WIKIPEDIA

Gradient descent

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient (or approximate gradient) of the function at the current point. If, instead, one takes steps proportional to the *positive* of the gradient, one approaches a local maximum of that function; the procedure is then known as **gradient ascent**.

Gradient descent is also known as **steepest descent**. However, gradient descent should not be confused with the method of steepest descent for approximating integrals.

Contents

Description

Gradient descent is based on the observation that if the multi-variable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases *fastest* if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, $-\nabla F(\mathbf{a})$. It follows that, if

$$\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla F(\mathbf{a}_n)$$

for $\gamma \in \mathbb{R}_+$ small enough, then $F(\mathbf{a_n}) \geq F(\mathbf{a_{n+1}})$. In other words, the term $\gamma \nabla F(\mathbf{a})$ is subtracted from $\mathbf{a}$ because we want to move against the gradient, toward the minimum. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of F, and considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ such that

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n),\ n \geq 0.$$

We have a monotonic sequence

$$F(\mathbf{x}_0) \geq F(\mathbf{x}_1) \geq F(\mathbf{x}_2) \geq \cdots,$$

so hopefully the sequence $(\mathbf{x}_n)$ converges to the desired local minimum. Note that the value of the *step size* γ is allowed to change at every iteration. With certain assumptions on the function F (for example, F convex and ∇F Lipschitz) and particular choices of γ (e.g., chosen either via a line search that satisfies the Wolfe conditions or the Barzilai-Borwein[1] method shown as following),

$$\gamma_n = \frac{\left|(\mathbf{x}_n - \mathbf{x}_{n-1})^T \left[\nabla F(\mathbf{x}_n) - \nabla F(\mathbf{x}_{n-1})\right]\right|}{\|\nabla F(\mathbf{x}_n) - \nabla F(\mathbf{x}_{n-1})\|^2}$$

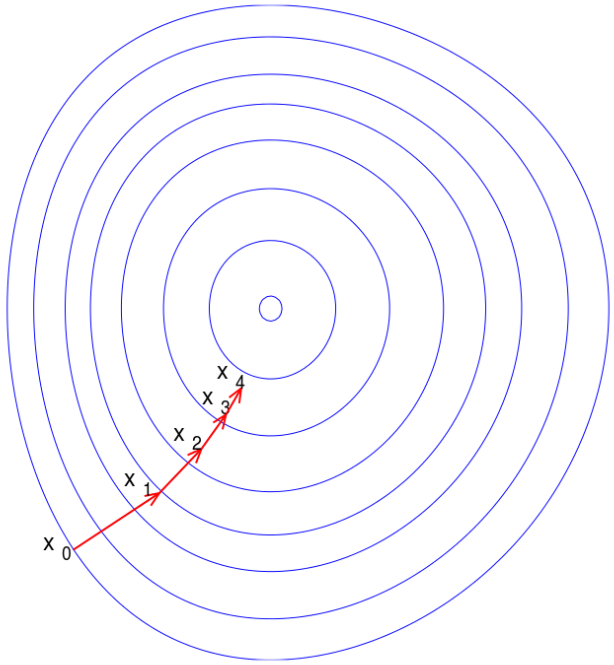


Illustration of gradient descent on a series of level sets.

convergence to a local minimum can be guaranteed. When the function F is convex, all local minima are also global minima, so in this case gradient descent can converge to the global solution.

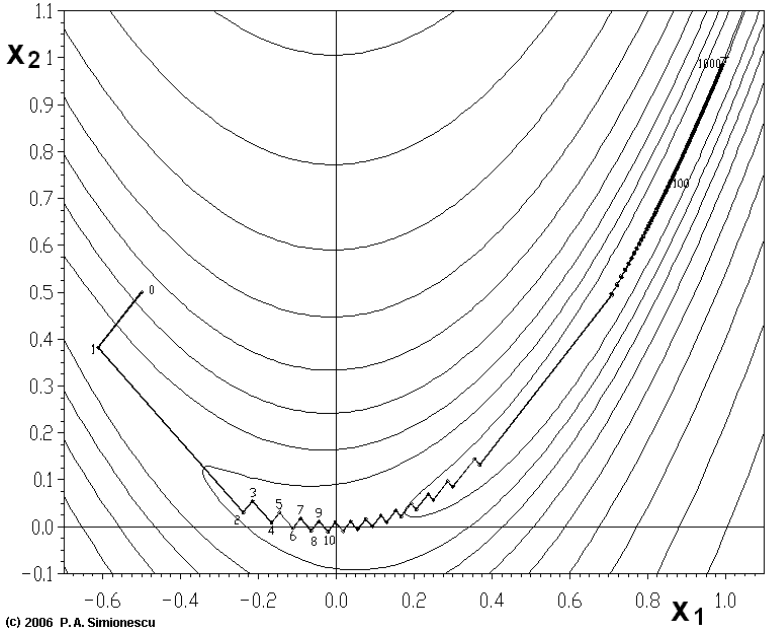
This process is illustrated in the adjacent picture. Here F is assumed to be defined on the plane, and that its graph has a bowl shape. The blue curves are the contour lines, that is, the regions on which the value of F is constant. A red arrow originating at a point shows the direction of the negative gradient at that point. Note that the (negative) gradient at a point is orthogonal to the contour line going through that point. We see that gradient *descent* leads us to the bottom of the bowl, that is, to the point where the value of the function F is minimal.

