# Optimization II

August 12, 2020

## 1 Introduction to optimization

In general, optimization is the process of finding and selecting the optimal element from a set of feasible candidates. In mathematical optimization, this problem is usually formulated as determining the extreme value of a function on a given domain. An extreme value, or an optimal value, can refer to either the minimum or maximum of the function, depending on the application and the specific problem. In this lab we are concerned with the optimization of realvalued functions of one or several variables, which optionally can be subject to a set of constraints that restricts the domain of the function. The applications of mathematical optimization are many and varied, and so are the methods and algorithms that must be employed to solve optimization problems. Since optimization is a universally important mathematical tool, it has been developed and adapted for use in many fields of science and engineering, and the terminology used to describe optimization problems varies between fields. For example, the mathematical function that is optimized may be called a cost function, loss function, energy function, or objective function, to mention a few. Here we use the generic term objective function. Optimization is closely related to equation solving because at an optimal value of a function, its derivative, or gradient in the multivariate case, is zero.

In this lab we discuss using SciPy's optimization module optimize for nonlinear optimization problems, and we will briefly explore using the convex optimization library cvxopt for linear optimization problems with linear constraints. This library also has powerful solvers for quadratic programming problems.

**cvxopt:** The convex optimization library cvxopt provides solvers for linear and quadratic optimization problems.

[1]:
```python
from scipy import optimize
import cvxopt
```

### 1.1 Classification of Optimization Problems

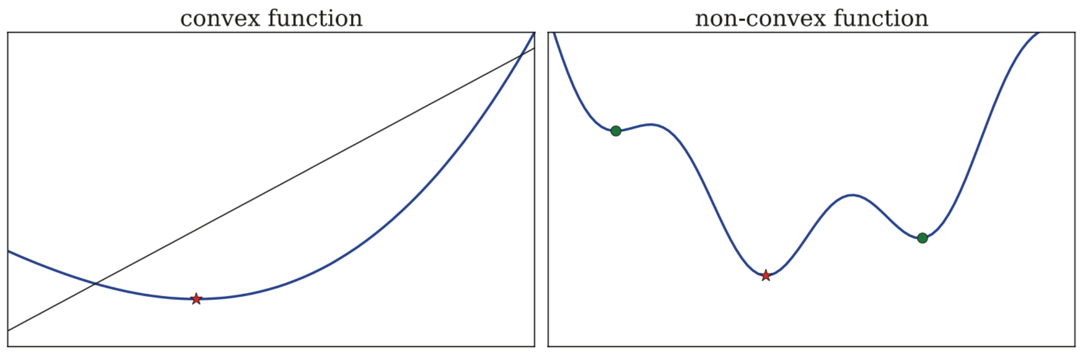
Here we restrict our attention to mathematical optimization of real-valued functions, with one or more dependent variables. Many mathematical optimization problems can be formulated in this way, but a notable exception is optimization of functions over discrete variables.
A general optimization problem of the type considered here can be formulated as a minimization problem, minx $f(x)$, subject to sets of $m$ equality constraints $g(x) = 0$ and $p$ inequality constraints $h(x) \leq 0$. Here $f(x)$ is a real-valued function of x, which can be a scalar or a vector $x = (x_0, x_1, \ldots, x_n)^T$, while g(x) and h(x) can be vector-valued functions: $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$

and $h : \mathbb{R}^n \to \mathbb{R}^p$. Note that maximizing $f(x)$ is equivalent to minimizing $\breve{}f(x)$, so without loss of generality, it is sufficient to consider only minimization problems.

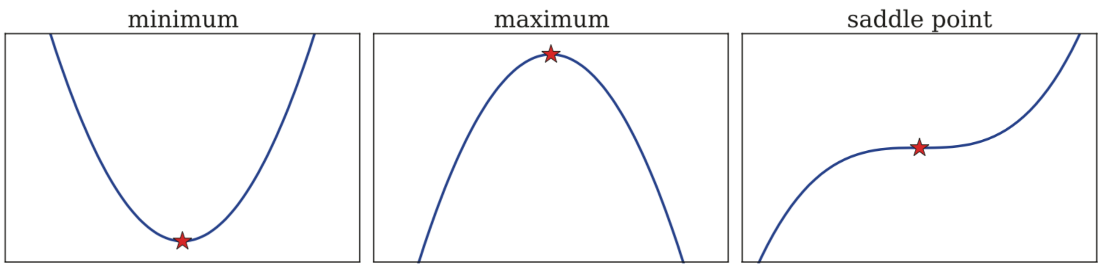
Optimization problems are classified depending on the properties of the functions $f(x)$, $g(x)$, and $h(x)$. First and foremost, the problem is univariate or one dimensional if $x$ is a scalar, $x \in \mathbb{R}$, and multivariate or multidimensional if $x$ is a vector, $x \in \mathbb{R}^n$. For highdimensional objective functions, with larger $n$, the optimization problem is harder and more computationally demanding to solve. If the objective function and the constraints all are linear, the problem is a linear optimization problem. If either the objective function or the constraints are nonlinear, it is a nonlinear optimization problem. With respect to constraints, important subclasses of optimization are unconstrained problems, and those with linear and nonlinear constraints. Finally, handling equality and inequality constraints requires different approaches.

### 1.1.1 Convex functions

An important subclass of nonlinear problems that can be solved efficiently is convex problems, which is directly related to the absence of strictly local minima and the existence of a unique global minimum. By definition, a function is convex on an interval $[a, b]$ if the values of the function on this interval lie below the line through the endpoints $(a, f(a))$ and $(b, f(b))$. This condition, which can be readily generalized to the multivariate case, implies a number of important properties, such as the existence of a unique minimum on the interval. Because of strong properties like this one, convex problems can be solved efficiently even though they are nonlinear. The concepts of local and global minima, and convex and nonconvex functions, are illustrated below



for the multivariate case, can be used to determine if a stationary point is a local minimum or not. In particular if the second-order derivative is positive, or the Hessian positive definite, when evaluated at stationary point x*, then x* is a local minimum. Negative second-order derivative, or negative definite Hessian, corresponds to a local maximum, and a zero second-order derivative, or an indefinite Hessian, corresponds to saddle point.



