Optimization: Part 1

Abhinav Anand, IIMB

2018/07/03

Background

Problems in finance and economics are often concerned with the behavior of agents who are considered to be utility maximizers. Utility functions are thought to be monotonic in their variables—more of a utility enhancing variable is better than less—and are thought to obey diminishing returns as the variables scale.[1] Additionally, real-life constraints ensure that variables are bounded. Hence optimization of functions under constraints forms an important discipline in the study of such subjects.

After linear optimization, quadratic optimization problems are the simplest since they can be captured in *quadratic forms* and can be represented via symmetric matrices with special properties. A distinguishing feature of quadratic optimization is the presence of first and second order conditions that characterize the nature of the extreme point.

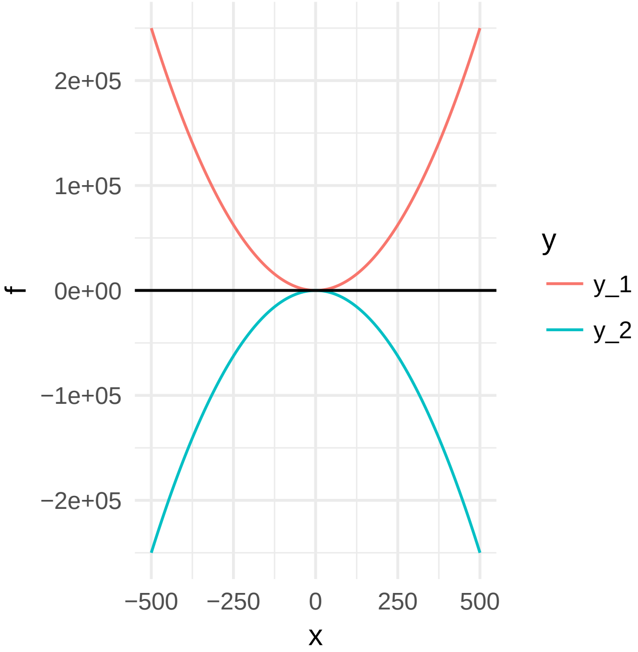
Quadratic Optimization

In one dimension, $x \in \mathbb{R}$ the simplest quadratic objective functions can be $f(x) = \{x^2, -x^2\}$ with the first attaining a global minimum and the second a global maximum at $x = 0$.

```
x <- -500:500
y_1 <- x^2
y_2 <- -x^2
data_l <- cbind(x, y_1, y_2) %>%
  dplyr::as_tibble() %>% #wide format
```

[1] This utility function can assume a variety of forms. For example, if the 'agent' is a firm, its utility is its profit function; if it's a government, it could be some (aggregate) social welfare function etc.

```
  tidyr::gather(.,
                y_1:y_2,
                key = 'y',
                value = 'f') #long format

ggplot(data_l, aes(x, f, color = y)) +
  geom_line() +
  geom_hline(yintercept = 0) +
  theme_minimal()
```



Quadratic Forms

For the case when $x = (x_1, x_2) \in \mathbb{R}^2$, a general quadratic form is the following:

$$Q(x) = a_{11}x_1^2 + a_{22}x_2^2 + 2a_{12}x_1x_2$$

which may be represented as a matrix:

$$Q(x) = [x_1x_2] \begin{bmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Similarly, when $x = (x_1, x_2, x_3) \in \mathbb{R}^3$, the form becomes

$$Q(x) = [x_1 x_2 x_3] \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

and in general, when $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$:

$$Q(x) = [x_1 \ldots x_n] \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & & \\ a_{1n} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

It's easy to see that $Q(0) = 0$. If $a > 0, ax^2 > 0$ and we can call the quadratic form "positive definite". Similarly, if $a < 0, ax^2 < 0$ and we call the form "negative definite". This carries over when $x \in \mathbb{R}^2$, since $Q(x_1, x_2) = x_1^2 + x_2^2 > 0$ (positive definite) and $Q(x_1, x_2) = -x_1^2 - x_2^2 > 0$ (negative definite). Additionally for the case when $Q(x_1, x_2) = x_1^2 - x_2^2 > 0$, the quadratic form is *indefinite*.