Examples

Gradient descent has problems with pathological functions such as the Rosenbrock function shown here.

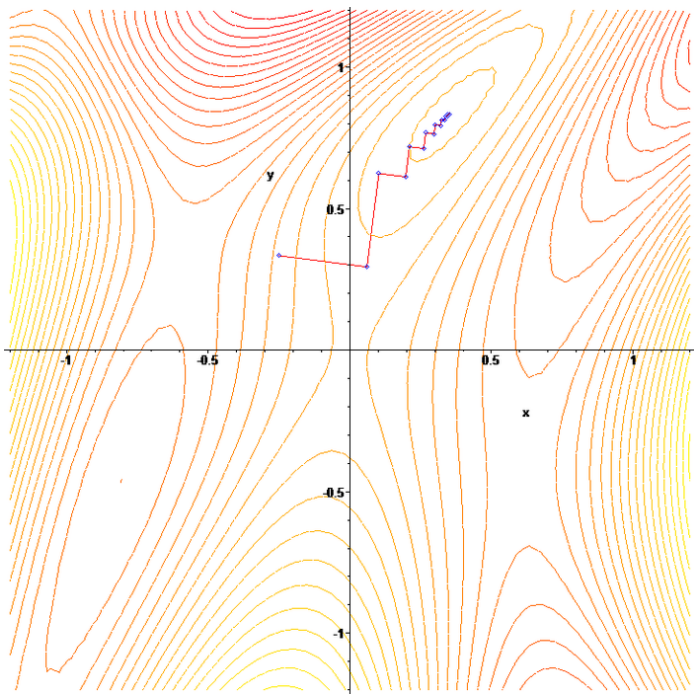
$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - {x_1}^2)^2.$$

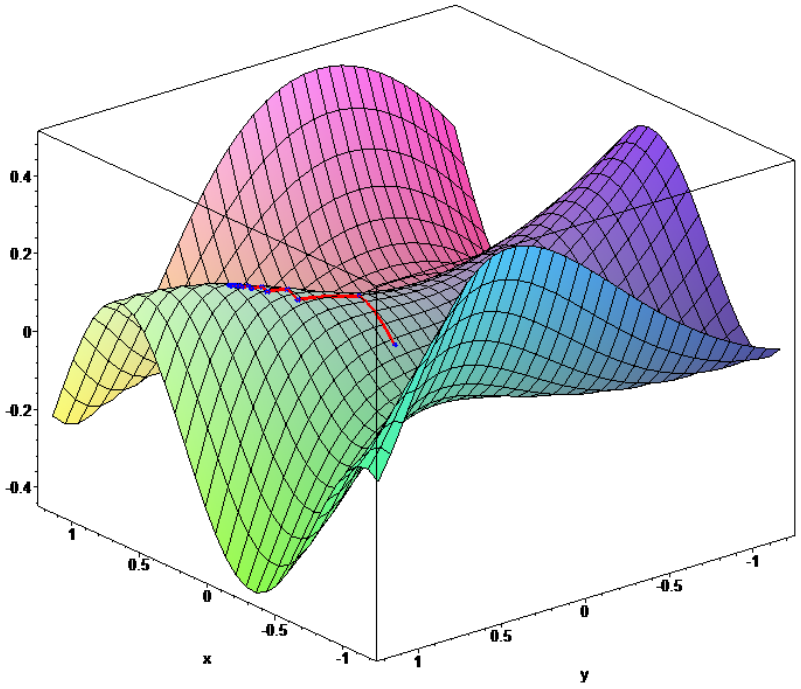
The Rosenbrock function has a narrow curved valley which contains the minimum. The bottom of the valley is very flat. Because of the curved flat valley the optimization is zigzagging slowly with small step sizes towards the minimum.



The zigzagging nature of the method is also evident below, where the gradient descent method is applied to

$$F(x, y) = \sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2 + 3\right)\cos(2x + 1 - e^y).$$





Limitations

For some of the above examples, gradient descent is relatively slow close to the minimum: technically, its asymptotic rate of convergence is inferior to many other methods. For poorly conditioned convex problems, gradient descent increasingly 'zigzags' as the gradients point nearly orthogonally to the shortest direction to a minimum point. For more details, see the § Comments below.

For non-differentiable functions, gradient methods are ill-defined. For locally Lipschitz problems and especially for convex minimization problems, bundle methods of descent are well-defined. Non-descent methods, like subgradient projection methods, may also be used.[2] These methods are typically slower than gradient descent. Another alternative for non-differentiable functions is to "smooth" the function, or bound the function by a smooth function. In this approach, the smooth problem is solved in the hope that the answer is close to the answer for the non-smooth problem (occasionally, this can be made rigorous).

