Optimization Techniques in Finance

1. Unconstraint optimization problems

Andrew Lesniewski

Baruch College
New York

Fall 2019

Outline

Optimization problems

- Finance is all about *optimization*.
- Optimization helps to:
  - (i) maximize profits,
  - (ii) minimize costs,
  - (iii) control risk.
- Optimization problems facing financial professionals may include:
  - (i) Optimally allocate assets in a portfolio.
  - (ii) Build and interest rate curve based on benchmark rate observations.
  - (iii) Fit an implied volatility model to option market data.
  - (iv) Fit a time series model into observed data.
  - (v) Unwind a large position in a way that minimizes transaction costs.
  - (vi) Design an optimal hedging strategy of a portfolio of assets.
  - (vii) Optimally fit a predictive model into observed data.
  - (viii) Optimally exercise an American option.

Optimization problems

- Optimization is part of *applied mathematics*.
- That means that optimization problems are formulated in terms of properties of suitably defined functions.
- Depending on the context, solving an optimization problem draws on different disciplines of mathematics.
- A very limited number of optimization problems can be solved in “closed form”, i.e. a ready to use explicit formula. Typically, solving an optimization problem requires designing an *algorithm* and implementing it in *computer code*.
- Not all algorithms are equal: we strongly prefer algorithms with good performance characteristics.

Optimization problems

- Typically, *optimization problems* are formulated in terms of an *objective function* $f(x)$, $x \in \Omega \subset \mathbb{R}^n$ (depending on context, also known as a *loss function*, a *cost function*, etc.). The variables x are the *parameters* of the problem, and n is the *dimension* of the problem.
- The set $\Omega \subset \mathbb{R}^n$ defines constraints on the optimization problem, and its elements are called *feasible points*.
- More complex problems (such as multi-stage problems) may involve multiple objective functions.
- An optimization problem consists in finding a value x^* of the parameters such that $f(x) > f(x^*)$ for all values of x in a neighborhood of x^*. Such an x^* is called a *local minimizer* of $f(x)$.
- A *global minimizer* has the property that $f(x) > f(x^*)$ for *all* values of $x \in \Omega$.

Unconstrained optimization problems

- Finding global minimizers is typically much harder than finding local minimizer, as an objective function may have multiple local minima within Ω.
- In the case of a *convex* objective function, its local minimum is necessarily global, and there is no difference between the two problems. We will return to convex optimization later.
- A problem in which finding a maximum is required can be reduced to the case of a minimization problem by considering the objective function $\tilde{f}(x) = -f(x)$. In these lectures we will usually consider minimization problems.
- An optimization problem with $\Omega = \mathbb{R}^n$ are called *unconstrained*.

Necessary optimality conditions

- Let x^* be an unconstrained local minimizer of an objective function $f(x)$, $x \in \mathbb{R}^n$, and assume that $f(x)$ is continuously differentiable in a vicinity of x^*. Then:
  - (i) Its gradient at x^* is zero,
  $$\nabla f(x^*) = 0. \tag{1}$$
  - (ii) If, additionally, $f(x)$ is twice continuously differentiable near x^*, then the Hessian at x^* is positive semidefinite:
  $$\nabla^2 f(x^*) \geq 0. \tag{2}$$
- The proof is quite quick: Choose a direction $d \in \mathbb{R}^n$ and define a function $g(t) = f(x^* + td)$, $t \in \mathbb{R}$. Since x^* is a local minimizer,

$$\begin{aligned} 0 &\leq \lim_{t \to 0+} \frac{g(t) - g(0)}{t} \\ &= g'(0) \\ &= \nabla f(x^*)^{\mathrm{T}} d. \end{aligned}$$

Necessary optimality conditions

- Since d is arbitrary, this is possible only if $\nabla f(x^*) = 0$.
- If $f(x)$ is twice continuously differentiable, then from Taylor's theorem:

$$\begin{aligned} g(t) &= g(0) + g'(0)t + \frac{1}{2} g''(0)t^2 + o(t^2) \\ &= g(0) + \frac{1}{2} d^{\mathrm{T}} \nabla^2 f(x^*) d\, t^2 + o(t^2). \end{aligned}$$

- Therefore,

$$\begin{aligned} 0 &\leq \lim_{t \to 0} \frac{g(t) - g(0)}{t^2} \\ &= \frac{1}{2} d^{\mathrm{T}} \nabla^2 f(x^*) d + \lim_{t \to 0} \frac{o(t^2)}{t^2} \\ &= \frac{1}{2} d^{\mathrm{T}} \nabla^2 f(x^*) d, \end{aligned}$$

and we are done.

Sufficient optimality conditions

- Let the objective function $f(x)$, $x \in \mathbb{R}^n$, be twice continuously differentiable in an open set $O \subset \mathbb{R}^n$. Suppose that a point $x^* \in O$ satisfies the conditions

  (i)
  $$\nabla f(x^*) = 0. \tag{3}$$

  (ii) the Hessian at x^* is (strictly) positive definite:
  $$\nabla^2 f(x^*) > 0. \tag{4}$$

- Then x^* is a strict local minimizer of $f(x)$.