In general, symmetric matrices are called positive (semi)definite, negative (semi)definite etc. according to the definiteness of the expression $Q(x) = x^\top Ax$.[2] Additionally for symmetric matrices, the eigenvalues are all real numbers. A highly useful fact to know is that for positive definite matrices, all eigenvalues are positive; for negative definite matrices, all eigenvalues are negative; and for indefinite matrices, some eigenvalues are positive and some are negative.

Quadratic Programming

In general, a quadratic programming problem can assume the following form:

$$\min x^\top Dx - d^\top x :$$

$$Ax \geq b$$

$$Ex = c$$

$$x \geq 0$$

[2] Hence if $\forall x \neq 0 \in \mathbb{R}^n, x^\top Ax \geq 0$ the form is positive semidefinite and so on.

Constraints must be linear and the objective must be quadratic. Without loss of generality, the matrix D may be assumed symmetric. However it must be positive semidefinite.

Computation: Quadratic Programming

The package `quadprog` includes subroutines for solving quadratic programming problems. We use the function `solve.QP()` for this purpose.

The general syntax for solving such quadratic programming problems is the following:

```r
quadprog::solve.QP(Dmat = ..., #the D matrix of quadratic coefficients
                   dvec = ..., #the d vector
                   Amat = ..., #the constraint matrix
                   bvec = ... #the constraint RHS
                   meq = 0, #how many first few equality constraints
                  )
```

Illustration: Solve the following quadratic program:

$$\min f(x) = 1/2x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2 :$$

$$x_1 + x_2 \leq 2$$

$$-x_1 + 2x_2 \leq 2$$

$$2x_1 + x_2 \leq 3$$

$$x_1, x_2 \geq 0$$

This problem may be rewritten in matrix form:

$$\min [x_1, x_2] \begin{bmatrix} 1/2, -1/2 \\ -1/2, 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - [2, 6] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} :$$

$$\begin{bmatrix} -1, -1 \\ 1, -2 \\ -2, -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} -2 \\ -2 \\ -3 \end{bmatrix}$$

We can solve this quadratic program via the following:

```
Dmat <- matrix(c(1/2,-1/2,-1/2,1),2,2)
dvec = c(2,6)
Amat = matrix(c(-1,-1,1,-2,-2,-1),2,3)
bvec = c(-2,-2,-3)

quadprog::solve.QP(Dmat, dvec, Amat, bvec)
```

```
## $solution
## [1] 0.6666667 1.3333333
##
## $value
## [1] -8.777778
##
## $unconstrained.solution
## [1] 20 16
##
## $iterations
## [1] 3 0
##
## $Lagrangian
## [1] 3.2222222 0.8888889 0.0000000
##
## $iact
## [1] 1 2
```

Application: Mean-Variance Optimization

A classic application of quadratic programming is its use in mean-variance optimization for portfolios. We illustrate it with the following example:

Suppose an investor has n assets in her portfolio. She wishes to find best weights for each portfolio. She likes high returns but also wants low risk.

She could frame this problem as one about minimizing the variance of the portfolio while subjecting returns to a certain lower bound.[3] Suppose she possesses securities with returns according to normal random variables $X_i \sim \mathcal{N}(\mu_i, \sigma_i)$; and has apportioned fractional weights w_i for each of them.

Then for two securities:

$$P_2 \sim \mathcal{N}(w_1\mu_1 + w_2\mu_2; w_1^2\sigma_1^2 + w_2^2\sigma_2^2 + 2w_1w_2cov(X_1, X_2))$$

and for n securities in general:

$$P_n \sim \mathcal{N}(w^\top \mu; w^\top \Sigma w)$$

where Σ is the $n \times n$ coavariance matrix.

Hence the optimization program for the investor becomes: minimize the variance subject to the constraint that the return is higher than some lower critical value.

$$\min w^\top \Sigma w :$$

$$w^\top \mu \geq \mu_*; w^\top e = 1; w \geq 0$$

This is of the same form as a quadratic program.

