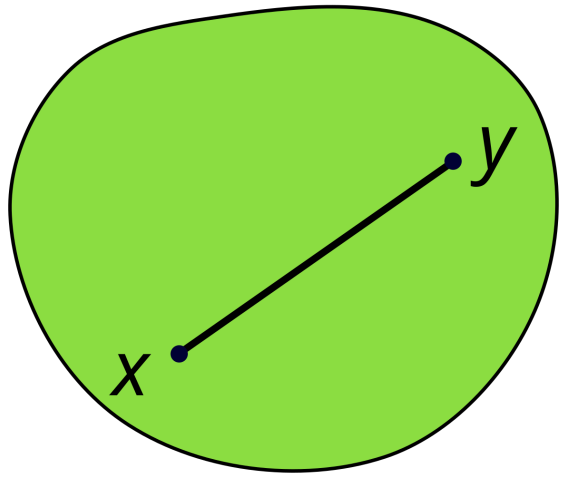
- if we have linear constraints, we usually write them as
  - **equality constraints**: $Ax = b$
  - **inequality constraints**: $Ax \leq b$

  where A is a matrix and b is a vector

convex constraints

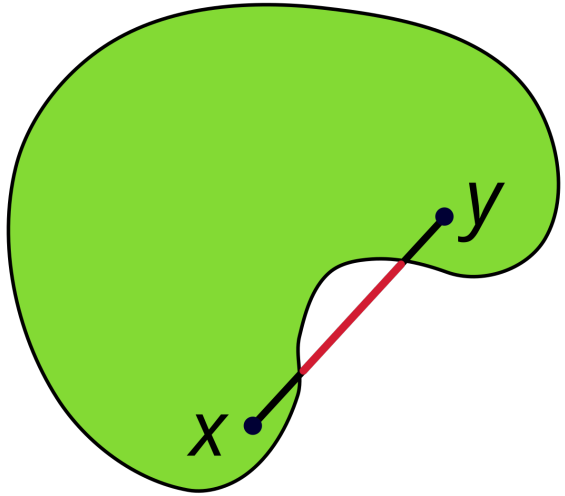
- a set $\mathcal{X}$ is said to be **convex** if

$$\forall x_a, x_b \in \mathcal{X}, \alpha \in (0,1)$$
$$\alpha x_a + (1-\alpha) x_b \in \mathcal{X}$$



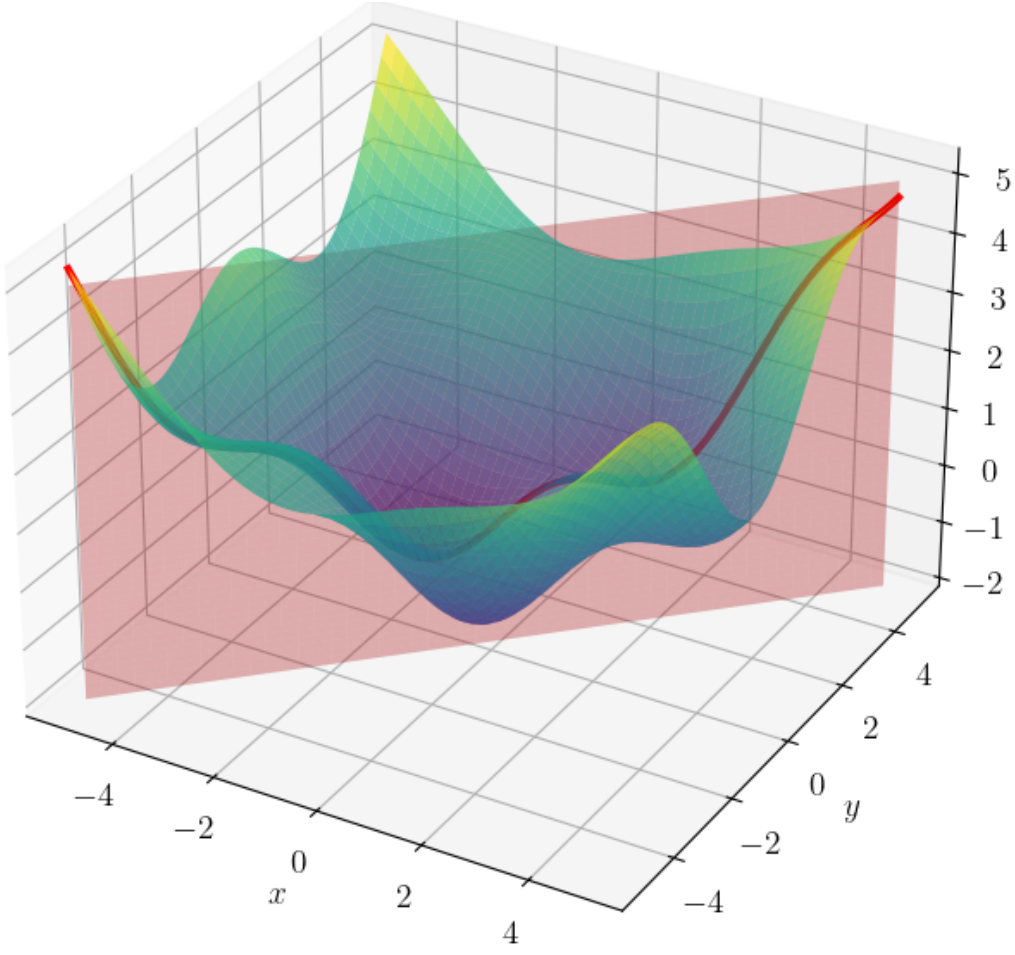
- a **convex constraint** is a constraint whose admissible region is a convex set

