Introduction to Optimization

by Loïc Quertenmont, PhD

LINFO1113 - 2019-2020

Programme

Cours	1	Librairies mathématiques, représentation des nombres en Python et erreurs liées
Cours	2,3	Résolution des systèmes linéaires
Cours	4,5	Interpolation et Régression Linéaires
Cours	6,7	Racines d'équations
Cours	8	Différentiation numérique
Cours	9	Intégration numérique
Cours	10	Equations Différentielles Ordinaires
Cours	**11**	**Introduction à l'optimisation**
Cours	12,13	Rappel / Répétition

Outline

- **Introduction**
- **Minimization along a line**
- **Powell's Method**
- **Downhill Simplex method**
- **Gradient Descent**

Introduction

Introduction

- Goal:
  - Find $\boldsymbol{x}$ that minimizes $F(\boldsymbol{x})$
  - Satisfying the equality constraint: $g(\boldsymbol{x}) = 0$
  - Satisfying the inequality constraint: $h(\boldsymbol{x}) \geq 0$

- Optimization is the term often used for minimizing or maximizing a function.
  - We can consider minimization only:
  - Maximization of $F(\boldsymbol{x})$ is equivalent to minimizing $-F(\boldsymbol{x})$.

Introduction

- The function $F(\boldsymbol{x})$,
  - Called the merit or objective function
    - Similar to loss/cost function in machine learning
  - Is the quantity that we wish to keep as small as possible, such as the cost or weight.

- The components of $\boldsymbol{x}$,
  - Called the design variables
  - Are the quantities that we are free to adjust.
    - Physical dimensions (lengths, areas, angles, and so on) are common examples of design variables.

Introduction

- Optimization is a large topic
  - Many books dedicated to it.

- We will just see an introduction
  - With a few methods for gentle problems
    - Reasonably well behaved
    - Not involving too many design variables.
  - By omitting the more sophisticated methods, we may actually not miss all that much.
    - All optimization algorithms are unreliable to a degree—any one may work on one problem and fail on another.
    - As a rule of thumb, by going up in sophistication we gain computational efficiency, but not necessarily reliability

Starting conditions

- Algorithms for minimization are iterative

- Require starting values $\boldsymbol{x_0}$ of the design variables $\boldsymbol{x}$.
  - If $F(\boldsymbol{x})$ has several local minima,
    $\boldsymbol{x_0}$ determines which of these will be computed.

  - No guaranteed to find the global optimal point.

  - One suggested procedure:
    - Run the optimization multiple times
      using different starting points
      and pick the best result.

Constrained optimization

- Often, $\boldsymbol{x}$ is also subjected to restrictions, or constraints
  - May have the form of equalities or inequalities.

- As an example:
  - Take the minimum weight design of a roof truss (“charpente”) that has to carry a certain loading.
  - Assume that the layout of the members is given, so that the design variables are the cross-sectional areas of the members.
  - Here the design is dominated by inequality constraints that consist of prescribed upper limits on the stresses and possibly the displacements

Constrained vs Unconstrained (1)

- Unconstrained optimization
  - Most optimization techniques are designed for this case
  - Minima are stationary points (points where $\nabla F(\boldsymbol{x}) = 0$)

- Constrained optimization
  - Minima are generally located where the $F(\boldsymbol{x})$ surface meets the constraints.
  - There are special algorithms for constrained optimization, but they are not easily accessible because of their complexity and specialization.
  - An easy approach, that can work in some cases, is to transform the problem into an unconstrained optimization problem

Constrained vs Unconstrained (2)

- Transforming Constrained → Unconstrained
  - Use an unconstrained optimization algorithm
  - but to modify the merit function so that any violation of constrains is heavily penalized

  - Consider minimizing $F(\boldsymbol{x})$ with the following constraints:
    - $g_i(\boldsymbol{x}) = 0$ for $i = 1,2,\dots,M$
    - $h_j(\boldsymbol{x}) \geq 0$ for $j = 1,2,\dots,N$

  - We can choose a new merit function

$$F^*(\boldsymbol{x}) = F(\boldsymbol{x}) + \lambda\, P(\boldsymbol{x}) \quad \text{with} \quad P(\boldsymbol{x}) = \sum_{\boldsymbol{i}}^{\boldsymbol{M}} (g_i(\boldsymbol{x}))^{\boldsymbol{2}} + \sum_{\boldsymbol{j}}^{\boldsymbol{N}} \max(0, h_j(\boldsymbol{x}))^{\boldsymbol{2}}$$

  - $P(\boldsymbol{x})$ is a penalty function and λ is a multiplier.

Constrained vs Unconstrained (3)

$$F^*(\boldsymbol{x}) = F(\boldsymbol{x}) + \lambda\, P(\boldsymbol{x}) \quad \text{with} \quad P(\boldsymbol{x}) = \sum_{\boldsymbol{i}}^{\boldsymbol{M}} (g_i(\boldsymbol{x}))^{\boldsymbol{2}} + \sum_{\boldsymbol{j}}^{\boldsymbol{N}} \max(0, h_j(\boldsymbol{x}))^{\boldsymbol{2}}$$

- The constraints are satisfied when $P(\boldsymbol{x}) = \boldsymbol{0}$
- If a constraint is violated:
  - Penalty proportional to the square of the violation
  - → Algorithm tries to avoid violation
  - The degree of avoidance being dependent on the magnitude of λ
  - If λ is small:
    - Optimization goes faster, because there is more “space” to operate
    - But constraint violation can be large
  - If λ is large:
    - Constraints are enforced
    - May result in a poorly conditioned procedure
  - Procedure:
    - Start with a small λ,
    - if solution is unacceptable w.r.t constraint,
      increase λ and use previous solution as a starting point
  - Problem if the conditions have widely different magnitudes
    - The offending constrained can be scaled

Non-Iterative Procedure

- If derivatives of $F(\boldsymbol{x})$ can be computed AND there is no inequality constraints
  - The optimal point can be obtained by calculus (without an iterative procedure)

- **If there is no constraints:**
  - The optimal point is the solution of the (non-linear) system of equations:

$$\boldsymbol{\nabla F(x) = 0}$$

- **If there are (only) equality constraints : $\boldsymbol{g_i(x) = 0}$ for $\boldsymbol{i = 1, 2, \dots, M}$**
  - We define: $F^*(\boldsymbol{x}) = F(\boldsymbol{x}) + \sum_{\boldsymbol{i}}^{\boldsymbol{M}} \lambda_i \, g_i(\boldsymbol{x})$
  - And solve the System: $\boldsymbol{\nabla F^*(x) = 0}$ for both $\boldsymbol{x}$ and λ_i's
  - The λ_i's are known as *Lagrange Multipliers*

- **For both equality and inequality constraints:**
  - The technique can also be extended to inequality constraints, but the solution of the resulting equations is not straightforward due to a lack of uniqueness.

Linear Programming

- Problems where the objective function $F(\boldsymbol{x})$ and the constraints $g(\boldsymbol{x})$ and $h(\boldsymbol{x})$ are linear functions of $\boldsymbol{x}$.
  - This is known as Linear Programming
  - **This will not be discussed in this lecture**
  - This can be solved easily using dedicated methods (Simplex method)
  - This is used mainly for operations research and cost analysis;
  - There are very few engineering applications.
  - Linear programming can also be useful in nonlinear optimization.
    - Many method rely in part on the simplex method.
    - Nonlinear constraints can often be solved by piecewise application of linear programming.
    - The simplex method is also used to compute search directions in the method of feasible directions.

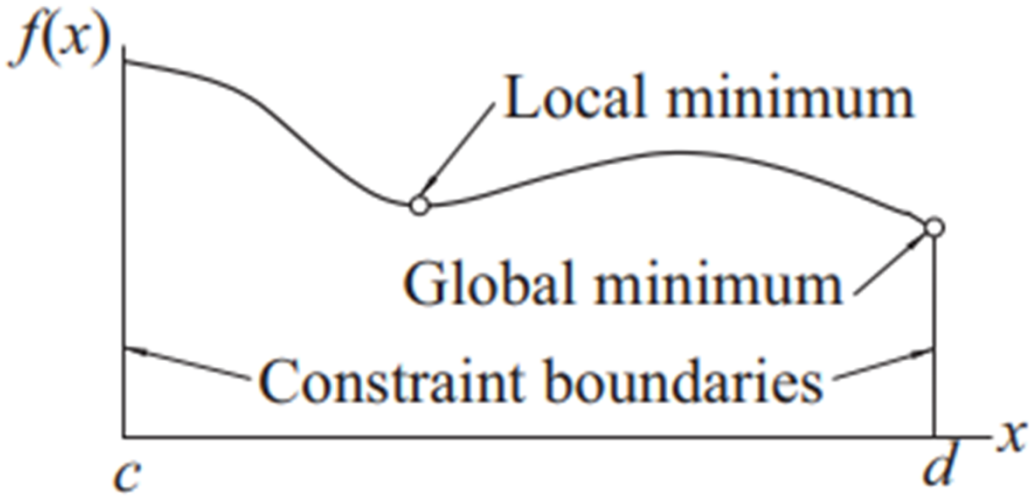
Minimization Along a Line

Minimization along a line (1)

- Minimizing a function f(x)
- Single variable x
- Constraints c ≤ x ≤ d.

- **Minimums are given by $f'(x) = 0$**

- **Two minimums**
  - Local
  - Global (at boundary d)



Minimization along a line (2)

- Finding the global minimum
  - Easy
  - Check all stationary points $f'(x) = 0$
  - Check all the constraint boundaries

- But then, why do we need an optimization algorithm?
  - We need one if $f(x)$ is difficult to differentiate
    - If $f(x)$ is a computer program for instance