Illustration: Suppose there are two normally distributed common stocks whose expected returns are $\mu^\top = (1.8\%, 2.5\%)$ and whose covariance matrix is:

$$\Sigma = \begin{bmatrix} 1.68, 0.34 \\ 0.34, 3.09 \end{bmatrix}$$

Hence the minimization program is:

$$\min 1.68w_1^2 + 3.09w_2^2 + 2 * 0.34w_1w_2 :$$

$$w_1 + w_2 = 1$$

[3] Risk is often assumed to be (roughly) measurable via the variation in a security's prices—hence variance. Volatility is the name for the standard deviation. The idea is that securities with more price fluctuations, and hence more variance, are more risky.

$$0.018w_1 + 0.025w_2 \geq 0.018$$

$$w_1, w_2 \geq 0$$

```r
D_mat <- matrix(c(1.68,0.34,0.34,3.09),2,2)
d_vec <- c(0, 0)
A_mat <- matrix(c(1,1,0.018,0.025),2,2)
b_vec <- c(1,0.018)
m_eq <- 1

quadprog::solve.QP(D_mat, d_vec, A_mat, b_vec, m_eq)
```

```
## $solution
## [1] 0.6723716 0.3276284
##
## $value
## [1] 0.620489
##
## $unconstrained.solution
## [1] 0 0
##
## $iterations
## [1] 2 0
##
## $Lagrangian
## [1] 1.240978 0.000000
##
## $iact
## [1] 1
```

Unconstrained Optimization

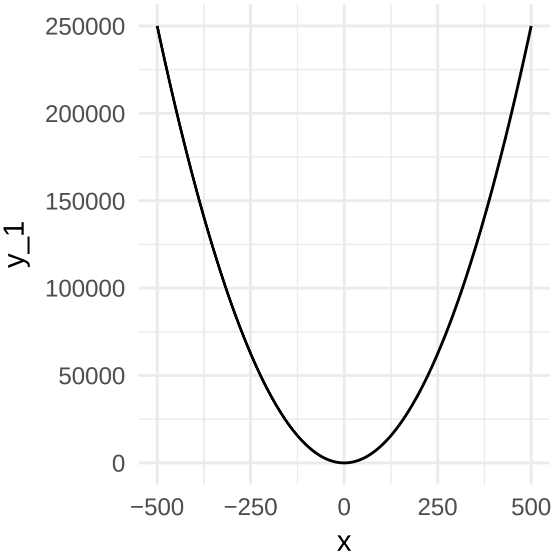
While it's clear what a maximum is in one dimension, what should be its definition in two dimensions and beyond? The central idea remains the same: x^* is the maximum if there is no other x in the domain of $f(x)$ such that $f(x) \geq f(x^*)$. Note that this definition is general and independent of the dimension of the underlying space from where we pick x. The only difference in dimensions one and beyond is the structure of the domain; and while in one dimension it must take the form $x_L \leq x \leq x_H$, one needs such conditions for each component when $x \in \mathbb{R}^n : x_{L1} \leq x_1 \leq x_{H1}, \ldots, x_{Ln} \leq x_n \leq x_{Hn}$. With this interpretation in mind, we are ready to formulate the necessary and sufficient conditions for the optimal of an unconstrained optimization program.

First Order Conditions

When the dimension of the domain is 1, $x_L \leq x \leq x_H$, the first order condition stipulates that at the *critical point* the derivative be 0.[4] Whether this critical point is a (local) maximum or (local) minimum or neither, depends on the *second order condition* on the derivative of the derivative.

The example of $f(x) = x^2$ illustrates this idea.

```
(plot_x_square <- ggplot(data = dplyr::as_tibble(cbind(x, y_1)),
       mapping = aes(x, y_1)) +
  geom_line() +
  theme_minimal())
```



End points or corner points are generally not considered critical points though they

[4] Why must this be so? Can you see the case for the plot $f(x) = x^2$ and reason why?

can contain extreme points. (For the plot $f(x) = x^2$ what is/are maximal points if $x \in [l, b]$?) Conventionally, critical points are thought to be points interior to the domain.

