

Announcements

1. How to avoid PR like: <https://github.com/GiggleLiu/ModernScientificComputing/pull/27>. Help desk event (Saturday night, hosted by Yusheng Zhao) will be announced in the #coding-club stream?
2. Why we should never add a big file into a Github repo?
3. The ChatGPT bot in [HKUST-GZ Zulip workspace](#), @ChatGPT (user stream: #chatgpt-discussion), credit Yijie Xu, Github repo: <https://github.com/yeahjack/chatgptzulipbot>.
4. Have to cancel the next lecture, we will not have final exam!



jinguoliu

Explain the pseudocode of AdaGrad.

ChatGPT 🐼

Here is the pseudocode for the AdaGrad algorithm:

```
Input: Learning rate alpha, Initial weights w, Initial gradient g, Tolerance epsilon

Initialize historical gradient sum r to 0

While not converged:
    Compute gradient g for current weights w
    Update historical gradient sum r as r = r + g^2
    Compute learning rate eta for each weight as eta = alpha / sqrt(r + epsilon)
    Update each weight w as w = w - eta * g
    Check for convergence by comparing the change in the loss function with the tolerance threshold epsilon
```

In this pseudocode, the algorithm takes as input the learning rate alpha, initial weights w, initial gradient g, and a tolerance threshold epsilon that determines when the algorithm has converged. The historical gradient sum r is initialized to 0 at the start of the algorithm.

The algorithm then enters a loop that continues until convergence is achieved. At each iteration of the loop, the gradient g is computed for the current weights w. The historical gradient sum r is updated by adding the square of the gradient g. The learning rate eta is then computed for each weight by dividing the learning rate alpha by the square root of the historical gradient sum r plus a small constant epsilon to prevent division by zero.

The weights are then updated by subtracting the product of the learning rate eta and the gradient g. The algorithm checks for convergence by comparing the change in the loss function with the tolerance threshold epsilon. If the change in the loss function is smaller than epsilon, the algorithm has converged and the loop is exited. Otherwise, the loop continues with the updated weights.

Table of Contents

Announcements

Optimization

Gradient free optimization

The downhill simplex method

Gradient based optimization

Gradient descent

Gradient descent with momentum

AdaGrad

Adam

The Julia package `Optimisers.jl`

Hessian based optimizers

Newton's Method

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm

Mathematical optimization

Convex optimization

Linear programming

Example

Assignments

Hint:

Optimization

Reference: **Scientific Computing - Chapter 6**

A general continuous optimization problem has the following form

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{subject to certain constraints}$$

The constraints may be either equality or inequality constraints.

Gradient free optimization

Gradient-free optimizers are optimization algorithms that do not rely on the gradient of the objective function to find the optimal solution. Instead, they use other methods such as random search, genetic algorithms, or simulated annealing to explore the search space and find the optimal solution. These methods are particularly useful when the objective function is non-differentiable or when the gradient is difficult to compute. However, gradient-free optimizers can be slower and less efficient than gradient-based methods, especially when the search space is high-dimensional.

There are several popular gradient-free optimizers, including:

- **Genetic algorithms:** These are optimization algorithms inspired by the process of natural selection. They use a population of candidate solutions and apply genetic operators such as crossover and mutation to generate new solutions.
- **Simulated annealing:** This is a probabilistic optimization algorithm that uses a temperature parameter to control the probability of accepting a worse solution. It starts with a high temperature that allows for exploration of the search space and gradually decreases the temperature to converge to the optimal solution.
- **Particle swarm optimization:** This is a population-based optimization algorithm that simulates the movement of particles in a search space. Each particle represents a candidate solution, and they move towards the best solution found so far.
- **Bayesian optimization:** This is a probabilistic optimization algorithm that uses a probabilistic model to approximate the objective function and guides the search towards promising regions of the search space.
- **Nelder-Mead algorithm:** This is a direct search method that does not require the computation of gradients of the objective function. Instead, it uses a set of simplex (a geometrical figure that generalizes the concept of a triangle to higher dimensions) to iteratively explore the search space and improve the objective function value. The Nelder-Mead algorithm is particularly effective in optimizing nonlinear and non-smooth functions, and it is widely used in engineering, physics, and other fields.

NOTE: [Optim.jl documentation](#) contains more detailed introduction of gradient free, gradient based and hessian based optimizers.

The downhill simplex method

Here are the steps involved in the one dimensional downhill simplex algorithm:

1. Initialize a one dimensional simplex, evaluate the function at the end points x_1 and x_2 and assume $f(x_2) > f(x_1)$.
2. Evaluate the function at $x_c = 2x_1 - x_2$.
3. Select one of the following operations
 1. If $f(x_c)$ is smaller than $f(x_1)$, **flip** the simplex by doing $x_1 \leftarrow x_c$ and $x_2 \leftarrow x_1$.
 2. If $f(x_c)$ is larger than $f(x_1)$, but smaller than $f(x_2)$, then $x_2 \leftarrow x_c$, goto case 3.
 3. If $f(x_c)$ is larger than $f(x_2)$, then **shrink** the simplex: evaluate f on $x_d \leftarrow (x_1 + x_2)/2$, if it is larger than $f(x_1)$, then $x_2 \leftarrow x_d$, otherwise $x_1 \leftarrow x_d, x_2 \leftarrow x_1$.
4. Repeat step 2-3 until convergence.

simplex1d (generic function with 1 method)

```
1 function simplex1d(f, x1, x2; tol=1e-6)
2     # initial simplex
3     history = [[x1, x2]]
4     f1, f2 = f(x1), f(x2)
5     while abs(x2 - x1) > tol
6         xc = 2x1 - x2
7         fc = f(xc)
8         if fc < f1 # flip
9             x1, f1, x2, f2 = xc, fc, x1, f1
10        else # shrink
11            if fc < f2 # let the smaller one be x2.
12                x2, f2 = xc, fc
13            end
14            xd = (x1 + x2) / 2
15            fd = f(xd)
16            if fd < f1 # update x1 and x2
17                x1, f1, x2, f2 = xd, fd, x1, f1
18            else
19                x2, f2 = xd, fd
20            end
21        end
22        push!(history, [x1, x2])
23    end
24    return x1, f1, history
25 end
```

(3.57628e-7, 1.27898e-13, [[-1.0, 6.0], [-1.0, 2.5], [0.75, -1.0], [-0.125, 0.75], [-0.125

```
1 simplex1d(x->x^2, -1.0, 6.0)
```

