Lecture 5: Numerical Methods I: Root Finding and Optimization

Programming and Numerical Methods for Economics—ECNM10115

Albert Rodriguez-Sala & Jacob Adenbaum
School of Economics, The University of Edinburgh

Winter 2024

This Class

Preliminaries: Symbolic Computation and Systems of Linear Equations

Preliminaries

In this class we will focus on numerical methods for solving non-linear systems of equations and optimization.

- ▷ SciPy is the main library for numerical methods in Python.
    - For this class, in particular, we will mostly use methods from scipy.optimize.
- ▷ Before we go to non-linear systems of equations and optimization, let's see some references for interesting methods not covered in the course:
    1. Symbolic computation.
    2. Solving systems of linear equations and linear algebra.

Symbolic vs numerical methods

In this part of the course we focus on numerical methods.

- ▷ A **numerical method** is an approximate computer method for solving a mathematical problem that often has no analytical solution.

However, we can also use Python to solve mathematical problems symbolically as in Wolfram Mathematica and other programs.

- ▷ **Symbolic computation:** solving mathematical problems **analytically**, that is with exact precision where the computer treats mathematical objects as symbols.

  Sympy is the module for symbolic mathematics in Python.

  - Introduction to Sympy and symbolic computation.
  - No numerical error, exact solutions.
  - The mathematical problem needs to have an analytical solution!

Systems of linear equations

Systems of linear equations: systems of equations in which all equations are linear. The system can be represented in the following matrix notation $b = Ax$

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_m \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & & a_{3n} \\ & \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_m \end{bmatrix}$$

- If the matrix A can be easily inverted, then the solution of the linear system of equations is

$$x = A^{-1}b$$

- If the matrix A cannot be easily inverted, we need more complex linear algebra methods.

Systems of linear equations

- Scipy.linalg is a great library for linear algebra computations.
- In particular, scipy.linalg.solve is the main routine to solve a linear system of equations. Depending on the type of matrix A, it will use different linear algebra methods.

Root Finding: Solving Non-Linear Equations

Systems of equations in economics

In economics we use systems of equations to

- **Defind/find equilibria:** economic equilbriums are systems of equations that need to hold given quantities and prices. *we might also have that agents optimize, aggregate resources hold, laws of motion, government budgets, etc.*
- **Solve First Order Conditions**: when we want to solve for optimal behavior or any other optimization problem.
    - FOC for the household/consumer optimal behavior.
    - FOC of firms.
    - FOC on econometric estimators: OLS, GLS, Maximum likelihood, GMM, etc.
- Other systems of equations in economics, like the calibration exercise in example 2.

Solving non-linear equations—N-Dimensional

A non-linear system of equations, is a system of equations where at least one of the equations is not linear. We can represent the system as follows

$$F(\mathbf{x}) = 0 \qquad \mathbf{x} \in \mathbb{R}^N$$

where $\mathbf{x}$ is a vector of N variables $\mathbf{x} = [x_1, x_2, .., x_N]$ and F is a set of N functions $F = [f^1, f^2, ..., f^N]$ on $\mathbf{x}$.

$$\begin{aligned} f^1(x_1, x_2, \ldots x_N) &= 0 \\ f^2(x_1, x_2, \ldots x_N) &= 0 \\ &\ldots \\ f^N(x_1, x_2, \ldots x_N) &= 0 \end{aligned}$$

Solving the system consists of finding the roots, that is to find the $\mathbf{x}^*$ such that

$$F(\mathbf{x}^*) = 0$$

Methods to solve non-linear systems of equations

There are 3 sets of methods to solve non-linear systems of equations with their own advantages and disadvantages.

1. **Derivative-based methods:** faster and more stable methods. They require that the functions are differentiable (no "kinks" or steps). Might not converge to the solution. Examples: **Newton's method**.
2. **Derivative-free methods**: less numerically stable, but do not require derivatives. Often easier to implement. Examples: **Secant (Broyden) method**. **Bisection method** only for univariate cases, slow. Both methods require functions to be continuous. Typically, convergences to the solution is achieved.
3. **Hybrid methods:** combination of derivative-based methods and derivative-free methods. **Powell's hybrid algorithm** used in *fsolve*.