- if both the **function** and the **constraints** are convex, we have a **convex optimization problem**



constrained optimization

- for example, minimizing a function subject to a linear equality constraint corresponds to finding the solution on the intersection between the function and a (hyper)plane



Lagrange multipliers

- Lagrange multipliers are used to solve constrained optimization problems

- for a problem with equality constraints

$$\min_x f(x), \quad \text{subject to} \quad g(x) = 0$$

- the **Lagrangian function** is defined as

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T g(x)$$

where λ are the Lagrange multipliers associated with the constraints

Lagrange multipliers

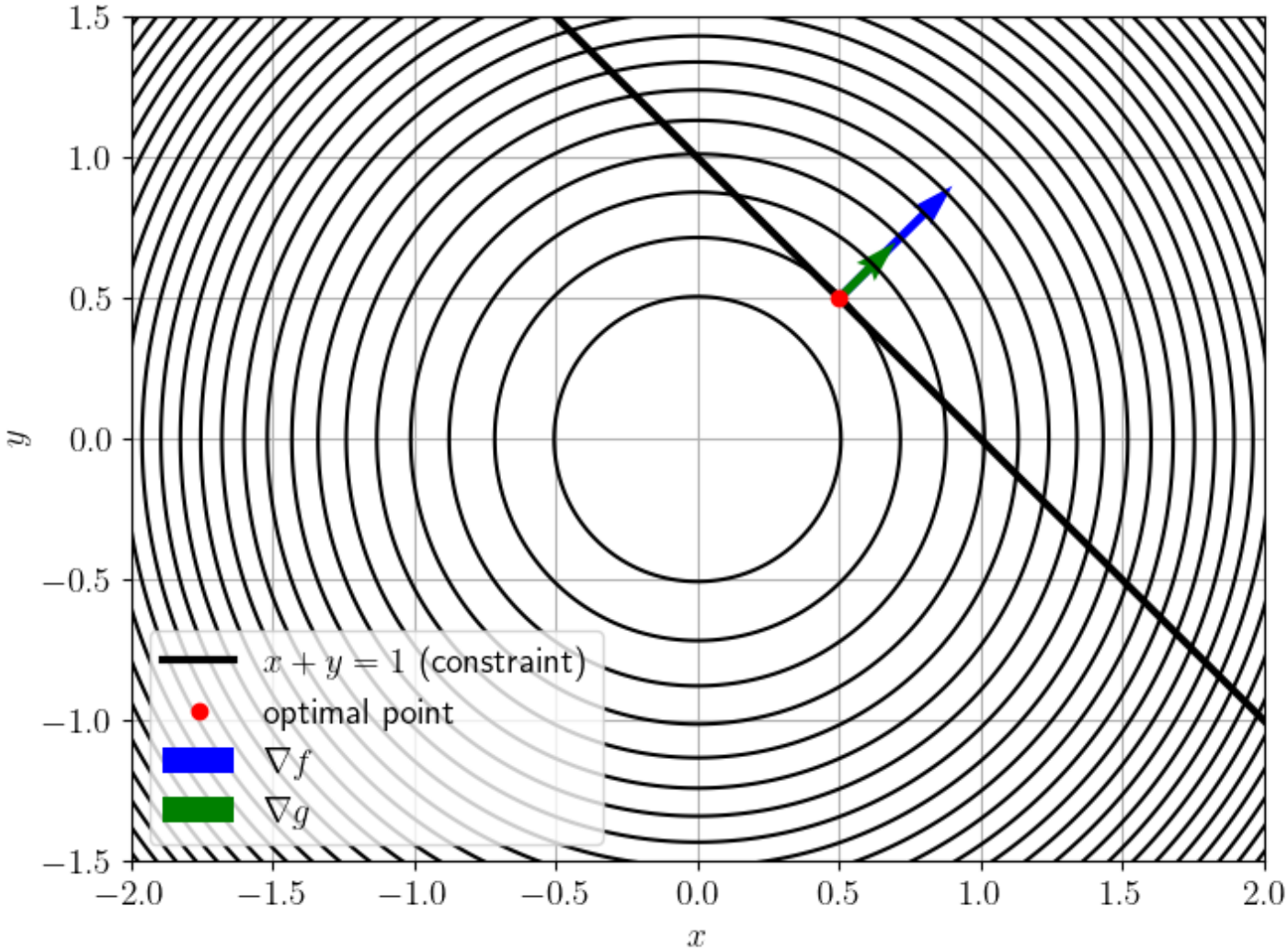
- the gradient of the objective function $\nabla f(x)$ must be aligned with the gradient of the constraint $\nabla g(x)$ at the optimal point
- this implies that at the optimal solution x^*, we have