Bracketing

- Before a minimization algorithm can be used, the minimum point must be bracketed.
  - Exactly like for root searching
- Bracketing procedure is simple:
  - Start with an initial value of x_0 and move downhill computing the function at x_1, x_2, x_3, ... until we reach the point x_n where $f(x)$ increases for the first time.
  - The minimum point is now bracketed in the interval (x_{n-2}, x_n).
- What should the step size $h_i = x_{i+1} - x_i$ be?
  - Constant h_i often requires too many steps.
  - Better to increase the size at every step,ie:

$$h_{i+1} = ch_i \quad \text{with } c > 1$$

  - Wide bracketing is OK

Bracketing

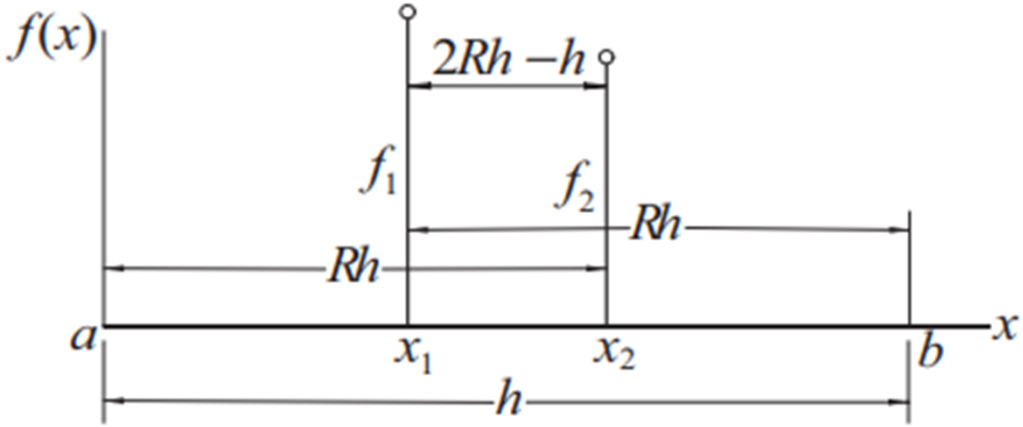
- Can you implement the algorithm ?

```
import math
def bracket(f,x1,h):
    c = 1.618033989
    f1 = f(x1)
    x2 = x1 + h; f2 = f(x2)
  # Determine downhill direction and change sign of h if needed
    if f2 > f1:
        h = -h
        x2 = x1 + h; f2 = f(x2)
      # Check if minimum between x1 - h and x1 + h
        if f2 > f1: return x2,x1 - h
  # Search loop
    for i in range (100):
        h = c*h
        x3 = x2 + h; f3 = f(x3)
        if f3 > f2: return x1,x3
        x1 = x2; x2 = x3
        f1 = f2; f2 = f3
    print("Bracket did not find a mimimum")
```

Golden Section Search (1)

- Counterpart of bisection used in finding roots of $f(x)$.

- Minimum of $f(x)$ is bracketed in $[a, b]$ of length h.
- To telescope the interval, we evaluate the function at
  - $x_1 = b - Rh$
  - $x_2 = a + Rh$
  - The constant R is to be determined shortly.

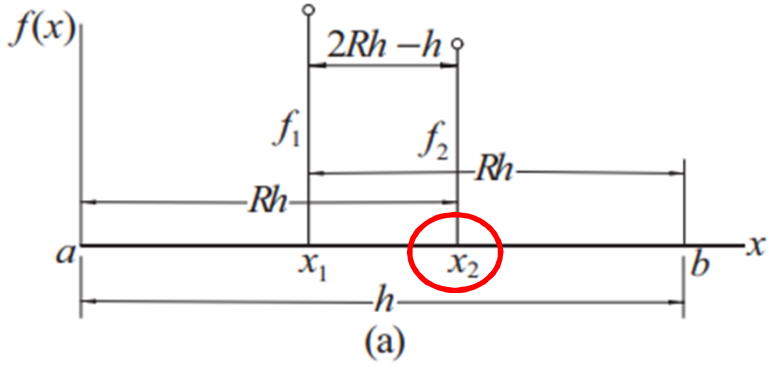


- If $f_1 > f_2$, the minimum lies in $[x_1, b]$;
- otherwise it is located in $[a, x_2]$.

Golden Section Search (2)

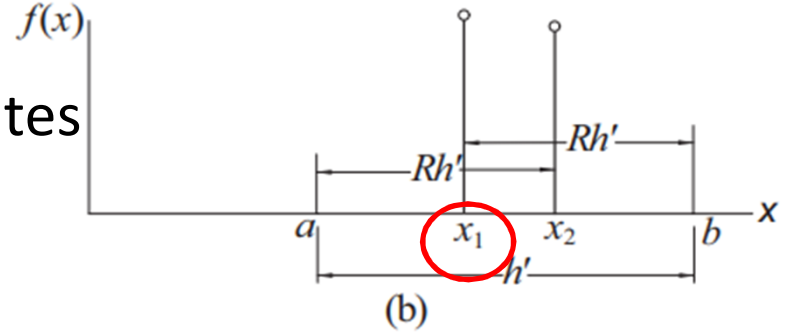
- Assuming $f_1 > f_2$:
  - $a \leftarrow x_1$ and $x_1 \leftarrow x_2$
  - The new interval $[a, b]$ has now the length $h' = Rh$

- To do a new telescoping, we need to evaluate the function at
  - $x_2 = a + Rh'$
  - No need to reevaluate at x_1, because we already did it at the first telescoping



(a)

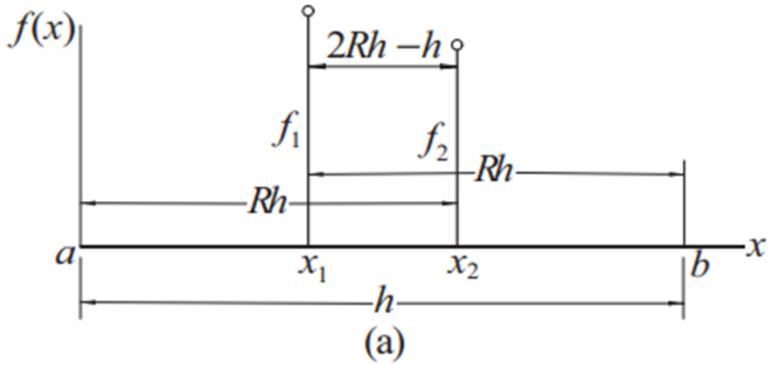
  - This works only if the figures are similar (if the same constant R locates x1 and x2 in both figures)



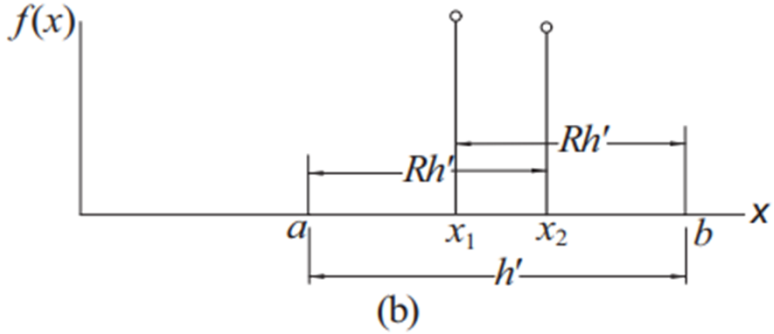
(b)

Golden Section Search (3)

- Similar figures:
  - In the first figure, the overlap region is defined by:
    $x_2 - x_1 = 2Rh - h$
  - The same distance in the second figure is:
    $x_1 - a = h' - Rh'$



  - *Equating the right side:*
    $2Rh - h = h' - Rh'$
  - Replacing h' by Rh
    $2R - 1 = R(1 - R)$



  - The solution is given by:
    $R = \frac{-1+\sqrt{5}}{2} = 0.61^{8}\,03^{3}\,989$ is the **golden ratio**

Golden Section Search (4)

- Remarks
  - Each telescoping reduces the interval length by a factor $R = 0.618$
    - This is not as good as the 0.5 of the bisection algorithm in root search
  - However, the golden search method achieves this reduction with one function evaluation
  - The number of telescoping operations required to reduce h from $|b - a|$ to an error tolerance ε is given by

$$|b - a|\, R^n = \varepsilon$$

  - Which yield:

$$n = \frac{\ln(\varepsilon/|b-a|)}{\ln(R)} = -2.078\,087 \ln\left(\frac{\varepsilon}{|b-a|}\right)$$

Golden Section Search (5)

- Can you implement the algorithm ?

```
def search(f,a,b,tol=1.0e-9):
    nIter = int(math.ceil(-2.078087*math.log(tol/abs(b-a))))
    R = 0.618033989
    C = 1.0 - R
  # First telescoping
    x1 = R*a + C*b; x2 = C*a + R*b
    f1 = f(x1); f2 = f(x2)
  # Main loop
    for i in range(nIter):
        if f1 > f2:
            a = x1
            x1 = x2; f1 = f2
            x2 = C*a + R*b; f2 = f(x2)
        else:
            b = x2
            x2 = x1; f2 = f1
            x1 = R*a + C*b; f1 = f(x1)
    if f1 < f2: return x1,f1
    else: return x2,f2
```

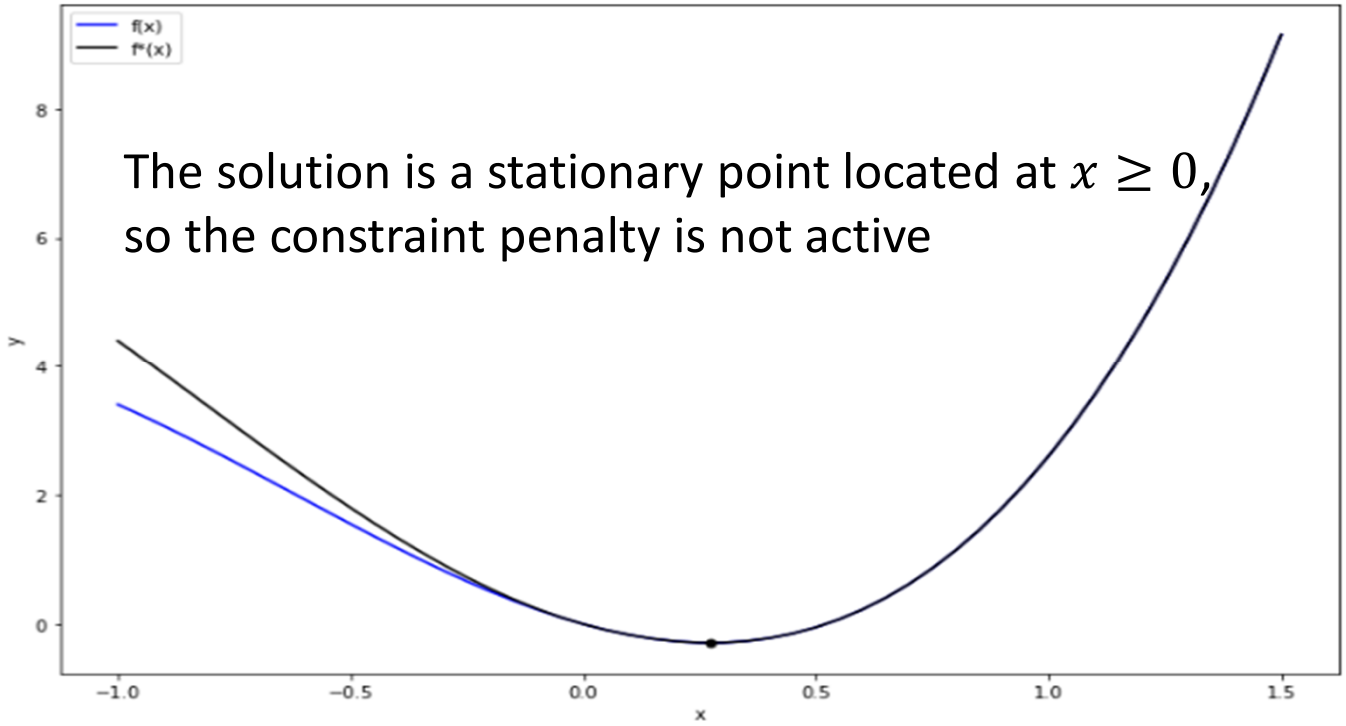
Example 1(1)