Functions can be separated into classes on the basis of how differentiable they are (the differentiability class). For example, all functions that are continuous form the C^0 class; those that are differentiable and their derivatives also continuous form the class C^1; those whose second derivatives are continuous form the class C^2 and so on.[5]

Theorem: If $x^* \in \mathbb{R}^n$ is an interior point in the domain of the real-valued function $F \in C^1$, then it is a critical point when

$$\frac{\partial F}{\partial x_i}(x^*) = 0, i \in \{1, 2, \dots, n\}$$

Second Order Conditions

Critical points are interior points at which all first partial derivatives are 0. However, this in itself is not enough for us to know if the critical point is a maximum, a minimum or neither. For this, we need to consider its second derivatives.

For the function $F \in C^2, x \in U \subset \mathbb{R}^n$, ($U$ is open in $\mathbb{R}^n$) the *Hessian* is the following second derivative matrix:

$$D^2F(x^*) = \begin{bmatrix} \frac{\partial F}{\partial x_1^2}(x^*) & \dots & \frac{\partial F}{\partial x_n \partial x_1}(x^*) \\ \vdots & \ddots & \vdots \\ \frac{\partial F}{\partial x_1 \partial x_n}(x^*) & \dots & \frac{\partial F}{\partial x_n^2}(x^*) \end{bmatrix}$$

Since for the function $F \in C^2$, the cross-partials $\frac{\partial F}{\partial x_i \partial x_j} = \frac{\partial F}{\partial x_j \partial x_i}$, the Hessian is symmetric.

The second order conditions are based on the Hessian. Just as in the one dimensional case, where there occurs a maximum at a critical point if the second derivative is negative; there occurs a maximum at the critical point $x^* \in U \subset \mathbb{R}^n$ if the Hessian is *negative definite*.

[5] A function is "smooth" if all derivatives are continous: $f \in C^\infty$; and analytical $f \in C^\omega$ if it is smooth *and* its Taylor approximation converges to its function value at all points in the domain.

Theorem: For the function $F \in C^2, x \in U \subset \mathbb{R}^n$ with critical point x^*:

1. If the Hessian $D^2F(x^*)$ is negative definite, then x^* is a local maximum.
2. If the Hessian $D^2F(x^*)$ is positive definite, then x^* is a local minimum.
3. If the Hessian $D^2F(x^*)$ is indefinite, then x^* is neither. (x^* is a "saddle point".)

Convex Functions and Global Minima

While the conditions above classify critical points as maximal, minimal or saddle; these are only local optima. However if we consider only a special class of functions called *convex*, we can guarantee that the local minimum will also be the global minimum!

A simple example of a convex function is $f(x) = x^2$. It has one minimum at $x = 0$ which is also its global minimum. The added wrinkle is that the domain of a convex function must be a *convex* set.

Convex Set

A set $U \subset \mathbb{R}^n$ is convex if for two points in it, the whole line connecting the two points must also fully lie in U. More formally, a set U is convex if $\forall t \in [0, 1]$

$$x, y \in U \Rightarrow tx + (1 - t)y \in U$$

(Can you see this geometrically?)

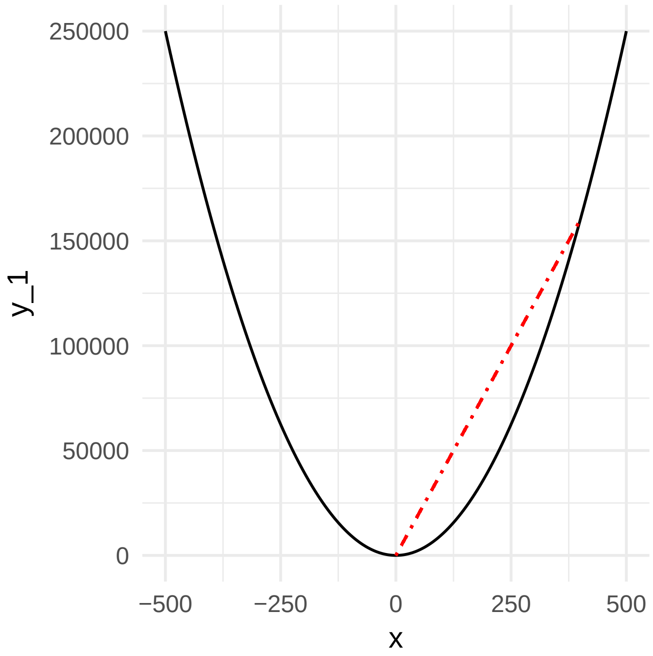
Convex Function

Now, we can define a convex *function* whose domain is a convex *set* by stipulating that the image of the line connecting x to y lie below the line itself. Hence a function F is convex if its domain $U \in \mathbb{R}^n$ is convex; and for all $x, y \in U$ and $\forall t \in [0, 1]$