The Nelder-Mead method is well summarized in this [wiki page](#). Here is a Julia implementation:

simplex (generic function with 1 method)

```
1 function simplex(f, x0; tol=1e-6, maxiter=1000)
2     n = length(x0)
3     x = zeros(n+1, n)
4     fvals = zeros(n+1)
5     x[1,:] = x0
6     fvals[1] = f(x0)
7     alpha = 1.0
8     beta = 0.5
9     gamma = 2.0
10    for i in 1:n
11        x[i+1,:] = x[i,:]
12        x[i+1,i] += 1.0
13        fvals[i+1] = f(x[i+1,:])
14    end
15    history = [x]
16    for iter in 1:maxiter
17        # Sort the vertices by function value
18        order = sortperm(fvals)
19        x = x[order,:]
20        fvals = fvals[order]
21        # Calculate the centroid of the n best vertices
22        xbar = dropdims(sum(x[1:n,:], dims=1) ./ n, dims=1)
23        # Reflection
24        xr = xbar + alpha*(xbar - x[n+1,:])
25        fr = f(xr)
26        if fr < fvals[1]
27            # Expansion
28            xe = xbar + gamma*(xr - xbar)
29            fe = f(xe)
30            if fe < fr
31                x[n+1,:] = xe
32                fvals[n+1] = fe
33            else
34                x[n+1,:] = xr
35                fvals[n+1] = fr
36            end
37        elseif fr < fvals[n]
38            x[n+1,:] = xr
39            fvals[n+1] = fr
40        else
41            # Contraction
42            if fr < fvals[n+1]
43                xc = xbar + beta*(x[n+1,:] - xbar)
44                fc = f(xc)
45                if fc < fr
46                    x[n+1,:] = xc
47                    fvals[n+1] = fc
48                else
49                    # Shrink
50                    for i in 2:n+1
51                        x[i,:] = x[1,:] + beta*(x[i,:] - x[1,:])
52                        fvals[i] = f(x[i,:])
53                    end
```

```

54         end
55     else
56         # Shrink
57         for i in 2:n+1
58             x[i,:] = x[1,:] + beta*(x[i,:] - x[1,:])
59             fvals[i] = f(x[i,:])
60         end
61     end
62 end
63 push!(history, x)
64 # Check for convergence
65 if maximum(abs.(x[2:end,:] .- x[1,:])) < tol && maximum(abs.(fvals[2:end] .-
fvals[1])) < tol
66     break
67 end
68 end
69 # Return the best vertex and function value
70 bestx = x[1,:]
71 bestf = fvals[1]
72 return (bestx, bestf, history)
73 end

```

The `simplex` function takes three arguments: the objective function `f`, the initial guess `x0`, and optional arguments for the tolerance `tol` and maximum number of iterations `maxiter`.

The algorithm initializes a simplex (a high dimensional triangle) with $n+1$ vertices, where n is the number of dimensions of the problem. The vertices are initially set to `x0` and `x0 + h_i`, where `h_i` is a small step size in the i th dimension. The function values at the vertices are also calculated.

The algorithm then iteratively performs **reflection**, **expansion**, **contraction**, and **shrink** operations on the simplex until convergence is achieved. The best vertex and function value are returned.

We use the [Rosenbrock function](#) as the test function.

`rosenbrock` (generic function with 1 method)

```

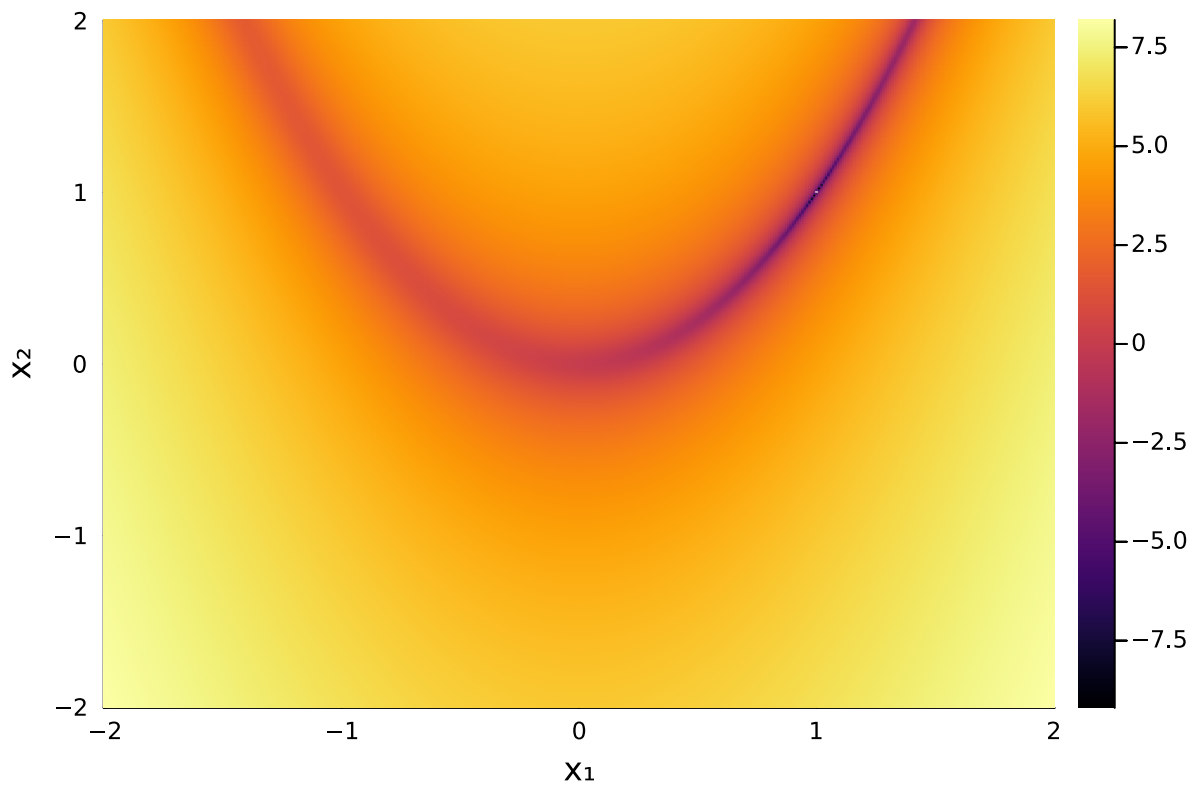
1 function rosenbrock(x)
2     (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2
3 end