[2]:
```
import matplotlib.pyplot as plt
import numpy as np
```

### 1.2 Univariate Optimization

Optimization of a function that only depends on a single variable is relatively easy. In addition to the analytical approach of seeking the roots of the derivative of the function, we can employ techniques that are similar to the root-finding methods for univariate functions, namely, **bracketing methods** and **Newton's method**. Like the bisection method for univariate root finding, it is possible to use bracketing and iteratively refine an interval using function evaluations alone. Refining an interval $[a, b]$ that contains a minimum can be achieved by evaluating the function at two interior points $x_1$ and $x_2$, $x_1 < x_2$, and selecting $[x_1, b]$ as new interval if $f(x_1) > f(x_2)$, and $[a, x_2]$ otherwise. This idea is used in the golden section search method, which additionally uses the trick of choosing $x_1$ and $x_2$ such that their relative positions in the [a, b] interval satisfy the golden ratio. In the SciPy optimize module, the function golden implements the golden search method.

**Newton's method** for root finding is an example of a quadratic approximation method that can be applied to find a function minimum, by applying the method to the derivative rather than the function itself. This yields the iteration formula $x_{k+1} = x_k - f'(x_k)/f''(x_k)$, which can converge quickly if started close to an optimal point but may not converge at all if started too far from the optimal value. This formula also requires evaluating both the derivative and the second-order derivative in each iteration. If analytical expressions for these derivatives are available, this can be a good method. If only function evaluations are available, the derivatives may be approximated using an analog of the secant method for root finding.

A combination of the two previous methods is typically used in practical implementations of univariate optimization routines, giving both stability and fast convergence. In SciPy's optimize module, the `brent` function is such a hybrid method, and it is generally the preferred method for optimization of univariate functions with SciPy.

**Exercise 1** As an example for illustrating these techniques, consider the following classic optimization problem:
Minimize the area of a cylinder with unit volume.

Here, suitable variables are the radius $r$ and height $h$ of the cylinder, and the objective function is $f([r, h]) = 2\pi r^2 + 2\pi rh$, subject to the equality constraint $g([r, h]) = \pi r^2 h - 1 = 0$. As this problem is formulated here, it is a two-dimensional optimization problem with an equality constraint. However, we can algebraically solve the constraint equation for one of the dependent variables, for example, $h = 1/\pi r^2$, and substitute this into the objective function to obtain an unconstrained one-dimensional optimization problem: $f(r) = 2\pi r^2 + 2/r$.

To solve this problem using SciPy's numerical optimization functions, we first define a Python function f that implements the objective function. To solve the optimization problem, we then pass this function to, for example, optimize.brent. Optionally we can use the brack keyword argument to specify a starting interval for the algorithm:

[48]:
```
def f(r):
        return 2 * np.pi * r**2 + 2 / r
r_min = optimize.brent(f, brack=(0.1, 4))
r_min
```

[48]:
```
0.5419260772557135
```

[49]:
```
f(r_min)
```

[49]:
```
5.535810445932086
```

Instead of calling optimize.brent directly, we could use the generic interface for scalar minimization problems optimize.minimize_scalar. Note that to specify a starting interval in this case, we must use the bracket keyword argument:

[50]:
```
optimize.minimize_scalar(f, bracket=(0.1, 4))
```

[50]:
```
     fun: 5.535810445932086
    nfev: 19
     nit: 15
 success: True
       x: 0.5419260772557135
```

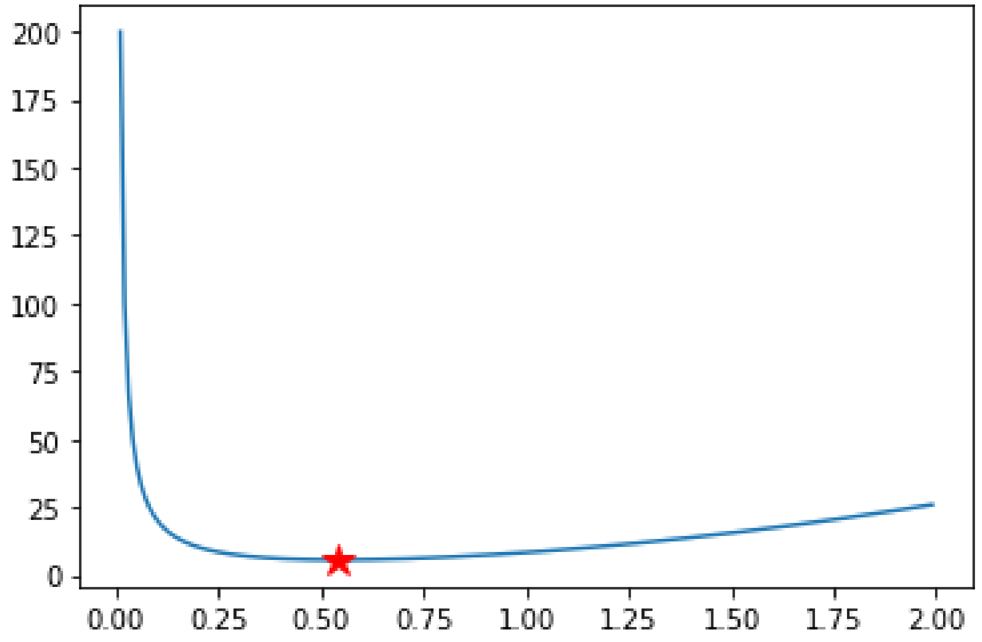
All these methods give that the radius that minimizes the area of the cylinder is approximately 0.54 ( and a minimum area of approximately 5.54. The objective function that we minimized in this example is plotted below, where the minimum is marked with a red star. When possible, it is a good idea to visualize the objective function before attempting a numerical optimization, because it can help in identifying a suitable initial interval or a starting point for the numerical optimization routine.

[51]:
```
import matplotlib.pyplot as plt
```

[52]:
```
r = np.arange(0,2,0.01)
f_r = f(r)
plt.plot(r,f_r)
plt.plot(r_min,f(r_min),'r*',markersize=12)
```

```
F:\Users\Yehia\anaconda3\envs\py3\lib\site-packages\ipykernel_launcher.py:2:
RuntimeWarning: divide by zero encountered in true_divide
```

```
[52]: [<matplotlib.lines.Line2D at 0x176650dd7c8>]
```



### 1.3 Unconstrained Multivariate Optimization

Multivariate optimization is significantly harder than the univariate optimization discussed in the previous section. In particular, the analytical approach of solving the nonlinear equations for roots of the gradient is rarely feasible in the multivariate case, and the bracketing scheme used in the golden search method is also not directly applicable. Instead we must resort to techniques that start at some point in the coordinate space and use different strategies to move toward a better approximation of the minimum point. The most basic approach of this type is to consider the gradient $\Delta f(x)$ of the objective function $f(x)$ at a given point $x$. In general, the negative of the gradient, $-\Delta f(x)$, always points in the direction in which the function $f(x)$ decreases the most. As minimization strategy, it is therefore sensible to move along this direction for some distance $\alpha_k$ and then iterate this scheme at the new point. This method is known as the steepest descent method, and it gives the iteration formula $x_{k+1} = x_k - \alpha_k \Delta f(x_k)$, where $\Delta_k$ is a free parameter known as the line search parameter that describes how far along the given direction to move in each iteration. An appropriate $\alpha_k$ can, for example, be selected by solving the one-dimensional optimization problem $min_{ak} = .f(x_k - \alpha_k \Delta f(x_k))$. This method is guaranteed to make progress and eventually converge to a minimum of the function, but the convergence can be quite slow because this method tends to overshoot along the direction of the gradient, giving a zigzag approach to the minimum.

