MECH 579 : Numerical Optimization

Department of Mechanical Engineering, McGill University

Assignment #4: Constrained Optimization and Model-Order Reduction

Name : Chiyoung Kwon
ID : 261258263
Department : Mechanical Engineering
Program : PhD

---

1. Constrained Rosenbrock Optimization

Minimize the Rosenbrock function defined below using `scipy.optimize`.
The function is non-convex and has a global minimum at $(1, 1)$.
The minimum lies at the bottom of a narrow parabolic valley that is curved on the $x - y$ plane.
The problem statement is as follows:

$$\text{minimize} \quad f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

$$\text{with respect to} \quad (x, y) \in \mathbb{R}^2$$

$$\text{subject to} \quad \hat{c}(x, y) = 1 - x^2 - y^2 \geq 0.$$

(a) Write a Python code to find the minimum of the function subject to the given constraint using the *SLSQP (Sequential Least Squares Programming)* method. Compute the gradient and provide it to the `scipy.optimize` library. Provide the following in a written report:

In [17]:
```
# ------------------------------------------------------------------ Clear variable
all = [var for var in globals() if var[0] != '_']
for var in all:
    del globals()[var]
del var, all
```

In [18]:
```
# ------------------------------------------------------------------ Import modules
import numpy as np
```

```
import torch
import scipy.optimize as sci_opt
import sys
sys.path.append('../')
from module_opt import *
import matplotlib.pyplot as plt
```

In [19]:
```
# ------------------------------------------------------------------ Define functio
# obj function
def f(x:np.ndarray) -> np.float64:
    return (1 - x[0])**2 + 100*(x[1] - x[0]**2)**2

# gradient of obj function
def grad_f(x:np.ndarray) -> np.float64:
    x = torch.tensor(x, requires_grad=True)
    f(x).backward()
    return x.grad.detach().numpy()

# constr function
def c(x:np.ndarray) -> np.float64:
    return 1 - x[0]**2 - x[1]**2

# gradient of constr function
def grad_c(x:np.ndarray) -> np.float64:
    x = torch.tensor(x, requires_grad=True)
    c(x).backward()
    return x.grad.detach().numpy()
```

In [20]:
```
# ------------------------------------------------------------------ xk Log of SLSQ
x_sqsqp_sci = []
def cb(xk):
    x_sqsqp_sci.append(xk.copy())
```

In [21]:
```
x0 = np.array([.1, .1])
x_sqsqp_sci.append(x0)
log_sqsqp_sci = sci_opt.minimize(fun=f, x0=x0, jac=grad_f, method='SLSQP', constrai
log_sqp_mine = sqp(f=f, ce=[], ci=[c], x0=x0, inner_opt=3, tol=1e-6, tol_inter=1e-4
```

```
/home/chiyoung0/NumericalOptimization_25/Assignment_4/../module_opt.py:497: RuntimeW
arning: divide by zero encountered in scalar divide
  mu = max(max_lp, ((grad_f_k @ p_k) + .5*sigma*pBp)/((1 - rho)*sum_c)) # penalty pa
rameter for constraint terms in phi1
```

```
 log - SQP
‖Δx‖ = 1.47e-01, x01 = [0.14218750, -0.04062500] | f = 1.1060e+00, ‖∇L‖ = 1.23e+01,
‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 6.93e-02, x02 = [0.20867842, -0.02118043] | f = 1.0451e+00, ‖∇L‖ = 1.35e+01,
‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 1.25e-01, x03 = [0.25140292, 0.09589510] | f = 6.6727e-01, ‖∇L‖ = 8.10e+00, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 6.49e-02, x04 = [0.31627409, 0.09574917] | f = 4.6931e-01, ‖∇L‖ = 1.19e+00, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 8.87e-02, x05 = [0.39732247, 0.13191077] | f = 4.3058e-01, ‖∇L‖ = 5.96e+00, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 1.77e-02, x06 = [0.40349427, 0.14852747] | f = 3.7621e-01, ‖∇L‖ = 3.06e+00, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 1.35e-01, x07 = [0.50119294, 0.24155868] | f = 2.5809e-01, ‖∇L‖ = 2.14e+00, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 1.66e-01, x08 = [0.61914778, 0.35887715] | f = 2.0491e-01, ‖∇L‖ = 7.21e+00, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 6.63e-02, x09 = [0.64803621, 0.41850449] | f = 1.2409e-01, ‖∇L‖ = 4.38e-01, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 1.79e-01, x10 = [0.75839124, 0.55918211] | f = 8.3895e-02, ‖∇L‖ = 5.41e+00, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 1.52e-02, x11 = [0.76086967, 0.57419854] | f = 5.9415e-02, ‖∇L‖ = 1.35e+00, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
```

```
‖Δx‖ = 5.17e-02, x12 = [0.78737843, 0.61863790] | f = 4.5384e-02, ‖∇L‖ = 2.06e-01, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 2.68e-03, ‖λ‖ = 0.00e+00, ‖v‖ = 9.67e-02


 log - SQP
‖Δx‖ = 1.34e-03, x13 = [0.78640918, 0.61770719] | f = 4.5675e-02, ‖∇L‖ = 7.30e-03, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 1.57e-06, ‖λ‖ = 0.00e+00, ‖v‖ = 1.21e-01


 log - SQP
‖Δx‖ = 1.89e-05, x14 = [0.78640874, 0.61768831] | f = 4.5678e-02, ‖∇L‖ = 5.25e-05, ‖
ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 1.22e-01

alpha_k did not meet the armijo condition ...

Final iterate: x* = [0.78640874 0.61768831]
f(x*)         = 0.045677535932855194
λ*            = [0.]
v*            = [0.12150877]
‖∇L(x*)‖     = 4.108e-05
‖ce(x*)‖∞   = 0.000e+00
‖ci(x*)‖∞ = 0.000e+00
/home/chiyoung0/NumericalOptimization_25/Assignment_4/../module_opt.py:507: RuntimeW
arning: invalid value encountered in scalar multiply
  Dphi1_0 = -p_k @ grad_f_k - mu*sum_c # directional derivative wrt pk at x0
/home/chiyoung0/NumericalOptimization_25/Assignment_4/../module_opt.py:510: RuntimeW
arning: invalid value encountered in scalar multiply
  Dphi1_0 = -pBp + (p_k @ A_lmbda) + (p_k @ A_nu) - mu*sum_c
/home/chiyoung0/NumericalOptimization_25/Assignment_4/../module_opt.py:506: RuntimeW
arning: invalid value encountered in scalar multiply
  phi1 = lambda alpha : f(x_k + alpha*p_k) + mu*(sum([abs(ce_j(x_k + alpha*p_k)) for
ce_j in ce]) + sum([np.maximum(-ci_j(x_k + alpha*p_k), 0) for ci_j in ci])) # l1 mer
it function
```

In [22]:
```
f_sqsqp_sci = []
grad_f_sqsqp_sci = []
c_sqsqp_sci = []

for xk in x_sqsqp_sci:
    f_sqsqp_sci.append(f(xk))
    grad_f_sqsqp_sci.append(np.linalg.norm(grad_f(xk)))
    c_sqsqp_sci.append(c(xk))
```

In [23]:
```
x_sqp_mine = log_sqp_mine[0]
f_sqp_mine = log_sqp_mine[1]
grad_f_sqp_mine = [np.linalg.norm(grad_f_k) for grad_f_k in log_sqp_mine[2]]
grad_L_sqp_mine = [np.linalg.norm(grad_L_k) for grad_L_k in log_sqp_mine[3]]
c_sqp_mine = log_sqp_mine[5]
```