Root-finding routines in Python

Scipy.optimize library contains many routines to solves systems of non-linear equations. the two main routines are

- scipy.optimize.fsolve: Main routine to solve systems of equations in Python. works as a hybrid method based on the Powell algorithm.
- scipy.optimize.root: routine to solve systems of equations and allows us to choose the method. The methods available are based on hybrid, derivative-free, or Netwon methods. For univariate cases, we have the routine *scipy.optimize.root_scalar*.

Example 1: univariate problem—1-D non-linear equation

Example 1: find the root of the following function univariate function

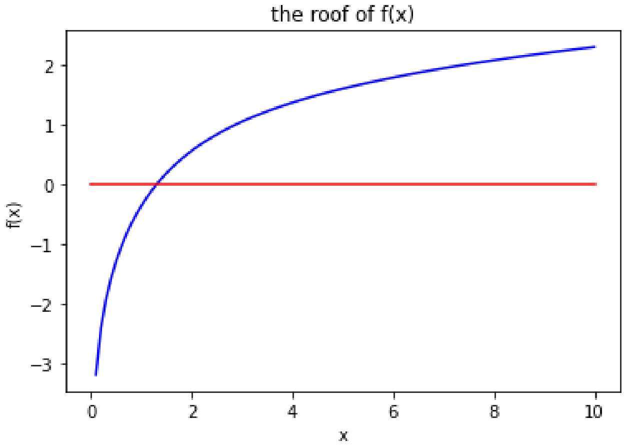
$$f(x) = ln(x) - e^{-x}$$

that is we want to find x^* such that $f(x^*) = 0$. As a first step we can plot the function to have a guess.

- Plotting is a very useful technique to understand our root-finding and optimization problems. Whenever possible ($N < 3$) plot the functions.
  - Does the function exhibits multiple maxima and minima?
  - continuous and differentiable?
  - Flat regions?

  Having a rough answer to these questions will allow us to choose the correct algorithm and initial value x_0.

Example 1: univariate problem



Example 1: solution

Let's **find the exact solution** with Python.

1. Program the function $f(x)$.

```
def f1_func(x):
    y = np.log(x) - np.exp(-x)
    return y
```

2. Set an initial value x_0. This is a value we guess and it is necessary to start the algorithm.

```
x0= 0.5  # initial value:
```

3. Use *fsolve* or *root_scalar* from Scipy.optimize with inputs $f(x)$ and initial value x_0 to find the root.

```
root_f1 = fsolve(f1_func, x0)
print('the root of the function is x*=', root_f1)
>> the root of the function is x*= [1.30979959]
```

A more complex example

The previous example was relatively easy: univariate function that was continuous and differentiable, we could do the plot, and the programming of the problem was simple.

Let's now see how to solve a more complex exercise.

Example 2: a calibration exercise with the Solow model

Example 2: Study three economies under the Solow growth model.

- assume the three economies have the same technology function $Y_t = AK_t^{\alpha}L_t^{1-\alpha}$, no population growth, depreciation rate δ, but they differ on their saving rates s_i. Parameter values are: $A = 2$, $\alpha = 0.3$, $\delta = 0.1$.
- The capital per worker law of motion in each economy i is

$$k_{t+1,i} = s_i A k_{t,i}^{\alpha} + (1-\delta)k_{t,i}$$

- And the capital in steady state is

$$k_i^* = \left(\frac{s_i A}{\delta}\right)^{\frac{1}{1-\alpha}}$$

- In the data, we observe country 2 is 20% richer than country 1, country 3 is 35% richer than country 1, and the capital-ouptut ratio in country 3 is 3. That is,

$$y_2 = 1.2y_1 \qquad y_3 = 1.35y_1 \qquad \frac{k_3}{y_3} = 3$$

Given the data observations, and considering that the 3 economies are in steady state, What are the saving rates s_1, s_2, s_3 of the 3 economies?