$$\nabla f(x^*) + \lambda^T \nabla g(x^*) = 0$$

- λ represents the "strength" of the constraint, adjusting the direction of the gradient

Lagrange multipliers

- intuitively, this means that there is no way to decrease the function by moving parallel to the constraint



Lagrange multipliers

- for a problem with only equality constraints, an optimal point must satisfy the following **conditions**:
  1. **stationarity**: $\nabla f(x^*) + \lambda^T \nabla g(x^*) = 0$
  2. **feasibility**: $g(x^*) = 0$

- these conditions are necessary and sufficient if the optimization problem is **convex**

KKT conditions

- for a problem with equality and inequality constraints

$$\min_x \quad f(x), \quad \text{subject to} \quad g_i(x) = 0,\ h_j(x) \leq 0$$

a point x^* must satisfy the **Karush-Kuhn-Tucker (KKT)** conditions

1. **stationarity**: $\nabla f(x^*) + \sum_i \lambda_i \nabla g_i(x^*) + \sum_j \mu_j \nabla h_j(x^*) = 0$
2. **primal feasibility**: $g_i(x^*) = 0,\ h_j(x^*) \leq 0$
3. **dual feasibility**: $\mu_j \geq 0$
4. **complementarity**: $\mu_j h_j(x^*) = 0$

- these conditions are necessary and sufficient if the optimization problem is **convex**

algorithmic differentiation

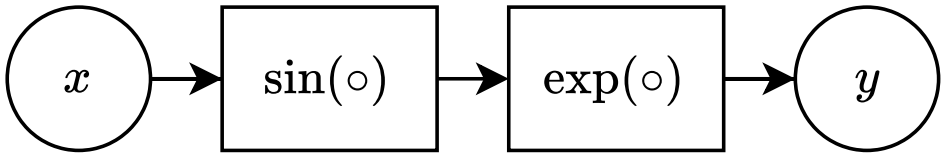
- optimization techniques require computing derivatives, and for large complicated functions this can be difficult to do symbolically

- it is possible to do **numerical differentiation**, by approximating infinitesimal increments with small finite increments, but this can be imprecise

- modern libraries (e.g., CasADi) can perform very efficient **algorithmic differentiation**, which gives exact results without the need of symbolic calculations

algorithmic differentiation

- the basic idea behind algorithmic differentiation is to represent a function as a **computational graph**
- for example, we can represent the function $y = e^{\sin x}$ as