In [24]:
```
x_sqsqp_sci
```

Out[24]:
```
[array([0.1, 0.1]),
 array([  5.5, -17.9]),
 array([1.80337776, 0.35516037]),
 array([0.26289431, 0.06300771]),
 array([0.33871204, 0.1079608 ]),
 array([0.60549344, 0.27539841]),
 array([0.56348045, 0.27320071]),
 array([0.54436567, 0.28004934]),
 array([0.60507065, 0.36809743]),
 array([0.83886403, 0.62948966]),
 array([0.82668467, 0.63313419]),
 array([0.77441116, 0.57914971]),
 array([0.7511412 , 0.55703409]),
 array([0.78043125, 0.60620139]),
 array([0.78659318, 0.61760767]),
 array([0.78641991, 0.61769229]),
 array([0.78641991, 0.61769229])]
```

In [25]:
```
x_sqp_mine
```

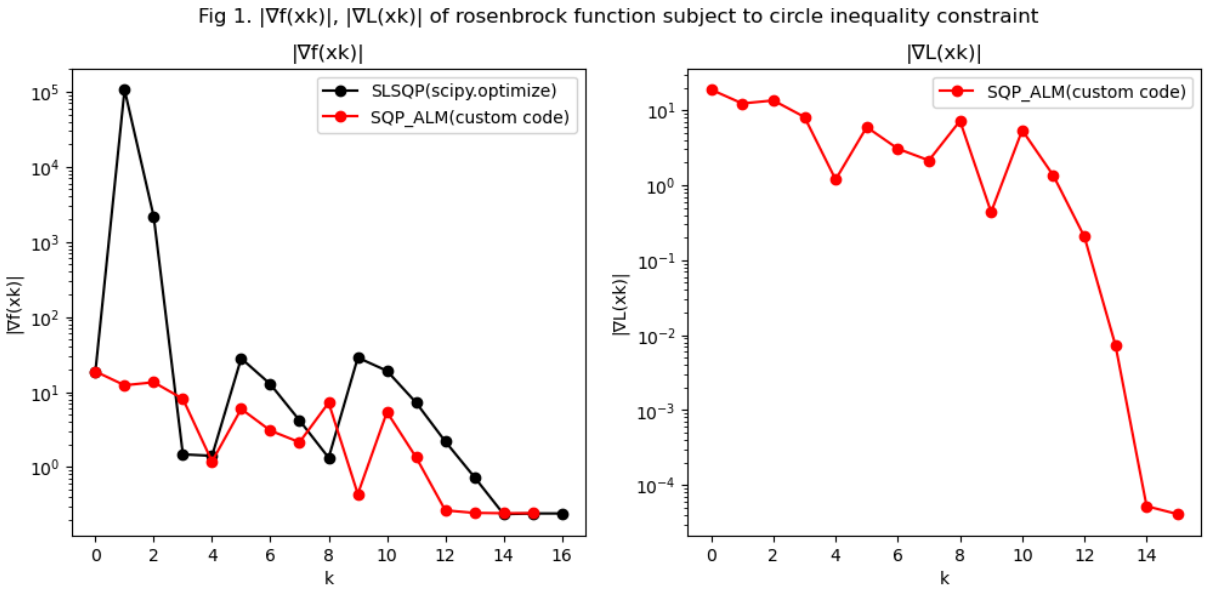
Out[25]:
```
[array([0.1, 0.1]),
 array([ 0.1421875, -0.040625 ]),
 array([ 0.20867842, -0.02118043]),
 array([0.25140292, 0.0958951 ]),
 array([0.31627409, 0.09574917]),
 array([0.39732247, 0.13191077]),
 array([0.40349427, 0.14852747]),
 array([0.50119294, 0.24155868]),
 array([0.61914778, 0.35887715]),
 array([0.64803621, 0.41850449]),
 array([0.75839124, 0.55918211]),
 array([0.76086967, 0.57419854]),
 array([0.78737843, 0.6186379 ]),
 array([0.78640918, 0.61770719]),
 array([0.78640874, 0.61768831]),
 array([0.78640874, 0.61768831])]
```

(i) Convergence of the gradient (y-axis: log of the gradient, x-axis: iteration) and a comparison of the convergence.

In [26]:
```
fig, axes = plt.subplots(1, 2, figsize=(6*2, 5))
axes[0].plot(grad_f_sqsqp_sci, 'ko-', label='SLSQP(scipy.optimize)')
axes[0].plot(grad_f_sqp_mine, 'ro-', label='SQP_ALM(custom code)')
axes[0].set_yscale('log')
axes[0].set_title('|∇f(xk)|')
axes[0].set_xlabel('k')
axes[0].set_ylabel('|∇f(xk)|')
axes[0].legend()

axes[1].plot(grad_L_sqp_mine, 'ro-', label='SQP_ALM(custom code)')
axes[1].set_yscale('log')
axes[1].set_title('|∇L(xk)|')
axes[1].set_xlabel('k')
axes[1].set_ylabel('|∇L(xk)|')
axes[1].legend()
```

```
fig.suptitle("Fig 1. |∇f(xk)|, |∇L(xk)| of rosenbrock function subject to circle ir
```

Out[26]:
```
Text(0.5, 0.98, 'Fig 1. |∇f(xk)|, |∇L(xk)| of rosenbrock function subject to circl
e inequality constraint')
```

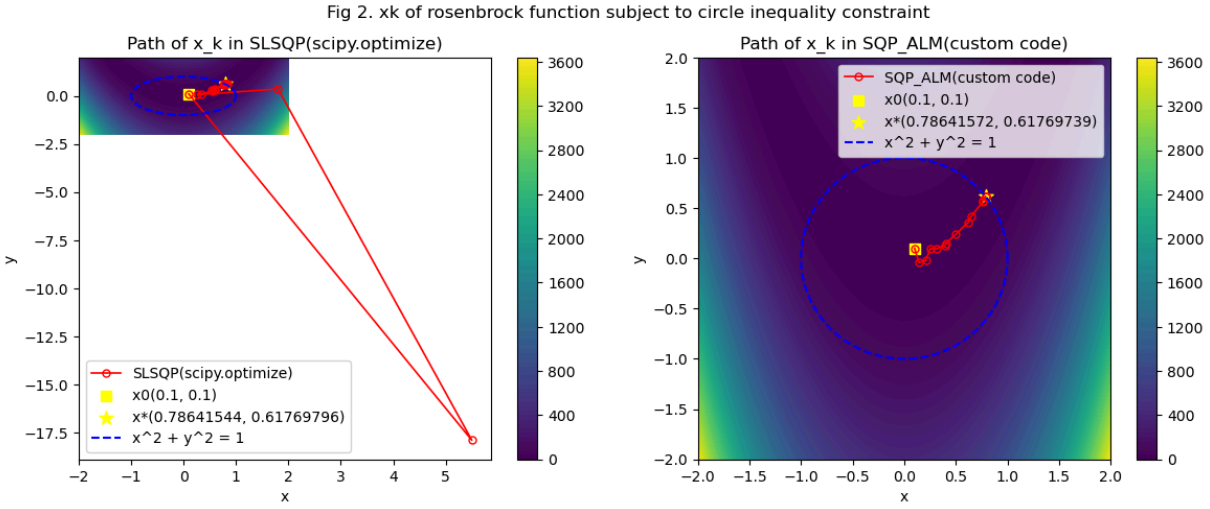


(ii) Contour plot of the path of the optimization algorithm.