Example 2: a calibration exercise with the Solow model

The previous problem consists of solving the next system of 3 non-linear equations $f(\mathbf{x}) = 0$, with 3 unknowns $\mathbf{x} = [s_1, s_2, s_3]$

$$A(k_2^*(s_2))^\alpha - 1.2A(k_1^*(s_1))^\alpha = 0 \tag{1}$$
$$A(k_3^*(s_3))^\alpha - 1.3A(k_1^*(s_1))^\alpha = 0 \tag{2}$$
$$\frac{k_3^*(s_3)}{A(k_3^*(s_3))^\alpha} - 3 = 0 \tag{3}$$

Where

$$k_i^*(s_i) = \left(\frac{s_i A}{\delta}\right)^{\frac{1}{1-\alpha}}$$

The root of the system is the $\mathbf{x}^* = \{s_1^*, s_2^*, s_3^*\}$ such that $F(\mathbf{x}^*) = 0$. Let's program this system of equations and solve it in Python.

Example 2: Solution

```
A, alpha, delta = 2, 0.3, 0.1
params = [A,alpha,delta]
def steady_state_ex2(s, params):  # Define the system of equations
    A,alpha,delta = params
    # from the Solow model we know that k* for each country k* is.
    k1 = (s[0]*A/delta)*(1/(1-alpha))
    k2 = (s[1]*A/delta)*(1/(1-alpha))
    k3 = (s[2]*A/delta)*(1/(1-alpha))
    # From the calibration, we know these equations need to be equal to 0 given s
    eq_1 = A*k2**(alpha) -1.2*A*k1**(alpha)
    eq_2 = A*k3**(alpha) -1.35*A*k1**(alpha)
    eq_3 = k3/(A*k3**(alpha))  - 3
    return  np.array([eq_1, eq_2, eq_3])

ss_func_ex2 = lambda s: steady_state_ex2(s,params)
s0= [0.1, 0.3, 0.5]  # initial guess
root_savings = fsolve(ss_func_ex2, s0)
print(root_savings)
>> array([0.16644348, 0.30563584, 0.45259806])
```

Numerical Optimization

Optimization problem

An optimization problem is characterized by the following elements:

1. a vector of **control variables** (choices), $\mathbf{x} \in \mathbb{R}^N$.
2. an **objective function** to be maximized or minimized, f, where $f : \mathbb{R}^N \to \mathbb{R}$.
3. and, potentially, a set of **constraints** $\mathbf{x} \in C \subseteq \mathbb{R}^N$.

with these elements, the maximization problem is

$$\begin{aligned} &\underset{\mathbf{x}}{\text{Max}}\ f(\mathbf{x}) \\ &\text{st: } \mathbf{x} \in C \end{aligned}$$

And equivalently for the minimization problem.

Practical aspects of optimization

- **maximization** of f is just **minimization** of $-f$.
- Our functions might take other inputs (parameters, other functions, data) that are not the control variables we optimize over.
- To apply minimization routines we need **to write the function with respect to a single input**: the vector of control variables $\mathbf{x}$.

Things to keep in mind before we start the optimization:

- What is the **criterion function** f? Might exhibit discountinuity, non-differentiable (or complex derivative)? flat regions?
- What are the **control variables**, $\mathbf{x}$, of the problem? Are the control variables **bounded**?
- Are there **constraints** in the problem? Can the problem be simplified by isolating some variables from the constraints?

Optimization methods

Search methods:

brute-force.

Iteration methods:

1. Derivative-based methods

    Newton-Rhapson

    Quasi-Newton methods: the BFGS method.

2. Derivative-fee methods:

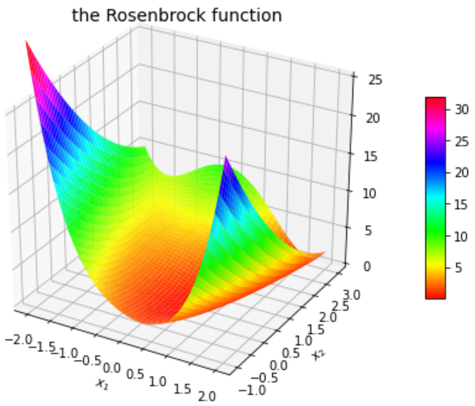
    Nelder-Mead method.

Example: the Rosenbrock function

To illustrate the different algorithms we will use the **Rosenbrock function** for $N = 2$.

$$f(\mathbf{x}) = f(x_1, x_2) = (1 - x_1)^2 + (x_2 - x_1^2)^2$$

- commonly used to test different optimization algorithms.
- The global minimum is difficult to find: inside a long, parabolic-shaped flat valley.
- Global minimum is at $\mathbf{x} = (1, 1)$ and $f(1, 1) = 0$.