Solution of a linear system

Gradient descent can be used to solve a system of linear equations, reformulated as a quadratic minimization problem, e.g., using linear least squares. The solution of

$$A\mathbf{x} - \mathbf{b} = 0$$

in the sense of linear least squares is defined as minimizing the function

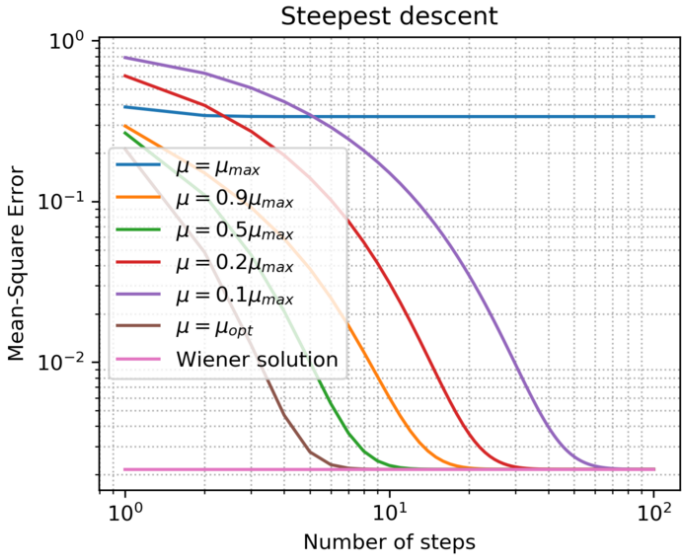
$$F(\mathbf{x}) = \|A\mathbf{x} - \mathbf{b}\|^2.$$

In traditional linear least squares for real A and $\mathbf{b}$ the Euclidean norm is used, in which case

$$\nabla F(\mathbf{x}) = 2A^T(A\mathbf{x} - \mathbf{b}).$$

In this case, the line search minimization, finding the locally optimal step size γ on every iteration, can be performed analytically, and explicit formulas for the locally optimal γ are known.[4]

For solving linear equations, this algorithm is rarely used, with the conjugate gradient method being one of the most popular alternatives. The speed of convergence of gradient descent depends on the ratio of the maximum to minimum eigenvalues of A^TA, while the speed of convergence of conjugate gradients has a more complex dependence on the eigenvalues, and can benefit from preconditioning. Gradient descent also benefits from preconditioning, but this is not done as commonly.



The Steepest-Descent Algorithm Applied to the Wiener filter[3].

Solution of a non-linear system

Gradient descent can also be used to solve a system of nonlinear equations. Below is an example that shows how to use the gradient descent to solve for three unknown variables, x_1, x_2, and x_3. This example shows one iteration of the gradient descent.

Consider the nonlinear system of equations

$$\begin{cases} 3x_1 - \cos(x_2x_3) - \frac{3}{2} = 0 \\ 4x_1^2 - 625x_2^2 + 2x_2 - 1 = 0 \\ \exp(-x_1x_2) + 20x_3 + \frac{10\pi - 3}{3} = 0 \end{cases}$$

Let us introduce the associated function

$$G(\mathbf{x}) = \begin{bmatrix} 3x_1 - \cos(x_2x_3) - \frac{3}{2} \\ 4x_1^2 - 625x_2^2 + 2x_2 - 1 \\ \exp(-x_1x_2) + 20x_3 + \frac{10\pi - 3}{3} \end{bmatrix},$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

One might now define the objective function

$$F(\mathbf{x}) = \frac{1}{2} G^{\mathrm{T}}(\mathbf{x}) G(\mathbf{x}) = \frac{1}{2} \left[\left(3x_1 - \cos(x_2 x_3) - \frac{3}{2} \right)^2 + \left(4x_1^2 - 625x_2^2 + 2x_2 - 1 \right)^2 + \left(\exp(-x_1 x_2) + 20x_3 + \frac{10\pi - 3}{3} \right)^2 \right],$$

which we will attempt to minimize. As an initial guess, let us use

$$\mathbf{x}^{(0)} = \mathbf{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

We know that

$$\mathbf{x}^{(1)} = \mathbf{0} - \gamma_0 \nabla F(\mathbf{0}) = \mathbf{0} - \gamma_0 J_G(\mathbf{0})^{\mathrm{T}} G(\mathbf{0}),$$

where the Jacobian matrix J_G is given by

$$J_G(\mathbf{x}) = \begin{bmatrix} 3 & \sin(x_2 x_3) x_3 & \sin(x_2 x_3) x_2 \\ 8x_1 & -1250x_2 + 2 & 0 \\ -x_2 \exp\left(-x_1 x_2\right) & -x_1 \exp(-x_1 x_2) & 20 \end{bmatrix}.$$

We calculate:

$$J_G(\mathbf{0}) = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 20 \end{bmatrix}, \qquad G(\mathbf{0}) = \begin{bmatrix} -2.5 \\ -1 \\ 10.472 \end{bmatrix}.$$

Thus

$$\mathbf{x}^{(1)} = \mathbf{0} - \gamma_0 \begin{bmatrix} -7.5 \\ -2 \\ 209.44 \end{bmatrix},$$

and

$$F(\mathbf{0}) = 0.5\left((-2.5)^2 + (-1)^2 + (10.472)^2\right) = 58.456.$$

Now, a suitable γ_0 must be found such that

$$F\left(\mathbf{x}^{(1)}\right) \leq F\left(\mathbf{x}^{(0)}\right) = F(\mathbf{0}).$$