In [27]:
```
step = .01
grid = np.meshgrid(np.arange(-2, 2+step, step), np.arange(-2, 2+step, step))
f_grid = f(grid)

# === 원 (x^2 + y^2 = 1) parametric ===
theta = np.linspace(0, 2*np.pi, 400)
circle_x = np.cos(theta)
circle_y = np.sin(theta)

fig, axes = plt.subplots(1, 2, figsize=(7*2, 5))

contourf_0 = axes[0].contourf(grid[0], grid[1], f_grid, levels=100)
axes[0].plot([xk[0] for xk in x_sqsqp_sci], [xk[1] for xk in x_sqsqp_sci],
             color='red', marker='o', markersize=5, markerfacecolor='none', markere
axes[0].scatter(x_sqsqp_sci[0][0], x_sqsqp_sci[0][1], color='yellow', marker='s', s
axes[0].scatter(x_sqsqp_sci[-1][0], x_sqsqp_sci[-1][1], color='yellow', marker='*',
axes[0].plot(circle_x, circle_y, 'b--', linewidth=1.5, label='x^2 + y^2 = 1')
axes[0].set_title('Path of x_k in SLSQP(scipy.optimize)')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].legend(loc='best')
fig.colorbar(contourf_0, ax=axes[0])

contourf_1 = axes[1].contourf(grid[0], grid[1], f_grid, levels=100)
axes[1].plot([xk[0] for xk in x_sqp_mine], [xk[1] for xk in x_sqp_mine],
             color='red', marker='o', markersize=5, markerfacecolor='none', markere
axes[1].scatter(x_sqp_mine[0][0], x_sqp_mine[0][1], color='yellow', marker='s', s=5
axes[1].scatter(x_sqp_mine[-1][0], x_sqp_mine[-1][1], color='yellow', marker='*', s
axes[1].plot(circle_x, circle_y, 'b--', linewidth=1.5, label='x^2 + y^2 = 1')
```

```
axes[1].set_title('Path of x_k in SQP_ALM(custom code)')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].legend(loc='best')
fig.colorbar(contourf_1, ax=axes[1])

fig.suptitle("Fig 2. xk of rosenbrock function subject to circle inequality constra
```

Out[27]:
```
Text(0.5, 0.98, 'Fig 2. xk of rosenbrock function subject to circle inequality con
straint')
```

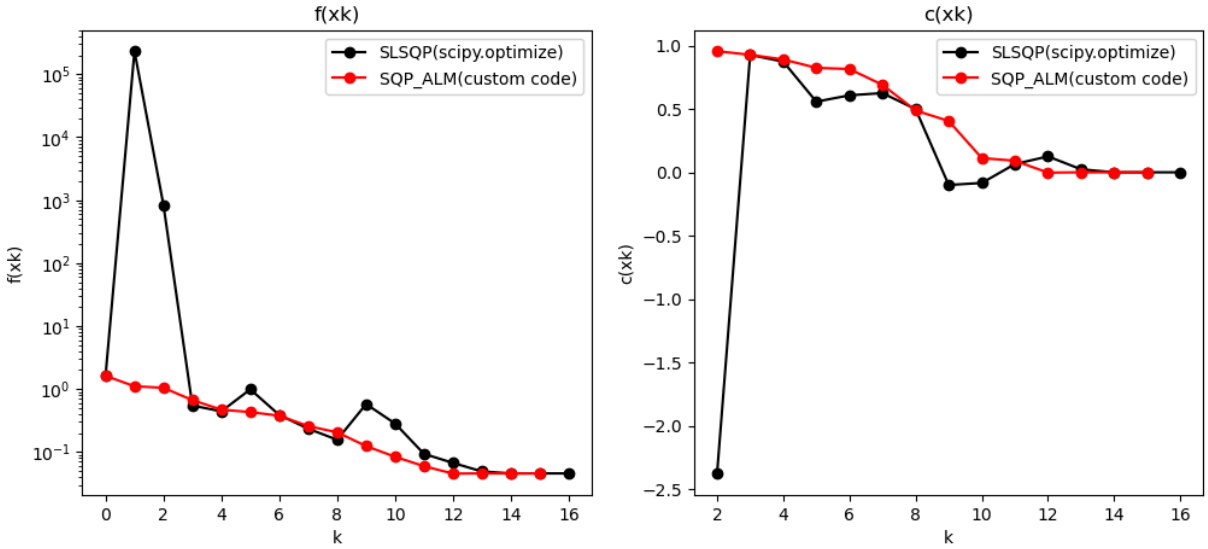


(iii) Discuss and compare the plots, as well as discuss the choice of parameters used in your results and their effect on the optimization.

In [28]:
```
fig, axes = plt.subplots(1, 2, figsize=(6*2, 5))
axes[0].plot(f_sqsqp_sci, 'ko-', label='SLSQP(scipy.optimize)')
axes[0].plot(f_sqp_mine, 'ro-', label='SQP_ALM(custom code)')
axes[0].set_yscale('log')
axes[0].set_title('f(xk)')
axes[0].set_xlabel('k')
axes[0].set_ylabel('f(xk)')
axes[0].legend()

axes[1].plot(np.arange(2, len(c_sqsqp_sci)), c_sqsqp_sci[2:], 'ko-', label='SLSQP(s
axes[1].plot(np.arange(2, len(c_sqp_mine)), c_sqp_mine[2:], 'ro-', label='SQP_ALM(c
# axes[1].set_yscale('log')
axes[1].set_title('c(xk)')
axes[1].set_xlabel('k')
axes[1].set_ylabel('c(xk)')
axes[1].legend()

fig.suptitle("Fig 3. f(xk) and c(xk) of rosenbrock function subject to circle inequ
```

Out[28]:
```
Text(0.5, 0.98, 'Fig 3. f(xk) and c(xk) of rosenbrock function subject to circle i
nequality constraint')
```

Fig 3. f(xk) and c(xk) of rosenbrock function subject to circle inequality constraint



I solved the given optimization problem using two methods: **SciPy's `optimize` SLSQP algorithm** and **my own SQP implementation**, in which the **intermediate solver for the QP subproblem is an Augmented Lagrangian Method (ALM)**. I then compared the results obtained from the two approaches. Since SciPy's SLSQP provides only the final solution and the log of the iterate sequence $\{x_k\}$, and does **not** expose the Lagrange multipliers or any related internal quantities, it was not possible to compute the Lagrangian gradient for SLSQP. Therefore, the plot of the Lagrangian gradient $\|\nabla L(x_k, \lambda_k)\|$ is presented **only for my SQP implementation**.

I am not fully aware of the exact internal mechanics of SciPy's SLSQP. However, from the observed behavior at $k = 1$ —where SLSQP exhibits a substantial overshoot compared to my SQP method—it seems likely that SLSQP performs a rather aggressive BFGS Hessian update during the early iterations. In contrast, my SQP implementation constructs the Hessian of the Lagrangian for the QP subproblem in a more conservative way. Moreover, I applied a **hot-start strategy**, where the step p_k^* obtained from the ALM at iteration k is used as the initial guess p^0 for solving the QP subproblem at iteration k+1. As a result, my SQP algorithm tends to exhibit a more stable and smooth optimization path.

2. Constrained Brequet Range equation

Take the problem of the Brequet Range equation from Assignment 2. According to the aircraft manufacturer, the maximum altitude that the aircraft can fly at is 2 · 104 m. Furthermore, in the area of interest the maximum speed the aircraft can fly at is 540 km/hr.

(a) Solve the same optimization problem as in Assignment 3 with the maximum constraints given above and appropriate minimum constraints using the `scipy.optimize`. Compute the gradient and provide it to the `scipy.optimize` library.