Example: quadratic objective function

- Let $A \in \mathrm{Mat}_n(\mathbb{R})$ be a positive definite matrix, $b \in \mathbb{R}^n$, and consider the objective function:
$$f(x) = \frac{1}{2} x^{\mathrm{T}} A x - x^{\mathrm{T}} b. \tag{5}$$
- The first order optimality condition reads
$$Ax^* = b,$$
i.e.
$$x^* = A^{-1} b. \tag{6}$$
- The Hessian
$$\nabla^2 f(x) = A \tag{7}$$
is positive definite (and constant). Therefore, (6) is a strict minimizer of (5).

Easy objective function: quadratic function

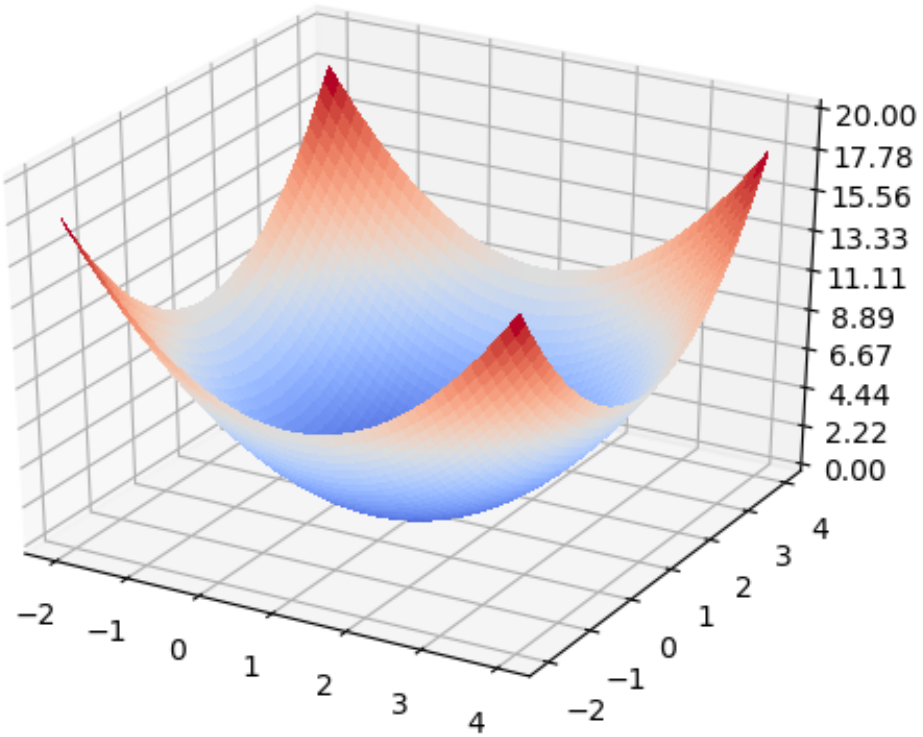
- An objective function that is easy to optimize is a quadratic function:

$$f(x, y) = (x - 1)^2 + (y - 1)^2. \tag{8}$$

- Inspection shows that it has a unique global minimum at $x = 1, y = 1$.
- This minimum is trivial to find analytically, and numerical algorithms usually have an easy time to find it.

Easy objective function: quadratic function

- The figure below shows a 3D graph of the quadratic function.



Hard objective function: Rosenbrock function

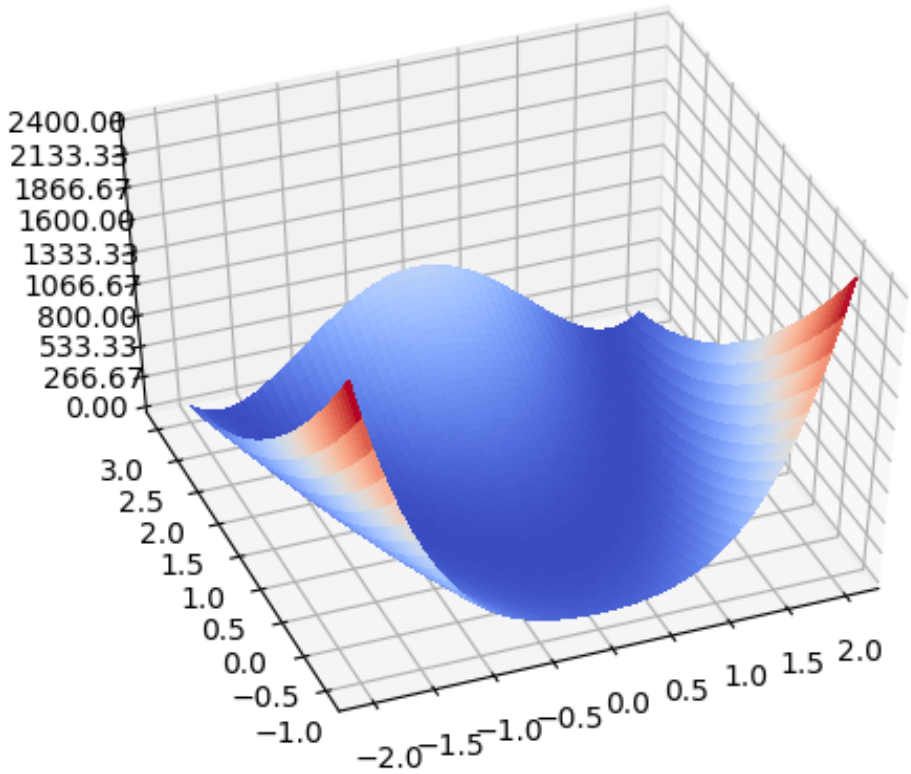
- A function, frequently used as a performance benchmark for testing optimization algorithms, is the *Rosenbrock function*:

$$f(x, y) = (x - 1)^2 + 100(y - x^2)^2. \tag{9}$$

- Inspection shows that it has a unique global minimum at $x = 1, y = 1$.
- This minimum, even though trivial to find analytically, lies inside a steep parabolic valley, and numerical algorithms struggle to find it.

Hard objective function: Rosenbrock function

- The figure below shows a 3D graph of the Rosenbrock function. Note the parabolic valley going through the point $(1, 1)$.



Sensitivity

- Suppose that we have a differentiable objective function that is parameterized by a set of parameters $\alpha \in \mathbb{R}^p$, i.e. the objective function is of the form $f(x, \alpha)$. Assume that for each α there is a local minimizer $x^*(\alpha)$, and that $x^*(\alpha)$ is a differentiable function of α.
- The first order optimality condition reads

$$\nabla_x f(x^*(\alpha), \alpha) = 0.$$

- Taking the gradient with respect to α and using the chain rule yields

$$\nabla_\alpha x^*(\alpha) \nabla_x^2 f(x^*(\alpha), \alpha) + \nabla_{x\alpha}^2 f(x^*(\alpha), \alpha) = 0,$$

where $\nabla_\alpha x(\alpha)$ is a $p \times n$ matrix of first derivatives $\partial x_i(\alpha)/\partial \alpha_k)$. Assuming invertibility of the Hessian, this yields

$$\nabla_\alpha x^*(\alpha) = -\nabla_{x\alpha}^2 f(x^*(\alpha), \alpha) \left(\nabla_x^2 f(x^*(\alpha), \alpha)\right)^{-1}. \tag{10}$$

Sensitivity

- This expresses the *sensitivity* of the minimizer to the parameter α.
- For example, if $f(x, \alpha) = f(x) - x^{\mathrm{T}}\alpha$, $\alpha \in \mathbb{R}^n$, then

$$\nabla_\alpha x^*(\alpha) = \left(\nabla_x^2 f(x^*(\alpha))\right)^{-1}. \tag{11}$$

- Sensitivity analysis will be a helpful tool in studying constraint optimization problems.

NLS problems

- In a *nonlinear least square (NLS) problem* the objective function is of the form:

$$f(x) = \frac{1}{2}\sum_{i=1}^{m}\phi_i(x)^2, \quad x \in \mathbb{R}^n, \tag{12}$$

with given functions $\phi_i(x)$, and $m \geq n$. The factor $\frac{1}{2}$ is introduced for convenience.

- Introducing the column vector $\phi(x) = (\phi_1(x), \ldots \phi_m(x))^{\mathrm{T}}$, we can conveniently write this objective function as

$$f(x) = \frac{1}{2}\phi(x)^{\mathrm{T}}\phi(x). \tag{13}$$

- NLS problems arise in *data fitting*: We are given a set of data points (observations) $(T_1, S_1), \ldots, (T_p, S_p)$ and a *data model* depending on a number of parameters $x_1, \ldots x_k$:

$$S = g(x, T). \tag{14}$$

The goal is to fit the data to the model.

- The model function $g(x, T)$ can be e.g. an explicitly defined function or a suitable spline function.

NLS problems

- Assume that

$$S_i = g(x, T_i) + \varepsilon_i, \tag{15}$$

where ε_i are residuals. One way to fit the model to the data is to minimize the sum of squares of the residuals, which leads to the following objective function:

$$f(x) = \frac{1}{2} \sum_{i=1}^{m} \left(g(x, T_i) - S_i\right)^2. \tag{16}$$

- In the simplest situation, x is two-dimensional, $x = (a, b) \in \mathbb{R}^2$, and the function $g(x, T)$ is linear:

$$\begin{aligned} S &= g(a, b, T) \\ &= a + bT. \end{aligned} \tag{17}$$

- This is the *linear* least square problem that can easily be solved in the closed form.

NLS problems

- The minimizer is given by the linear regression formulas:

$$b^* = \frac{\sum_i (S_i - \overline{S})(T_i - \overline{T})}{\sum_i (T_i - \overline{T})^2}, \qquad (18)$$
$$a^* = \overline{S} - b^* \, \overline{T},$$

  where $\overline{T} = \frac{1}{p} \sum_{i=1}^{p} T_i$, and $\overline{S} = \frac{1}{p} \sum_{i=1}^{p} S_i$.

- Except for the linear case, NLS problems cannot be solved in closed form. NLS problems tend to be hard (think about it: the Rosenbrock function can be viewed as an NLS problem), and require refined numerical algorithms.