$$tF(x) + (1 - t)F(y) \geq F(tx + (1 - t)y)$$

We can check if this condition is satisfied for our one dimensional convex function $f(x) = x^2$.

```
plot_x_square +
  geom_segment(aes(x = 0,
                   y = 0,
                   xend = 400,
                   yend = 400^2
                   ),
               linetype = "dotdash",
               color = "red"
               )
```



Concave Function

A *concave* function is simply the negative of a convex function. In one dimension, $-x^2, \log(x), \sqrt{x}$ etc. are some commonly known concave functions. For these, local maxima are global maxima. Note that just like the convex functions, domains of concave functions must be convex too.

Theorem: If $F : U \to \mathbb{R}$, (where $U \subset \mathbb{R}^n$ is convex and open) $F \in C^2$ and convex then

1. $D^2F(x)$ is positive semidefinite for $x \in U$
2. $F(y) - F(x) \geq DF(x) \cdot (y - x)$ for $x, y \in U$
3. If for $x^* \in U, DF(x^*) = 0$ then x^* is the global minimum

Computation: Unconstrained Optimization

There are several special packages written in `R` for solving optimization problems. A core set of packages is included with the installation of `R`, with more than 12,500 additional packages (as of May 2018) available at the "Comprehensive R Archive Network" (CRAN). Additionally, packages related to a topic, say, Econometrics or Optimization or Differential Equations etc. can be found by looking them up in CRAN Task Views: https://cran.r-project.org/web/views/.

We quote here from CRAN Task Views: Optimization

> For one-dimensional unconstrained function optimization there is `optimize()` which searches an interval for a minimum or maximum. Function `optim()` provides an implementation of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, bounded BFGS, conjugate gradient (CG), Nelder-Mead, and simulated annealing (SANN) optimization methods. It utilizes gradients, if provided, for faster convergence. Typically it is used for unconstrained optimization but includes an option for box-constrained optimization.

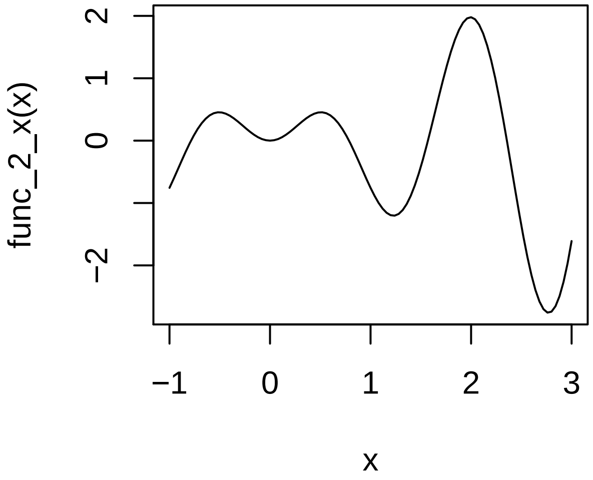
One-Dimension

The `optimize()` function is used for one dimensional problems. Its first argument is the function to be optimized and the second argument is the solution search space as provided by the user. It may only be used to detect local extrema.