Newton's method for multivariate optimization is a modification of the steepest descent method that can improve convergence. As in the univariate case, Newton's method can be viewed as a local quadratic approximation of the function, which when minimized gives an iteration scheme. In the multivariate case, the iteration formula $x_{k+1} = x_k - H_f^{-1}(x_k)\Delta f(x_k)$, where compared to the steepest descent method, the gradient has been replaced with the gradient multiplied from the left with the inverse of Hessian matrix for the function. In general this alters both the direction and the length of the step, so this method is not strictly a steepest descent method and may not converge if started too far from a minimum. However, when close to a minimum, it converges quickly. As usual there is a trade-off between convergence rate and stability. As it is formulated here, Newton's method requires both the gradient and the Hessian of the function

In SciPy, Newton's method is implemented in the function `optimize.fmin_ncg`. This function takes the following arguments: a Python function for the objective function, a starting point, a Python function for evaluating the gradient, and (optionally) a Python function for evaluating the Hessian.

**Exercise 2** To see how this method can be used to solve an optimization problem, we consider the following problem:

$$min_x f(x)$$

where the objective function is

$$f(x) = (x_1 - 1)^4 + 5(x_2 - 1)^2 - 2x_1x_2$$

To apply Newton's method, we need to calculate the gradient and the Hessian. For this particular case, this can easily be done by hand. However, it can also be computed using sympy library

[8]:
```
def f(X):
    return (X[0]-1)**4 + 5*(X[1]-1)**2 - 2*X[0]*X[1]
def f_diff (X):
    diff = np.array([-2*X[1]+4*(X[0]-1)**3 , -2*X[0]+10*X[1]-10])
    return diff
def f_H(X):
    H = np.array([[(12*(X[0]-1)**2)-2, -2], [-2, 10]])
    return H
```

Now the functions f, f_diff, and f_H are vectorized Python functions on the form that, for example, `optimize.fmin_ncg` expects, and we can proceed with a numerical optimization of the problem at hand by calling this function. In addition to the functions that we have prepared, we also need to give a starting point for the Newton method. Here we use (0, 0) as the starting point.

[9]:
```
x_opt = optimize.fmin_ncg(f, (0, 0), fprime=f_diff, fhess=f_H)
```

```
Optimization terminated successfully.
         Current function value: -3.867223
         Iterations: 12
         Function evaluations: 15
         Gradient evaluations: 15
         Hessian evaluations: 12
```

[10]:
```
x_opt
```

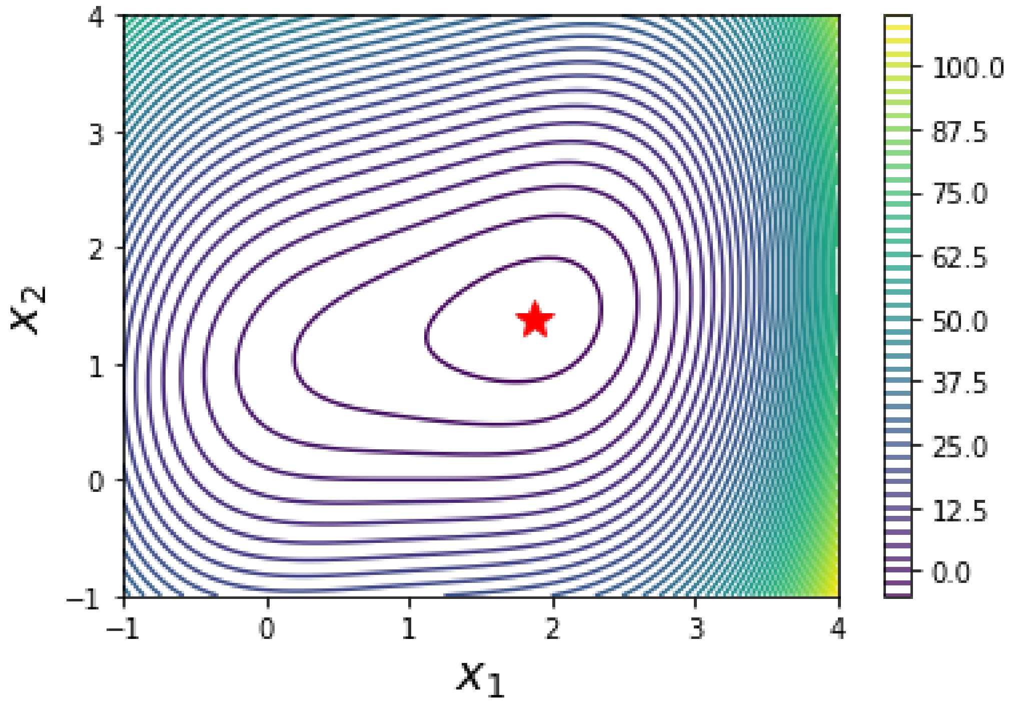
```
[10]: array([1.88292861, 1.37658572])
```

The routine found a minimum point at $(x_1, x_2) = (1.88292613, 1.37658523)$, and diagnostic information about the solution was also printed to standard output, including the number of iterations and the number of function, gradient, and Hessian evaluations that were required to arrive at the solution.

```
[53]: def f_plot(X,Y):
          return (X-1)**4 + 5*(Y-1)**2 - 2*X*Y
```

```
[12]: fig, ax = plt.subplots(figsize=(6, 4))
      x_ = y_ = np.linspace(-1, 4, 100)
      X, Y = np.meshgrid(x_, y_)
      c = ax.contour(X, Y, f_plot(X, Y), 50)
      ax.plot(x_opt[0], x_opt[1], 'r*', markersize=15)
      ax.set_xlabel(r"$x_1$", fontsize=18)
      ax.set_ylabel(r"$x_2$", fontsize=18)
      plt.colorbar(c, ax=ax)
```

```
[12]: <matplotlib.colorbar.Colorbar at 0x176635ae548>
```



In practice, it may not always be possible to provide functions for evaluating both the gradient and the Hessian of the objective function, and often it is convenient with a solver that only requires function evaluations. For such cases, several methods exist to numerically estimate the gradient or the Hessian or both. Methods that approximate the Hessian are known as quasi-

Newton methods, and there are also alternative iterative methods that completely avoid using the Hessian. Two popular methods are the Broyden-Fletcher-Goldfarb-Shanno (BFGS) and the conjugate gradient methods, which are implemented in SciPy as the functions `optimize.fmin_bfgs` and `optimize.fmin_cg`.