- to evaluate the function we assigning a value to x and move forward through the graph, assigning intermediate values to auxiliary variables (in this case we only need one variable v)

$$x = 1.2$$
$$v = \sin(x) = 0.93$$
$$y = \exp(v) = 2.54$$

algorithmic differentiation

- if we now want to compute $y' = dy/dx$, we can start at the beginning of the graph, then propagate forward using the **chain rule**

$$\begin{aligned} x &= 1.2 & x' &= 1 \\ v &= \sin(x) = 0.93 & v' &= \cos(x) \cdot x' = 0.36 \\ y &= \exp(v) = 2.54 & y' &= \exp(v) \cdot v' = 0.91 \end{aligned}$$

- this is called **forward-mode** algorithmic differentiation

algorithmic differentiation

- there is a different way to obtain the derivative: we first perform a **forward pass** in which we evaluate the function, and then we perform a **backward pass** in which we evaluate the **adjoint variables** $\bar{y}$, $\bar{v}$ and $\bar{x}$

$$
\begin{aligned}
x &= 1.2 \\
v &= \sin(x) = 0.93 \\
y &= \exp(v) = 2.54 \\
\bar{y} &= 1 \\
\bar{v} &= \bar{y} \cdot dy/dv = 1.0 \cdot \exp(v) = 2.54 \\
\bar{x} &= \bar{v} \cdot dv/dx = 2.54 \cdot \cos(x) = 0.91
\end{aligned}
$$

- this is called **reverse-mode** algorithmic differentiation

algorithmic differentiation

- both forward-mode and backward-mode apply the chain rule, but one works from the inputs to the outputs, and the other works from the outputs to the inputs
- forward-mode works best when we have **few inputs** and **many outputs**
- reverse-mode works best when we have **many inputs** and a **single scalar output** (very common in optimization)
- a particular application of reverse-mode algorithmic differentiation is **backpropagation**, used for computing the gradient of a neural network

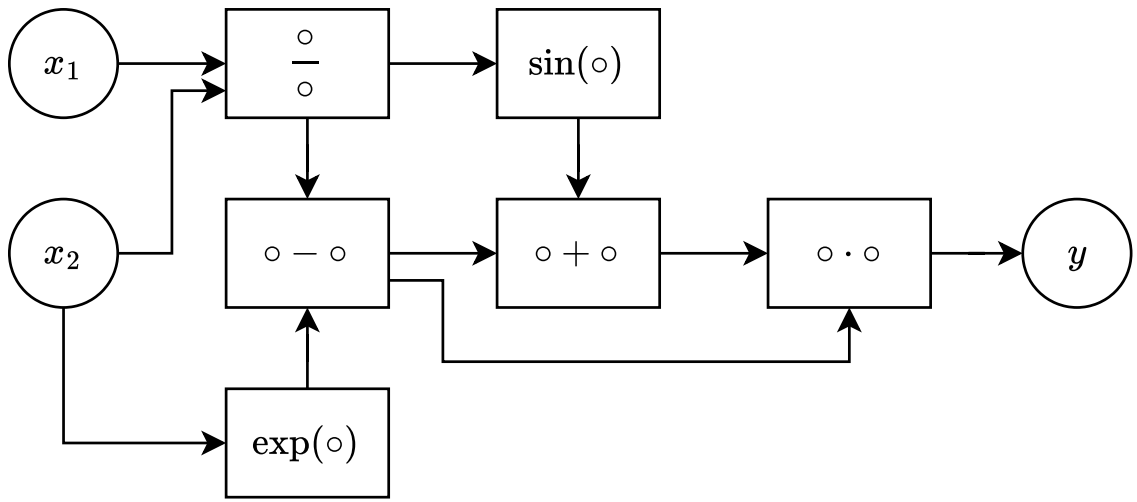
algorithmic differentiation

- for example, the function

$$y = \left(\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2} \right)\left(\frac{x_1}{x_2} - e^{x_2}\right)$$

can be represented as the following sequence of operations

v_1	x_1/x_2
v_2	$\sin(v_1)$
v_3	$\exp(x_2)$
v_4	$v_1 - v_3$
v_5	$v_2 + v_4$
y	$v_5 \cdot v_4$



algorithmic differentiation software

- **CasADi**: a library for automatic differentiation and optimization
- **autograd**: the algorithmic differentiation module used in the machine learning library PyTorch
- **GradientTape**: the algorithmic differentiation module used in the machine learning library TensorFlow
- **JAX**: another popular library for algorithmic differentiation, known for efficient use of parallelization