```

```

1 using Plots

```



```

1 let
2     x = -2:0.01:2
3     y = -2:0.01:2
4     f = [rosenbrock((a, b)) for b in y, a in x]
5     heatmap(x, y, log.(f); label="log(f)", xlabel="x1", ylabel="x2")
6 end

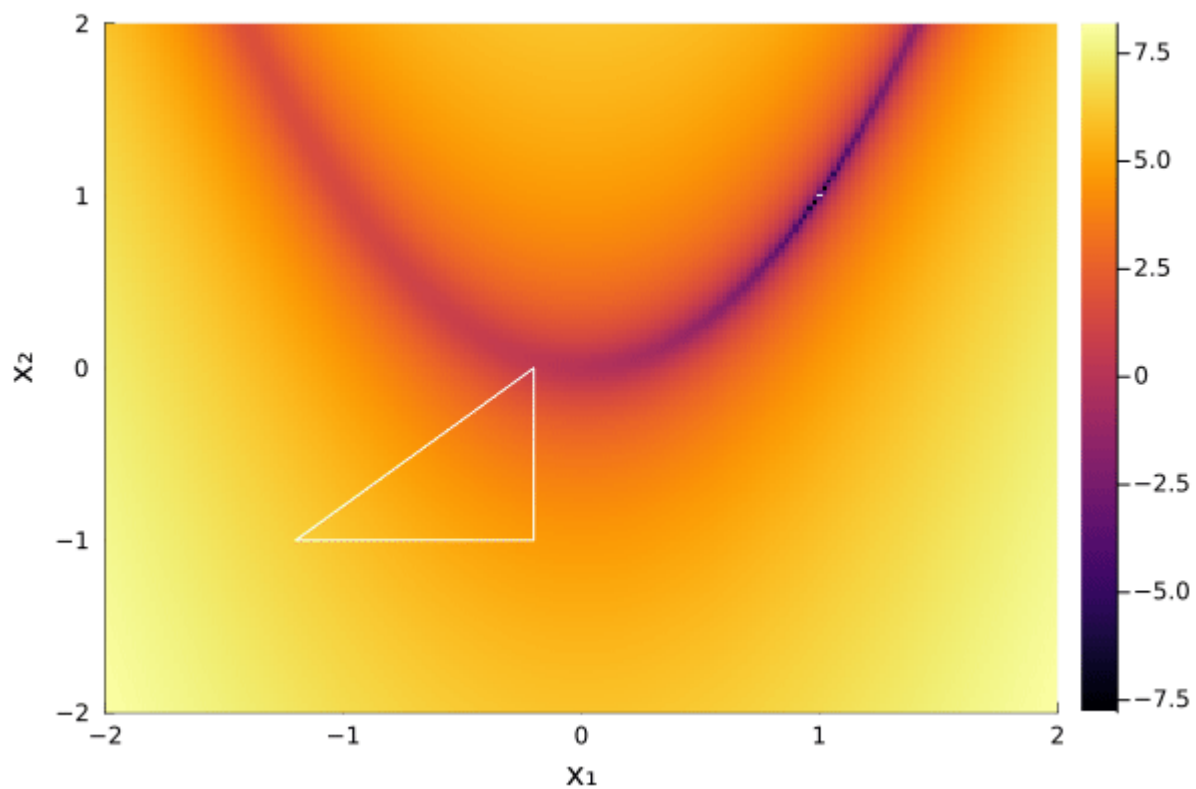
```

show_triangles (generic function with 1 method)

```

1 function show_triangles(history)
2     x = -2:0.02:2
3     y = -2:0.02:2
4     f = [rosenbrock((a, b)) for b in y, a in x]
5     @gif for item in history
6         plt = heatmap(x, y, log.(f); label="log(f)", xlabel="x1", ylabel="x2", xlim=
            (-2, 2), ylim=(-2, 2))
7         plot!(plt, [item[:,1]..., item[1,1]], [item[:,2]..., item[1, 2]]; label="",
            color="white")
8     end fps=5
9 end

```



```
1 let
2     bestx, bestf, history = simplex(rosenbrock, [-1.2, -1.0]; tol=1e-3)
3     @info "converged in $(length(history)) steps, with error $bestf"
4     show_triangles(history)
5 end
```

converged in 72 steps, with error 6.087010938696048e-7

Saved animation to /tmp/jl_49Ylw9ivOd.gif

```
1 using Optim
```



```

* Status: success

* Candidate solution
  Final objective value:      9.211146e-10

* Found with
  Algorithm:      Nelder-Mead

* Convergence measures
   $\sqrt{(\sum (y_i - \bar{y})^2)/n} \leq 1.0e-08$ 

* Work counters
  Seconds run:      0 (vs limit Inf)
  Iterations:      59
  f(x) calls:      117

```

```

1 let
2   # Set the initial guess
3   x0 = [-1, -1.0]
4   # Set the optimization options
5   options = Optim.Options(iterations = 1000)
6   # Optimize the Rosenbrock function using the simplex method
7   result = optimize(rosenbrock, x0, NelderMead(), options)
8   # Print the optimization result
9   result
10 end

```

Gradient based optimization

If $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is differentiable, then the vector-valued function $\nabla f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ defined by

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{pmatrix}$$

is called the gradient of f .

Gradient descent is based on the observation that changing \mathbf{x} slightly towards the negative gradient direction always decrease f in the first order perturbation.

$$f(\mathbf{x} - \epsilon \nabla f(\mathbf{x})) \approx f(\mathbf{x}) - \epsilon \nabla f(\mathbf{x})^T \nabla f(\mathbf{x}) = f(\mathbf{x}) - \epsilon \|\nabla f(\mathbf{x})\|_2 < f(\mathbf{x})$$

Gradient descent

In each iteration, the update rule of the gradient descent method is

$$\theta_{t+1} = \theta_t - \alpha g_t$$

where

- θ_t is the values of variables at time step t .
- g_t is the gradient at time t along θ_t , i.e. $\nabla_{\theta_t} f(\theta_t)$.
- α is the learning rate.

One can obtain the gradient with `ForwardDiff`.

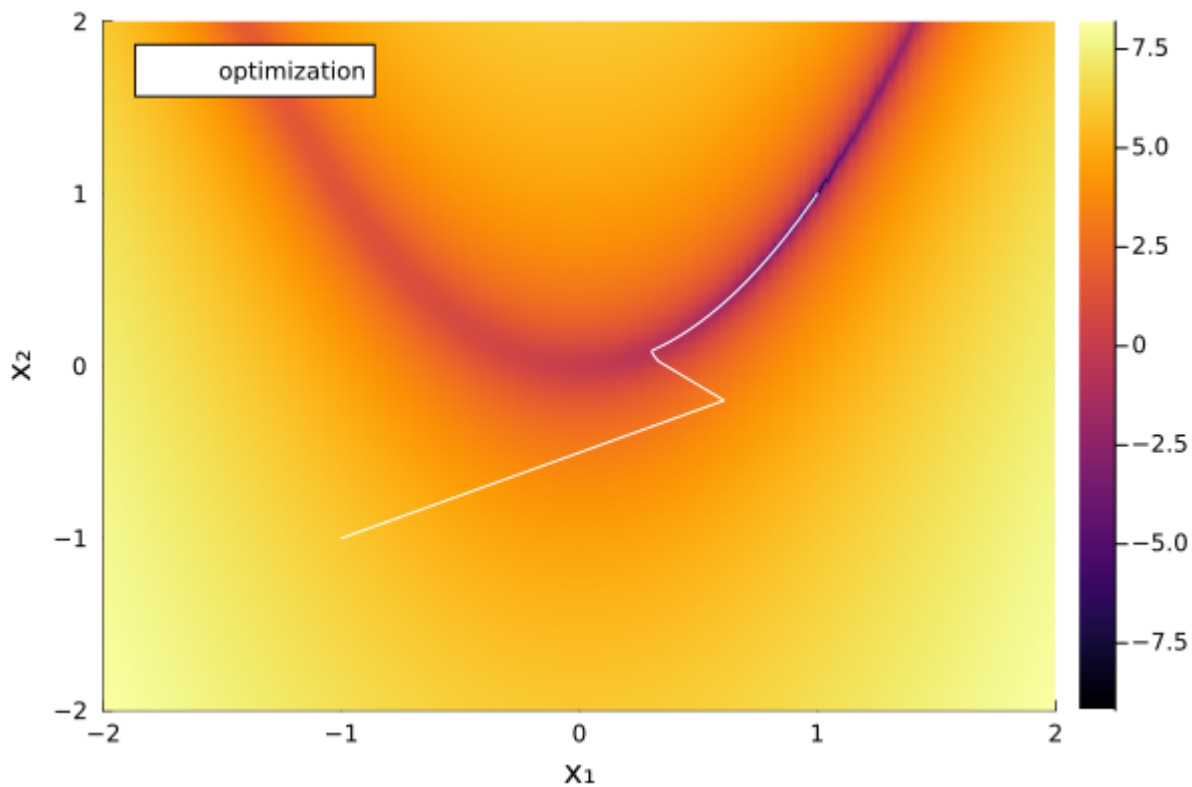
```
1 using ForwardDiff
```

```
[-800.0, 400.0]
```

```
1 ForwardDiff.gradient(rosenbrock, [1.0, 3.0])
```