- Use golden search to find x that minimizes:
  - $f(x) = 1.6x^3 + 3x^2 - 2x$
  - With $x \geq 0$
- Solution
  - The minimum is either a stationary point with $x \geq 0$ or is at the constraint boundary $x = 0$
  - $f^*(x) = f(x) + \lambda[\min(0, x)]^2$
  - $f(x) = 1.6x^3 + 3x^2 - 2x$

```
def f(x):
    return 1.6*x**3 + 3.0*x**2 - 2.0*x

def fstar(x):
    lam = 1.0 # Constraint multiplier
    c = np.minimum(0, x) # Constraint function
    return f(x) + lam*c**2

xStart = 1.0
h      = 0.01
x1,x2  = bracket(fstar,xStart,h)
print("bracket", x1, x2)

x,fMin = search(fstar,x1,x2)
print("x =",x)
print("f(x) =",fMin)
```

bracket = [0.546393, -0.213738]
x_min = 0.27349
f(x_min) = -0.289859

Example 1(2)



- Analytical solution can be obtained easily by solving

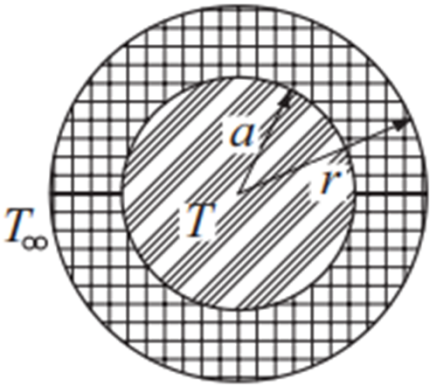
$$f'(x) = 4.8x^2 + 6x - 2 = 0$$

- Solution is $x = 0.273^4 94$
  - This is the only positive root
- The other possibility for the global minimum is x=0 (constraint boundary)
  - But f(0) is larger than f at the stationary point
- So the global minimum is $x = 0.2734^9$

Example 2 (1)

A wire carrying an electric current is surrounded by rubber insulation of outer radius r. The resistance of the wire generates heat, which is conducted through the insulation and convected into the surrounding air. The temperature of the wire can be shown to be

$$T = \frac{q}{2\pi}\left(\frac{\ln(r/a)}{k} + \frac{1}{hr}\right) + T_\infty$$



where

q = rate of heat generation in wire = 50 W/m

a = radius of wire = 5 mm

k = thermal conductivity of rubber = 0.16 W/m · K

h = convective heat-transfer coefficient = 20 W/m^2 · K

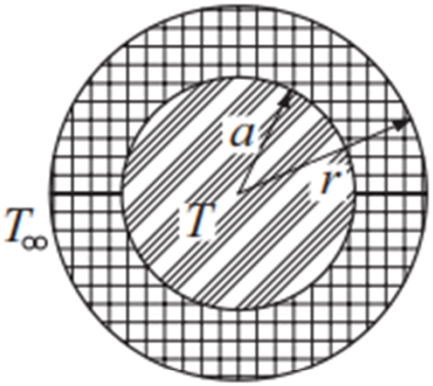
T_∞ = ambient temperature = 280 K

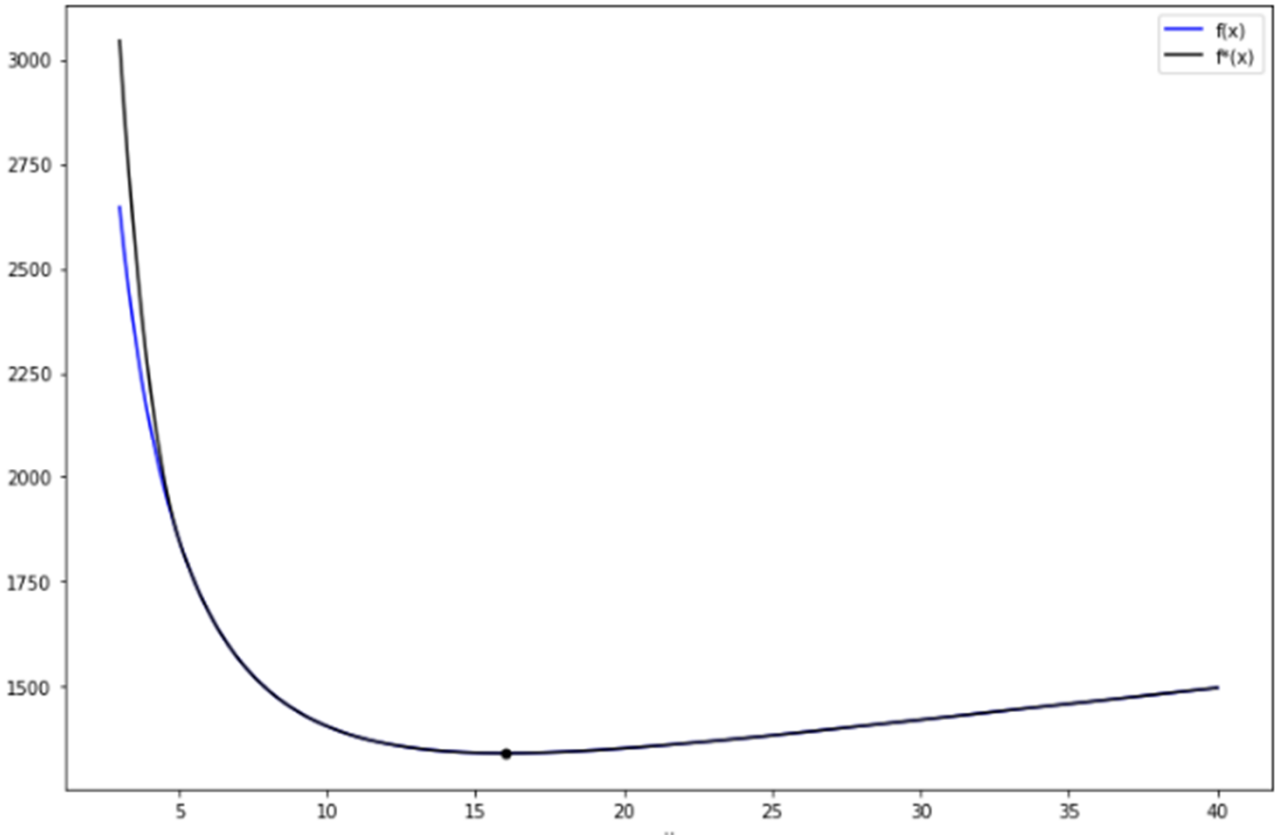
Find r that minimizes T.

```
def f(r):
    return (q/2*np.pi)*((np.log(r/a)/k) + (1/(h*r))  ) + Tinf

def fstar(x):
    lam = 100.0 # Constraint multiplier
    c = np.minimum(0, x-a) # Constraint function
    return f(x) + lam*c**2

xStart = a+1.0
h      = 0.01
x1,x2  = bracket(fstar,xStart,h)
print("bracket", x1, x2)

x,fMin = search(fstar,x1,x2)
print("x =",x)
print("f(x) =",fMin)
```

Example 2 (2)

- bracket = [11.19383; 19.6238]
- x = 15.99999
- f(x) = 1341.83





Powell's Method

Powell's method (1)

- Golden Search works fine for one-dimensional problems
- What if we want to optimize multiple variable at once?
- The objective is to minimize $F(\boldsymbol{x})$, where the components of $\boldsymbol{x}$ are the n independent design variables
- One way to tackle the problem is to use a succession of one-dimensional minimizations to close in on the optimal point

Powell's method (2)

- The procedure is this one:
  - Choose a point $\boldsymbol{x_0}$ in the design space.
  - Loop over $i = 1, 2, 3, \ldots$:
    - Choose a vector $\boldsymbol{v_i}$.
    - Minimize F($\mathbf{x}$) along the line through $\boldsymbol{x_{i-1}}$ in the direction of $\boldsymbol{v_i}$.
      - Let the minimum point be $\boldsymbol{x_i}$.
    - If $| \boldsymbol{x_i} - \boldsymbol{x_{i-1}} | < \varepsilon$ exit loop

- The minimization along a line can be done using one-dimensional optimization method (golden search)
- How to choose the vectors $\boldsymbol{v_i}$?

Conjugate Directions (1)

- Consider the quadratic function
  - $F(\boldsymbol{x}) = c - \sum_i b_i x_i + \frac{1}{2} \sum_i \sum_j A_{ij} x_i x_j$
  - $F(\boldsymbol{x}) = c - \boldsymbol{b}^{\boldsymbol{T}} \boldsymbol{x} + \frac{1}{2} \boldsymbol{x}^{\boldsymbol{T}} \boldsymbol{A} \boldsymbol{x}$ (using matrix notation)

- Differentiation along x_i is:
  - $\frac{\partial F}{\partial x_i} = -b_i + \sum_j A_{ij} x_j$
  - $\boldsymbol{\nabla} F = -\boldsymbol{b} + \boldsymbol{A} \boldsymbol{x}$ (using matrix notation)

- $\boldsymbol{\nabla} F$ is the gradient of F

Conjugate Directions (2)

- Consider the change of ∇F when we move from point $\boldsymbol{x_0}$ in the direction of a vector $\boldsymbol{u}$ (moved distance is s)

$$\boldsymbol{x} = \boldsymbol{x_0} + s\boldsymbol{u}$$

- The gradient at $\boldsymbol{x}$ is then

$$\nabla F_{|\boldsymbol{x_0}+s\boldsymbol{u}} = -\boldsymbol{b} + \boldsymbol{A}(\boldsymbol{x_0} + s\boldsymbol{u})$$

$$\nabla F_{|\boldsymbol{x_0}+s\boldsymbol{u}} = \nabla F_{|\boldsymbol{x_0}} + s\boldsymbol{A}\boldsymbol{u}$$

  - The change in the gradient is $s\boldsymbol{A}\boldsymbol{u}$

- If the change is perpendicular to a vector **v**; (equivalent to say If $\boldsymbol{v}^T\boldsymbol{A}\boldsymbol{u} = 0$), the directions of **u** and **v** are said to be mutually conjugate (non-interfering)
- The implication is that once we have minimized **F(x)** in the direction of **v**, we can move along **u** without ruining the previous minimization

Conjugate Directions (3)

- For a quadratic function of n independent variables
  It is possible to construct n mutually conjugate directions.

- Therefore, it would take precisely n line minimizations along these directions to reach the minimum point.