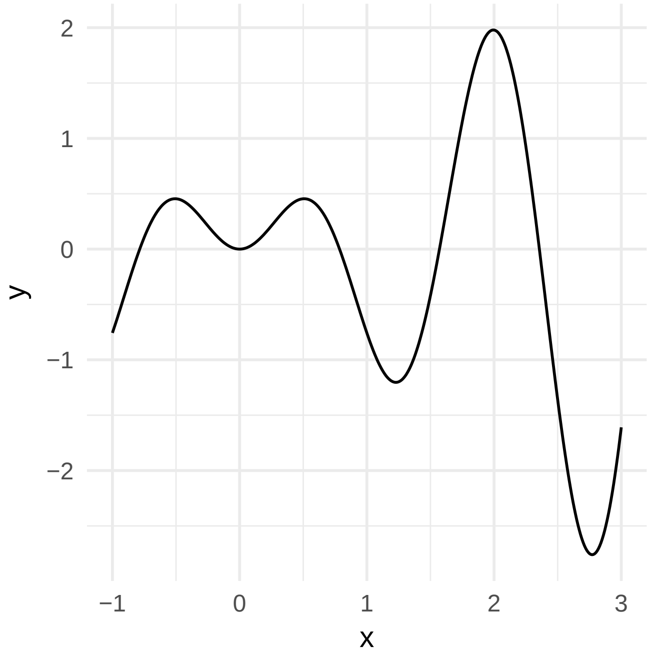
To illustrate, consider $xsin(4x)$:

```r
# Another function for optimization
func_2_x <- function(x) {x*sin(4*x)}
```

```r
curve(func_2_x, -1, 3) #note base R graphics function curve()
```



```r
# Via ggplot()
x <- seq(-1, 3, 0.01)
y <- func_2_x(x)
data_plot_gg <- cbind(x, y) %>% tibble::as_tibble()

ggplot(data_plot_gg, aes(x, y)) +
  geom_line() +
  theme_minimal()
```



```
# Using optimize()
(optim_min_mult <- optimize(f = func_2_x,
                            interval = c(0, 3)
                            ) #default = minimum
)
```

```
## $minimum
## [1] 1.228297
##
## $objective
## [1] -1.203617
```

It seems from the plot of $xsin(4x)$ that the local minimum is at around 1.2 and the global minimum is at around 2.8. However, our invocation of the `optimize()` function yields a minimum of 1.2282971, which is not the global minimum even though it lies within our set search limits $(2.8 \in c(0, 3))$. This is so because the function `optimize()` returns the first minimum it encounters in the search algorithm. Since we can observe the plot, we should supply a better default search region to capture the global minimum:

```r
optimize(func_2_x, c(1.5, 3))
```

```
## $minimum
## [1] 2.771403
##
## $objective
## [1] -2.760177
```

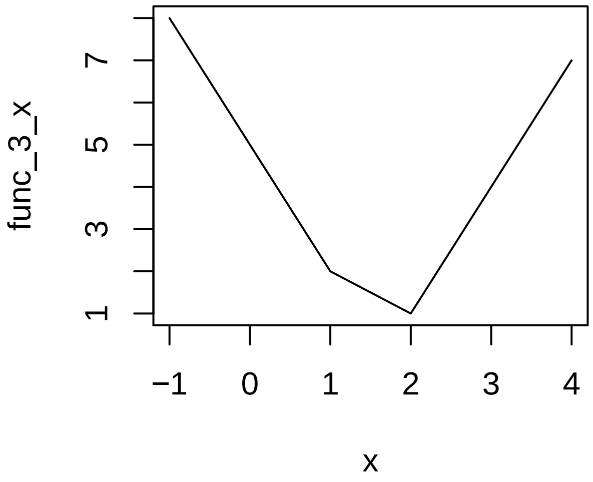
In order to find the maximum:

```r
optimize(func_2_x, c(1.5, 3), maximum = TRUE)
```

```
## $maximum
## [1] 1.994653
##
## $objective
## [1] 1.979182
```