gradient_descent (generic function with 1 method)

```
1 function gradient_descent(f, x; niters::Int, learning_rate::Real)
2     history = [x]
3     for i=1:niters
4         g = ForwardDiff.gradient(f, x)
5         x -= learning_rate * g
6         push!(history, x)
7     end
8     return history
9 end
```



```

1 let
2   x0 = [-1, -1.0]
3   history = gradient_descent(rosenbrock, x0; niters=10000, learning_rate=0.002)
4   @info rosenbrock(history[end])
5
6   # plot
7   show_history(history)
8 end

```

8.937860566104837e-9

show_history (generic function with 1 method)

```

1 function show_history(history)
2   x = -2:0.01:2
3   y = -2:0.01:2
4   f = [rosenbrock((a, b)) for b in y, a in x]
5   plt = heatmap(x, y, log.(f); label="log(f)", xlabel="x1", ylabel="x2", xlim=(-2,
6   2), ylim=(-2, 2))
7   plot!(plt, getindex.(history, 1), getindex.(history, 2); label="optimization",
8   color="white")
9 end

```

The problem of gradient descent: easy trapped by plateaus.

Gradient descent with momentum

We can add a "momentum" term to the weight update, which helps the optimization algorithm to move more quickly in the right direction and avoid getting stuck in local minima.

The intuition behind the momentum method can be understood by considering a ball rolling down a hill. Without momentum, the ball would roll down the hill and eventually come to a stop at the bottom. However, with momentum, the ball would continue to roll past the bottom of the hill and up the other side, before eventually coming to a stop at a higher point. This is because the momentum of the ball helps it to overcome small bumps and obstacles in its path and continue moving in the right direction.

In each iteration, the update rule of gradient descent method with momentum is

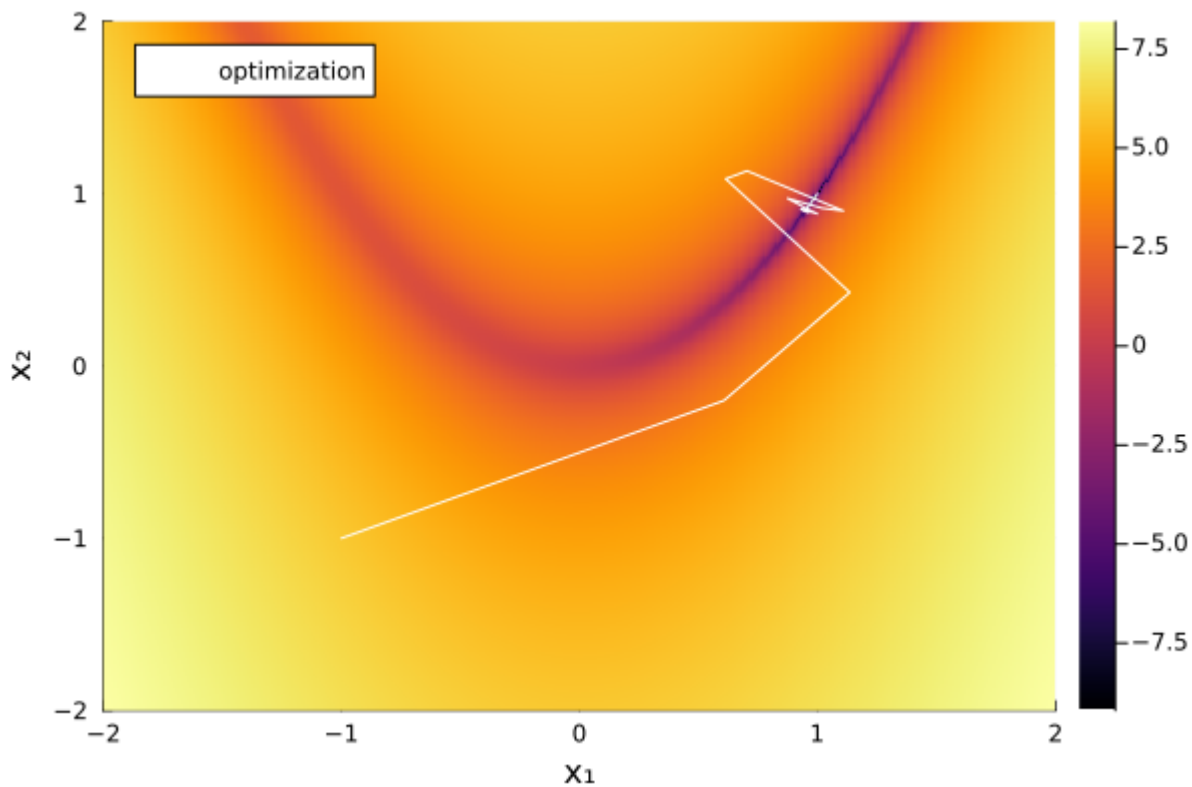
$$\begin{aligned}v_{t+1} &= \beta v_t - \alpha g_t \\ \theta_{t+1} &= \theta_t + v_{t+1}\end{aligned}$$

where

- g_t is the gradient at time t along θ_t , i.e. $\nabla_{\theta_t} f(\theta_t)$.
- α is the initial learning rate.
- β is the parameter for the gradient accumulation.

gradient_descent_momentum (generic function with 1 method)

```
1 function gradient_descent_momentum(f, x; niters::Int, β::Real, learning_rate::Real)
2     history = [x]
3     v = zero(x)
4     for i=1:niters
5         g = ForwardDiff.gradient(f, x)
6         v = β .* v .- learning_rate .* g
7         x += v
8         push!(history, x)
9     end
10    return history
11 end
```



```
1 let
2   x0 = [-1, -1.0]
3   history = gradient_descent_momentum(rosenbrock, x0; niters=10000,
4   learning_rate=0.002,  $\beta$ =0.5)
5   @info rosenbrock(history[end])
6   # plot
7   show_history(history)
8 end
```

3.004455068371721e-17

The problem of momentum based method, easily got overshooted. Moreover, it is not **scale-invariant**.

AdaGrad

AdaGrad is an optimization algorithm used in machine learning for solving convex optimization problems. It is a gradient-based algorithm that adapts the learning rate for each parameter based on the historical gradient information. The main idea behind AdaGrad is to give more weight to the parameters that have a smaller gradient magnitude, which allows for a larger learning rate for those parameters.