- If $F(\boldsymbol{x})$ is not a quadratic function, the quadratic development can be treated as a local approximation of the objective function, obtained by truncating the Taylor series expansion of $F(\boldsymbol{x})$ about $\boldsymbol{x_0}$

$$F(\mathbf{x}) \approx F(\mathbf{x}_0) + \nabla F(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

  - Now the conjugate directions based on the quadratic form are only approximations, valid in the close vicinity of $\boldsymbol{x_0}$. Consequently, it would take several cycles of n line minimizations to reach the optimal point.

Conjugate Directions (4)

- The various conjugate gradient methods use different techniques for constructing conjugate directions.

- The zero-order methods work with $F(\boldsymbol{x})$ only,

- The first-order methods use both $F(\boldsymbol{x})$ and $\boldsymbol{\nabla} F$.
  - The first-order methods are computationally more efficient, of course, but the input of $\boldsymbol{\nabla} F$, if it is available at all, can be very tedious.

Powell's Algorithm

- Powell's method is a zero-order method, only using $F(\boldsymbol{x})$.
- The basic algorithm is as follows:

  - Start:
    - Choose a point $\boldsymbol{x_0}$ in the design space.
    - Choose the starting vectors $\boldsymbol{v_i}$
      (often $\boldsymbol{v_i} = \boldsymbol{e_i}$ = unit vector in the direction of component $\boldsymbol{x_i}$)

  Initialization

  - Cycle:
    - Loop over $i = 1, 2, 3, \ldots$:
      - Minimize F(**x**) along the line through $\boldsymbol{x_{i-1}}$ in the direction of $\boldsymbol{v_i}$.
      - Let the minimum point be $\boldsymbol{x_i}$.
    - $\boldsymbol{v_{n+1}} \leftarrow \boldsymbol{x_0} - \boldsymbol{x_n}$

  Build a new conjugate vector

    - Minimize F(**x**) along the line through $\boldsymbol{x_0}$ in the direction of $\boldsymbol{v_{n+1}}$.
    - Let the minimum point be $\boldsymbol{x_{n+1}}$.
    - If $| \boldsymbol{x_{n+1}} - \boldsymbol{x_0} | < \varepsilon$ exit loop

  Minimize along the new vector

    - Loop over $i = 1, 2, 3, \ldots$:
      - $\boldsymbol{v_i} \leftarrow \boldsymbol{v_{i+1}}$ ($\boldsymbol{v_1}$ is discarded; the other vectors are reused)
    - $\boldsymbol{x_0} \leftarrow \boldsymbol{x_{n+1}}$

  Reset variables for a new cycle

Powell's Algorithm

- Remarks:

- The vectors $\boldsymbol{v_{n+1}}$ produced in successive cycles are mutually conjugate.
  - The minimum point of a quadratic surface is reached in precisely n cycles.
  - In practice, the merit function is seldom quadratic, but as long as it can be approximated locally by a quadratic function, Powell's method will work.
  - Of course, it usually takes more than n cycles to arrive at the minimum of a non-quadratic function.

- Note that it takes n line minimizations to construct each conjugate direction.

Illustration (one cycle)

- **In two dimension (2 vectors)**

1. At start, we have:
   - $\boldsymbol{x_0}, \boldsymbol{v_1}, \boldsymbol{v_2}$

2. We find the distance s_1 that minimizes $\boldsymbol{F}(\boldsymbol{x_0} + s_1\boldsymbol{v_1})$
   - Moving to $\boldsymbol{x_1} = \boldsymbol{x_0} + s_1\boldsymbol{v_1}$

3. We find the distance s_2 that minimizes $\boldsymbol{F}(\boldsymbol{x_1} + s_2\boldsymbol{v_2})$
   - Moving to $\boldsymbol{x_2} = \boldsymbol{x_1} + s_2\boldsymbol{v_2}$

4. We find the new search direction $\boldsymbol{v_3}$
   - $\boldsymbol{v_3} = \boldsymbol{x_2} - \boldsymbol{x_0}$

5. We find the distance s_3 that minimizes $\boldsymbol{F}(\boldsymbol{x_0} + s_3\boldsymbol{v_3})$
   - Moving to $\boldsymbol{x_3} = \boldsymbol{x_0} + s_3\boldsymbol{v_3}$

6. We finished the cycle

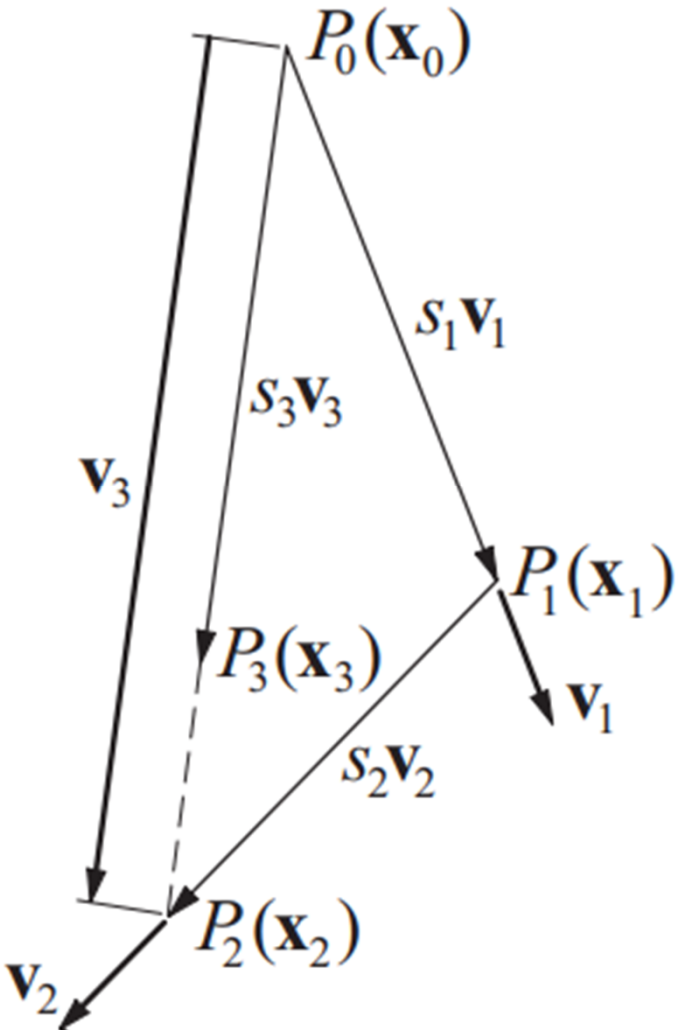
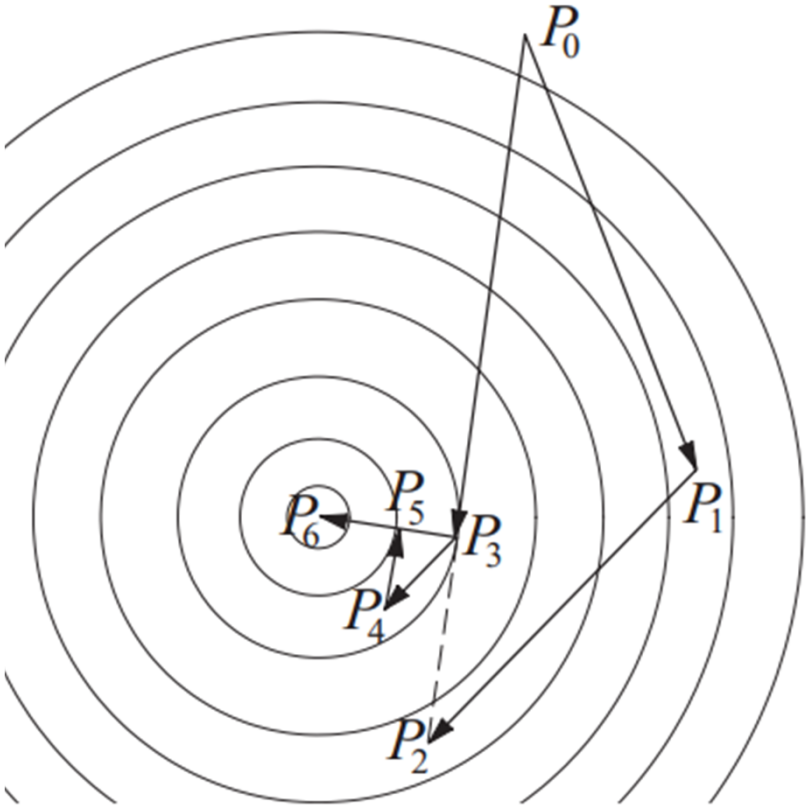


Illustration (second cycle)

- **In two dimension (2 vectors)**
- First cycle moved the point from $\boldsymbol{P_0}$ to $\boldsymbol{P_3}$
- Second cycle moved the point from $\boldsymbol{P_3}$ to $\boldsymbol{P_6}$
- $\boldsymbol{P_6}$ is the optimal point
- The directions $\boldsymbol{P_0P_3}$ and $\boldsymbol{P_3P_6}$ are mutually conjugate



Remarks

- Powell's method does have a major issue

- if $F(\boldsymbol{x})$ is not a quadratic, the algorithm tends to produce search directions that gradually become linearly dependent, thereby ruining the progress toward the minimum.

- The source of the problem is the automatic discarding of v_1 at the end of each cycle.
  - It is better to throw out the direction that resulted in the largest decrease of $F(\boldsymbol{x})$.
  - It seems counterintuitive to discard the best direction, but it is likely to be close to the direction added in the next cycle, thereby contributing to linear dependence.
  - As a result of the change, the search directions cease to be mutually conjugate, so that a quadratic form is no longer minimized in n cycles. This is not a significant loss because in practice $F(\boldsymbol{x})$ is seldom a quadratic.