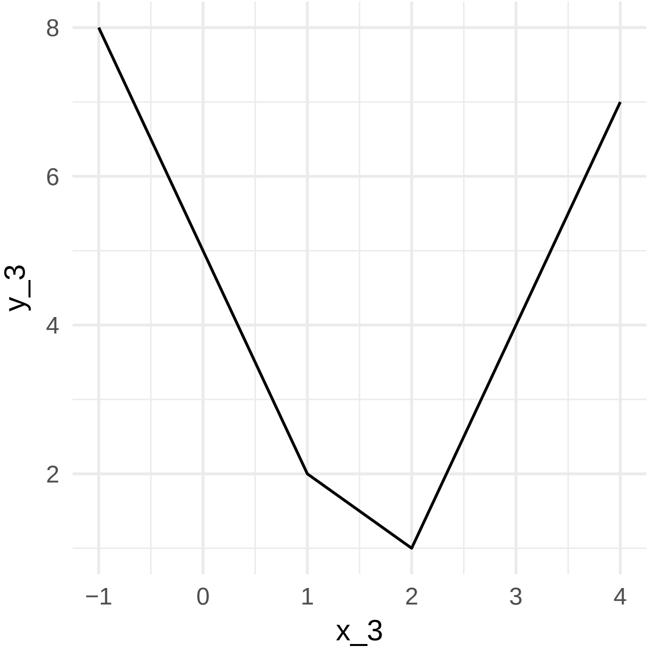
The optimize() function works with non-differentiable functions too:

```r
func_3_x <- function(x) {abs(x-1)+2*abs(x-2)}

plot(func_3_x, -1, 4) #note generic function plotting in base R
```



```r
# Via ggplot
x_3 <- seq(-1, 4, 0.01)
```

```
y_3 <- func_3_x(x_3)

ggplot(tibble::as_tibble(cbind(x_3,y_3)),
       aes(x_3, y_3)
       ) +
  geom_line() +
  theme_minimal()
```



```
(optim_x_3 <- optimize(func_3_x, c(0,3)))
```

```
## $minimum
## [1] 1.999991
##
## $objective
## [1] 1.000009
```

Multidimensional Programming

We use the package `optim()` for solving multidimensional programs.

```
optim(par = ..., #initial value vector (parameters)
      fn = ..., #objective function (minimization)
      lower = -Inf, #lower bound
      upper = Inf, #upper bound
      method = '', #default method = "Nelder-Mead"
      control = ..., #maximum iterations
      )
```

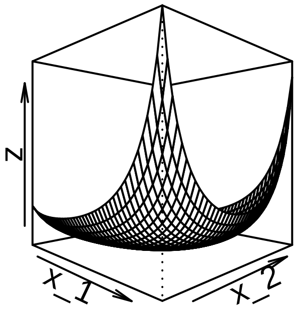
Consider the function $f(x) = 2(x_1 - 1)^2 + 5(x_2 - 3)^2 + 10$

```
func_optim <- function(x)
{
  2*(x[1]-1)^2 + 5*(x[2]-3)^2 + 10
}

optim_min <- optim(c(1,1), func_optim) #x=(x_1,x_2)=c(1,1)
print(optim_min)
```

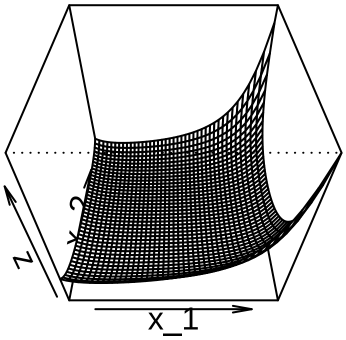
```
## $par
## [1] 1.000168 3.000232
##
## $value
## [1] 10
##
## $counts
## function gradient
##       75       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Another example: $f(x) = \frac{1}{x_1} + \frac{1}{x_2} + \frac{1-x_2}{x_2(1-x_1)} + \frac{1}{(1-x_1)(1-x_2)}$

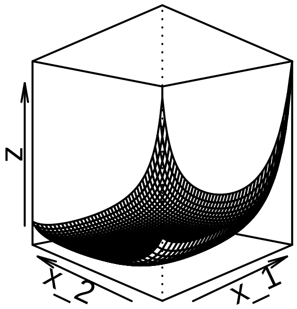
```r
func_optim_2 <- function(x1, x2)
{
  return(1/x1 + 1/x2 +
    (1-x2)/(x2*(1-x1)) +
    1/((1-x1)*(1-x2))
    )
}

x_1 <- x_2 <- seq(0.1, 0.9, 0.02)
z <- outer(x_1, x_2, FUN = "func_optim_2")