In each iteration, the update rule of AdaGrad is

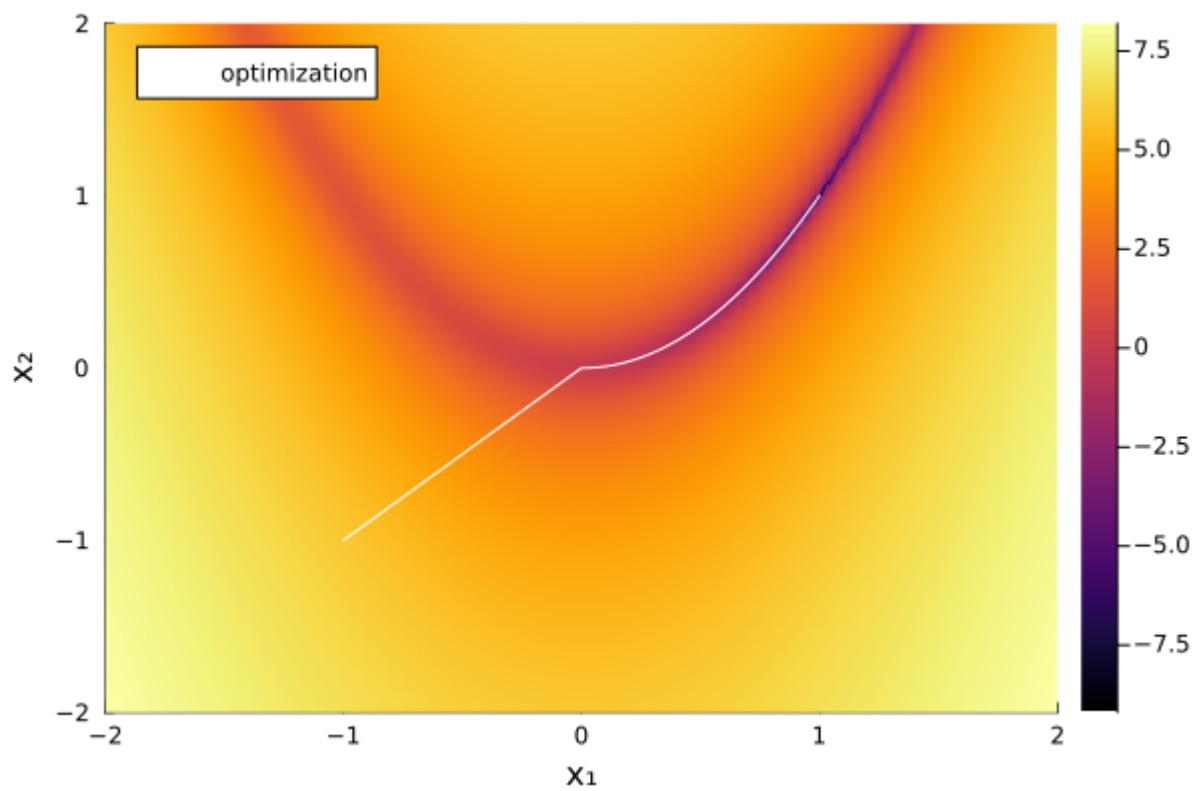
$$\begin{aligned}r_t &= r_t + g_t^2 \\ \eta &= \frac{\alpha}{\sqrt{r_t + \epsilon}} \\ \theta_{t+1} &= \theta_t - \eta \odot g_t\end{aligned}$$

where

- θ_t is the values of variables at time t .
- α is the initial learning rate.
- g_t is the gradient at time t along θ_t
- r_t is the historical squared gradient sum, which is initialized to 0 .
- ϵ is a small positive number.
- \odot is the element-wise multiplication.

adagrad_optimize (generic function with 1 method)

```
1 function adagrad_optimize(f, x; niters, learning_rate, ε=1e-8)
2     rt = zero(x)
3     η = zero(x)
4     history = [x]
5     for step in 1:niters
6         Δ = ForwardDiff.gradient(f, x)
7         @. rt = rt + Δ .^ 2
8         @. η = learning_rate ./ sqrt.(rt + ε)
9         x = x .- Δ .* η
10        push!(history, x)
11    end
12    return history
13 end
```



```
1 let
2   x0 = [-1, -1.0]
3   history = adagrad_optimize(rosenbrock, x0; niters=10000, learning_rate=1.0)
4   @info rosenbrock(history[end])
5
6   # plot
7   show_history(history)
8 end
```

8.215967416285362e-9

Adam

The Adam optimizer is a popular optimization algorithm used in deep learning for training neural networks. It stands for Adaptive Moment Estimation and is a variant of stochastic gradient descent (SGD) that is designed to be more efficient and effective in finding the optimal weights for the neural network.

The Adam optimizer maintains a running estimate of the first and second moments of the gradients of the weights with respect to the loss function. These estimates are used to adaptively adjust the learning rate for each weight parameter during training. The first moment estimate is the mean of the gradients, while the second moment estimate is the uncentered variance of the gradients.

The Adam optimizer combines the benefits of two other optimization algorithms: AdaGrad, which adapts the learning rate based on the historical gradient information, and RMSProp, which uses a moving average of the squared gradients to scale the learning rate.

The Adam optimizer has become a popular choice for training deep neural networks due to its fast convergence and good generalization performance. It is widely used in many deep learning frameworks, such as TensorFlow, PyTorch, and Keras.

In each iteration, the update rule of Adam is

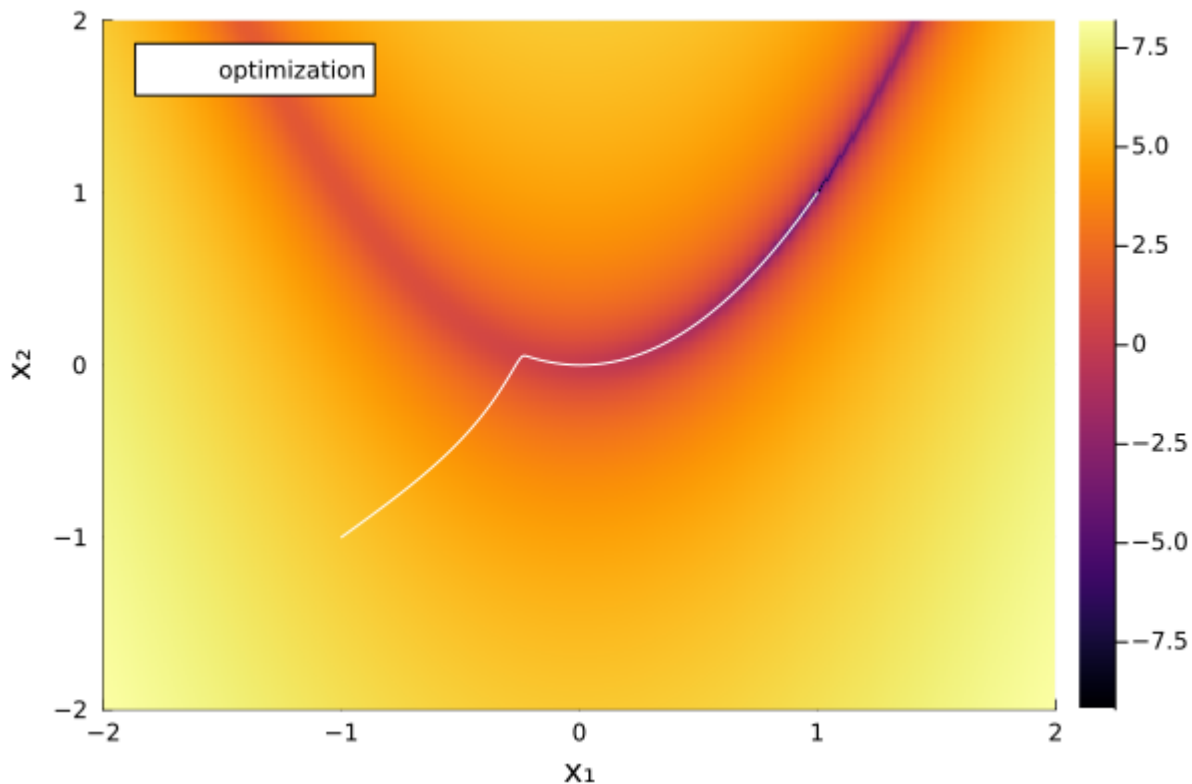
$$\begin{aligned}v_t &= \beta_1 v_{t-1} + (1 - \beta_1) g_t \\s_t &= \beta_2 s_{t-1} + (1 - \beta_2) g_t^2 \\ \hat{v}_t &= v_t / (1 - \beta_1^t) \\ \hat{s}_t &= s_t / (1 - \beta_2^t) \\ \theta_{t+1} &= \theta_t - \eta \frac{\hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon}\end{aligned}$$