NLS problems

- In finance, NLS problems arise when we face the task of interpolating a cross section of market data. This amounts to the problem of *curve building* in a variety of contexts. Here are a few important examples.
- Given a snapshot of benchmark swap rates of given tenors, build a smooth interpolating curve that allows us to compute the swap rate of any tenor.
- Given a snapshot of the US Treasuries, fit a smooth curve that allows to carry out rich-cheap analysis.
- Given a snapshot of benchmark credit spreads, build a smooth interpolating curve that allows us to compute the spread of any maturity.
- Given a snapshot of option prices of the same expiration (and underlying) but varying strikes, build an interpolation curve that allows us to predict volatility smile for any strike.

NLS problems

- We will now develop an number of algorithms for solving NLS problems. Each of these algorithms is *recursive*, meaning that it starts with an initial guess for x^* and continues until an exit criterion is met.
- Before we move on, notice that the gradient of the objective function (12) (or (13)) is given by

$$\begin{aligned} \nabla f(x) &= \sum_{i=1}^{m} \phi_i(x) \nabla \phi_i(x) \\ &= J(x)^{\mathrm{T}} \phi(x), \end{aligned} \tag{19}$$

  where $J(x)_{ij} = \nabla_j \phi_i(x)$ is the $m \times n$ Jacobian matrix.
- Likewise, the Hessian of $f(x)$ is given by

$$\begin{aligned} \nabla^2 f(x) &= \sum_{i=1}^{m} \left(\nabla \phi_i(x) \nabla \phi_i(x)^{\mathrm{T}} + \phi_i(x) \nabla^2 \phi_i(x) \right) \\ &= J(x)^{\mathrm{T}} J(x) + \sum_{i=1}^{m} \phi_i(x) \nabla^2 \phi_i(x). \end{aligned} \tag{20}$$

NLS problems

- As discussed earlier, finding a local minimizer x^* of $f(x)$ requires solving the equation
$$\nabla f(x^*) = 0. \tag{21}$$
- If x is close to x^*, we can use Taylor's expansion to approximate
$$\begin{aligned} 0 &= \nabla f(x^*) \\ &\approx \nabla f(x) + \nabla^2 f(x)(x^* - x). \end{aligned}$$
- Assuming that the matrix $\nabla^2 f(x)$ is nonsingular for all x, this suggest the following *Newton method* update rule:
  (i) Start with an initial guess x_0 for x^*.
  (ii) For $k = 1, 2 \ldots,$
$$x_{k+1} = x_k - \nabla^2 f(x_k)^{-1} \nabla f(x_k). \tag{22}$$
- This observation is the starting point of several powerful algorithms for solving the NLS problem.

- In some ways, the scheme above is very attractive: it can be proved that if x_0 is sufficiently close to the local minimizer x^*, then the sequence x_k converges rapidly to x^*, namely quadratically in the error $\|x_{k+1} - x_k\|$.
- Notice, however, some of the practical challenges that it faces:
  - (i) Finding an appropriate initial guess may be difficult.
  - (ii) Expressions (19) and (20) for the gradient and Hessian of $f(x)$, respectively, require computing the first and second derivatives of the functions $\phi_i(x)$. Preferably, these derivatives should be calculated in closed form. Typically, even in the case of moderate dimensionality n of the problem, this is infeasible. It is especially true for the second derivatives.
  - (iii) One may try to calculate the derivatives numerically, as finite differences. In the case of a moderate / large dimensionality of the problem, this is computationally prohibitively expensive, and the algorithm is inefficient.
  - (iv) There is no guarantee that the Hessian in (22) is invertible. Even if it is, there is a cost associated with this operation at each iteration step.

Gradient descent method

- The *gradient descent method* is the simplest approximation to (22), in which assume that the Hessian is a multiple of the identity matrix.
- The scheme can thus be formulated as
  (i) Start with an initial guess x_0 for x^*.
  (ii) For $k = 1, 2 \ldots,$
  $$x_{k+1} = x_k - \alpha \nabla f(x_k). \tag{23}$$
  Here the parameter $\alpha > 0$ is called the *step size* or *learning rate*.
- The value of α can be either determined by trial and error or by means of a *line search* algorithm. In the latter approach, in each step, we numerically search for a value α_k satisfying
  $$\alpha_k = \arg\min_{\alpha} f(x_k - \alpha \nabla f(x_k)). \tag{24}$$
- Gradient descent methods are easy to implement, but their performance tends to be mixed. Convergence (if any) is linear and slow, and it requires many iterations.
- Among the descent methods, the gradient descent method is locally the best, as it moves the next iteration in the direction of the steepest descent of $f(x)$.