The BFGS method is a quasi-Newton method that can gradually build up numerical estimates of the Hessian, and also the gradient, if necessary. The conjugate gradient method is a variant of the steepest descent method and does not use the Hessian, and it can be used with numerical estimates of the gradient obtained from only function evaluations. With these methods, the number of function evaluations that are required to solve a problem is much larger than for Newton's method, which on the other hand also evaluates the gradient and the Hessian. Both `optimize.fmin_bfgs` and `optimize. fmin_cg` can optionally accept a function for evaluating the gradient, but if not provided, the gradient is estimated from function evaluations.

The preceding problem given, which was solved with the Newton method, can also be solved using the `optimize.fmin_bfgs` and `optimize.fmin_cg`, without providing a function for the Hessian:

[13]:
```
x_opt = optimize.fmin_bfgs(f, (0, 0), fprime=f_diff)
```

```
Optimization terminated successfully.
         Current function value: -3.867223
         Iterations: 9
         Function evaluations: 13
         Gradient evaluations: 13
```

[14]:
```
x_opt
```

[14]:
```
array([1.88292645, 1.37658596])
```

[15]:
```
x_opt = optimize.fmin_cg(f, (0, 0), fprime=f_diff)
```

```
Optimization terminated successfully.
         Current function value: -3.867223
         Iterations: 8
         Function evaluations: 18
         Gradient evaluations: 18
```

[16]:
```
x_opt
```

[16]:
```
array([1.88292612, 1.37658523])
```

Note that here, as shown in the diagnostic output from the optimization solvers in the preceding section, the number of function and gradient evaluations is larger than for Newton's method. As already mentioned, both of these methods can also be used without providing a function for the gradient as well, as shown in the following example using the `optimize.fmin_bfgs` solver:

[17]:
```
x_opt = optimize.fmin_bfgs(f, (0, 0))
```

```
Optimization terminated successfully.
         Current function value: -3.867223
         Iterations: 9
         Function evaluations: 39
         Gradient evaluations: 13
```

[18]:
```
x_opt
```

[18]:
```
array([1.88292645, 1.37658596])
```

**Notes:**
In general, the BFGS method is often a good first approach to try, in particular if neither the gradient nor the Hessian is known. If only the gradient is known, then the BFGS method is still the generally recommended method to use, although the conjugate gradient method is often a competitive alternative to the BFGS method. If both the gradient and the Hessian are known, then Newton's method is the method with the fastest convergence in general. However, it should be noted that although the BFGS and the conjugate gradient methods theoretically have slower convergence than Newton's method, they can sometimes offer improved stability and can therefore be preferable.
Each iteration can also be more computationally demanding with Newton's method compared to quasi-Newton methods and the conjugate gradient method, and especially for large problems, these methods can be faster in spite of requiring more iterations.

### 1.3.1 Getting Global Minimum

The methods for multivariate optimization that we have discussed so far all converge to a local minimum in general. For problems with many local minima, this can easily lead to a situation when the solver easily gets stuck in a local minimum, even if a global minimum exists. Although there is no complete and general solution to this problem, a practical approach that can partially alleviate this problem is to use a brute force search over a coordinate grid to find a suitable starting point for an iterative solver. At least this gives a systematic approach to find a global minimum within given coordinate ranges. In SciPy, the function `optimize.brute` can carry out such a systematic search.

**Exercise 3** To illustrate this method, consider the problem of minimizing the function

$$4 sin x\pi + 6 sin y\pi + (x-1)^2 + (y-1)^2$$

which has a large number of local minima. This can make it tricky to pick a suitable initial point for an iterative solver. To solve this optimization problem with SciPy, we first define a Python function for the objective function:

[66]:
```
def f(X):
    x, y = X
    return (4 * np.sin(np.pi * x) + 6 * np.sin(np.pi * y)) + (x - 1)**2 + (y -␣
 ↪1)**2
```

To systematically search for the minimum over a coordinate grid, we call optimize. brute with the objective function f as the first parameter and a tuple of slice objects as the second argument, one for each coordinate. The slice objects specify the coordinate grid over which to search for a minimum value. Here we also set the keyword argument finish=None, which prevents the optimize.brute from automatically refining the best candidate.

[67]:
```
x_start = optimize.brute(f, (slice(-3, 5, 0.5), slice(-3, 5, 0.5)), finish=None)
x_start
```

[67]:
```
array([1.5, 1.5])
```

[68]:
```
f(x_start)
```

[68]:
```
-9.5
```

On the coordinate grid specified by the given tuple of slice objects, the optimal point is $(x_1, x_2) = (1.5, 1.5)$, with corresponding objective function minimum -9.5. This is now a good starting point for a more sophisticated iterative solver, such as optimize. `fmin_bfgs`:

[69]:
```
x_opt = optimize.fmin_bfgs(f, x_start)
```

```
Optimization terminated successfully.
         Current function value: -9.520229
         Iterations: 4
         Function evaluations: 21
         Gradient evaluations: 7
```