where

- θ_t is the values of variables at time t .
- η is the initial learning rate.
- g_t is the gradient at time t along θ .
- v_t is the exponential average of gradients along θ .
- s_t is the exponential average of squares of gradients along θ .
- β_1, β_2 are hyperparameters.

adam_optimize (generic function with 1 method)

```
1 function adam_optimize(f, x; niters, learning_rate, β1=0.9, β2=0.999, ε=1e-8)
2     mt = zero(x)
3     vt = zero(x)
4     βp1 = β1
5     βp2 = β2
6     history = [x]
7     for step in 1:niters
8         Δ = ForwardDiff.gradient(f, x)
9         @. mt = β1 * mt + (1 - β1) * Δ
10        @. vt = β2 * vt + (1 - β2) * Δ^2
11        @. Δ = mt / (1 - βp1) / (√(vt / (1 - βp2)) + ε) * learning_rate
12        βp1, βp2 = βp1 * β1, βp2 * β2
13        x = x .- Δ
14        push!(history, x)
15    end
16    return history
17 end
```



```
1 let
2     x0 = [-1, -1.0]
3     history = adam_optimize(rosenbrock, x0; niters=10000, learning_rate=0.01)
4     @info rosenbrock(history[end])
5
6     # plot
7     show_history(history)
8 end
```

9.64781526674972e-15

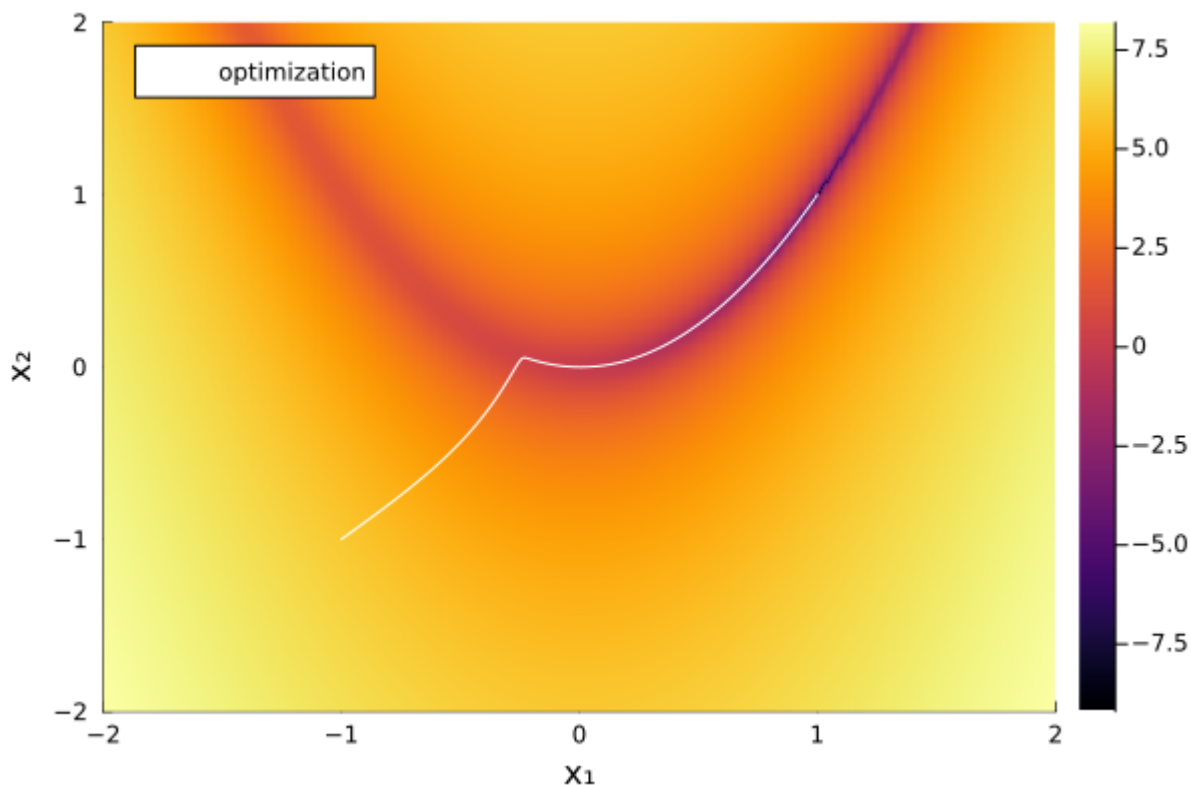
The Julia package Optimisers.jl

```
1 import Optimisers
```

```
gradient_based_optimizer = AdamW
```

```
learning_rate = 0.01
```

The different optimizers are introduced in the [documentation page](#)



```
1 let
2     x0 = [-1, -1.0]
3     method = eval(:(Optimisers.$(Symbol(gradient_based_optimizer))(learning_rate)))
4     state = Optimisers.setup(method, x0)
5     history = [x0]
6     for i=1:10000
7         grad = ForwardDiff.gradient(rosenbrock, x0)
8         state, x0 = Optimisers.update(state, x0, grad)
9         push!(history, x0)
10    end
11    @info rosenbrock(history[end])
12
13    # plot
14    show_history(history)
15 end
```

9.661831723258281e-15

Hessian based optimizers

Newton's Method

Newton's method is an optimization algorithm used to find the roots of a function, which can also be used to find the minimum or maximum of a function. The method involves using the first and second derivatives of the function to approximate the function as a quadratic function and then finding the minimum or maximum of this quadratic function. The minimum or maximum of the quadratic function is then used as the next estimate for the minimum or maximum of the original function, and the process is repeated until convergence is achieved.