Gauss-Newton method

- The *Gauss-Newton method* is based on the assumption that the function $\phi_i(x)$ can be locally well approximated by linear functions:

$$\phi(x+h) \approx \phi(x) + J(x)h. \tag{25}$$

- This implies that

$$\begin{aligned} f(x+h) &\approx f(x) + \nabla f(x)^{\mathrm{T}} h + \frac{1}{2} h^{\mathrm{T}} \nabla^2 f(x) h \\ &= f(x) + h^{\mathrm{T}} J(x)^{\mathrm{T}} \phi(x) + \frac{1}{2} h^{\mathrm{T}} J(x)^{\mathrm{T}} J(x) h, \end{aligned} \tag{26}$$

  where we have used expressions (19) and (20) for the gradient and Hessian of $f(x)$, respectively.
- Notice the absence of the second derivatives in (20) from the term quadratic in h. This is a consequence of the linear approximation assumption. This is a boon to the computational efficiency of algorithm.

Gauss-Newton method

- Effectively, we thus approximate the Hessian of $f(x)$ as follows:

$$\nabla^2 f(x) \approx J(x)^{\mathrm{T}} J(x). \tag{27}$$

- This approximation guarantees that the Hessian is always positive semidefinite. Also, it turns out that dropping the second derivatives does not, in practice, affect the convergence of the process.
- We now choose the optimal h by minimizing the quadratic function in h on the right hand side of (26). This yields the condition

$$J(x)^{\mathrm{T}} J(x) h = -J(x)^{\mathrm{T}} \phi(x). \tag{28}$$

- The function $f(x)$ descents along the direction of h, as (28) implies that

$$\begin{aligned} h^{\mathrm{T}} \nabla f(x) &= J(x)^{\mathrm{T}} \phi(x) h \\ &= -h^{\mathrm{T}} J(x)^{\mathrm{T}} J(x) h \\ &\leq 0. \end{aligned}$$

Gauss-Newton method

- The Gauss-Newton scheme can be thus formulate as follows. We start at x_0 and iterate:
$$x_{k+1} = x_k - \left(J(x_k)^{\mathrm{T}} J(x_k)\right)^{-1} J(x_k)^{\mathrm{T}} \phi(x_k). \tag{29}$$
- The key step of the method is solving equation (28), which amounts to inverting the matrix $J(x_k)^{\mathrm{T}} J(x_k)$.
- In case if $J(x)$ is poorly conditioned, this inverse may not exist.
- It may be beneficial to refine the method by introducing the learning rate α,
$$x_{k+1} = x_k - \alpha\left(J(x_k)^{\mathrm{T}} J(x_k)\right)^{-1} J(x_k)^{\mathrm{T}} \phi(x_k), \tag{30}$$
which can be either set *a priori* or determined at each step by conducting a line search. Under suitable technical assumptions, this method is guaranteed to converge.
- Unlike the Newton method, the Gauss-Newton method typically converges linearly in h.

Levenberg-Marquardt method

- The *Levenberg-Marquardt method* is a clever hybrid of the gradient descent and Gauss-Newton methods.
- Namely, we introduce a *damping parameter* $\mu > 0$, and determine the step size by solving the equation

$$\big(J(x)^{\mathrm{T}} J(x) + \mu I\big) h = -J(x)^{\mathrm{T}} \phi(x). \tag{31}$$

- Notice that the presence of μ has the following effects on the problem:
  - (i) The matrix $J(x)^{\mathrm{T}} J(x) + \mu I$ is strictly positive definite, and the existence of its inverse is guaranteed.
  - (ii) For small values of μ, the iteration step is similar to the iteration step in the Gauss-Newton method.
  - (iii) For large values of μ, the iteration step is similar to the iteration step in the gradient descent method.
- The value of μ will be updated throughout the iteration process, and it renders line search unnecessary.

Levenberg-Marquardt method

- In order to initialize μ, one usually chooses it be of the order of magnitude of maximum diagonal term of the matrix $J(x_0)^{\mathrm{T}} J(x_0)$,

$$\mu_0 = \beta \max_i \left(J(x_0)^{\mathrm{T}} J(x_0) \right)_{ii}, \tag{32}$$

  where $\beta \approx 10^{-3}$ is determined empirically.

- The update process is determined by the *gain ratio*, defined as the ratio of the projected change in $f(x)$ to the change in $f(x)$ projected by the quadratic approximation (26):

$$\rho = -\frac{f(x_k) - f(x_k + h)}{h^{\mathrm{T}} J(x_k)^{\mathrm{T}} \phi(x_k) + \frac{1}{2} h^{\mathrm{T}} J(x_k)^{\mathrm{T}} J(x_k) h}, \tag{33}$$

  which can also be expressed as

$$\rho = 2 \frac{f(x_k) - f(x_k + h)}{h^{\mathrm{T}} \left(\mu h - J(x_k)^{\mathrm{T}} \phi(x_k) \right)}. \tag{34}$$

Levenberg-Marquardt method

- Note that the denominator in (34) is always positive. Indeed, from (31) we have:

$$\begin{aligned} h^{\mathrm{T}}\left(\mu h - J(x_k)^{\mathrm{T}}\phi(x_k)\right) &= 2\mu h^{\mathrm{T}}h + h^{\mathrm{T}}J(x_k)^{\mathrm{T}}J(x_k)h \\ &> 0. \end{aligned}$$

- Large value of ρ indicates that the quadratic approximation to $f(x)$ is accurate. We should *decrease* the value of μ in order to make the process more Gauss-Newton like, and accelerate the convergence.
- On the other hand, a small value of ρ (also negative) indicates that the approximation is poor and we are on the wrong path. In this case, the value of μ should be *increased* to make the process more gradient descent like, and also decrease the step size.
- The updates of μ are done in terms of a scaling factor λ (frequently chosen as $\lambda = 2$): if the gain ratio is positive, the value of μ is reduced by a factor depending on ρ, otherwise it is scaled up according to $\mu \to \lambda\mu$.