This can be done with any of a variety of line search algorithms. One might also simply guess $\gamma_0 = 0.001$, which gives

$$\mathbf{x}^{(1)} = \begin{bmatrix} 0.0075 \\ 0.002 \\ -0.20944 \end{bmatrix}.$$

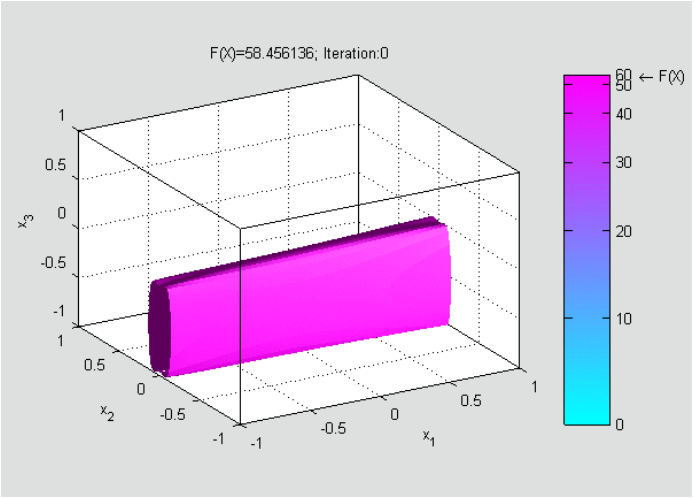
Evaluating the objective function at this value, yields

$$F\left(\mathbf{x}^{(1)}\right) = 0.5\left((-2.48)^2 + (-1.00)^2 + (6.28)^2\right) = 23.306.$$

The decrease from $F(\mathbf{0}) = 58.456$ to the next step's value of

$$F\left(\mathbf{x}^{(1)}\right) = 23.306$$

is a sizable decrease in the objective function. Further steps would reduce its value further, until an approximate solution to the system was found.



An animation showing the first 83 iterations of gradient descent applied to this example. Surfaces are isosurfaces of $F(\mathbf{x}^{(n)})$ at current guess $\mathbf{x}^{(n)}$, and arrows show the direction of descent. Due to a small and constant step size, the convergence is slow.

Comments

Gradient descent works in spaces of any number of dimensions, even in infinite-dimensional ones. In the latter case, the search space is typically a function space, and one calculates the Fréchet derivative of the functional to be minimized to determine the descent direction.[5]

The gradient descent can take many iterations to compute a local minimum with a required accuracy, if the curvature in different directions is very different for the given function. For such functions, preconditioning, which changes the geometry of the space to shape the function level sets like concentric circles, cures the slow convergence. Constructing and applying preconditioning can be computationally expensive, however.

The gradient descent can be combined with a line search, finding the locally optimal step size γ on every iteration. Performing the line search can be time-consuming. Conversely, using a fixed small γ can yield poor convergence.

Methods based on Newton's method and inversion of the Hessian using conjugate gradient techniques can be better alternatives.[6][7] Generally, such methods converge in fewer iterations, but the cost of each iteration is higher. An example is the BFGS method which consists in calculating on every step a matrix by which the gradient vector is multiplied to go into a "better" direction, combined with a more sophisticated line search algorithm, to find the "best" value of γ. For extremely large problems, where the computer-memory issues dominate, a limited-memory method such as L-BFGS should be used instead of BFGS or the steepest descent.

Gradient descent can be viewed as applying Euler's method for solving ordinary differential equations $x'(t) = -\nabla f(x(t))$ to a gradient flow.

Computational examples

The following code examples apply the gradient descent algorithm to find the minimum of the function $f(x) = x^4 - 3x^3 + 2$, with derivative $f'(x) = 4x^3 - 9x^2$.

Solving for $4x^3 - 9x^2 = 0$ and evaluation of the second derivative at the solutions shows the function has a plateau point at 0 and a global minimum at $x = \frac{9}{4}$.

Python

Here is an implementation in the Python programming language:

```
next_x = 6  # We start the search at x=6
gamma = 0.01  # Step size multiplier
precision = 0.00001  # Desired precision of result
max_iters = 10000  # Maximum number of iterations

# Derivative function
def df(x):
    return 4 * x**3 - 9 * x**2

for _i in range(max_iters):
    current_x = next_x
    next_x = current_x - gamma * df(current_x)

    step = next_x - current_x
    if abs(step) <= precision:
        break

print("Minimum at {}".format(next_x))

# The output for the above will be something like
# "Minimum at 2.2499646074278457"
```

Scala

Here is an implementation in the Scala programming language

```
import math._

val curX = 6
val gamma = 0.01
val precision = 0.00001
val previousStepSize = 1 / precision

def df(x: Double): Double = 4 * pow(x, 3) - 9 * pow(x, 2)

def gradientDescent(precision: Double, previousStepSize: Double, curX: Double): Double = {
    if (previousStepSize > precision) {
         val newX = curX + -gamma * df(curX)
         println(curX)
         gradientDescent(precision, abs(newX - curX), newX)
    } else curX
}

val ans = gradientDescent(precision, previousStepSize, curX)
println(s"The local minimum occurs at $ans")
```