In [29]:
```
sys.path.append('../Assignment_3')
import breguet_range_equation_fromprofessor
import importlib
importlib.reload(breguet_range_equation_fromprofessor)
from breguet_range_equation_fromprofessor import *
```

In [30]:
```
# obj function for scipy(numpy type)
def f2(x):
    x_t = torch.as_tensor(x)
    return -breguet_range(x_t[0], x_t[1], 162400.0, 100.0).numpy().astype(np.float6

# obj function for AD using torch(torch type)
def f2_for_AD_troch(x):
    return -breguet_range(x[0], x[1], 162400.0, 100.0) # negative sign for maximum

# gradient of obj function
def grad_f2(x:np.ndarray) -> np.float64:
    x = torch.tensor(x, requires_grad=True)
    f2_for_AD_troch(x).backward() # torch type
    return x.grad.detach().numpy().astype(np.float64)

def c1(x): # v >= 1 [m/s]
    return x[0] - 1

def grad_c1(x):
    x = torch.tensor(x, requires_grad=True)
    c1(x).backward()
    return x.grad.detach().numpy()

def c2(x): # v <= 540*1000/3600 [m/s]
    return -x[0] + 540*1000/3600

def grad_c2(x):
    x = torch.tensor(x, requires_grad=True)
    c2(x).backward()
    return x.grad.detach().numpy()

def c3(x): # v >= 1 [m]
    return x[1] - 1

def grad_c3(x):
    x = torch.tensor(x, requires_grad=True)
    c3(x).backward()
    return x.grad.detach().numpy()

def c4(x): # v <= 20000 [m]
    return -x[1] + 2*1e4

def grad_c4(x):
    x = torch.tensor(x, requires_grad=True)
    c4(x).backward()
    return x.grad.detach().numpy()
```

In [31]:
```
# ------------------------------------------------------------------ xk Log of SLSQ
x_sqsqp_sci2 = []
```

```
def cb2(xk):
    x_sqsqp_sci2.append(xk.copy())
```

In [32]:
```
x0_2 = np.array([50.0, 10000.0])
x_sqsqp_sci2.append(x0_2)

constraints2 = [
    {'type': 'ineq', 'fun': c1, 'jac': grad_c1},
    {'type': 'ineq', 'fun': c2, 'jac': grad_c2},
    {'type': 'ineq', 'fun': c3, 'jac': grad_c3},
    {'type': 'ineq', 'fun': c4, 'jac': grad_c4},
]

log_sqsqp_sci2 = sci_opt.minimize(fun=f2, x0=x0_2, jac=grad_f2, method='SLSQP', con
log_sqp_mine2 = sqp(f=f2, ce=[], ci=[c1, c2, c3, c4], x0=x0_2, inner_opt=3, tol=1e-
```

```
 log - SQP
‖Δx‖ = 9.84e+01, x01 = [148.38012554, 9999.77148155] | f = -1.0380e+04, ‖∇L‖ = 1.65e
+01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 7.05e+00, x02 = [141.33773233, 10000.17996551] | f = -1.0439e+04, ‖∇L‖ = 8.55
e-01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 8.60e-01, x03 = [141.30096925, 10001.03876003] | f = -1.0440e+04, ‖∇L‖ = 8.55
e-01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 4.39e+00, x04 = [141.33206054, 10005.42462359] | f = -1.0444e+04, ‖∇L‖ = 8.55
e-01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 2.19e+01, x05 = [141.47637087, 10027.35729717] | f = -1.0462e+04, ‖∇L‖ = 8.55
e-01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 1.10e+02, x06 = [142.19919310, 10137.08691684] | f = -1.0556e+04, ‖∇L‖ = 8.58
e-01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 5.50e+02, x07 = [145.83005042, 10687.32734091] | f = -1.1030e+04, ‖∇L‖ = 8.74
e-01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 0.00e+00


 log - SQP
‖Δx‖ = 2.23e+03, x08 = [150.00000000, 12921.44604315] | f = -1.2716e+04, ‖∇L‖ = 1.42
e+01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 1.80e-09, ‖λ‖ = 0.00e+00, ‖v‖ = 2.55e+01


 log - SQP
‖Δx‖ = 2.49e+03, x09 = [149.99999919, 15412.74819746] | f = -1.2871e+04, ‖∇L‖ = 1.00
e+01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 1.05e+02


 log - SQP
‖Δx‖ = 1.15e+03, x10 = [149.99999983, 14265.59407516] | f = -1.3101e+04, ‖∇L‖ = 6.82
e-01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 8.04e+01


 log - SQP
‖Δx‖ = 8.62e+01, x11 = [150.00000051, 14351.83650324] | f = -1.3102e+04, ‖∇L‖ = 1.45
e-01, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 5.07e-07, ‖λ‖ = 0.00e+00, ‖v‖ = 8.36e+01


 log - SQP
```

```
‖Δx‖ = 2.37e+00, x12 = [150.00000054, 14354.20377839] | f = -1.3102e+04, ‖∇L‖ = 1.46
e-04, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 5.40e-07, ‖λ‖ = 0.00e+00, ‖v‖ = 8.39e+01


 log - SQP
‖Δx‖ = 1.37e-02, x13 = [149.99999954, 14354.19009597] | f = -1.3102e+04, ‖∇L‖ = 6.91
e-07, ‖ce‖∞ = 0.00e+00, ‖ci‖∞ = 0.00e+00, ‖λ‖ = 0.00e+00, ‖v‖ = 8.39e+01

alpha_k did not meet the armijo condition ...

Final iterate: x* = [  149.99999954 14354.19009597]
f(x*)         = -13102.381588438351
λ*            = [0.]
v*            = [ 0.         83.85967372  0.          0.        ]
‖∇L(x*)‖     = 9.800e-05
‖ce(x*)‖∞   = 0.000e+00
‖ci(x*)‖∞ = 0.000e+00
```

In [33]:
```
f_sqsqp_sci2 = []
grad_f_sqsqp_sci2 = []
c1_sqsqp_sci2 = []
c2_sqsqp_sci2 = []
c3_sqsqp_sci2 = []
c4_sqsqp_sci2 = []

for xk in x_sqsqp_sci2:
    f_sqsqp_sci2.append(-f2(xk))
    grad_f_sqsqp_sci2.append(np.linalg.norm(grad_f2(xk)))
    c1_sqsqp_sci2.append(c1(xk))
    c2_sqsqp_sci2.append(c2(xk))
    c3_sqsqp_sci2.append(c3(xk))
    c4_sqsqp_sci2.append(c4(xk))
```

In [34]:
```
x_sqp_mine2 = log_sqp_mine2[0]
f_sqp_mine2 = [-f_k for f_k in log_sqp_mine2[1]]
grad_f_sqp_mine2 = [np.linalg.norm(grad_f2_k) for grad_f2_k in log_sqp_mine2[2]]
grad_L_sqp_mine2 = [np.linalg.norm(grad_L_k) for grad_L_k in log_sqp_mine[3]]
c1_sqp_mine2 = [c_k[0] for c_k in log_sqp_mine2[5]]
c2_sqp_mine2 = [c_k[1] for c_k in log_sqp_mine2[5]]
c3_sqp_mine2 = [c_k[2] for c_k in log_sqp_mine2[5]]
c4_sqp_mine2 = [c_k[3] for c_k in log_sqp_mine2[5]]
```

In [35]:
```
resolution = .1
step_V = 50*resolution
step_h = 5000*resolution
V_limit = 200 # [m/s]
h_limit = 20000 # [m]
grid = np.meshgrid(np.arange(0.01, V_limit+step_V, step_V), np.arange(0.01, h_limit
Range = -f2(grid)
```

```
/tmp/ipykernel_2324/3867621435.py:3: UserWarning: Creating a tensor from a list of n
umpy.ndarrays is extremely slow. Please consider converting the list to a single num
py.ndarray with numpy.array() before converting to a tensor. (Triggered internally a
t /pytorch/torch/csrc/utils/tensor_new.cpp:253.)
  x_t = torch.as_tensor(x)
```