Levenberg-Marquardt method

- An important element of a search algorithm is a set of *exit criteria*, which stop the algorithm once it has converged or it failed to converge.
  - (i) Ideally, the search should stop if $\nabla f(x^*) = J(x^*)^{\mathrm{T}}\phi(x^*) = 0$. We can implement it numerically by choosing a small $\varepsilon_1 > 0$, and require that all components of the vector $\nabla f(x_k)$ have absolute values less than ε_1. This is equivalent to stating that

$$\|J(x_k)^{\mathrm{T}}\phi(x_k)\|_\infty \leq \varepsilon_1. \tag{35}$$

    Here, the L^∞*-norm* $\|X\|_\infty$ of a vector X is defined as $\max_i |X_i|$.
  - (ii) Another criterion guarantees exit, if the changes in x_k are slow. We choose another small number ε_2 and verify if

$$\|x_{k+1} - x_k\| \leq \varepsilon_2(\|x_k\| + \varepsilon_2). \tag{36}$$

  - (iii) Lastly, in order to avoid running an infinite loop, one has to choose the maximum number of iterations k_{max}

Levenberg-Marquardt method

- Let us now summarize the Levenberg-Marquardt method in an algorithmic form:

choose x_0, initialize μ according to (32), and $\lambda = 2$
$done = (\|J(x_0)^{\mathrm{T}}\phi(x_0)\|_\infty \leq \varepsilon_1)$
while (*not done* and $k \leq k_{max}$)
  $h = -(J(x_k)^{\mathrm{T}}J(x_k) + \mu I)^{-1}J(x_k)\phi(x_k)$
  if $\|h\| \leq \varepsilon_2(\|x_k\| + \varepsilon_2)$
    $done = true$
  else
    calculate ρ according to (33)
    if $\rho > 0$
      $x_{k+1} = x_k + h$
      $done = (\|J(x_{k+1})^{\mathrm{T}}\phi(x_{k+1})\|_\infty \leq \varepsilon_1)$
      update $\mu \to \mu \max(1/3, 1 - (2\rho - 1)^3)$, and $\lambda = 2$
    else
      update $\mu \to \lambda\mu$, and $\lambda \to 2\lambda$

Levenberg-Marquardt method

- The Levenberg-Marquard method is implemented in `scipy`.

Data fitting without NLS

- Data fitting problems in finance often lead to optimization problems which are not of the NLS type.
- For example, in models requiring maximum likelihood estimation (MLE) from historical data, the objective function is typically not a sum squares of the residuals between the data and the model. Instead, it is the probability of the observed data given given the (unknown) parameters of the probability distribution.
- For example, many problems lead to the task of minimizing the Kullback-Leibler divergence between the observed frequencies p and a probability model $q(x)$,

$$KL(p||q(x)) = \sum_{i=1}^{m} p_i \log \frac{p_i}{q_i(x)} . \tag{37}$$

  This objective function clearly is not of the NLS form.
- Quasi-Newton methods have been designed to problems like this. The most popular among them is the *BFGS method* (for Broyden, Fletcher, Goldfarb, and Shanno)and its variations.

BFGS method

- Our goal is to find a local minimizer of an objective function $f(x)$, not necessary expressed as a sum of squares. We initialize the search with x_0 and iterate.
- At the $(k+1)$-th step of iteration we approximate the value of $f(x_{k+1})$ by an expression quadratic in the direction of the step from x_k. Namely, for x in the vicinity of x_k, we can write

$$f(x) \approx f(x_k) + \nabla f(x_k)^{\mathrm{T}} h + \frac{1}{2} h^{\mathrm{T}} B_k h, \tag{38}$$

  where B_k is a positive definite matrix.
- Naively, B_k should be the Hessian at x_k, but, as we know, this leads to serious performance issues, and the challenge is to choose B_k smartly.
- The minimizer of (38) is

$$h = -B_k^{-1} \nabla f(x_k), \tag{39}$$

  and x_{k+1} is given by

$$x_{k+1} = x_k + \alpha_k h. \tag{40}$$

BFGS method

- The value of the multiplier α_k is determined from the line search:

$$\alpha_k = \arg\min_{\alpha} f(x_k - \alpha \nabla f(x_k)). \tag{41}$$

- In the BFGS method, the matrix B_k is constructed iteratively as follows. Given B_k, we require that the updated value B_{k+1} satisfies the *secant equation*

$$B_{k+1}(x_{k+1} - x_k) = \nabla f(x_{k+1}) - \nabla f(x_k) \tag{42}$$

(remember B_k is supposed to mimic the Hessian!).

- This can be written as

$$B_{k+1} s_k = y_k, \tag{43}$$

where $s_k = x_{k+1} - x_k$ and $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$.