[70]:
```
x_opt
```

[70]:
```
array([1.47586906, 1.48365787])
```

[71]:
```
f(x_opt)
```

[71]:
```
-9.520229273055016
```

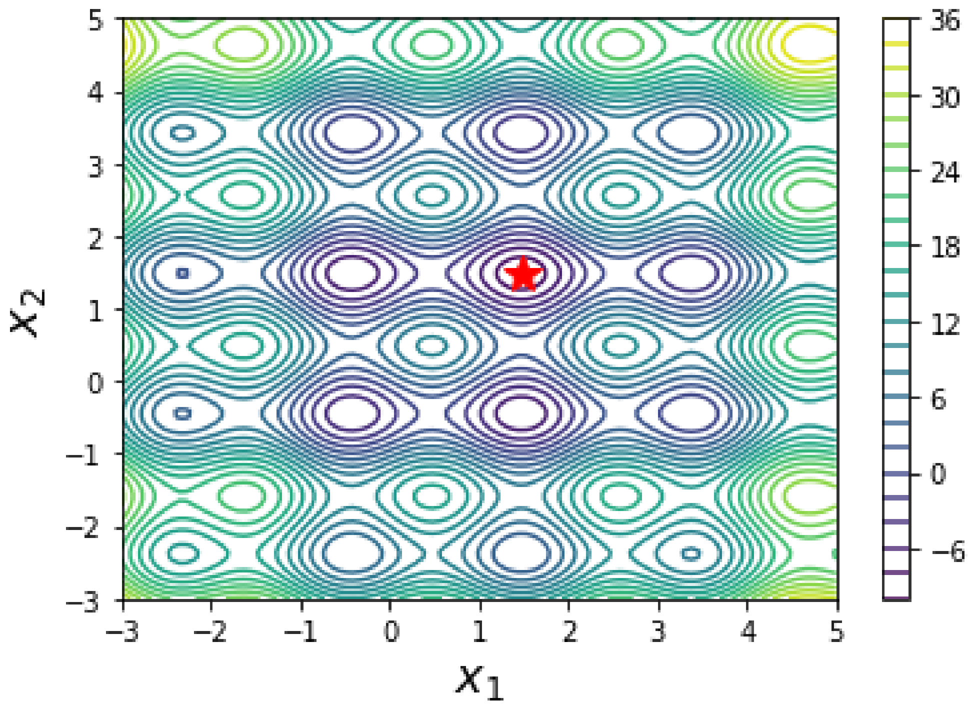
Here the BFGS method gave the final minimum point (x1,x2) = (1.47586906,1.48365788), with the minimum value of the objective function -9.52022927306. For this type of problem, guessing the initial starting point easily results in that the iterative solver converges to a local minimum, and the systematic approach that optimize.brute provides is frequently useful.

[72]:
```
def func_X_Y_to_XY(f, X, Y):
    """
    Wrapper for f(X, Y) -> f([X, Y])
    """
    s = np.shape(X)
    return f(np.vstack([X.ravel(), Y.ravel()])).reshape(*s)
```

[73]:
```
fig, ax = plt.subplots(figsize=(6, 4))
x_ = y_ = np.linspace(-3, 5, 100)
X, Y = np.meshgrid(x_, y_)
```

```
c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 25)
ax.plot(x_opt[0], x_opt[1], 'r*', markersize=15)
ax.set_xlabel(r"$x_1$", fontsize=18)
ax.set_ylabel(r"$x_2$", fontsize=18)
plt.colorbar(c, ax=ax)
```

```
[73]: <matplotlib.colorbar.Colorbar at 0x17665325d48>
```



In this section, we have explicitly called functions for specific solvers, for example, optimize.fmin_bfgs. However, like for scalar optimization, SciPy also provides a unified interface for all multivariate optimization solver with the function `optimize.minimize`, which dispatches out to the solver-specific functions depending on the value of the method keyword argument (remember, the univariate minimization function that provides a unified interface is `optimize.scalar_minimize`). For clarity, here we have favored explicitly calling functions for specific solvers, but in general it is a good idea to use optimize.minimize, as this makes it easier to switch between different solvers.

For example, to solve the previous example, we could have used the following:

```
[74]: result = optimize.minimize(f, x_start, method= 'BFGS')
      result
```

```
[74]:       fun: -9.520229273055016
      hess_inv: array([[2.41596001e-02, 4.61008275e-06],
      [4.61008275e-06, 1.63490348e-02]])
     jac: array([-7.15255737e-07, -7.15255737e-07])
 message: 'Optimization terminated successfully.'
    nfev: 21
     nit: 4
    njev: 7
  status: 0
 success: True
       x: array([1.47586906, 1.48365787])
```

The optimize.minimize function returns an instance of `optimize.OptimizeResult` that represents the result of the optimization. In particular, the solution is available via the x attribute of this class.

[75]:
```
result.x
```

[75]:
```
array([1.47586906, 1.48365787])
```

### 1.4 Nonlinear least square problems

In general, a least square problem can be viewed as an optimization problem with the objective function

$$g(\beta) = \sum_{i=0}^{m} r_i(\beta)^2 = ||r(\beta)||^2$$

where $r(\beta)$ is a vector with the residuals $r_i(\beta) = y_i - f(x_i, \beta)$ for a set of $m$ observations $(x_i, y_i)$. Here $\beta$ is a vector with unknown parameters that specifies the function $f(x, \beta)$.

If this problem is nonlinear in the parameters $\beta$, it is known as a nonlinear least square problem, and since it is nonlinear, it cannot be solved with the linear algebra techniques. Instead, we can use the multivariate optimization techniques described in the previous section, such as Newton's method or a quasi-Newton method. However, this nonlinear least square optimization problem has a specific structure, and several methods that are tailored to solve this particular optimization problem have been developed. One example is the Levenberg-Marquardt method, which is based on the idea of successive linearizations of the problem in each iteration.

In SciPy, the function optimize.leastsq provides a nonlinear least square solver that uses the Levenberg-Marquardt method. #### Exercise 4 To illustrate how this function can be used, consider a nonlinear model on the form

$$f(x, \beta) = \beta_0 + \beta_1 \exp\left(-\beta_2 x^2\right)$$

and a set of observations $(x_i, y_i)$.

In the following example, we simulate the observations with random noise added to the true values, and we solve the minimization problem that gives the best least square estimates of the parameters $\beta$. To begin with, we define a tuple with the true values of the parameter vector $\beta$ and a Python function for the model function. This function, which should return the $y$ value corresponding to a given $x$ value, takes as first argument the variable $x$, and the following arguments are the unknown function parameters:

```
[58]: beta = (0.25, 0.75, 0.5)
      def f(x, b0, b1, b2):
          return b0 + b1 * np.exp(-b2 * x**2)
```

Once the model function is defined, we generate randomized data points that simulate the observations.

```
[29]: xdata = np.linspace(0, 5, 50)
      y = f(xdata, *beta)
      ydata = y + 0.05 * np.random.randn(len(xdata))
```

With the model function and observation data prepared, we are ready to start solving the nonlinear least square problem. The first step is to define a function for the residuals given the data and the model function, which is specified in terms of the yet-to-be determined model parameters $\beta$

```
[60]: def g(beta):
          return ydata - f(xdata, *beta)
```