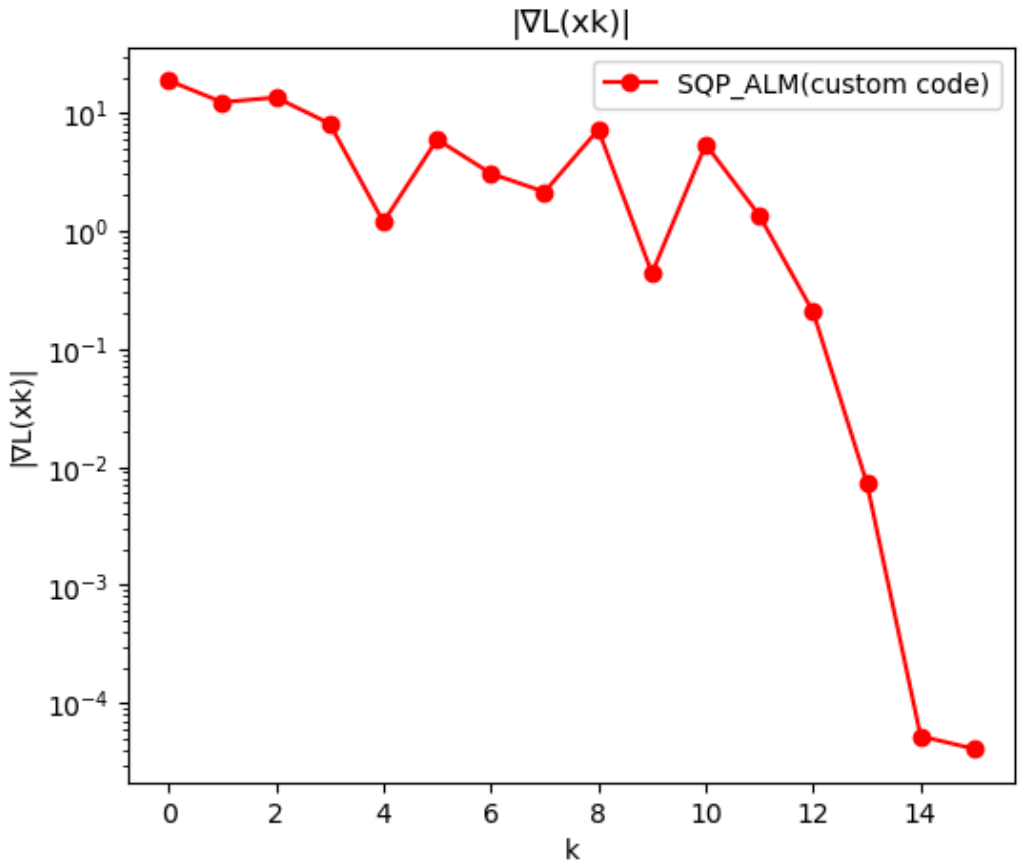
(i) Convergence gradient of the Lagrangian (y-axis: log of the gradient, x-axis: iteration) and a comparison of the convergence.

Scipy does not provide its history of gradient of Lagrangian function.
Therefore I plotted only the history of gradient of Lagrangian function within the solution using my own SQP code.

In [36]:
```
fig, axes = plt.subplots(1, 1, figsize=(6, 5))
axes.plot(grad_L_sqp_mine2, 'ro-', label='SQP_ALM(custom code)')
axes.set_yscale('log')
axes.set_title('|∇L(xk)|')
axes.set_xlabel('k')
axes.set_ylabel('|∇L(xk)|')
axes.legend()

fig.suptitle("Fig 4. |∇L(xk)| of Breguet range equation subject to V, h bound const
```

Out[36]:
```
Text(0.5, 0.98, 'Fig 4. |∇L(xk)| of Breguet range equation subject to V, h bound c
onstraints')
```

Fig 4. |∇L(xk)| of Breguet range equation subject to V, h bound constraints



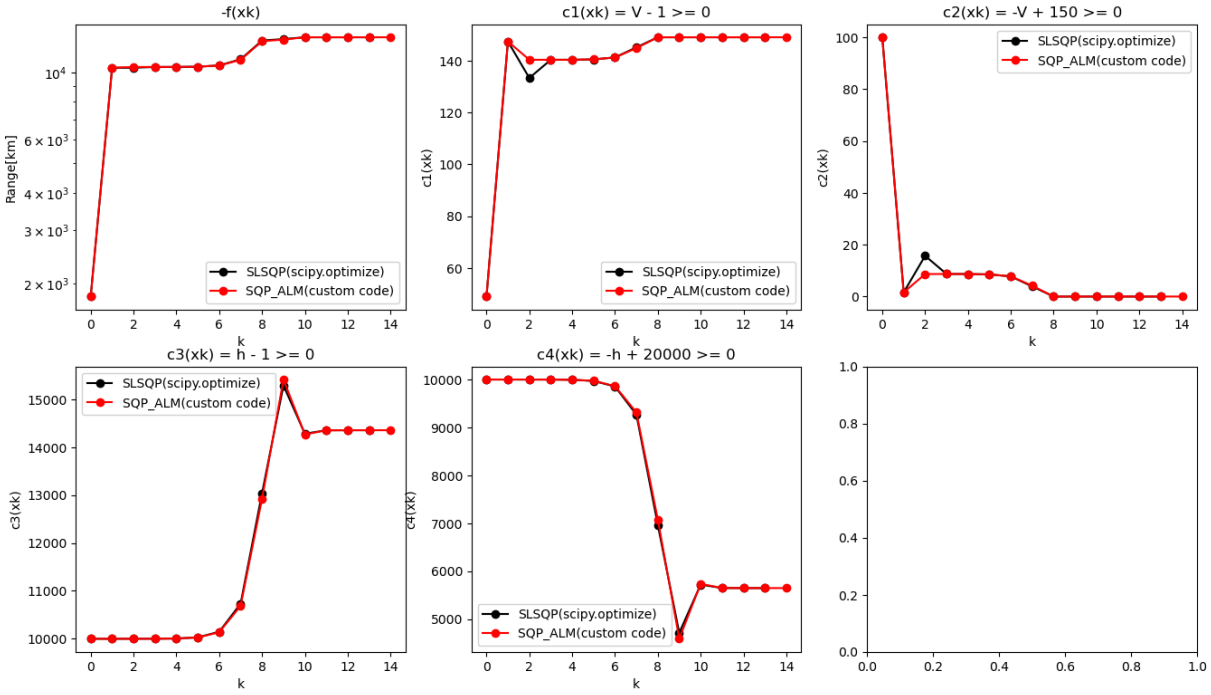
(ii) Convergence of the range and constraints as a function of design iterations.

In [38]:
```
fig, axes = plt.subplots(2, 3, figsize=(8*2, 3*3))
axes[0, 0].plot(f_sqsqp_sci2, 'ko-', label='SLSQP(scipy.optimize)')
```

```
axes[0, 0].plot(f_sqp_mine2, 'ro-', label='SQP_ALM(custom code)')
axes[0, 0].set_yscale('log')
axes[0, 0].set_title('-f(xk)')
axes[0, 0].set_xlabel('k')
axes[0, 0].set_ylabel('Range[km]')
axes[0, 0].legend()

axes[0, 1].plot(c1_sqsqp_sci2, 'ko-', label='SLSQP(scipy.optimize)')
axes[0, 1].plot(c1_sqp_mine2, 'ro-', label='SQP_ALM(custom code)')
axes[0, 1].set_title('c1(xk) = V - 1 >= 0')
axes[0, 1].set_xlabel('k')
axes[0, 1].set_ylabel('c1(xk)')
axes[0, 1].legend()

axes[0, 2].plot(c2_sqsqp_sci2, 'ko-', label='SLSQP(scipy.optimize)')
axes[0, 2].plot(c2_sqp_mine2, 'ro-', label='SQP_ALM(custom code)')
axes[0, 2].set_title('c2(xk) = -V + 150 >= 0')
axes[0, 2].set_xlabel('k')
axes[0, 2].set_ylabel('c2(xk)')
axes[0, 2].legend()

axes[1, 0].plot(c3_sqsqp_sci2, 'ko-', label='SLSQP(scipy.optimize)')
axes[1, 0].plot(c3_sqp_mine2, 'ro-', label='SQP_ALM(custom code)')
axes[1, 0].set_title('c3(xk) = h - 1 >= 0')
axes[1, 0].set_xlabel('k')
axes[1, 0].set_ylabel('c3(xk)')
axes[1, 0].legend()

axes[1, 1].plot(c4_sqsqp_sci2, 'ko-', label='SLSQP(scipy.optimize)')
axes[1, 1].plot(c4_sqp_mine2, 'ro-', label='SQP_ALM(custom code)')
axes[1, 1].set_title('c4(xk) = -h + 20000 >= 0')
axes[1, 1].set_xlabel('k')
axes[1, 1].set_ylabel('c4(xk)')
axes[1, 1].legend()

fig.suptitle("Fig 5. -f(xk) and c1(xk), c2(xk), c3(xk), c4(xk) of Breguet range equ
```