Search Methods

Search methods: Create a grid over the control variables and evaluate the function at each point on the grid. Select the best point. Then refine the grid in the neighborhood of the best point. and continue until the accuracy is ”good enough”.

- **Pros:** does not require any functional assumptions and it is a **global optimization method**.
- **Cons: inefficient**. suppose we have 5 variables (N=5) and 100 points, then there would be 100^5 points to check.
- Implementation in Python: scipy.optimize.brute. Computes the function’s value at each point of a multidimensional grid of points, to find the global minimum of the function.

Example: brute-force method on the Rosenbrock function

```
# defined the function in terms of a vector-variable X
def rosen_func(X):
    x1, x2 = X[0], X[1]
    return (1-x1)**2+(x2-x1**2)**2
# 1. Minimization using brute-force ----------
ranges_X = ((-2,2),(-1,3))
print('Brute-force method ---------')
qe.tic()
res1 = optimize.brute(rosen_func, ranges_X)
qe.toc()
print(res1)
>> Brute-force method ---------
TOC: Elapsed: 0:00:0.00
[1.0000081  1.00002356]
# not bad...
```

Iterative methods

Iterative methods: numerical optimization problems that improve on the optimized value in each round using some well-defined rule that modifies last-round's value. Iterative methods are defined by the following structure

1. An **initial guess** $\mathbf{x}_0$.
2. The **iteration method** for choosing $\mathbf{x}^{(j+1)}$ given that we're at $\mathbf{x}^{(j)}$ at iteration j.
3. **stopping criterion** for deciding when the numerical procedure has effectively converged. That is $|f(\mathbf{x}^{(j)}) - f(\mathbf{x}^{(j-1)}) < \varepsilon|$ for a very small ε.[1]

The iteration can be defined as follows

$$\mathbf{x}^{(j+1)} = \mathbf{x}^{(j)} + a^j d^j$$

where a is the **stepsize** (how much) and d is the **direction** of the movement (where) at iteration j.

[1]Note that convergence can also be in terms of $\mathbf{x}^{(j}$ or using other metrics rather than absolute value. The default value in scipy.minimize for BFGS or Nelder-Mead is $\varepsilon = 10^{-4}$.

Derivative-based methods: Newton-Rhapson method

Newton-Rhapson method: The idea behind the method is to maximize/minimize the second-order Taylor approximation of the function $f(x)$,

$$f(x) \approx f(x^j) + f'(x^j)(x - x^j) + \frac{1}{2}(x - x^j)f''(x^j)(x - x^j)^2$$

solving the first-order condition,

$$f'(x^j) + f''(x^j)(x - x^j) = 0$$

yields the **Newton-Rhapson iteration rule**

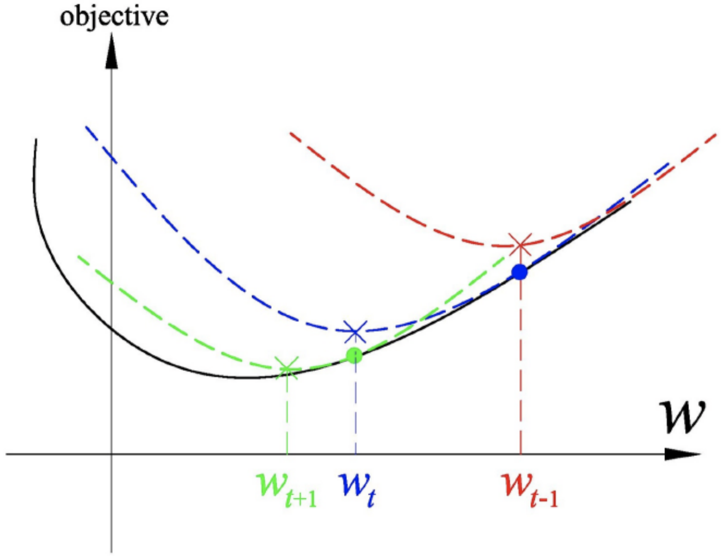
$$x^{j+1} = x^j + \frac{f'(x^j)}{f''(x^j)}$$

In the case of $N > 1$, the **Newton-Rhapson iteration rule is**

$$\mathbf{x}^{j+1} = \mathbf{x}^j - [H(\mathbf{x}^j)]^{-1}J(\mathbf{x}^j)$$

Where $J(\mathbf{x}^j)$ and $H(\mathbf{x}^j)$ are the *Jacobian* and the *Hessian* matrix evaluated at point $\mathbf{x}^j$.