Clojure

The following is a Clojure implementation:

```
(defn df [x] (- (* 4 x x x) (* 9 x x)))

(defn gradient-descent [thresh iters gamma x df]
  (loop [prev-x x
         cur-x (- x (* gamma (df x)))
         iters iters]
    (if (or (< (Math/abs (- cur-x prev-x)) thresh)
            (zero? iters))
      cur-x
      (recur cur-x
             (- cur-x (* gamma (df cur-x)))
             (dec iters)))))

(prn "The minimum is found at: " (gradient-descent 0.00001 10000 0.01 6 df))

;; This will output: "The minimum is found at: " 2.2499646074278457
```

one could skip the precision check and then use a more general functional programming pattern, using reduce:

```
(defn gradient-descent [iters gamma x df]
  (reduce (fn [x _]
            (- x (* gamma (df x))))
          x
          (range iters)))

(prn "The minimum is found at: " (gradient-descent 10000 0.01 6 df))

;; This will output: "The minimum is found at: " 2.249999999999999
```

The last pattern can also be achieved through infinite lazy sequences, like so:

```
(defn gradient-descent [n-iters gamma x df]
  (nth (iterate (fn [x] (- x (* gamma (df x)))) x)
       n-iters))

(prn "The minimum is found at: " (gradient-descent 10000 0.01 6 df))

;; This will output: "The minimum is found at: " 2.249999999999999
```

Java

Same implementation, to run in Java with JShell (Java 9 minimum): `jshell <scriptfile>`

```
import java.util.function.Function;
import static java.lang.Math.pow;
import static java.lang.System.out;

double gamma = 0.01;
double precision = 0.00001;

Function<Double,Double> df = x ->  4 * pow(x, 3) - 9 * pow(x, 2);

double gradientDescent(Function<Double,Double> f) {

    double curX = 6.0;
    double previousStepSize = 1.0;

    while (previousStepSize > precision) {
        double prevX = curX;
        curX -= gamma * f.apply(prevX);
        previousStepSize = abs(curX - prevX);
    }
    return curX;
}

double res = gradientDescent(df);
out.printf("The local minimum occurs at %f", res);
```

Julia

Here is the same algorithm implemented in the Julia programming language.

```
cur_x = 6.0
gamma = 0.01
precision = 0.00001
previous_step_size = 1.0

df(x) = 4x^3 - 9x^2

while previous_step_size > precision
    prev_x = cur_x
    cur_x -= gamma * df(prev_x)
    previous_step_size = abs(cur_x - prev_x)
end

println("The local minimum occurs at $cur_x");
```

C

Here is the same algorithm implemented in the C programming language.

```
#include <stdio.h>
#include <stdlib.h>

double dfx(double x) {
    return 4*(x*x*x) - 9*(x*x);
}

double abs_val(double x) {
    return x > 0 ? x : -x;
}

double gradient_descent(double dx, double error, double gamma, unsigned int max_iters) {
    double c_error = error + 1;
    unsigned int iters = 0;
    double p_error;
    while(error < c_error && iters < max_iters) {
        p_error = dx;
        dx -= dfx(p_error) * gamma;
        c_error = abs_val(p_error-dx);
        printf("\nc_error %f\n", c_error);
        iters++;
    }
    return dx;
}

int main() {
    double dx = 6;
    double error = 1e-6;
```

```
    double gamma = 0.01;
    unsigned int max_iters = 10000;
    printf("The local minimum is: %f\n", gradient_descent(dx, error, gamma, max_iters));
    return 0;
}
```

The above piece of code has to be modified with regard to step size according to the system at hand and convergence can be made faster by using an adaptive step size. In the above case the step size is not adaptive. It stays at 0.01 in all the directions which can sometimes cause the method to fail by diverging from the minimum.

JavaScript (ES6)

Here is the JavaScript (ES6) version:

```
const convergeGradientDescent = (
gradientFunction, // the gradient Function
x0 = 0, // starting position
maxIters = 1000, // Maximum number of iterations
precision = 0.00001,  //  Desired precision of result
gamma = 0.01  // Step size multiplier
) => {
  let step = 2*precision
  let iter = 0
  let x = x0
  while (true){
    step = gamma * gradientFunction(x)
    x -= step
    iter++
    console.log(iter)
    if(iter > maxIters) throw Error("Exceeded maximum iterations")
    if(Math.abs(step) < precision) {
      console.log(`Minimum at: ${x}`)
      console.log(`After ${iter} iteration(s)`)
      return(x)
    }
  }
}

// TEST
// gradient function, df
const df = (x) => (4*x-9)*x*x
convergeGradientDescent(df,6)
//2.2499646074278457
//Minimum at: 2.2499646074278457
//after 70 iteration(s)
```