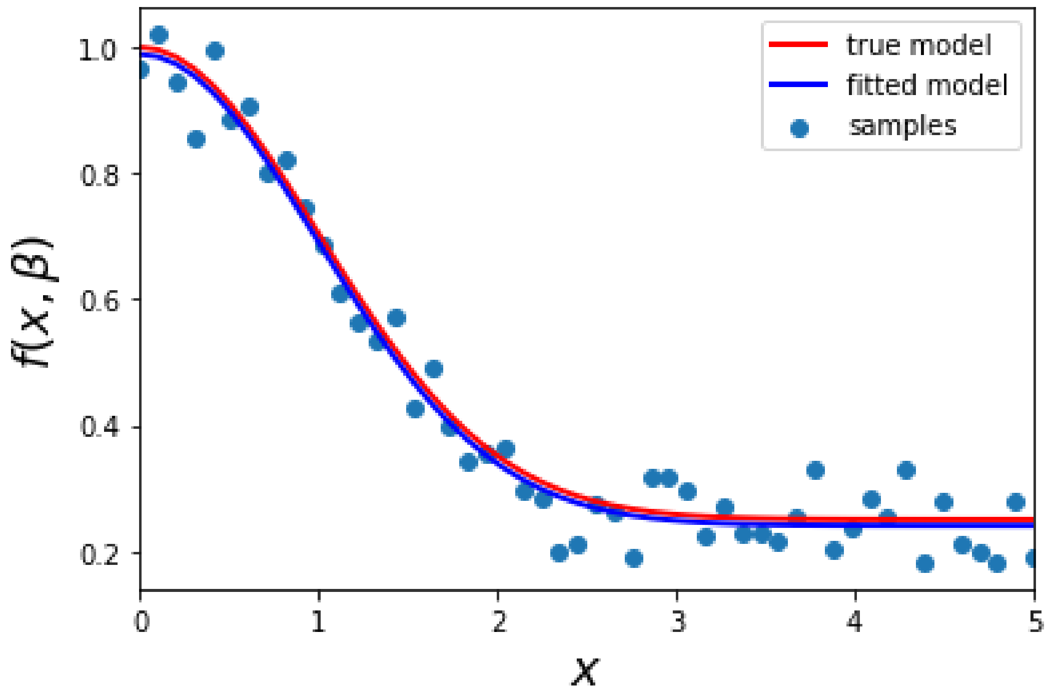
```
[61]: beta_start = (1, 1, 1)
      beta_opt, beta_cov = optimize.leastsq(g, beta_start)
      beta_opt
```

```
[61]: array([0.24065944, 0.74824422, 0.50705326])
```

Here the best fit is quite close to the true parameter values (0.25, 0.75, 0.5), as defined earlier. By plotting the observation data and the model function for the true and fitted function parameters, we can visually confirm that the fitted model seems to describe the data well

```
[62]: fig, ax = plt.subplots()
      ax.scatter(xdata, ydata, label='samples')
      ax.plot(xdata, y, 'r', lw=2, label='true model')
      ax.plot(xdata, f(xdata, *beta_opt), 'b', lw=2, label='fitted model')
      ax.set_xlim(0, 5)
      ax.set_xlabel(r"$x$", fontsize=18)
      ax.set_ylabel(r"$f(x, \beta)$", fontsize=18)
      ax.legend()
```

```
[62]: <matplotlib.legend.Legend at 0x176651b2a48>
```



The SciPy optimize module also provides an alternative interface to nonlinear least square fitting, through the function `optimize.curve_fit`. This is a convenience wrapper around `optimize.leastsq`, which eliminates the need to explicitly define the residual function for the least square problem. The previous problem could therefore be solved more concisely using the following:

[63]:
```
beta_opt, beta_cov = optimize.curve_fit(f, xdata, ydata)
beta_opt
```

[63]:
```
array([0.24065944, 0.74824422, 0.50705326])
```

## 1.5 Constrained Optimization

Constraints add another level of complexity to optimization problems, and they require a classification of their own. A simple form of constrained optimization is the optimization where the coordinate variables are subject to some bounds. For example: $min_x f(x)$ subject to $0 \leq x \leq 1$. The constraint $0 \leq x \leq 1$ is simple because it only restricts the range of the coordinate without dependencies on the other variables. This type of problems can be solved using the L-BFGS-B method in SciPy, which is a variant of the BFGS method we used earlier. This solver is available through the function `optimize.fmin_l_bgfs_b` or via `optimize.minimize` with the method argument set to `'L-BFGS-B'`. To define the coordinate boundaries, the bound keyword argument must be used, and its value should be a list of tuples that contain the minimum and maximum value of each constrained variable. If the minimum or maximum value is set to None, it is interpreted as an unbounded.

**Exercise 5** As an example of solving a bounded optimization problem with the L-BFGS-B solver, consider minimizing the objective function

$$f(x) = (x_1 - 1)^2 - (x_2 - 1)^2$$

subject to the constraints

$$2 \leq x_1 \leq 3 \; and \; 0 \leq x_2 \leq 2$$

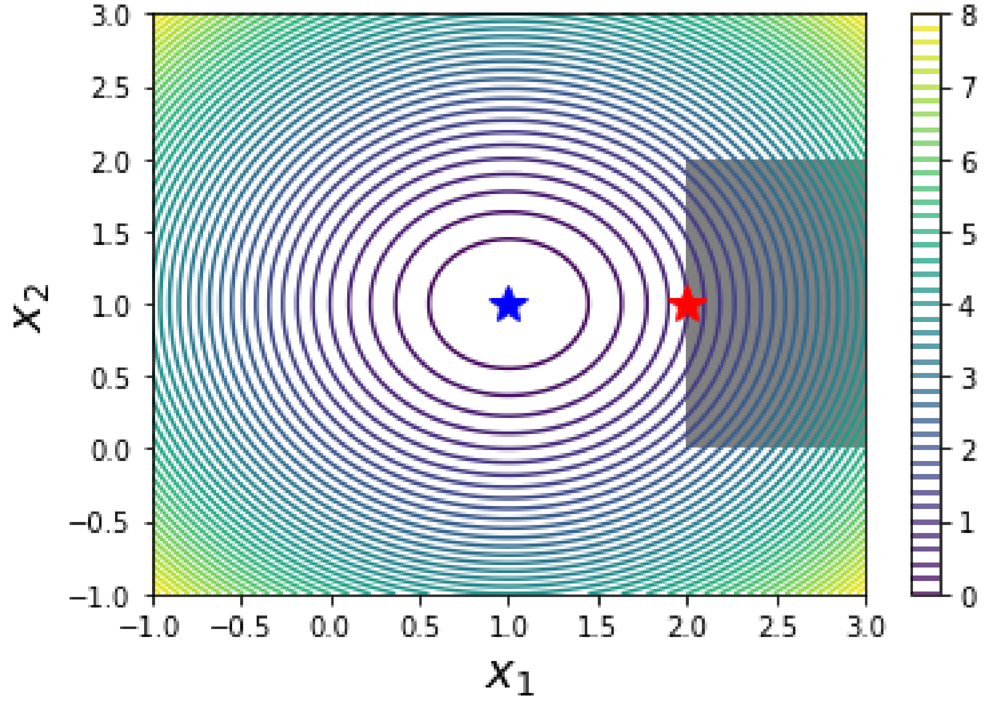
.