Algorithm

```
import numpy as np
from goldSearch import *
import math

def powell(F,x,h=0.1,tol=1.0e-6):

    def f(s): return F(x + s*v)    # F in direction of v

    n = len(x)                     # Number of design variables
    df = np.zeros(n)               # Decreases of F stored here
    u = np.identity(n)             # Vectors v stored here by rows
    for j in range(30):            # Allow for 30 cycles:
        xOld = x.copy()            # Save starting point
        fOld = F(xOld)
      # First n line searches record decreases of F
        for i in range(n):
            v = u[i]
            a,b = bracket(f,0.0,h)
            s,fMin = search(f,a,b)
            df[i] = fOld - fMin
            fOld = fMin
            x = x + s*v
      # Last line search in the cycle
        v = x - xOld
        a,b = bracket(f,0.0,h)
        s,fLast = search(f,a,b)
        x = x + s*v
      # Check for convergence
        if math.sqrt(np.dot(x-xOld,x-xOld)/n) < tol: return x,j+1
      # Identify biggest decrease & update search directions
        iMax = np.argmax(df)
        for i in range(iMax,n-1):
            u[i] = u[i+1]
        u[n-1] = v
    print("Powell did not converge")
```

Example 3 (1)

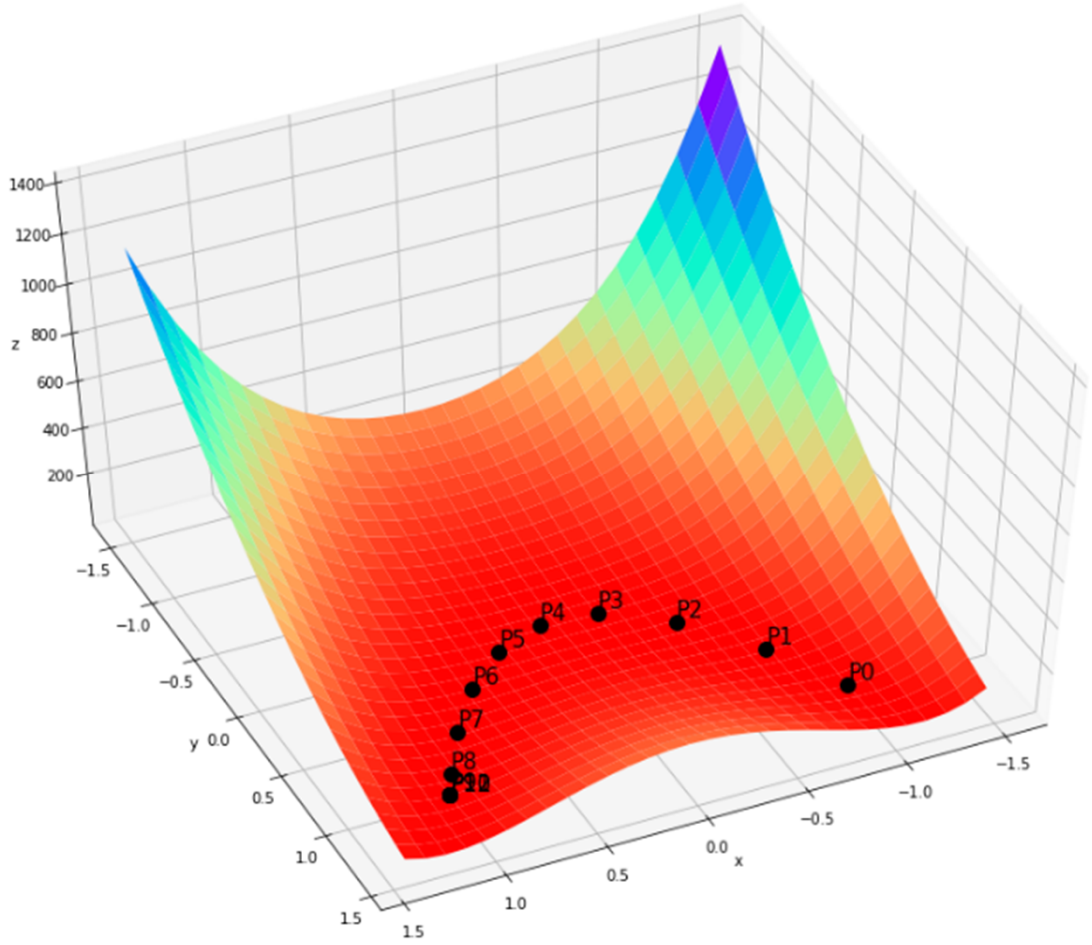
- Find the minimum of
  - $F = 10^0(y - x^2)^2 + (1 - x)^2$

```
def F(x):
    return 100.0*(x[1] - x[0]**2)**2 + (1 - x[0])**2

xStart = np.array([-1.0, 1.0])
sequences = []
xMin,nIter = powell(F,xStart, sequences=sequences)

print("x =",xMin)
print("F(x) =",F(xMin))
print("Number of cycles =",nIter)
```

```
x = [1. 1.]
F(x) = 3.7175070158540l8e-29
Number of cycles = 12
```

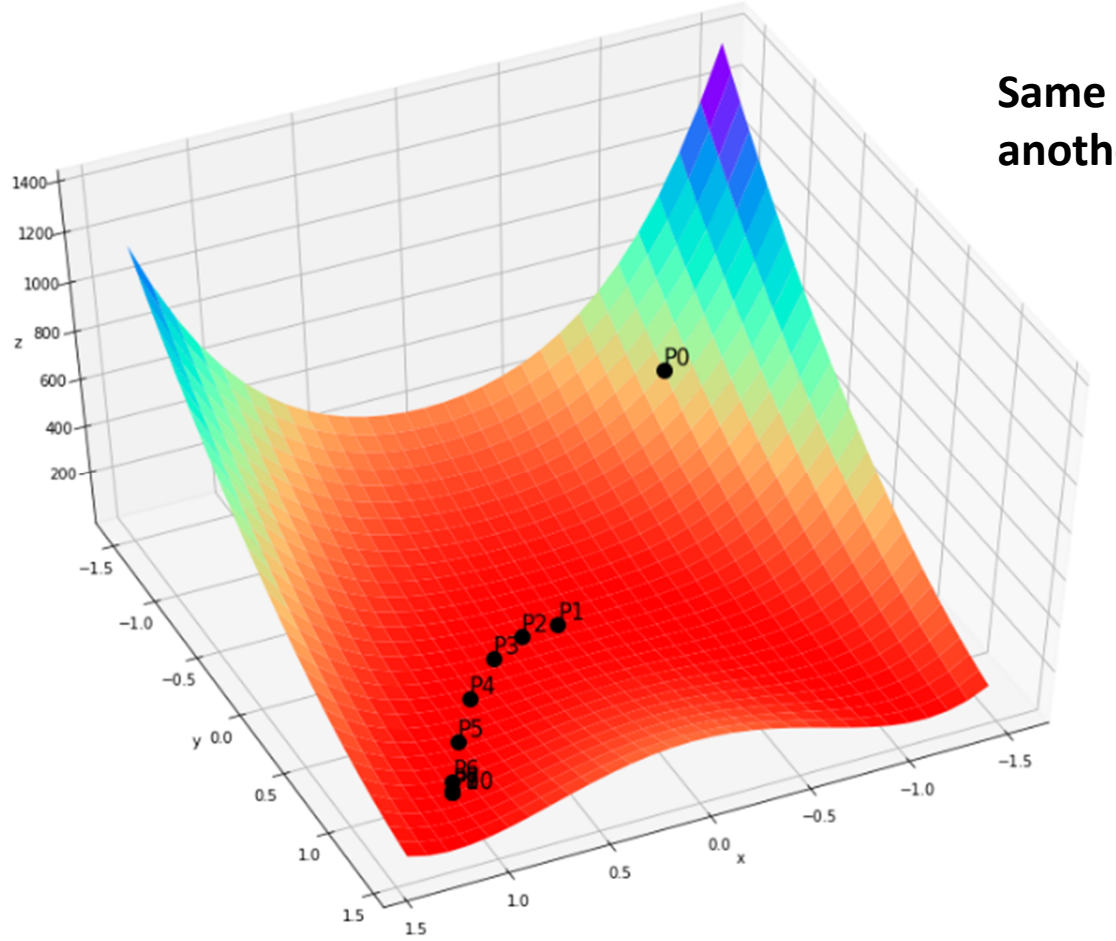
Example 3 (2)

- Find the minimum of
  - $F = 10^0(y - x^2)^2 + (1 - x)^2$



Example 3 (2)

- Find the minimum of
  - $F = 10^0(y - x^2)^2 + (1 - x)^2$



Same thing using another starting point

Example 4 (1)

- Use Powell to determine the smallest distance from the point (5, 8) to the curve xy = 5.
- This is a constrain optimization problem
  - $F(x, y) = (x - 5)^2 + (y - 8)^2$
  - With $xy - 5 = 0$
- $F^*(x, y) = (x - 5)^2 + (y - 8)^2 + \lambda(xy - 5)^2$

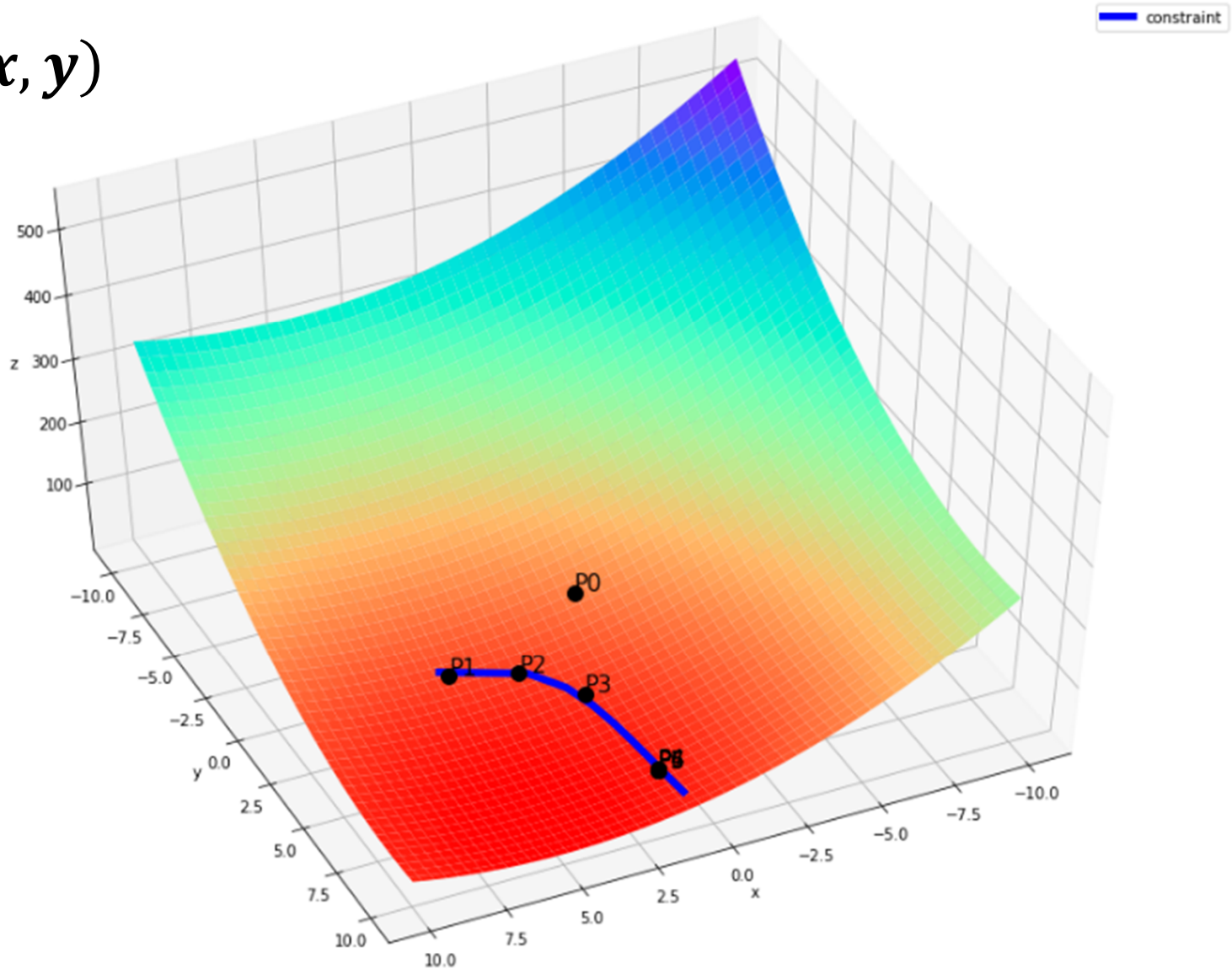
```
def F(x):
    return (x[0]-5)**2 + (x[1]-8)**2

def Fstar(x):
    lam = 1.0 # Constraint multiplier
    c = (x[0]*x[1])-5 # Constraint function
    return F(x) + lam*c**2


xStart = np.array([1.0, 5.0])
sequences = []
xMin,nIter = powell(Fstar,xStart, sequences=sequences)

print("x =",xMin)
print("F(x) =",F(xMin))
print("Number of cycles =",nIter)
```

executed in 20ms, finished 13:35:08 2019-11-22

```
x = [0.73306761 7.58776385]
F(x) = 18.37665070484751
Number of cycles = 5
```