Derivative-based methods: 1D illustration Newton-Rhapson method



Derivative-based methods: Newton-Rhapson method

Newton's method exploits information about the slope and curvature of the objective function to determine which direction (d) and how far to move (a) from a point $\mathbf{x}^{(j)}$.

- **Pros:** fast and robust method if assumptions on f hold.
- **Cons:** strong assumptions on f: requires f to have well-defined and easy to compute Jacobian and Hessian matrix.

In Scipy there is not the Newton-Rhapson method for optimization—only there is the Newton-Rhapson method for solving non-linear equations. The reason is that computing the Hessian is usually computationally expensive. Thus, in **Scipy there are Quasi-Newton methods** that do not require to compute the Hessian matrix.

Derivative-based methods: Quasi-Newton methods

Quasi-Newton methods: methods that follow the Newton-Raphson method but they approximate the inverse of the Hessian matrix $(H^j)^{-1}$ with a matrix B^j.[2]

Quasi-Newton Methods:

- Gradient Ascend method: $B^j = I$. [3]
- Davidson-Fletcher-Powell (DFP) method.
- Broyden-Fletcher-Goldfarb-Shano (BFGS) method.

[2] B^j is required to be both symmetric and positive definite. B^j does not contain 2nd order derivatives.
[3] Typically referred to as Steepest Descent method.

Derivative-based methods: BFGS method

Broyden-Fletcher-Goldfarb-Shano (BFGS) method: instead of requiring the full Hessian matrix at the point $\mathbf{x}_{j+1}$ to be computed as B_{j+1}, the approximate Hessian at iteration j is updated as follows[4]

$$B_{j+1} = B_j + \frac{1}{d^{\mathrm{T}} u} \left(d w^{\mathrm{T}} + d w^{\mathrm{T}} - \frac{w^{\mathrm{T}} u}{d^{\mathrm{T}} u} d d^{\mathrm{T}} \right),$$

where $d = \mathbf{x}_{j+1} - \mathbf{x}_j$, $u = J(\mathbf{x}_{j+1}) - J(\mathbf{x}_j)$, and $w = d - B_j u$.

[4]Based on the Sherman–Morrison formula to compute an updated matrix inverse

Derivative-based methods: BFGS method

- Implementation in Python, scipy.optimize.minimize(method='BFGS'). Note that BFGS is the default method in scipy.optimize.minimize routine.[5]
- **Pros:** fast and robust. Requires few function evaluations.
- **Cons:** requires derivatives, and does not ensure that the global maximum or minimum is found—local optimization method.
- **Trick:** to find the global maximum/minimum use multiple initial values!

[5]Similarly, BFGS is the default method on the fminunc in Matlab (main local minimization routine).

Example: BFGS method on the Rosenbrock function

```
def rosen_func(X):
    x1, x2 = X[0], X[1]
    return (1-x1)**2+(x2-x1**2)**2

x0 = [0,0]  #play with different x0s
print('BFGS method ---------')
qe.tic()
res2 = optimize.minimize(rosen_func,x0)
qe.toc()
print(res2.x)
>> BFGS method ---------
TOC: Elapsed: 0:00:0.01
[0.99999995 0.99999991]
# Not bad neither
```

```
# minimize output:
print(res2)
>> fun: 2.998637572518479 3e-15
 hess_inv: array([[0.50009319, 0.99925837],
[0.99925837, 2.49866596]])
jac: array([-9.18226667e-08,  4.54839240e-08])
message: 'Optimization terminated successfully
     nfev: 30
      nit: 8
     njev: 10
   status: 0
  success: True
        x: array([0.99999995, 0.99999991])
```

Derivative-free methods: Nelder-Mead method

Nelder-Mead method: a derivative-free iteration method based on simplex direct search.