To solve this problem, we first define a Python function for the objective functions and tuples with the boundaries for each of the two variables in this problem, according to the given constraints. For comparison, in the following code, we also solve the unconstrained optimization problem with the same objective function, and we plot a contour graph of the objective function where the unconstrained and constrained minimum values are marked with blue and red stars, respectively

[78]:
```
def f(X):
    x, y = X
    return (x - 1)**2 + (y - 1)**2
x_opt = optimize.minimize(f, [1, 1], method='BFGS').x
bnd_x1, bnd_x2 = (2, 3), (0, 2)
x_cons_opt = optimize.minimize(f, [1, 1], method='L-BFGS-B',bounds=[bnd_x1,␣
 ↪bnd_x2]).x
```

[77]:
```
fig, ax = plt.subplots(figsize=(6, 4))
x_ = y_ = np.linspace(-1, 3, 100)
X, Y = np.meshgrid(x_, y_)
c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 50)
ax.plot(x_opt[0], x_opt[1], 'b*', markersize=15)
ax.plot(x_cons_opt[0], x_cons_opt[1], 'r*', markersize=15)
bound_rect = plt.Rectangle((bnd_x1[0], bnd_x2[0]),
                           bnd_x1[1] - bnd_x1[0], bnd_x2[1] -bnd_x2[0],␣
 ↪facecolor="grey")
ax.add_patch(bound_rect)
ax.set_xlabel(r"$x_1$", fontsize=18)
ax.set_ylabel(r"$x_2$", fontsize=18)
plt.colorbar(c, ax=ax)
```

[77]:
```
<matplotlib.colorbar.Colorbar at 0x17665486348>
```



Constraints that are defined by equalities or inequalities that include more than one variable are somewhat more complicated to deal with. However, there are general techniques also for this type of problems. One of these techniques is Lagrange multipliers. This method can be extended to handle inequality constraints as well, and there exist various numerical methods of applying this approach. One example is the method known as sequential least square programming, abbreviated as SLSQP, which is available in the SciPy as the `optimize.slsqp` function and via `optimize.minimize` with `method='SLSQP'`. The optimize.minimize function takes the keyword argument constraints, which should be a list of dictionaries that each specifies a constraint. The allowed keys (values) in this dictionary are type ('eq' or 'ineq'), fun (constraint function), jac (Jacobian of the constraint function), and args (additional arguments to constraint function and the function for evaluating its Jacobian).

**Exercise 6** To illustrate this technique, consider the problem of maximizing the volume of a rectangle with sides of length $x_0$, $x_1$, and $x_2$, subject to the constraint that the total surface area should be unity:

$$g(x) = 2x_1x_2 + 2x_0x_2 + 2x_1x_0 - 1 = 0$$

.

[36]:
```
def f(X):
    return -X[0] * X[1] * X[2]
def g(X):
    return 2 * (X[0]*X[1] + X[1] * X[2] + X[2] * X[0]) - 1
```

Note that since the SciPy optimization functions solve minimization problems, and here we are interested in maximization, the function $f$ is here the negative of the original objective function. Next we define the constraint dictionary for $g(x) = 0$ and finally call the optimize.minimize function

[37]:
```
constraint = dict(type='eq', fun=g)
result = optimize.minimize(f, [0.5, 1, 1.5],␣
↪method='SLSQP',constraints=[constraint])
result
```

[37]:
```
     fun: -0.06804136862287297
     jac: array([-0.16666925, -0.16666542, -0.16666526])
 message: 'Optimization terminated successfully'
    nfev: 77
     nit: 18
    njev: 18
  status: 0
 success: True
       x: array([0.40824188, 0.40825127, 0.40825165])
```

[38]:
```
result.x
```

[38]:
```
array([0.40824188, 0.40825127, 0.40825165])
```

To solve problems with inequality constraints, all we need to do is to set `type='ineq'` in the constraint dictionary and provide the corresponding inequality function.

**Exercise 7** To demonstrate minimization of a nonlinear objective function with a nonlinear inequality constraint, we return to the quadratic problem

$$(x_0 - 1)^2 + (x_1 - 1)^2$$

but in this case with inequality constraint

$$g(x) = x_1 - 1.75 - (x_0 - 0.75)^4 \geq 0$$

.
As usual, we begin by defining the objective function and the constraint function, as well as the constraint dictionary:

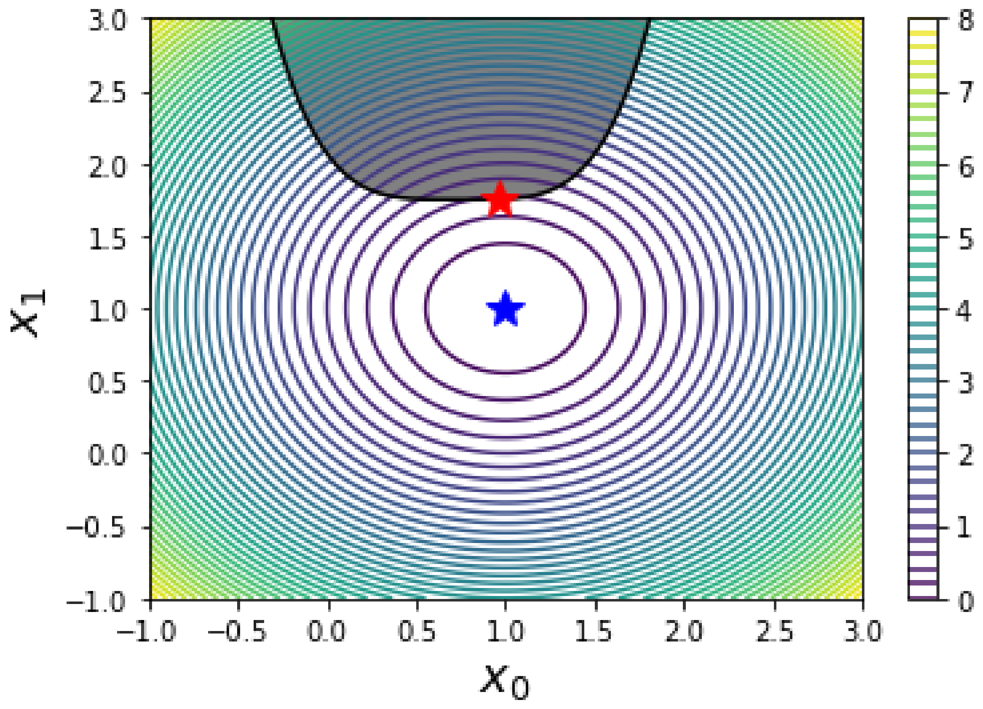
[79]:
```
def f(X):
    return (X[0] - 1)**2 + (X[1] - 1)**2
def g(X):
    return X[1] - 1.75 - (X[0] - 0.75)**4

constraints = [dict(type='ineq', fun=g)]
```