Out[38]: Text(0.5, 0.98, 'Fig 5. -f(xk) and c1(xk), c2(xk), c3(xk), c4(xk) of Breguet range equation subject to V, h bound constraints \n (f(x) is negative reverse version of Breguet range eqn for minimization, such that means -f(x) is positive)')

Fig 5. -f(xk) and c1(xk), c2(xk), c3(xk), c4(xk) of Breguet range equation subject to V, h bound constraints
(f(x) is negative reverse version of Breguet range eqn for minimization, such that means -f(x) is positive)

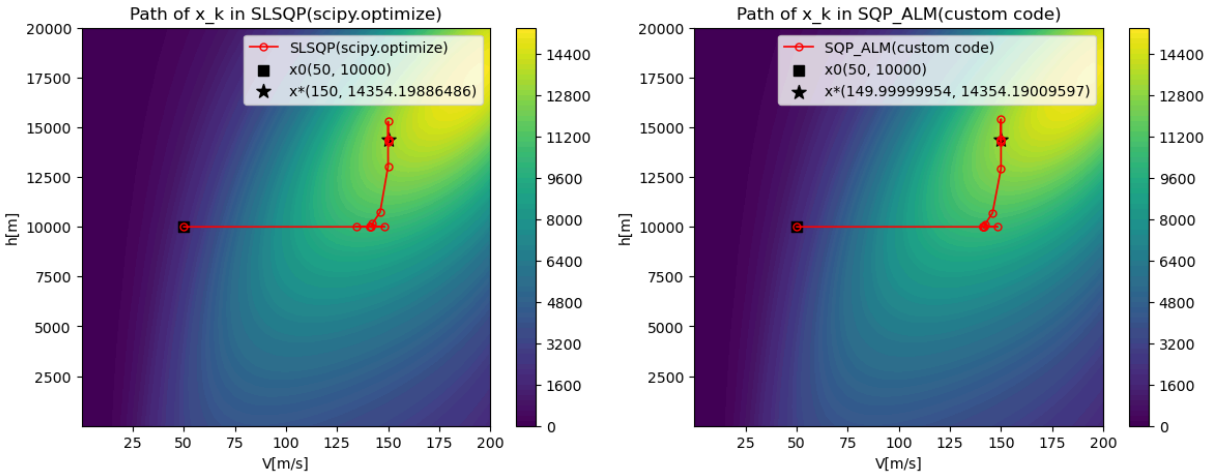


(iii) Contour plot of the path of the optimization algorithm.

In [215…

```
fig2, axes2 = plt.subplots(1, 2, figsize=(7*2, 5))

contourf2_0 = axes2[0].contourf(grid[0], grid[1], Range, levels=100)
axes2[0].plot([xk[0] for xk in x_sqsqp_sci2], [xk[1] for xk in x_sqsqp_sci2],
              color='red', marker='o', markersize=5, markerfacecolor='none', markere
axes2[0].scatter(x_sqsqp_sci2[0][0], x_sqsqp_sci2[0][1], color='black', marker='s',
axes2[0].scatter(x_sqsqp_sci2[-1][0], x_sqsqp_sci2[-1][1], color='black', marker='*
axes2[0].set_title('Path of x_k in SLSQP(scipy.optimize)')
axes2[0].set_xlabel('V[m/s]')
axes2[0].set_ylabel('h[m]')
axes2[0].legend(loc='best')
fig2.colorbar(contourf2_0, ax=axes2[0])

contourf2_1 = axes2[1].contourf(grid[0], grid[1], Range, levels=100)
axes2[1].plot([xk[0] for xk in x_sqp_mine2], [xk[1] for xk in x_sqp_mine2],
              color='red', marker='o', markersize=5, markerfacecolor='none', markere
axes2[1].scatter(x_sqp_mine2[0][0], x_sqp_mine2[0][1], color='black', marker='s', s
axes2[1].scatter(x_sqp_mine2[-1][0], x_sqp_mine2[-1][1], color='black', marker='*',
axes2[1].set_title('Path of x_k in SQP_ALM(custom code)')
axes2[1].set_xlabel('V[m/s]')
axes2[1].set_ylabel('h[m]')
axes2[1].legend(loc='best')
fig2.colorbar(contourf2_1, ax=axes2[1])
```

Out[215… <matplotlib.colorbar.Colorbar at 0x7b4657aa6210>


Path of x_k in SLSQP(scipy.optimize)
SLSQP(scipy.optimize)
x0(50, 10000)
x*(150, 14354.19886486)
h[m]
V[m/s]
Path of x_k in SQP_ALM(custom code)
SQP_ALM(custom code)
x0(50, 10000)
x*(149.99999954, 14354.19009597)
h[m]
V[m/s]

In [90]:
```
all = [var for var in globals() if var[0] != '_']
for var in all:
    del globals()[var]
del var, all
```

In [91]:
```
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import sys
sys.path.append('../Assignment_3')
from breguet_range_equation_fromprofessor import *
```

In [92]:
```
def obj_func(x:torch.tensor) -> torch.tensor:
    return breguet_range(x[:, 0], x[:, 1], 162400.0, 100.0)
```