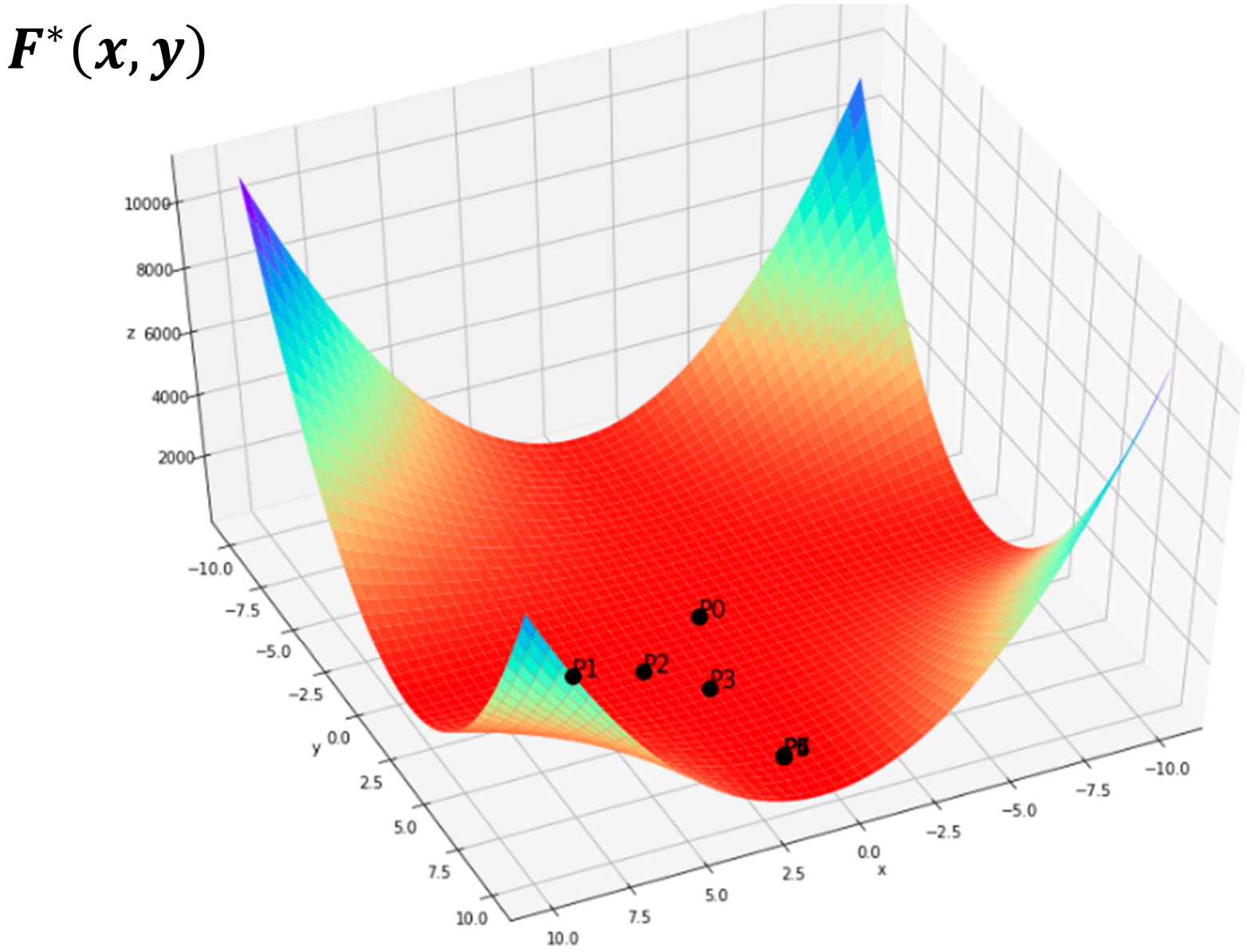
Example 4 (2)

$\boldsymbol{F(x, y)}$



Example 4 (3)

$F^*(x, y)$



Example 4 (4)

- Let's check again the solution
  - x=0.73306759
  - y=7.58776401
- **xy = 5.56**
  - Not fulfilling the constraint
  - The small value of λ favored speed of convergence over accuracy. Because the violation of the constraint xy = 5 is clearly unacceptable, we ran the program again with λ = 10 000 and changed the starting point to (0.73306761, 7.58776385), the end point of the first run
  - x = 0.65561311, y=7.62653598 →xy=5.0000

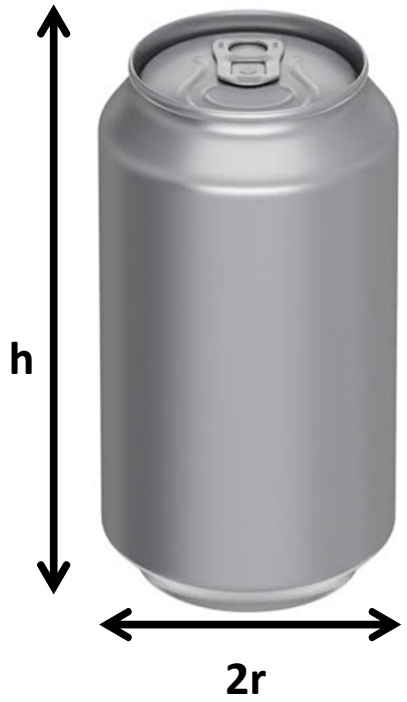
Example 5 (1)

- Find out the shape of the cheapest cylinder can, that can contains 33cl of liquid.

- Aluminum thickness for top/bottom cap of the can is 0.015cm
- Aluminum thickness for the barrel of the can is 0.01cm
- Aluminum price is 1,34$/kg

Example 5 (2)

- Cap surface $= \pi r^2$
- Cap weight $= \pi r^2 * 0.01^5 * \rho$
- They are 2 caps

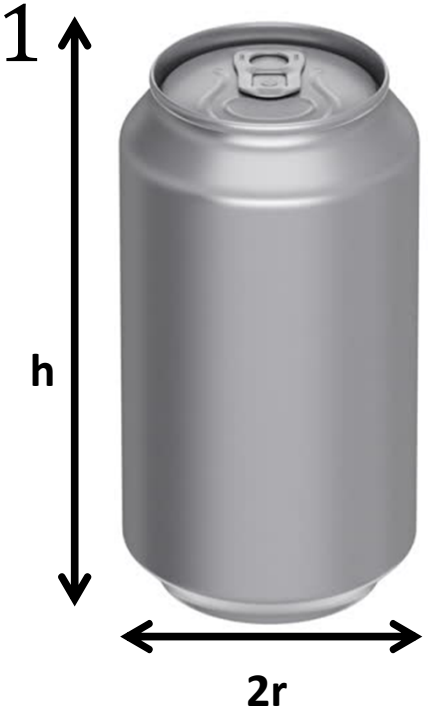


- Cylinder surface $= \mathrm{h} * 2\pi r$
- Cylinder weight $= \mathrm{h} * 2\pi r * 0.01 * \rho$

- Can volume $= \mathrm{h} * \pi r^2 \geq 33^0 \mathrm{cm}^3$

Example 5 (3)

- $F(r,h) = 2\pi r^2 * 0.01^5 + 2\pi rh * 0.01$
- With $\mathrm{h} * \pi r^2 \geq 33^0 \mathrm{cm}^3$

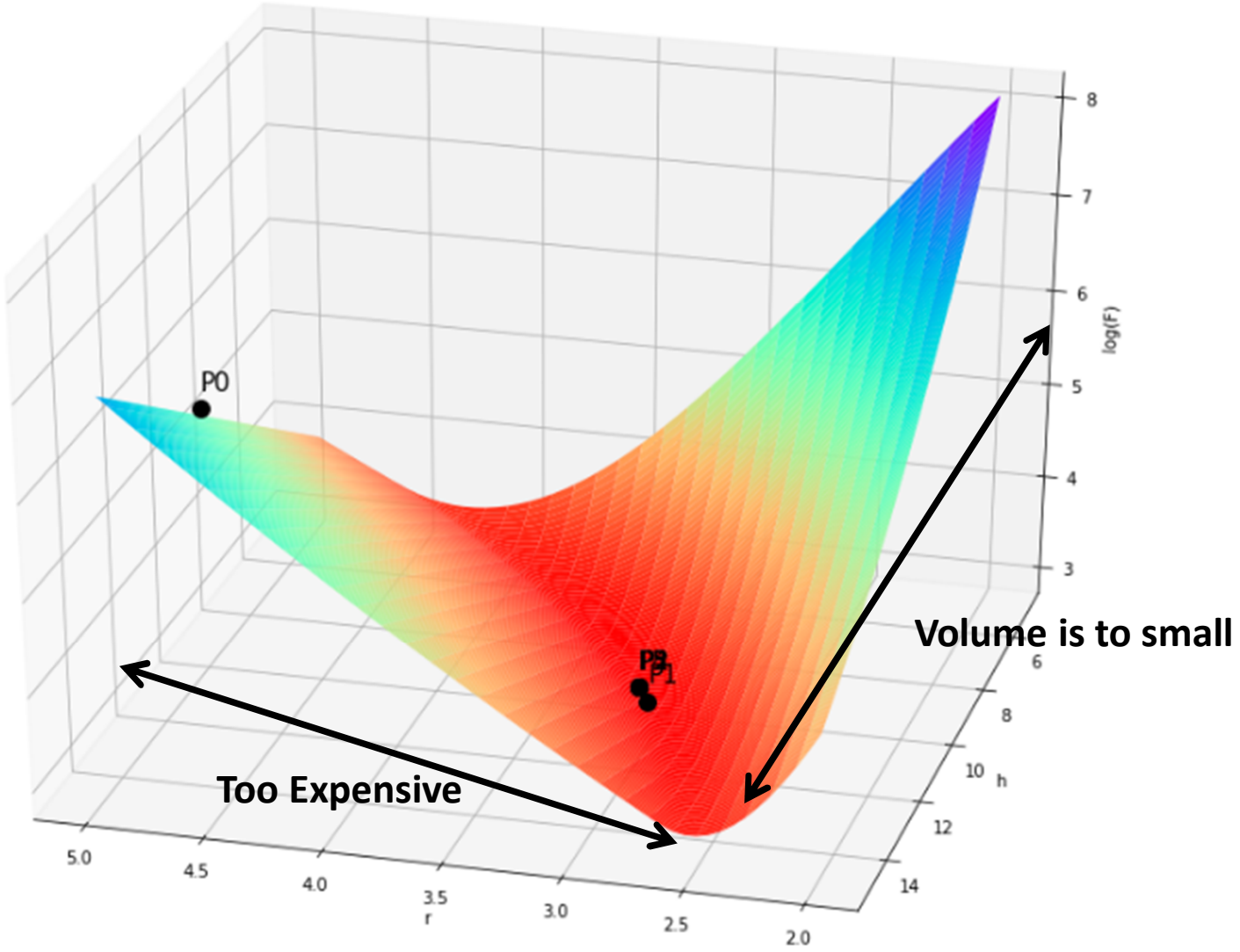



```
def F(x):
    r = x[0]
    h = x[1]
    return 2*np.pi*(0.015*r**2 + 0.01*r*h)

def volume(x):
    r = x[0]
    h = x[1]
    return np.pi*h*r**2

def Fstar(x):
    lam = 1.0E-4 #10000.0 # Constraint multiplier
    c = np.minimum(0,volume(x)-330) # Constraint function
    return F(x) + lam*c**2


xStart = np.array([5.0, 10.0])
#xStart = np.array([3.27049441, 9.81148308])
sequences = []
xMin,nIter = powell(Fstar,xStart, sequences=sequences)

print("x =",xMin)
print("F(x) =",F(xMin))
print("Number of cycles =",nIter)
print("Volume=", volume(xMin))
```