And here follows an example of gradient descent used to fit price against squareMeter on random values, written in JavaScript, using ECMAScript 6 features:

```
// adjust training set size

const M = 10;
```

```
// generate random training set

const DATA = [];

const getRandomIntFromInterval = (min, max) =>
  Math.floor(Math.random() * (max - min + 1) + min);

const createRandomPortlandHouse = () => ({
  squareMeter: getRandomIntFromInterval(0, 100),
  price: getRandomIntFromInterval(0, 100),
});

for (let i = 0; i < M; i++) {
  DATA.push(createRandomPortlandHouse());
}

const _x = DATA.map(date => date.squareMeter);
const _y = DATA.map(date => date.price);

// linear regression and gradient descent

const LEARNING_RATE = 0.0003;

let thetaOne = 0;
let thetaZero = 0;

const hypothesis = x => thetaZero + thetaOne * x;

const learn = (x, y, alpha) => {
  let thetaZeroSum = 0;
  let thetaOneSum = 0;

  for (let i = 0; i < M; i++) {
    thetaZeroSum += hypothesis(x[i]) - y[i];
    thetaOneSum += (hypothesis(x[i]) - y[i]) * x[i];
  }

  thetaZero = thetaZero - (alpha / M) * thetaZeroSum;
  thetaOne = thetaOne - (alpha / M) * thetaOneSum;
}

const MAX_ITER = 1500;
for (let i = 0; i < MAX_ITER; i++) {
  learn(_x, _y, LEARNING_RATE);
  console.log('thetaZero ', thetaZero, 'thetaOne', thetaOne);
}
```

MATLAB

The following MATLAB code demonstrates a concrete solution for solving the non-linear system of equations presented in the previous section:

$$\begin{cases} 3x_1 - \cos(x_2 x_3) - \frac{3}{2} & = 0 \\ 4x_1^2 - 625x_2^2 + 2x_2 - 1 & = 0 \\ \exp(-x_1 x_2) + 20x_3 + \frac{10\pi - 3}{3} & = 0. \end{cases}$$

```
% Multi-variate vector-valued function G(x)
G = @(x) [
    3*x(1) - cos(x(2)*x(3)) - 3/2           ;
    4*x(1)^2 - 625*x(2)^2 + 2*x(2) - 1      ;
    exp(-x(1)*x(2)) + 20*x(3) + (10*pi-3)/3];

% Jacobian of G
JG = @(x) [
    3,                   sin(x(2)*x(3))*x(3),   sin(x(2)*x(3))*x(2) ;
    8*x(1),              -1250*x(2)+2,          0                   ;
    -x(2)*exp(-x(1)*x(2)), -x(1)*exp(-x(1)*x(2)), 20                 ];

% Objective function F(x) to minimize in order to solve G(x)=0
F = @(x) 0.5 * sum(G(x).^2);

% Gradient of F (partial derivatives)
dF = @(x) JG(x).' * G(x);

% Parameters
GAMMA = 0.001;    % step size (learning rate)
MAX_ITER = 1000;  % maximum number of iterations
FUNC_TOL = 0.1;   % termination tolerance for F(x)

fvals = [];       % store F(x) values across iterations
progress = @(iter,x) fprintf('iter = %3d: x = %-32s, F(x) = %f\n', ...
    iter, mat2str(x,6), F(x));

% Iterate
iter = 1;         % iterations counter
x = [0; 0; 0];    % initial guess
fvals(iter) = F(x);
progress(iter, x);
while iter < MAX_ITER && fvals(end) > FUNC_TOL
    iter = iter + 1;
    x = x - GAMMA * dF(x);  % gradient descent
    fvals(iter) = F(x);     % evaluate objective function
    progress(iter, x);      % show progress
end

% Plot
plot(1:iter, fvals, 'LineWidth',2); grid on;
title('Objective Function'); xlabel('Iteration'); ylabel('F(x)');

% Evaluate final solution of system of equations G(x)=0
disp('G(x) = '); disp(G(x))

% Output:
%
% iter =   1: x = [0;0;0]                          , F(x) = 58.456136
```

```
% iter =   2: x = [0.0075;0.002;-0.20944]         , F(x) = 23.306394
% iter =   3: x = [0.015005;0.0015482;-0.335103]  , F(x) = 10.617030
% ...
% iter = 187: x = [0.683335;0.0388258;-0.52231]   , F(x) = 0.101161
% iter = 188: x = [0.684666;0.0389831;-0.522302]  , F(x) = 0.099372
%
% (converged in 188 iterations after exceeding termination tolerance for F(x))
```

R

The following R programming language code is an example of implementing gradient descent algorithm to find the minimum of the function $f(x) = x^4 - 3x^3 + 2$ in previous section. Note that we are looking for f's minimum by solving its derivative being equal to zero.

$$\nabla f(x) = 4x^3 - 9x^2 = 0$$

The x can be updated with the gradient descent method every iteration in the form of

$$x^{(k+1)} = x^{(k)} - \gamma \nabla f\left(x^{(k)}\right)$$

where k = 1, 2, ..., maximum iteration, and γ is the step size multiplier.

```
# set up a stepsize multiplier
gamma = 0.003

# set up a number of iterations
iter = 500

# define the objective function f(x) = x^4 - 3*x^3 + 2
objFun = function(x) return(x^4 - 3*x^3 + 2)

# define the gradient of f(x) = x^4 - 3*x^3 + 2
gradient = function(x) return((4*x^3) - (9*x^2))

# randomly initialize a value to x
set.seed(100)
x = floor(runif(1, 0, 10))

# create a vector to contain all xs for all steps
x.All = numeric(iter)

# gradient descent method to find the minimum
for(i in seq_len(iter)){
        x = x - gamma*gradient(x)
        x.All[i] = x
        print(x)
}

# print result and plot all xs for every iteration
```

```
print(paste("The minimum of f(x) is ", objFun(x), " at position x = ", x, sep = ""))
plot(x.All, type = "l")
```