In [93]:
```
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        n_inputs = 2;
        n_layer1 = 50;
        n_layer2 = 100;
        n_layer3 = 50;
        n_outputs = 1;
        # Specify layers of the neural network.
        self.hidden1 = torch.nn.Linear(n_inputs, n_layer1) # 2 inputs, n_layer1 neu
        self.hidden2 = torch.nn.Linear(n_layer1, n_layer2) # n_layer1 inputs, n_lay
        self.hidden3 = torch.nn.Linear(n_layer2, n_layer3) # n_layer2 inputs, n_lay
        self.output_layer = torch.nn.Linear(n_layer3, n_outputs) # n_layer3 inputs,

        # Initialize weights and biases.
        torch.nn.init.eye_(self.hidden1.weight);
        torch.nn.init.ones_(self.hidden1.bias);
        torch.nn.init.eye_(self.hidden2.weight);
        torch.nn.init.ones_(self.hidden2.bias);
        torch.nn.init.eye_(self.hidden3.weight);
        torch.nn.init.ones_(self.hidden3.bias);
        torch.nn.init.eye_(self.output_layer.weight);
        torch.nn.init.ones_(self.output_layer.bias);

    def forward(self, x):
        x = torch.relu(self.hidden1(x))
        x = torch.relu(self.hidden2(x))
        x = torch.relu(self.hidden3(x))
        x = self.output_layer(x)
        return x

class Net_test(torch.nn.Module):
    def __init__(self):
        super(Net_test, self).__init__()
        n_inputs = 2;
        n_layer1 = 10;
        n_layer2 = 10;
```

```
        n_outputs = 1;
        # Specify layers of the neural network.
        self.hidden1 = torch.nn.Linear(n_inputs, n_layer1) # 2 inputs, n_layer1 neu
        self.hidden2 = torch.nn.Linear(n_layer1, n_layer2) # n_layer1 inputs, n_lay
        self.output_layer = torch.nn.Linear(n_layer2, n_outputs) # n_layer3 inputs,

        # Initialize weights and biases.
        torch.nn.init.eye_(self.hidden1.weight);
        torch.nn.init.ones_(self.hidden1.bias);
        torch.nn.init.eye_(self.hidden2.weight);
        torch.nn.init.ones_(self.hidden2.bias);
        torch.nn.init.eye_(self.output_layer.weight);
        torch.nn.init.ones_(self.output_layer.bias);

    def forward(self, x):
        x = torch.relu(self.hidden1(x))
        x = torch.relu(self.hidden2(x))
        x = self.output_layer(x)
        return x

def get_trained_objfunc(n_samples_1d):

    # Data to train the neural network.
    x_1d =  torch.linspace(10, 200, n_samples_1d);
    y_1d =  torch.linspace(1250, 25000, n_samples_1d);
    x_samples,y_samples = np.meshgrid(x_1d, y_1d);
    xy_tensor = torch.from_numpy(np.concatenate((x_samples.reshape(n_samples_1d**2,
    f_exact = obj_func(xy_tensor).unsqueeze(1);

    f_scaled = f_exact / f_exact.abs().max()
    print(f'f_exact.abs().max() : {f_exact.abs().max()}')
    f_neural_net = Net()
    loss_function = torch.nn.MSELoss() # Using squared L2 norm of the error.
    optimizer = torch.optim.Adam(f_neural_net.parameters(), lr=0.005)  # Using Adam

    log_loss = []

    print("Training neural network of objective function");
    for epoch in range(10000):
        optimizer.zero_grad()
        outputs = f_neural_net(xy_tensor)
        loss = loss_function(outputs, f_scaled)
        loss.backward() # Backpropagation and computing gradients w.r.t. weights an
        optimizer.step() # Update weights and biases.

        log_loss.append(loss.detach().numpy())

        if epoch % 500 == 0:
            print("Epoch {}: Loss = {}".format(epoch, loss.detach().numpy()))

    print("Finished training neural network of objective function");

    return f_neural_net, log_loss;

def get_trained_objfunc2(n_samples_1d):
```

```
    # Data to train the neural network.
    x_1d =  torch.linspace(10, 200, n_samples_1d);
    y_1d =  torch.linspace(1250, 25000, n_samples_1d);
    x_samples,y_samples = np.meshgrid(x_1d, y_1d);
    xy_tensor = torch.from_numpy(np.concatenate((x_samples.reshape(n_samples_1d**2,
    f_exact = obj_func(xy_tensor).unsqueeze(1);

    f_scaled = f_exact / f_exact.abs().max()
    print(f'f_exact.abs().max() : {f_exact.abs().max()}')
    f_neural_net = Net_test()
    loss_function = torch.nn.MSELoss() # Using squared L2 norm of the error.
    optimizer = torch.optim.Adam(f_neural_net.parameters(), lr=0.005)  # Using Adam

    log_loss = []

    print("Training neural network of objective function");
    for epoch in range(10000):
        optimizer.zero_grad()
        outputs = f_neural_net(xy_tensor)
        loss = loss_function(outputs, f_scaled)
        loss.backward() # Backpropagation and computing gradients w.r.t. weights an
        optimizer.step() # Update weights and biases.

        log_loss.append(loss.detach().numpy())

        if epoch % 500 == 0:
            print("Epoch {}: Loss = {}".format(epoch, loss.detach().numpy()))

    print("Finished training neural network of objective function");

    return f_neural_net, log_loss;
```

3. Breguet Range eqn using Neural Network(NN)

Solve the unconstrained Brequet Range equation from Assignment 2 but use a Neural Network approach.

You may use the provided code to optimize the Rosenbrock as a starting point.

In your report, please provide the following:

(a) Train a Neural Network to compute the Brequet Range Equation. Define the initial sample set to be 50 data points (pairs of velocity and altitude) and decide on the number of epochs.

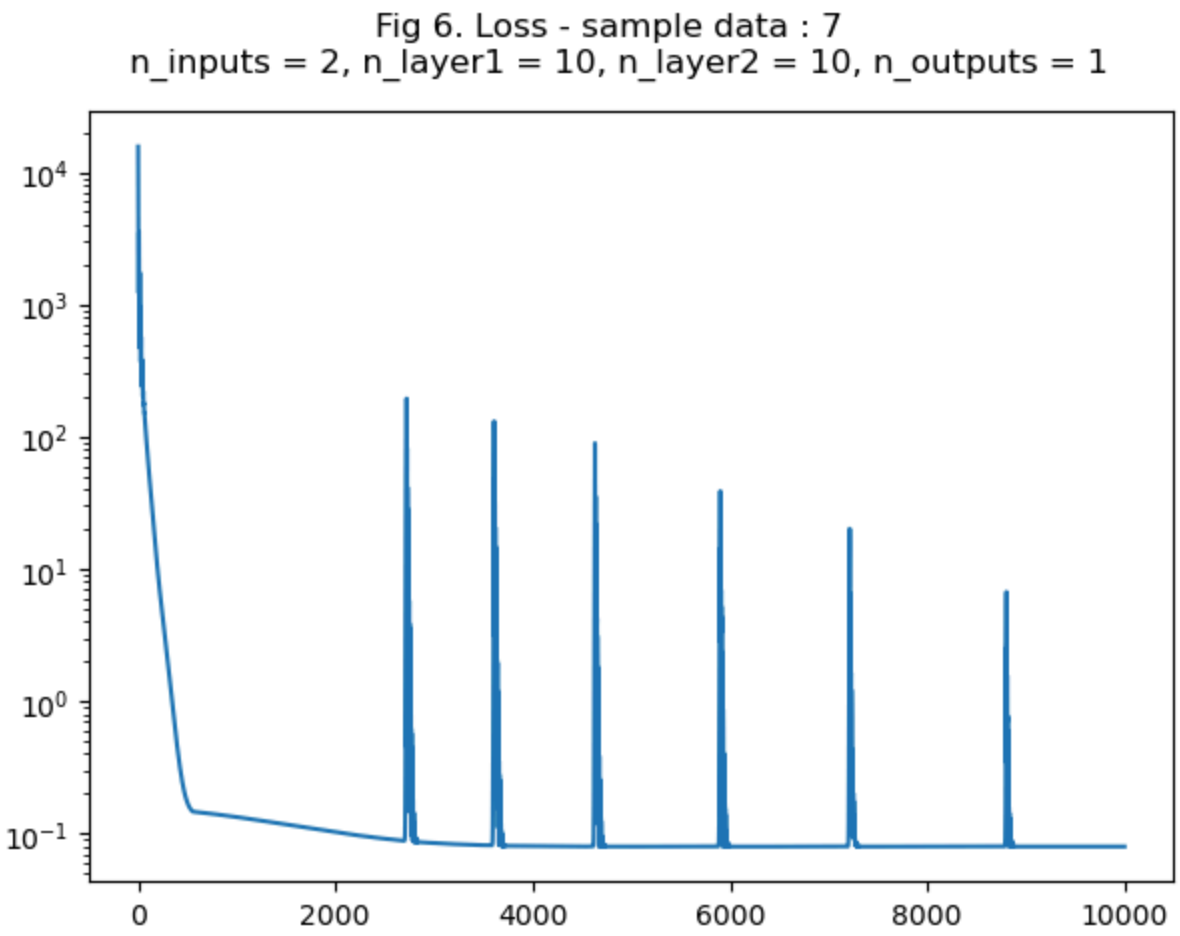
- Provide a plot of the convergence of the loss function as a number of epochs. (Note: Use 10000 as the maximum number of epochs and you may choose toincrease it if necessary)