executed in 34ms, finished 14:51:05 2019-11-22

```
x = [3.16352213 9.49056672]
F(x) = 2.8296592821069595
Number of cycles = 5
Volume= 298.38966195574517
```

Example 5 (4)


P0
P1
Too Expensive
Volume is to small
log(F)
8
7
6
5
4
3
5.0
4.5
4.0
3.5
3.0
2.5
2.0
r
6
8
10
12
14
h

Downhill Simplex Method

Downhill Simplex Method

- The Downhill Simplex method
  - Also known as Nelder-Mead method

- Multi-variate non-linear optimization.
- It is a zero order method (not using derivative)

- Much slower than Powell's method in most cases, but makes up for it in robustness. It often works in problems where Powell's method hangs up.

- Nothing to do with the "Simplex" Method which is another method for **LINEAR** optimization

Principle (1)

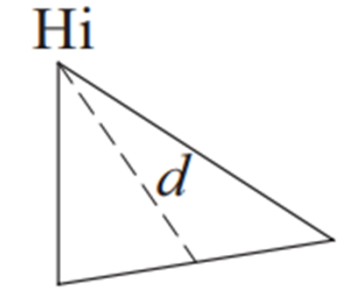
- "Bracket" the optimal point in the n-dimension space

- Bracketing is done using a simplex
  - A simplex in n-dimensional space is a figure of n + 1 vertices connected by straight lines and bounded by polygonal faces
  - In 1D, a simplex is a line
  - In 2D, a simplex is a triangle
  - In 3D, a simplex is a tetrahedron

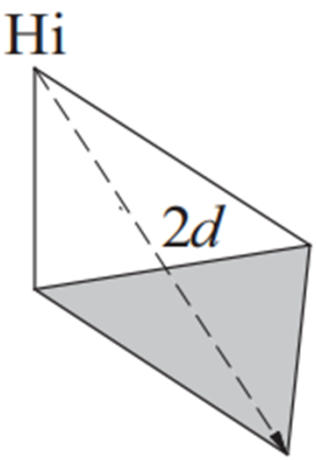
- The idea is to employ a moving simplex in the design space to surround the optimal point and then shrink the simplex until its dimensions reach a specified error tolerance

Principle (2)

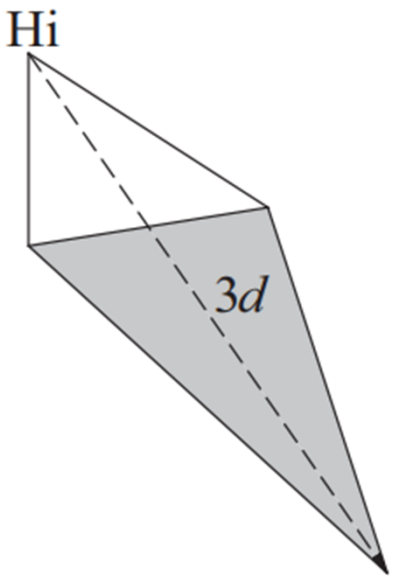
- Possible simplex transformation (in 2D)
- Direction of the move is determined by values of $F(\boldsymbol{x})$ at the vertices
- **Hi** is the vertex with the largest $F(\boldsymbol{x})$
- **Lo** is the vertex with the lowest $F(\boldsymbol{x})$
- The magnitude of a move is controlled by the distance *d* measured from the Hi vertex to the centroid of the opposing face



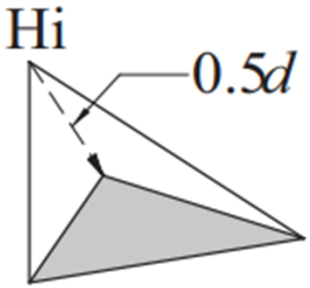
Original simplex



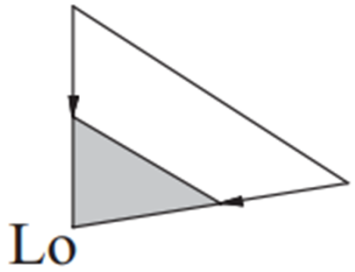
Reflection



Expansion

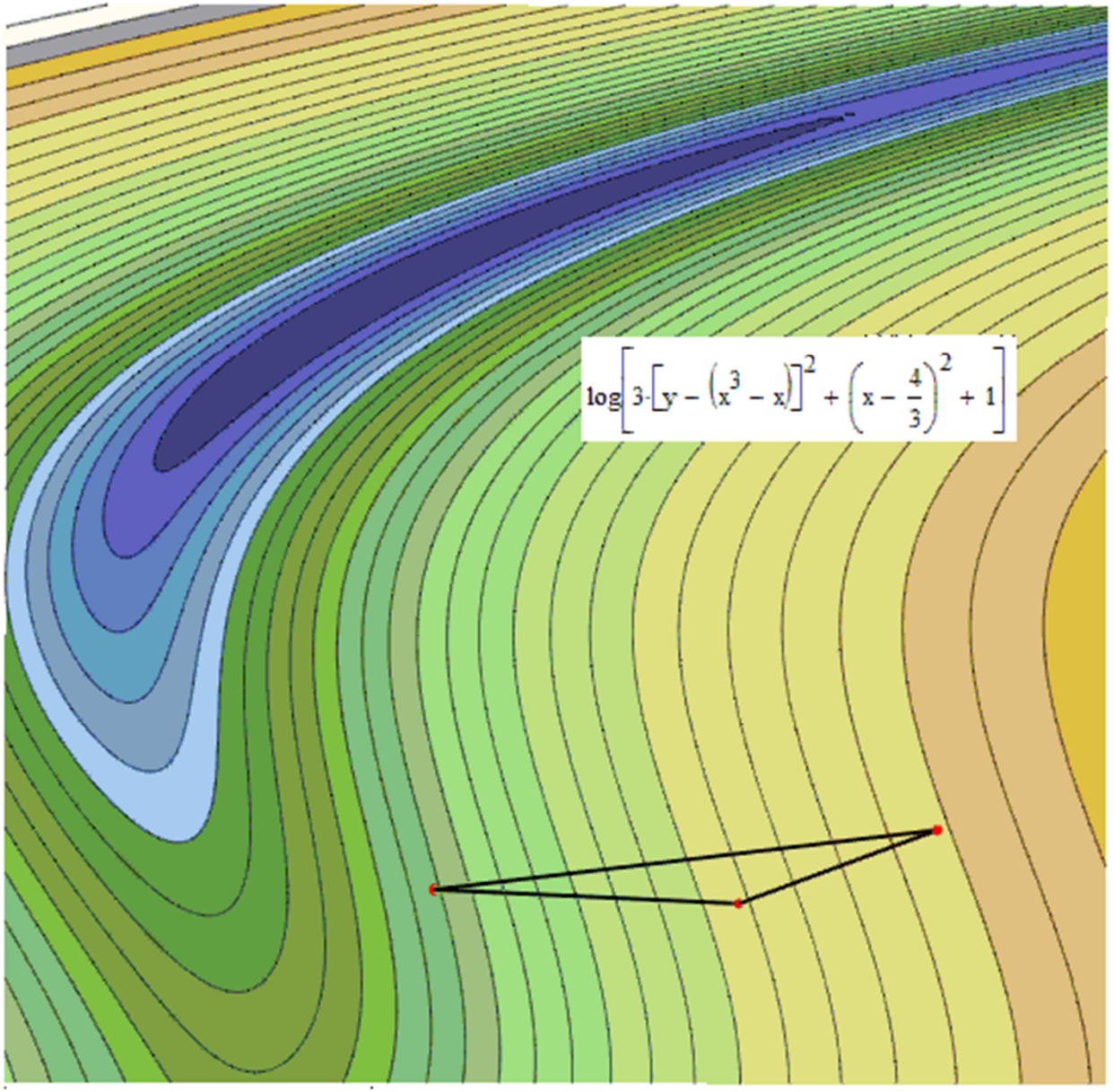


Contraction



Shrinkage

Principle (3)



The Nelder–Mead method or downhill simplex method or amoeba method is a commonly used nonlinear optimization technique

u = 0

x = 11.7

y = 5.375

f(x, y) = 0.897

Created in Mathcad 15 by Vladimir Sabanin and Valery Ochkov (2014/05/21)

Algorithm

Choose a starting simplex.
Cycle until $d \leq \varepsilon$ (ε being the error tolerance):
    Try reflection.
    if new vertex $\leq$ old *Lo*: accept reflection
        Try expansion.
        if new vertex $\leq$ old *Lo*: accept expansion.
    else:
        if new vertex $>$ old *Hi*:
            Try contraction.
            if new vertex $\leq$ old *Hi*: accept contraction.
        else: use shrinkage.
End cycle.