# Different perspectives
persp(x_1, x_2, z, theta = 45, phi = 0)
```



```r
persp(x_1, x_2, z, theta = 0, phi = 45)
```



```r
persp(x_1, x_2, z, theta = -45, phi = 0)
```



```
func_opt_2 <- function(x)
{
  x1 <- x[1]
  x2 <- x[2]
  return((1/x1 + 1/x2 +
            (1-x2)/(x2*(1-x1)) +
            1/((1-x1)*(1-x2)))
         )
}

optim(c(0.5, 0.5), func_opt_2)
```

```
## $par
## [1] 0.3636913 0.5612666
##
## $value
## [1] 9.341785
##
## $counts
## function gradient
##       55       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Application: Maximum Likelihood Estimation

While informally the words 'probability' and 'likelihood' can be used somewhat interchangeably, in statistics there is an important difference between the two. Probability is the plausibility of an event happening given certain model parameter values; while likelihood is the plausibility of the model parameter values given certain observed events.[6]

Illustration: Suppose we flip a coin where p_H denotes the probability of being heads. Suppose we get two heads in a row. Suppose we assume the coin flips are independent and identically distributed. Then

$$\Pr(\{H,H\}|p_H = 0.5) = 0.5 \cdot 0.5 = 0.25 = \mathcal{L}(p_H = 0.5|\{H,H\})$$

More formally, for the "likelihood function" $\mathcal{L}(\cdot)$

$$\Pr_\theta(X = x) := \mathcal{L}(\theta|X = x)$$

or for continuous random variables X,

$$f_\theta(x) := \mathcal{L}(\theta|X = x)$$

Since we care most about the point in the domain where the likelihood reaches its maximum, we often compute the *log*-likelihood $l(\cdot) = \ln(\mathcal{L}(\cdot))$ since the logarithm is strictly increasing function and therefore must possess the same argmax.

The maximum likelihood estimate is:

$$\hat{\theta} \in \{\text{argmax}_{\theta\in\Theta}\mathcal{L}(\theta; x)\}$$

For some models, MLEs have closed forms. For many others no closed-form solutions to the maximization problem is known; and for many others, there may be multiple solutions or no solutions.

If samples are independent and identically distributed:

[6] Tellingly, the likelihoods, when added, do not need to add up to 1 unlike probabilities.

$$\hat{l}(\theta; x) = \max \frac{1}{n} \sum_{i=1}^{n} \ln(f(x_i|\theta))$$

Note that this is the sample analogue of maximimzing the expected log-likelihood: $\max \mathbb{E}[\ln(f(x_i|\theta))]$.

Computation: Maximum Likelihood

Suppose for a normal random variable $X \sim \mathcal{N}(\mu, \sigma)$ there are iid observations $\{x_i\}_{i=1}^{n}$. Then the log-likelihood function is:

$$l(\mu, \sigma) = \sum_{i=1}^{n} \ln(f(x_i)|\theta = (\mu, \sigma))$$

$$l(\mu, \sigma) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2$$

Taking first order conditions with respect to μ and σ^2 yields maximum likelihood estimates

$$\hat{\mu} = \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\widehat{\sigma^2} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Let's check if we get the same results with the `optim()` function

```
x_vec <- rnorm(1000, 5, 2) #1000 normals with mean 5 and sigma 2

neg_log_likeli <- function(theta)
{
  n <- length(x_vec)
  NLL <- sum((x_vec - theta[1])^2)/(2*theta[2]) +
    (n/2)*log(theta[2])

  return(NLL)
}
```

```r
optim(theta <- c(0,1), #(0,1) starting
      neg_log_likeli,
      hessian=TRUE #print the Hessian matrix
      )
```

```
## $par
## [1] 5.084675 4.163636
##
## $value
## [1] 1213.22
##
## $counts
## function gradient
##       81       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##              [,1]        [,2]
## [1,] 240.17470344 -0.06452115
## [2,]  -0.06452115 28.84493057
```

```r
optim(theta <- c(0.5,1.2), #different starting point
      neg_log_likeli,
      hessian=TRUE
      )
```

```
## $par
## [1] 5.083225 4.165530
##
## $value
## [1] 1213.22
##
## $counts
## function gradient
##       79       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##              [,1]        [,2]
## [1,] 240.06549381  0.01912008
## [2,]   0.01912008 28.79248171
```

```
mean(x_vec)
```

```
## [1] 5.083557
```

```
sd(x_vec)
```

```
## [1] 2.041572
```