BFGS method

- Premultiplying (43) by s_k^{T}, we see that the following *curvature condition* has to be satisfied:

$$s_k^{\mathrm{T}} y_k > 0. \tag{44}$$

  If the curvature condition is satisfied, the secant equation (43) has a unique solution.
- For non-convex functions, the curvature condition is not always satisfied. In such a case, it has to be enforced by means of choosing an appropriate step size α_k.
- We will now assume that the updated B_{k+1} equals to B_k plus two positive definite rank 1 matrices:

$$B_{k+1} = B_k + auu^{\mathrm{T}} + bvv^{\mathrm{T}}, \tag{45}$$

  where $a, b \in \mathbb{R}$, and $u, v \in \mathbb{R}^n$.
- How do we choose the parameters above?

BFGS method

- This in itself is an optimization in which we minimize the Frobenius (i.e. L^2) norm between the symmetric matrices B_{k+1} and B_k under the constraint that the secant equation holds. The solution turns out to be

$$
\begin{aligned}
u &= y_k, \\
v &= B_k s_k \\
a &= \frac{1}{s_k^{\mathrm{T}} y_k}, \\
b &= -\frac{1}{s_k^{\mathrm{T}} B_k s_k},
\end{aligned}
$$

which leads to the following update equation for B_{k+1}:

$$
B_{k+1} = B_k - \frac{B_k s_k s_k^{\mathrm{T}} B_k}{s_k^{\mathrm{T}} B_k s_k} + \frac{y_k y_k^{\mathrm{T}}}{s_k^{\mathrm{T}} y_k}. \tag{46}
$$

BFGS method

- We are now almost ready, except for the choice of the initial matrix B_0. Unfortunately, there is no standard recipe for the choice of B_0.
- One possible choice is the Hessian of $f(x)$ calculated at the initial guess x_0 (either in closed form, or using finite differences.)
- Another choice is a multiple of the identity matrix, $B_0 = aI$, where a is user determined.
- For the exit criterion we use $\|\nabla f(x_k)\|_\infty < \varepsilon$, where ε is user determined.

BFGS method

- Let us now summarize the BFGS method in an algorithmic form:

initialize x_0 and B_0
while ($\|\nabla f(x_k)\|_\infty > \varepsilon$) and $k \leq k_{max}$
    compute the search direction (39)
    carry out the line search (41)
    set $s_k = \alpha_k h$, update $x_{k+1} = x_k + s_k$, and calculate y_k
    update B_k to B_{k+1} according to equation (46)

BFGS method

- The algorithm above requires inverting the matrix B_k at each step of the iteration.
- There is a version of the BFGS method, which operates in terms of the inverses $H_k = B_k^{-1}$ only.
- Namely, one can verify (with some pain) that the update equation (46) can be formulated in terms of the inverses as follows:

$$H_{k+1} = \left(I - \frac{s_k y_k^{\mathrm{T}}}{s_k^{\mathrm{T}} y_k}\right) H_k \left(I - \frac{y_k s_k^{\mathrm{T}}}{s_k^{\mathrm{T}} y_k}\right) + \frac{s_k s_k^{\mathrm{T}}}{s_k^{\mathrm{T}} y_k} . \tag{47}$$

- The BFGS method is implemented in the Python package `scipy`.

Global optimization

- In many applications, the objective function has multiple local minima, or at least the existence of a unique global minimum cannot be guaranteed.
- Furthermore, objective functions can be non-differentiable, noisy, exhibit flat directions, etc.
- In situations like this, analytic methods or Newton-like methods are not applicable.
- A number of approaches for solving such optimization problems have been proposed. We will focus on a particularly neat stochastic search method, namely *differential evolution*.

Hard objective function: Ackley function

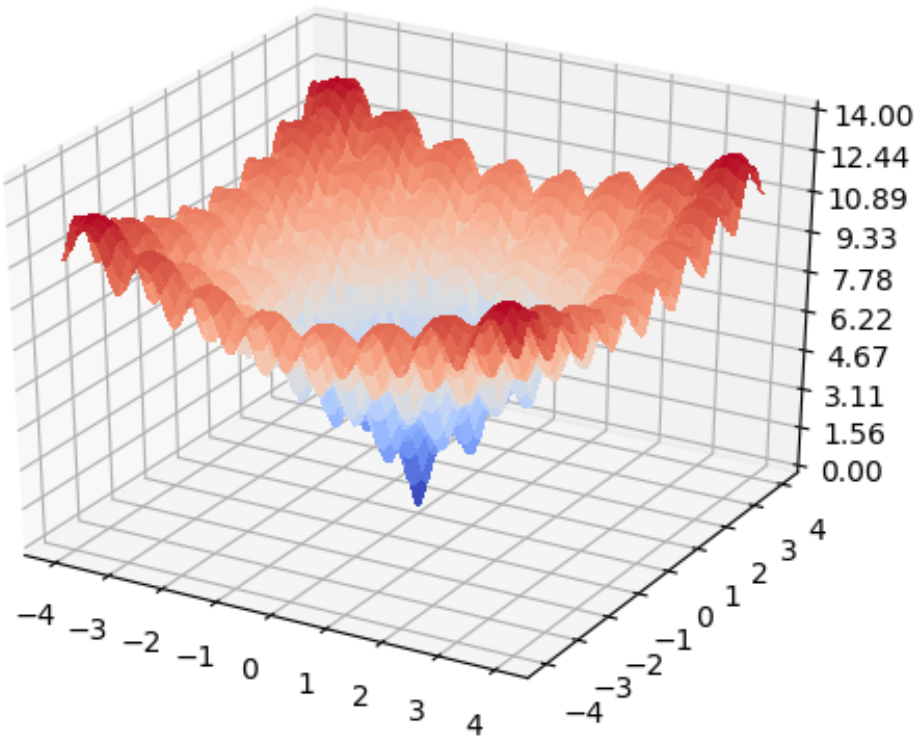
- Another function, frequently used as a performance benchmark for testing optimization algorithms, is the *Ackley function*:

$$f(x,y) = 20 + e - 20\,\exp\Big(-0.2\sqrt{0.5(x^2+y^2)}\Big) - \exp\Big(0.5\big(\cos(2\pi x)+\cos(2\pi y)\big)\Big). \tag{48}$$

- Inspection shows that it has a unique global minimum at $x = 0, y = 0$.
- This minimum, even though trivial to find analytically, is hidden among plenty of local minima and maxima, which makes Newton-type methods impractical.

Hard objective function: Ackley function

- The figure below shows a 3D graph of the Ackley function. Note the multitude of local minima and maxima.



Differential evolution

- As before, let $f(x)$, $x \in \mathbb{R}^n$ be an objective function. We do not require $f(x)$ to be differentiable.
- Each of the components of x is restricted to a finite interval:

$$L_i \leq x_i \leq U_i. \tag{49}$$

  The limits are chosen by the user. Tighter limits lead to a better performance of the algorithm.
- We select the population size $N \geq 4$, and randomly generate the initial population: $x_{1,0}, \ldots, x_{N,0}$.
- We will evolve this population through a number of generations G according to the rules motivated by genetics.

Differential evolution

- *Mutation*. We choose a constant *mutation factor* $F \in [0, 2]$, determined by the user.
- For a given vector $x_{i,G}$ (we start with $G = 0$), we randomly select three vectors $x_{r_1,G}, x_{r_2,G}, x_{r_3,G}$, such that the four indices i, r_1, r_2, r_3 are all distinct.
- We form the *donor (mutant) vectors*:

$$v_{i,G+1} = x_{r_1,G} + F(x_{r_2,G} - x_{r_3,G}), \tag{50}$$

for each element of the population $i = 1, \ldots, N$.

Differential evolution

- *Crossover*. We choose a user determined *crossover constant* $CR \in [0, 1]$.
- For each $i = 1, \ldots, N$ we form a *trial vector* $u_{i,G+1}$ constructed as follows.
  - (i) Generate random numbers $u_j \sim U(0, 1), j = 1 \ldots, n$.
  - (ii) Generate a random integer r from the range $1, \ldots, n$.
  - (iii) For $j = 1 \ldots, n$, set

$$u_{j,i,G+1} = \begin{cases} v_{j,i,G+1}, & \text{if } u_j \leq CR \text{ or } j = r, \\ x_{j,i,G}, & \text{if } u_j > CR \text{ and } j \neq r. \end{cases} \tag{51}$$

Differential evolution

- *Selection*. The target vector $x_{i,G}$ is now compared with the trial vector $u_{i,G+1}$, and the one with the lower value of the objective function is passed on to the next generation.
- Namely, for $i = 1, \ldots, N$,

$$x_{i,G+1} = \begin{cases} u_{i,G+1}, & \text{if } f(u_{i,G+1}) < f(x_{i,G}) \\ x_{i,G} & \text{otherwise.} \end{cases} \tag{52}$$

Differential evolution

- Choosing the right parameters for the differential evolution requires trial and error. Usually, $F = 0.5$, and $CR = 0.1$ work well. Population size N should be a function of the dimensionality of the problem. A useful rule of thumb is $N = 5n$.
- Implementation of differential evolution requires exit criteria imposed by the user. One of them is (of course) the maximum number of generations G_{max}.
- Another (more important) criterion is based on the assessment of whether $f_G^* = \min_i f(x_{i,G})$ approximates the minimum of $f(x)$ close enough.
- In practice, this is best done by monitoring the distribution of the values of $f(x_{i,G})$. We exit when $\max_i |f(x_{i,G}) - f_G^*|$ falls below a threshold ε.

References


[1] Kochenderfer, M. J., and Wheeler, T. A.: *Algorithms for Optimization*, MIT Press (2019).


[2] Madsen, K., Nielsen, K. B., and Tingleff, O.: *Methods for Non-Linear Least Squares Problems* Technical University Denmark (2014).


[3] Nocedal, J. and Wright, S. J.: *Numerical Optimization*, Springer (2006).


[4] Storn, R., and Price, K.: Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces, *Journal of Global Optimization*, **11**, 341–359 (1997).