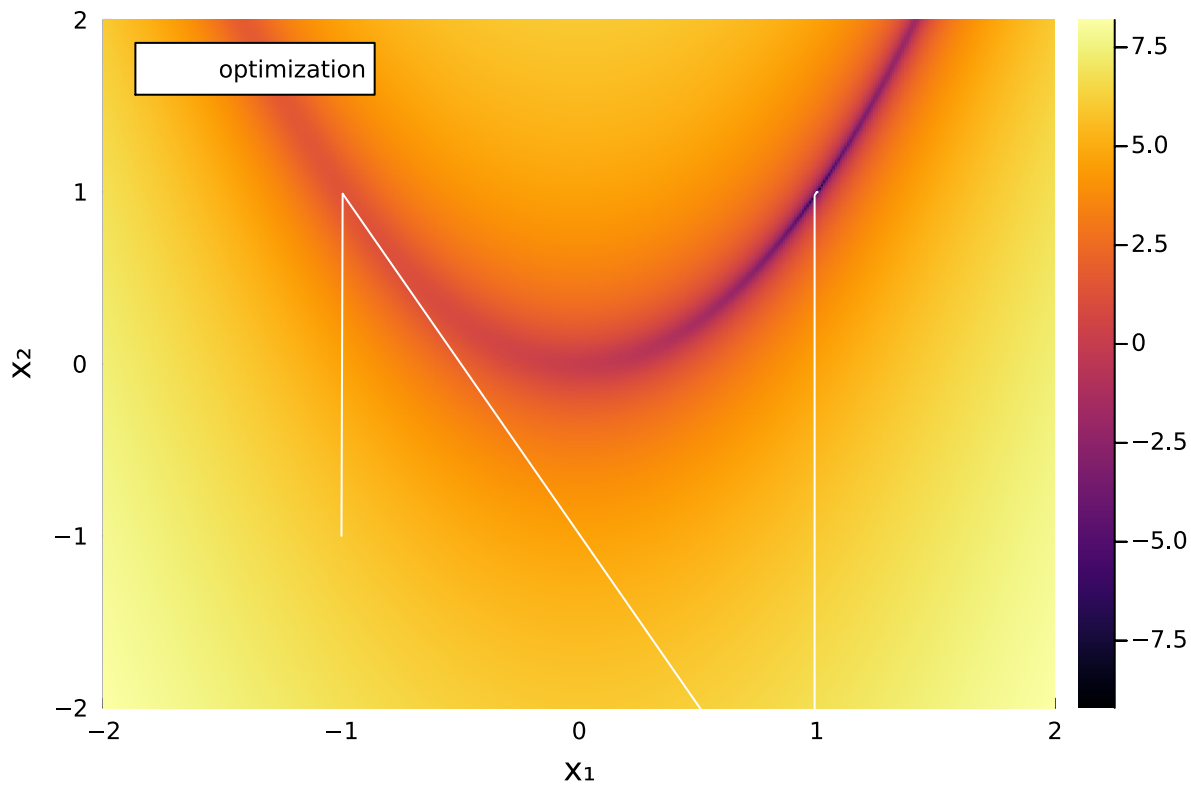
$$\begin{aligned}H_k p_k &= -g_k \\ x_{k+1} &= x_k + p_k\end{aligned}$$

where

- g_k is the gradient at time k along x_k .

newton_optimizer (generic function with 1 method)

```
1 function newton_optimizer(f, x; tol=1e-5)
2     k = 0
3     history = [x]
4     while k < 1000
5         k += 1
6         gk = ForwardDiff.gradient(f, x)
7         hk = ForwardDiff.hessian(f, x)
8         dx = -hk \ gk
9         x += dx
10        push!(history, x)
11        sum(abs2, dx) < tol && break
12    end
13    return history
14 end
```



```
1 let
2   x0 = [-1, -1.0]
3   history = newton_optimizer(rosenbrock, x0; tol=1e-5)
4   @info "number iterations = $(length(history)), got $(rosenbrock(history[end]))"
5
6   # plot
7   show_history(history)
8 end
```

number iterations = 6, got 3.4781872520856105e-23

The drawback of Newton's method is, the Hessian is very expensive to compute! While gradients can be computed with the automatic differentiation method with constant overhead. The Hessian requires $O(n)$ times more resources, where n is the number of parameters.

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm

The BFGS method is a popular numerical optimization algorithm used to solve unconstrained optimization problems. It is an iterative method that seeks to find the minimum of a function by iteratively updating an estimate of the inverse Hessian matrix of the function.

The BFGS method belongs to a class of quasi-Newton methods, which means that it approximates the Hessian matrix of the function using only first-order derivative information. The BFGS method updates the inverse Hessian matrix at each iteration using information from the current and previous iterations. This allows it to converge quickly to the minimum of the function.

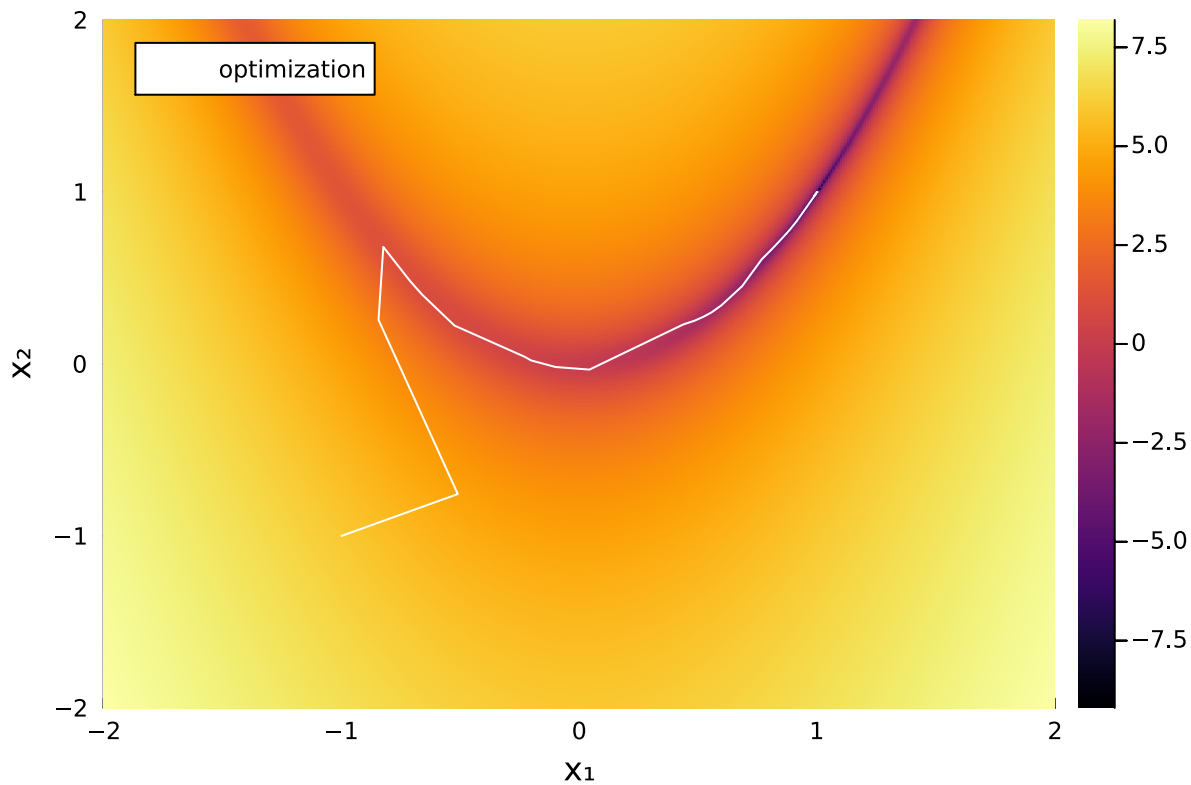
The BFGS method is widely used in many areas of science and engineering, including machine learning, finance, and physics. It is particularly well-suited to problems where the Hessian matrix is too large to compute directly, as it only requires first-order derivative information.