- A **simplex** is the simplest possible polytope in any given space.[6] A simplex in 1D is a line segment, a simplex in 2D is a triangle, in 3D a tetrahedro, etc.
- **Updating scheme**: the iterative scheme forms a new simplex at each iteration by reflecting away from the vertex with the largest value of f, or by contracting toward the vertex with the smallest value of f .

[6]polytopes are geometric objects with flat sides (e.g. polytopes of two dimensions are polygones, and in three dimensions polyhedrons).

Derivative-free methods: Nelder-Mead method

- Animation of the Nelder-Mead iterations for the Rosenbrock function N=2.
- scipy.optimize.minimize(method='nelder-mead') scipy routine for the Nelder-Mead algorithm.
- Pros: robust (to e.g. noise in objective function) and does not require derivatives.
- Cons: slow convergence and does not ensure that the global maximum or minimum is found (local potimization method).

Nelder-Mead method on the Rosenbrock function

```
def rosen_func(X):
    x1, x2 = X[0], X[1]
    return (1-x1)**2+(x2-x1**2)**2

x0 = [0,0] # play with other x0s
print('Nelder-Mead method ---------')
qe.tic()
res3=optimize.minimize(rosen_func,x0,method='nelder-mead')
qe.toc()
print(res3.x)
Nelder-Mead method ---------
TOC: Elapsed: 0:00:0.00
[1.00001608 1.0000338 ]
# not bad neither but required more iterations and it is less precise
```

```
 print(res3.x)
 final_simplex: (array([[1.00001608, 1.0000338 ],
[1.00001032, 1.00003561],
[0.99997718, 0.99996132]]), array([2.61305939e-10, 
fun: 2.613059385742404e-10
message: 'Optimization terminated successfully.'
        nfev: 129
        nit: 67
        status: 0
        success: True
         x: array([1.00001608, 1.0000338 ])
```

Scipy Minimize

We have seen that Scipy.optimize.minimize is the main routine for local optimization in Python (see the source code of minimize). Besides the methods discussed before, minimize also contains

- **Bounded optimization**: Many variables in economics can only take particular values–i.e. non-negativity.
  - **L-BFGS-B:** BFGS algorithm but allowing for bounds.
  - **Powell:** derivative-free and allows for bounds.
- **Constrained optimization.**
  - **SLSQP**: Bounds and constraints.
  - **trust-constr**: The most versatile constrained minimization algorithm implemented in SciPy and the **most appropriate for large-scale problems**. That's what SciPy says and what I found out with personal experience!
- Its 1-D version is minimize_scalar.

Local vs global optimizers

Except for the search method (brute-force), all the methods seen in this class are **local optimization methods**. Thus, using them might not ensure the global maximum or minimum is found. To solve for the global, you can

1. Use local optimization routines with many different starting values $\mathbf{x}_0$. If given different $\mathbf{x}_0$, you find different $\mathbf{x}^*$, then chose the $\mathbf{x}^*$ such that $f(\mathbf{x}^*)$ is the highest/lowest.
2. Use **global optimization methods**:
   - Brute-force method.
   - Simulated-annealing (or dual-annealing in scipy.optimize).
   - Basin-Hopping method.

Further examples: Rosenbrock function in 5-D

Let's minimize the Rosenbrock function in 5 dimensions, $N = 5$. [7]

$$\sum_{i=1}^{N-1} (1 - x_i)^2 + (x_{i+1} - x_i^2)^2$$

1. by brute-force
2. by BFGS
3. by Nelder-Mead

Use minimization routines that include bounds. Use the following bounds $x_i \in [0.5, 2]$ for all $i \in \{1, 2, 3, 4, 5\}$.

4. by L-BFGS-B
5. by Powell

[7] The global minimum is at x=[1,1,1,1,1].

Further examples: Rosenbrock function N = 2 with constraints

Solve the following problem

$$\begin{aligned} &\min_{x_1,x_2} (1-x_1)^2 + (x_2 - x_1^2)^2 \\ &\text{st:} \\ &x_1^2 + x_2^2 \leq 1 \end{aligned}$$

1. by COBYLA
2. by SLSQP
3. by trust-constr.