Haskell

Here is an implementation in the Haskell programming language:

```
df :: Double -> Double
df x = 4 * x ** 3 - 9 * x ** 2

gradientDescent :: Double -> Double -> Double -> Double -> IO Double
gradientDescent gamma precision previousStepSize curX =
  if previousStepSize > precision
    then do
      let newX = curX - (gamma * df curX)
      print curX
      gradientDescent gamma precision (abs (newX - curX)) newX
    else return curX

runGradientDescent :: IO Double
runGradientDescent = gradientDescent gamma precision previousStepSize curX
 where
  curX             = 6
  gamma            = 0.01
  precision        = 0.0001
  previousStepSize = 1 / precision
```

Extensions

Gradient descent can be extended to handle constraints by including a projection onto the set of constraints. This method is only feasible when the projection is efficiently computable on a computer. Under suitable assumptions, this method converges. This method is a specific case of the forward-backward algorithm for monotone inclusions (which includes convex programming and variational inequalities).[8]

Fast gradient methods

Another extension of gradient descent is due to Yurii Nesterov from 1983,[9] and has been subsequently generalized. He provides a simple modification of the algorithm that enables faster convergence for convex problems. For unconstrained smooth problems the method is called the Fast Gradient Method (FGM) or the Accelerated Gradient Method (AGM). Specifically, if the differentiable function F is convex and ∇F is Lipschitz, and it is not assumed that F is strongly convex, then the error in the objective value generated at each step k by the gradient descent method will be bounded by $\mathcal{O}\left(\frac{1}{k}\right)$. Using the Nesterov acceleration technique, the

error decreases at $\mathcal{O}\left(\frac{1}{k^2}\right)$.[10] It is known that the rate $\mathcal{O}\left(k^{-2}\right)$ for the decrease of the cost function is optimal for first-order optimization methods. Nevertheless, there is the opportunity to improve the algorithm by reducing the constant factor. The optimized gradient method (OGM)[11] reduces that constant by a factor of two and is an optimal first-order method for large-scale problems.[12]

For constrained or non-smooth problems, Nesterov's FGM is called the fast proximal gradient method (FPGM), an acceleration of the proximal gradient method.

The momentum method

Yet another extension, that reduces the risk of getting stuck in a local minimum, as well as speeds up the convergence considerably in cases where the process would otherwise zig-zag heavily, is the *momentum method*, which uses a momentum term in analogy to "the mass of Newtonian particles that move through a viscous medium in a conservative force field".[13] This method is often used as an extension to the backpropagation algorithms used to train artificial neural networks.[14][15]

An analogy for understanding gradient descent

The basic intuition behind gradient descent can be illustrated by a hypothetical scenario. A person is stuck in the mountains and is trying to get down (i.e. trying to find the minima). There is heavy fog such that visibility is extremely low. Therefore, the path down the mountain is not visible, so he must use local information to find the minima. He can use the method of gradient descent, which involves looking at the steepness of the hill at his current position, then proceeding in the direction with the steepest descent (i.e. downhill). If he was trying to find the top of the mountain (i.e. the maxima), then he would proceed in the direction steepest ascent (i.e. uphill). Using this method, he would eventually find his way down the mountain. However, assume also that the steepness of the hill is not immediately obvious with simple observation, but rather it requires a sophisticated instrument to measure, which the person happens to have at the moment. It takes quite some time to measure the steepness of the hill with the instrument, thus he should minimize his use of the instrument if he wanted to get down the mountain before sunset. The difficulty then is choosing the frequency at which he should measure the steepness of the hill so not to go off track.


Fog in the mountains

In this analogy, the person represents the algorithm, and the path taken down the mountain represents the sequence of parameter settings that the algorithm will explore. The steepness of the hill represents the slope of the error surface at that point. The instrument used to measure steepness is differentiation (the slope of the error surface can be calculated by taking the derivative of the squared error function at that point). The direction he chooses to travel in aligns with the gradient of the error surface at that point. The amount of time he travels before taking another measurement is the learning rate of the algorithm. See Backpropagation § Limitations for a discussion of the limitations of this type of "hill climbing" algorithm.