$$\begin{aligned}
 B_k p_k &= -g_k && // \text{Newton method like update rule} \\
 \alpha_k &= \operatorname{argmin} f(x + \alpha p_k) && // \text{using line search} \\
 s_k &= \alpha_k p_k \\
 x_{k+1} &= x_k + s_k \\
 y_k &= g_{k+1} - g_k \\
 B_{k+1} &= B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k^T}{s_k^T B_k s_k}
 \end{aligned}$$

where

- B_k is an approximation of the Hessian matrix, which is initialized to identity.
- g_k is the gradient at time k along x_k .

We can show $B_{k+1} s_k = y_k$ (secant equation) is satisfied.



```

1 let
2     # Set the initial guess
3     x0 = [-1.0, -1.0]
4     # Set the optimization options
5     options = Optim.Options(iterations = 1000, store_trace=true, extended_trace=true)
6     # Optimize the Rosenbrock function using the simplex method
7     result = optimize(rosenbrock, x->ForwardDiff.gradient(rosenbrock, x), x0, BFGS(),
8                       options, inplace=false)
9     # Print the optimization result
10    @info result
11    show_history([t.metadata["x"] for t in result.trace])
12 end

```

* Status: success

* Candidate solution

Final objective value: 3.405044e-28

* Found with

Algorithm: BFGS

* Convergence measures

$ x - x' $	= 4.05e-09	≠ 0.0e+00
$ x - x' / x' $	= 4.05e-09	≠ 0.0e+00
$ f(x) - f(x') $	= 4.60e-18	≠ 0.0e+00
$ f(x) - f(x') / f(x') $	= 1.35e+10	≠ 0.0e+00
$ g(x) $	= 7.01e-13	≤ 1.0e-08

* Work counters

Seconds run: 0 (vs limit Inf)
 Iterations: 30
 f(x) calls: 84
 ∇f(x) calls: 84

Mathematical optimization

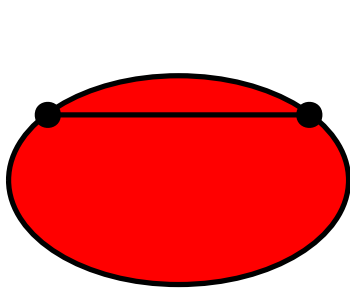
Convex optimization

A set $S \subseteq \mathbb{R}^n$ is convex if it contains the line segment between any two of its points, i.e.,

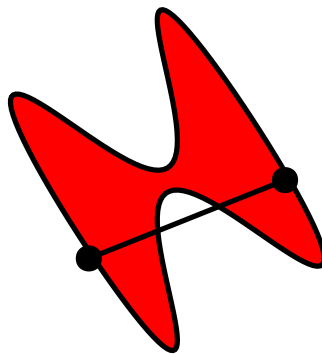
$$\{\alpha \mathbf{x} + (1 - \alpha) \mathbf{y} : 0 \leq \alpha \leq 1\} \subseteq S$$

for all $\mathbf{x}, \mathbf{y} \in S$.

```
1 using Luxor
```



convex set

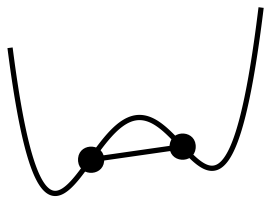


nonconvex set

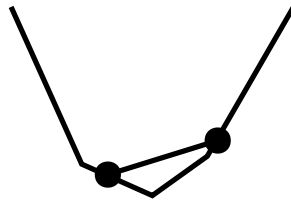
A function $f : S \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$ is convex on a convex set S if its graph along any line segment in S lies on or below the chord connecting the function values at the endpoints of the segment, i.e., if

$$f(\alpha \mathbf{x} + (1 - \alpha) \mathbf{y}) \leq \alpha f(\mathbf{x}) + (1 - \alpha) f(\mathbf{y})$$

for all $\alpha \in [0, 1]$ and all $\mathbf{x}, \mathbf{y} \in S$.



nonconvex



convex



strictly convex

Any local minimum of a convex function f on a convex set $S \subseteq \mathbb{R}^n$ is a global minimum of f on S .

Linear programming

Linear programs are problems that can be expressed in canonical form as

Find a vector	\mathbf{x}
that maximizes	$\mathbf{c}^T \mathbf{x}$
subject to	$A\mathbf{x} \leq \mathbf{b}$
and	$\mathbf{x} \geq \mathbf{0}$.

Here the components of \mathbf{x} are the variables to be determined, \mathbf{c} and \mathbf{b} are given vectors (with \mathbf{c}^T indicating that the coefficients of \mathbf{c} are used as a single-row matrix for the purpose of forming the matrix product), and A is a given matrix.

Example

<https://jump.dev/JuMP.jl/stable/tutorials/linear/diet/>

[JuMP.jl documentation](#) also contains mathematical models such as **semidefinite programming** and **integer programming**.

Assignments

1. Show the following graph $G = (V, E)$ has a unit-disk embedding.

```
V = 1, 2, ..., 10
E = [(1, 2), (1, 3),
      (2, 3), (2, 4), (2, 5), (2, 6),
      (3, 5), (3, 6), (3, 7),
      (4, 5), (4, 8),
      (5, 6), (5, 8), (5, 9),
      (6, 7), (6, 8), (6, 9),
      (7, 9), (8, 9), (8, 10), (9, 10)]
```

So what is uni-disk embedding of a graph? Ask Chat-GPT with the following question

What is a unit-disk embedding of a graph?

Hint:

To solve this issue, you can utilize an optimizer. Here's how:

1. Begin by assigning each vertex with a coordinate. You can represent the locations of all vertices as a $2 \times n$ matrix, denoted as \mathbf{x} , where each column represents a coordinate of vertices in a two-dimensional space.
2. Construct a loss function, denoted as $f(\mathbf{x})$, that returns a positive value as punishment if any connected vertex pair (v, w) has a distance $\text{dist}(\mathbf{x}_v, \mathbf{x}_w) > 1$ (the unit distance), or if any disconnected vertex pair has a distance smaller than 1. If all unit-disk constraints are satisfied, the function returns 0.
3. Use an optimizer to optimize the loss function $f(\mathbf{x})$. If the loss can be reduced to 0, then the corresponding \mathbf{x} represents a unit-disk embedding. If not, you may need to try multiple times to ensure that your optimizer does not trap you into a local minimum.