Implementation

```
def downhill(F,xStart,side=0.1,tol=1.0e-6):
    n = len(xStart)                 # Number of variables
    x = np.zeros((n+1,n))
    f = np.zeros(n+1)

  # Generate starting simplex
    x[0] = xStart
    for i in range(1,n+1):
        x[i] = xStart
        x[i,i-1] = xStart[i-1] + side
  # Compute values of F at the vertices of the simplex
    for i in range(n+1): f[i] = F(x[i])

  # Main loop
    for k in range(500):
      # Find highest and lowest vertices
        iLo = np.argmin(f)
        iHi = np.argmax(f)
      # Compute the move vector d
        d = (-(n+1)*x[iHi] + np.sum(x,axis=0))/n
      # Check for convergence
        if math.sqrt(np.dot(d,d)/n) < tol: return x[iLo]

      # Try reflection
        xNew = x[iHi] + 2.0*d
        fNew = F(xNew)
        if fNew <= f[iLo]:          # Accept reflection
            x[iHi] = xNew
            f[iHi] = fNew
          # Try expanding the reflection
            xNew = x[iHi] + d
            fNew = F(xNew)
            if fNew <= f[iLo]:      # Accept expansion
                x[iHi] = xNew
                f[iHi] = fNew
        else:
          # Try reflection again
            if fNew <= f[iHi]:      # Accept reflection
                x[iHi] = xNew
                f[iHi] = fNew
            else:
              # Try contraction
                xNew = x[iHi] + 0.5*d
                fNew = F(xNew)
                if fNew <= f[iHi]: # Accept contraction
                    x[iHi] = xNew
                    f[iHi] = fNew
                else:
                  # Use shrinkage
                    for i in range(len(x)):
                        if i != iLo:
                            x[i] = (x[i] - x[iLo])*0.5
                            f[i] = F(x[i])
    print("Too many iterations in downhill")
    return x[iLo]
```

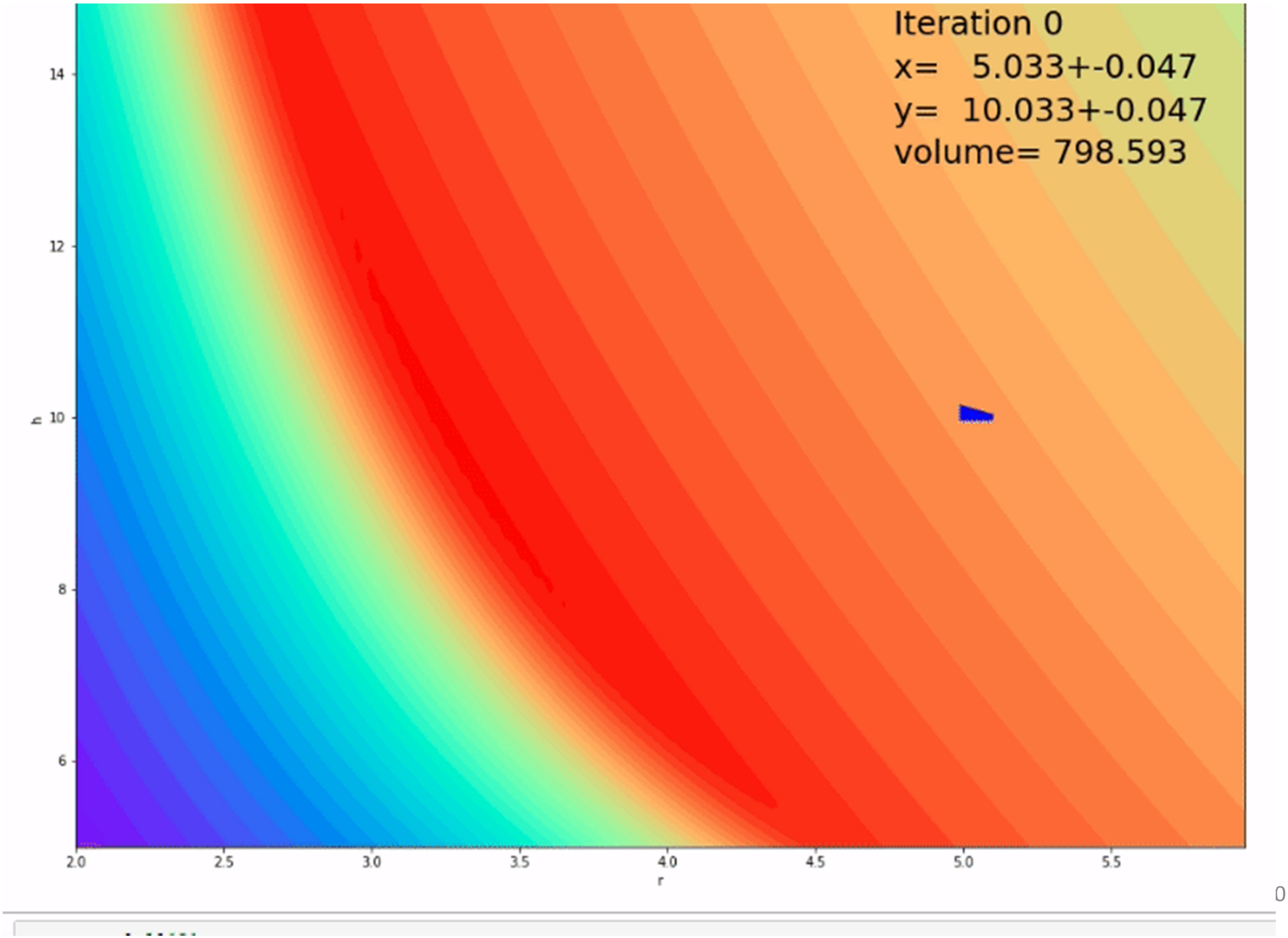
- Staring point
- Starting Simplex $(\boldsymbol{x_0} + b\boldsymbol{e_i})$

- Loop
  - Find Hi and Lo
  - Compute **d**
  - Stop if **d** is small

  - Try Reflection vs Lo ?
    - If yes, Try expansion ?

  - Try Reflection vs Hi ?
    - If no, Try contraction ?
      - If no, Try shrinkage ?

Example With Can optimization



Gradient Descent

Gradient Descent

- Highly used by machine learning libraries
- Order one method (using the derivative of $f(\boldsymbol{x})$)
- Iterative method
- Basic idea is to move the point $\boldsymbol{x}$ in the design space toward the direction of the gradient of $f(\boldsymbol{x})$

- $\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \, \boldsymbol{\nabla} f(\boldsymbol{x})$
- $x_i \leftarrow x_i - \eta \frac{\partial f(\boldsymbol{x})}{\partial x_i}$ *equivalent component notation*

- η *is a scalar called the learning rate*

Algorithm

- Choose an initial
  - vector of parameters $\boldsymbol{x_0}$
  - learning rate η.

- Repeat until an approximate minimum is obtained:

  - $\boldsymbol{x_{i+1} \leftarrow x_i - \eta \nabla f(x_i)}$

  - $\eta \leftarrow \eta \times 0.95$ *(optionally decrease the learning rate overtime)*

Illustration

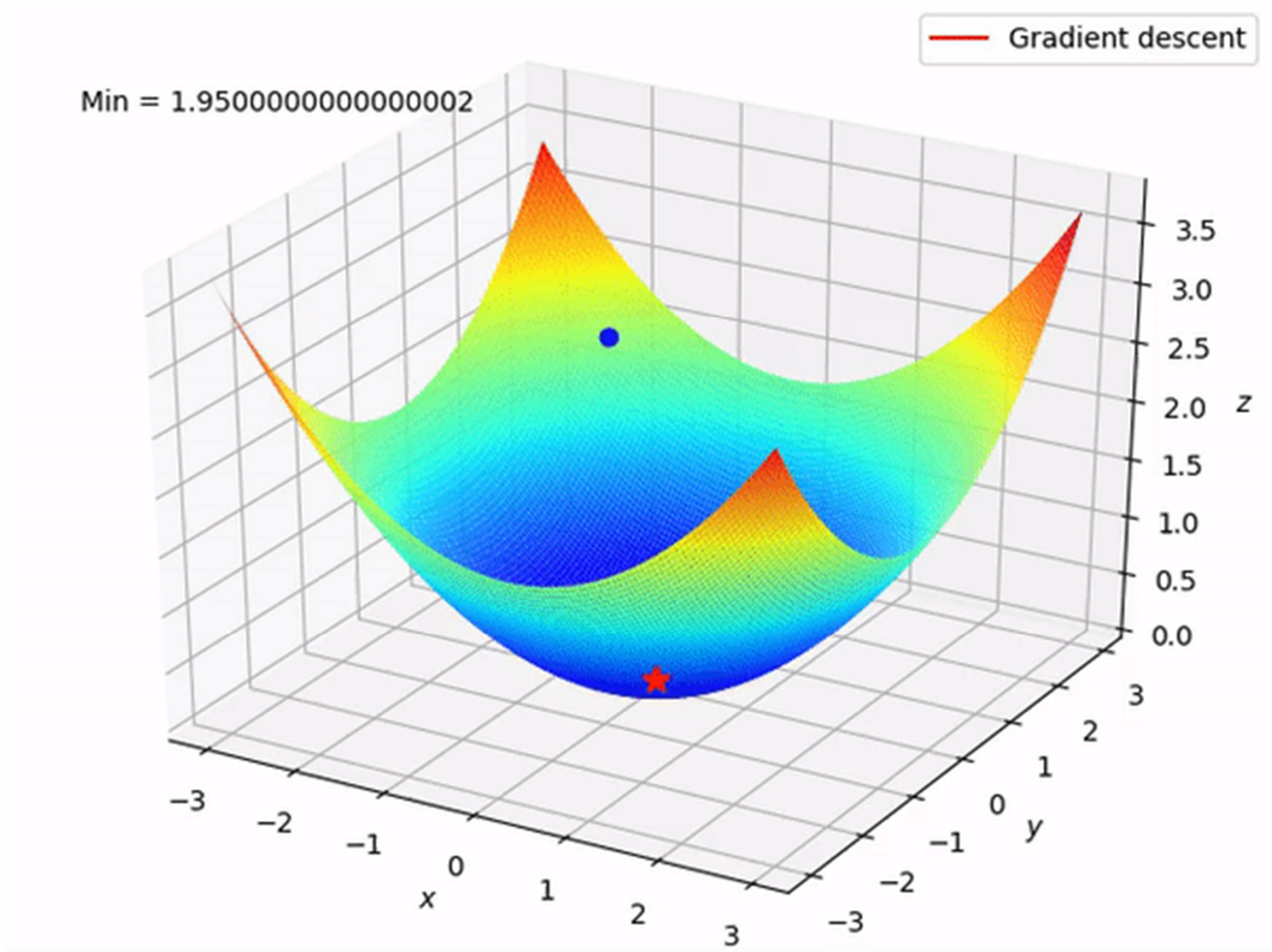


Image from:
http://www.xpertup.com/2018/05/11/loss-functions-and-optimization-algorithms/