See also

- Conjugate gradient method
- Stochastic gradient descent
- Rprop
- Delta rule

- Wolfe conditions
- Preconditioning
- Broyden–Fletcher–Goldfarb–Shanno algorithm
- Davidon–Fletcher–Powell formula
- Nelder–Mead method
- Gauss–Newton algorithm
- Hill climbing

References

1. Barzilai, Jonathan; Borwein, Jonathan M. (1988). "Two-Point Step Size Gradient Methods". *IMA Journal of Numerical Analysis*. **8** (1): 141–148. doi:10.1093/imanum/8.1.141 (https://doi.org/10.1093%2Fimanum%2F8.1.141).
2. Kiwiel, Krzysztof C. (2001). "Convergence and efficiency of subgradient methods for quasiconvex minimization". *Mathematical Programming, Series A*. **90** (1). Berlin, Heidelberg: Springer. pp. 1–25. doi:10.1007/PL00011414 (https://doi.org/10.1007%2FPL00011414). ISSN 0025-5610 (https://www.worldcat.org/issn/0025-5610). MR 1819784 (https://www.ams.org/mathscinet-getitem?mr=1819784).
3. Haykin, Simon S. Adaptive filter theory. Pearson Education India, 2008. - p. 108-142, 217-242
4. Yuan, Ya-xiang (1999). "Step-sizes for the gradient method" (ftp://lsec.cc.ac.cn/pub/yyx/papers/p0504.pdf) (PDF). *AMS/IP Studies in Advanced Mathematics*. **42** (2): 785.
5. Akilov, G. P.; Kantorovich, L. V. (1982). *Functional Analysis* (2nd ed.). Pergamon Press. ISBN 0-08-023036-9.
6. Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P. (1992). *Numerical Recipes in C: The Art of Scientific Computing* (2nd ed.). New York: Cambridge University Press. ISBN 0-521-43108-5.
7. Strutz, T. (2016). *Data Fitting and Uncertainty: A Practical Introduction to Weighted Least Squares and Beyond* (2nd ed.). Springer Vieweg. ISBN 978-3-658-11455-8.
8. P. L. Combettes and J.-C. Pesquet, "Proximal splitting methods in signal processing" (https://arxiv.org/pdf/0912.3522v4.pdf), in: *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, (H. H. Bauschke, R. S. Burachik, P. L. Combettes, V. Elser, D. R. Luke, and H. Wolkowicz, Editors), pp. 185-212. Springer, New York, 2011.
9. Nesterov, Yu. (2004). *Introductory Lectures on Convex Optimization : A Basic Course*. Springer. ISBN 1-4020-7553-7.
10. Vandenberghe, Lieven (2019). "Fast Gradient Methods" (https://www.seas.ucla.edu/~vandenbe/236C/lectures/fgrad.pdf) (PDF). *Lecture notes for EE236C at UCLA*.
11. Kim, D.; Fessler, J. A. (2016). "Optimized First-order Methods for Smooth Convex Minimization". *Math. Prog.* **151** (1–2): 81–107. arXiv:1406.5468 (https://arxiv.org/abs/1406.5468). doi:10.1007/s10107-015-0949-3 (https://doi.org/10.1007%2Fs10107-015-0949-3).
12. Drori, Yoel (2017). "The Exact Information-based Complexity of Smooth Convex Minimization". *Journal of Complexity*. **39**: 1–16. arXiv:1606.01424 (https://arxiv.org/abs/1606.01424). doi:10.1016/j.jco.2016.11.001 (https://doi.org/10.1016%2Fj.jco.2016.11.001).
13. Qian, Ning (January 1999). "On the momentum term in gradient descent learning algorithms" (https://web.archive.org/web/20140508200053/http://brahms.cpmc.columbia.edu/publications/momentum.pdf) (PDF). *Neural Networks*. **12** (1): 145–151. CiteSeerX 10.1.1.57.5612 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.5612). doi:10.1016/S0893-6080(98)00116-6 (https://doi.org/10.1016%2FS0893-6080%2898%2900116-6). Archived from the original (http://brahms.cpmc.columbia.edu/publications/momentum.pdf) (PDF) on 8 May 2014. Retrieved 17 October 2014.
14. "Momentum and Learning Rate Adaptation" (http://www.willamette.edu/~gorr/classes/cs449/momrate.html). Willamette University. Retrieved 17 October 2014.
15. Geoffrey Hinton; Nitish Srivastava; Kevin Swersky. "The momentum method" (https://www.coursera.org/lecture/neural-networks/the-momentum-method-Oya9a). *Coursera*. Retrieved 2 October 2018. Part of a lecture series for the Coursera online course Neural Networks for Machine Learning (https://www.coursera.org/learn/neural-networks).

Further reading

- Boyd, Stephen; Vandenberghe, Lieven (2004). "Unconstrained Minimization" (https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf#page=471) (PDF). *Convex Optimization*. New York: Cambridge University Press. pp. 457–520. ISBN 0-521-83378-7.
- Snyman, Jan A.; Wilke, Daniel N. (2018). *Practical Mathematical Optimization - Basic Optimization Theory and Gradient-Based Algorithms*. Springer Optimization and Its Applications. **133** (2 ed.). Springer. ISBN 978-3-319-77585-2.. (Python module pmo.py (http://extras.springer.com/2018/978-3-319-77585-2))

External links

- Using gradient descent in C++, Boost, Ublas for linear regression (http://codingplayground.blogspot.it/2013/05/learning-linear-regression-with.html)
- Series of Khan Academy videos discusses gradient ascent (https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/gradient-and-directional-derivatives/v/gradient)
- Online book teaching gradient descent in deep neural network context (http://neuralnetworksanddeeplearning.com/chap1.html#learning_with_gradient_descent)
- "Gradient Descent, How Neural Networks Learn" (https://www.youtube.com/watch?v=IHZwWFHWa-w&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2). *3Blue1Brown*. October 16, 2017 – via YouTube.