Next, we are ready to solve the optimization problem by calling the `optimize.minimize` function. For comparison, here we also solve the corresponding unconstrained problem.

```
[80]: x_opt = optimize.minimize(f, (0, 0), method='BFGS').x
      x_cons_opt = optimize.minimize(f, (0, 0),␣
       ↪method='SLSQP',constraints=constraints).x
```

```
[41]: fig, ax = plt.subplots(figsize=(6, 4))
      x_ = y_ = np.linspace(-1, 3, 100)
      X, Y = np.meshgrid(x_, y_)
      c = ax.contour(X, Y, func_X_Y_to_XY(f, X, Y), 50)
      ax.plot(x_opt[0], x_opt[1], 'b*', markersize=15)
      ax.plot(x_, 1.75 + (x_-0.75)**4, 'k-', markersize=15)
      ax.fill_between(x_, 1.75 + (x_-0.75)**4, 3, color='grey')
      ax.plot(x_cons_opt[0], x_cons_opt[1], 'r*', markersize=15)
      ax.set_ylim(-1, 3)
      ax.set_xlabel(r"$x_0$", fontsize=18)
      ax.set_ylabel(r"$x_1$", fontsize=18)
      plt.colorbar(c, ax=ax)
```

```
[41]: <matplotlib.colorbar.Colorbar at 0x17664e26688>
```



For optimization problems with only inequality constraints, SciPy provides an alternative solver using the constrained optimization by linear approximation (COBYLA)method. This solver is accessible either through `optimize.fmin_cobyla` or `optimize. minimize` with `method='COBYLA'`. The previous example could just as well have been solved with this solver, by replacing method='SLSQP' with method='COBYLA'.

## 1.6 Linear Programming

In the previous section, we considered methods for very general optimization problems, where the objective function and constraint functions all can be nonlinear. However, at this point it is worth taking a step back and considering a much more restricted type of optimization problem, namely, linear programming, where the objective function is linear and all constraints are linear equality or inequality constraints. The class of problems is clearly much less general, but it turns out that linear programming has many important applications, and they can be solved vastly more efficiently than general nonlinear problems. The reason for this is that linear problems have properties that enable completely different methods to be used.
In particular, the solution to a linear optimization problem must necessarily lie on a constraint boundary, so it is sufficient to search the vertices of the intersections of the linear constraint functions. This can be done efficiently in practice. A popular algorithm for this type of problems is known as simplex, which systematically moves from one vertex to another until the optimal vertex has been reached. There are also more recent interior point methods that efficiently solve linear programming problems. With these methods, linear programming problems with thousands of variables and constraints are readily solvable.

Linear programming problems are typically written in the so-called standard form: $min_x c^T x$ where $Ax \leq b$ and $x \geq 0$. Here $c$ and $x$ are vectors of length $n$, and $A$ is a $mn$ matrix and $b$ a $m$-vector.

**Exercise 8** For example, consider the problem of minimizing the function

$$f(x) = -x_0 + 2x_1 - 3x_2$$

, subject to the three inequality constraints

$$x_0 + x_1 \leq 1, \; -x_0 + 3x_1 \leq 2, \; and \; -x_1 + x_2 \leq 3$$

. On the standard form, we have $c = (-1, 2, -3)$, $b = (1, 2, 3)$, and A = $\begin{pmatrix} 1 & 1 & 0 \\ -1 & 3 & 0 \\ 0 & -1 & 1 \end{pmatrix}$

To solve this problem, here we use the cvxopt library, which provides the linear programming solver with the `cvxopt.solvers.lp` function. This solver expects as arguments the $c$, $A$, and $b$ vectors and matrix used in the standard form introduced in the preceding text, in the given order. The `cvxopt` library uses its own classes for representing matrices and vectors, but fortunately they are interoperable with NumPy arrays via the array interface and can therefore be cast from one form to another using the `cvxopt.matrix` and `np.array` functions.

To solve the stated example problem using the cvxopt library, we therefore first create NumPy arrays for the $A$ matrix and the $c$ and $b$ vectors and convert them to `cvxopt matrices` using the `cvxpot.matrix` function:

[82]:
```
c = np.array([-1.0, 2.0, -3.0])
A = np.array([[ 1.0, 1.0, 0.0], [-1.0, 3.0, 0.0],[ 0.0, -1.0, 1.0]])
b = np.array([1.0, 2.0, 3.0])
A_ = cvxopt.matrix(A)
b_ = cvxopt.matrix(b)
c_ = cvxopt.matrix(c)
```

The `cvxopt` compatible matrices and vectors c_, A_, and b_ can now be passed to the linear programming solver `cvxopt.solvers.lp`:

```
[83]: sol = cvxopt.solvers.lp(c_, A_, b_)
```

```
Optimal solution found.
```

```
[84]: sol
```

```
[84]: {'x': <3x1 matrix, tc='d'>,
       'y': <0x1 matrix, tc='d'>,
       's': <3x1 matrix, tc='d'>,
       'z': <3x1 matrix, tc='d'>,
       'status': 'optimal',
       'gap': 0.0,
       'relative gap': 0.0,
       'primal objective': -10.0,
       'dual objective': -10.0,
       'primal infeasibility': 0.0,
       'primal slack': -0.0,
       'dual slack': 2.7755575615628914e-17,
       'dual infeasibility': 2.967195843610875e-17,
       'residual as primal infeasibility certificate': None,
       'residual as dual infeasibility certificate': None,
       'iterations': 0}
```

```
[45]: x = np.array(sol['x'])
      x
```

```
[45]: array([[0.25],
             [0.75],
             [3.75]])
```

```
[46]: sol['primal objective']
```

```
[46]: -10.0
```

The solution to the optimization problem is given in terms of the vector x, which in this particular example is $x = (0.25, 0.75, 3.75)$, which corresponds to the $f(x)$ value $-10$. With this method and the `cvxopt.solvers.lp` solver, linear programming problems with hundreds or thousands of variables can readily be solved. All that is needed is to write the optimization problem on the standard form and create the c, A, and b arrays.