In [94]:
```
# sample data : 7 / n_inputs = 2; n_layer1 = 10; n_layer2 = 10; n_outputs = 1
fr_a_1, log_loss_a_1 = get_trained_objfunc2(7)
```

```
f_exact.abs().max() : 14874.58984375
Training neural network of objective function
Epoch 0: Loss = 15582.796875
Epoch 500: Loss = 0.170798659324646
Epoch 1000: Loss = 0.13346655666828156
Epoch 1500: Loss = 0.1172540932893753
Epoch 2000: Loss = 0.10258111357688904
Epoch 2500: Loss = 0.09152934700250626
Epoch 3000: Loss = 0.08490324020385742
Epoch 3500: Loss = 0.08179004490375519
Epoch 4000: Loss = 0.08047069609165192
Epoch 4500: Loss = 0.08003736287355423
Epoch 5000: Loss = 0.07991273701190948
Epoch 5500: Loss = 0.07989367097616196
Epoch 6000: Loss = 0.07994727045297623
Epoch 6500: Loss = 0.07989688962697983
Epoch 7000: Loss = 0.07989685982465744
Epoch 7500: Loss = 0.07990603893995285
Epoch 8000: Loss = 0.07990584522485733
Epoch 8500: Loss = 0.07990581542253494
Epoch 9000: Loss = 0.07991507649421692
Epoch 9500: Loss = 0.07991498708724976
Finished training neural network of objective function
```

In [95]:
```
fig, ax = plt.subplots(1, 1, figsize=(7, 5))
ax.plot(log_loss_a_1, label='loss')
ax.set_yscale('log')
fig.suptitle('Fig 6. Loss - sample data : 7 \n n_inputs = 2, n_layer1 = 10, n_layer
```

Out[95]:
```
Text(0.5, 0.98, 'Fig 6. Loss - sample data : 7 \n n_inputs = 2, n_layer1 = 10, n_l
ayer2 = 10, n_outputs = 1')
```



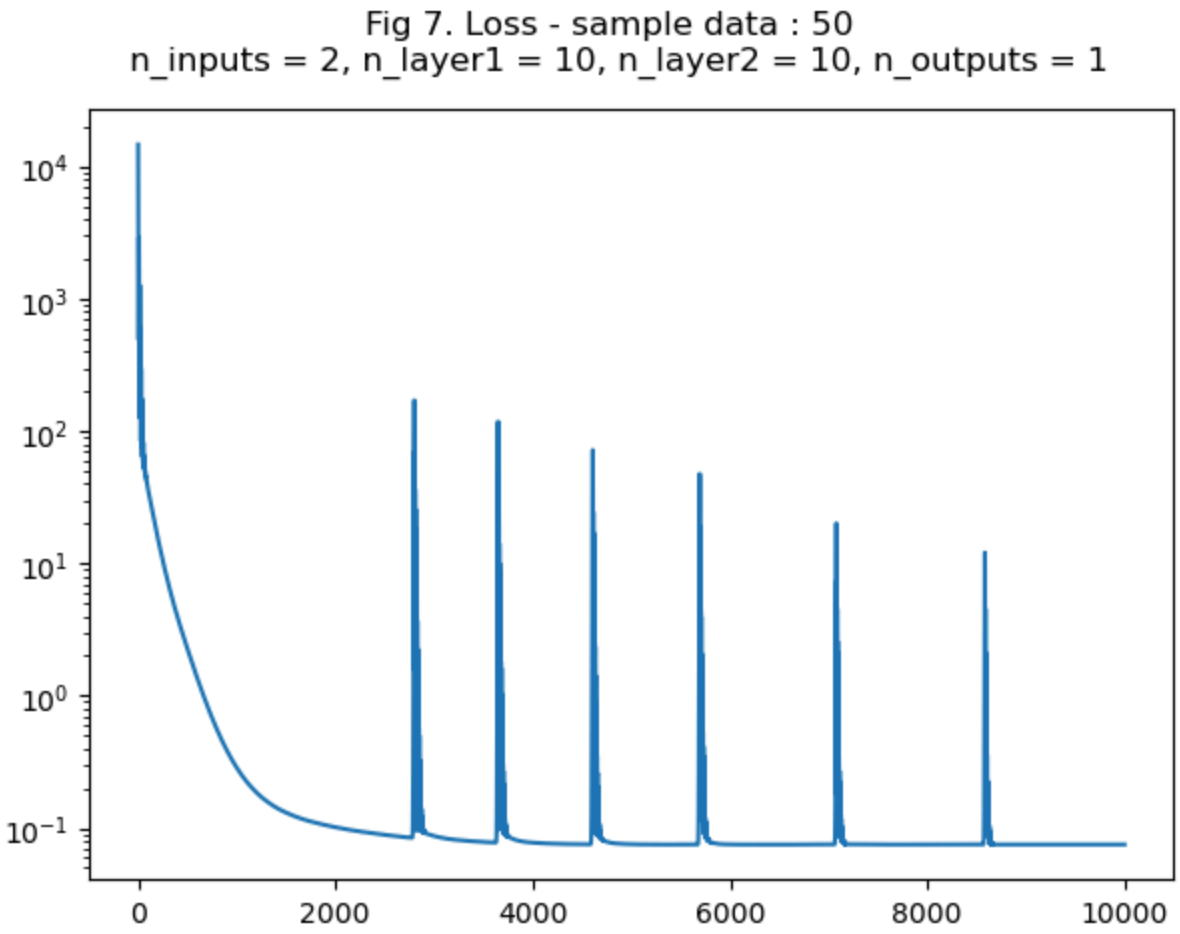
- Change the size of the initial sample set and compare the convergence of the loss function as a number of epochs.

In [96]:
```
# sample data : 50 / n_inputs = 2; n_layer1 = 10; n_layer2 = 10; n_outputs = 1
fr_a_2, log_loss_a_2 = get_trained_objfunc2(50)
```

```
f_exact.abs().max() : 15278.9013671875
Training neural network of objective function
Epoch 0: Loss = 14700.4130859375
Epoch 500: Loss = 2.2624950408935547
Epoch 1000: Loss = 0.28561949729919434
Epoch 1500: Loss = 0.130758136510849
Epoch 2000: Loss = 0.1014384776353836
Epoch 2500: Loss = 0.0888722762465477
Epoch 3000: Loss = 0.08708067238330841
Epoch 3500: Loss = 0.07849319279193878
Epoch 4000: Loss = 0.07756180316209793
Epoch 4500: Loss = 0.0754406675696373
Epoch 5000: Loss = 0.07548630237579346
Epoch 5500: Loss = 0.075055330991745
Epoch 6000: Loss = 0.07521291822195053
Epoch 6500: Loss = 0.07505495846271515
Epoch 7000: Loss = 0.07505445927381516
Epoch 7500: Loss = 0.07508036494255066
Epoch 8000: Loss = 0.07508016377687454
Epoch 8500: Loss = 0.07507995516061783
Epoch 9000: Loss = 0.07512041926383972
Epoch 9500: Loss = 0.07511994987726212
Finished training neural network of objective function
```

In [97]:
```
fig, ax = plt.subplots(1, 1, figsize=(7, 5))
ax.plot(log_loss_a_2, label='loss')
ax.set_yscale('log')
fig.suptitle('Fig 7. Loss - sample data : 50 \n n_inputs = 2, n_layer1 = 10, n_laye
```

Out[97]:
```
Text(0.5, 0.98, 'Fig 7. Loss - sample data : 50 \n n_inputs = 2, n_layer1 = 10, n_
layer2 = 10, n_outputs = 1')
```



- Change the Neural Network model itself. You may choose at least two different parameters: such as number of hidden layers, size of nodes per hidden layer, etc. Compare the convergence of the loss function between the cases.

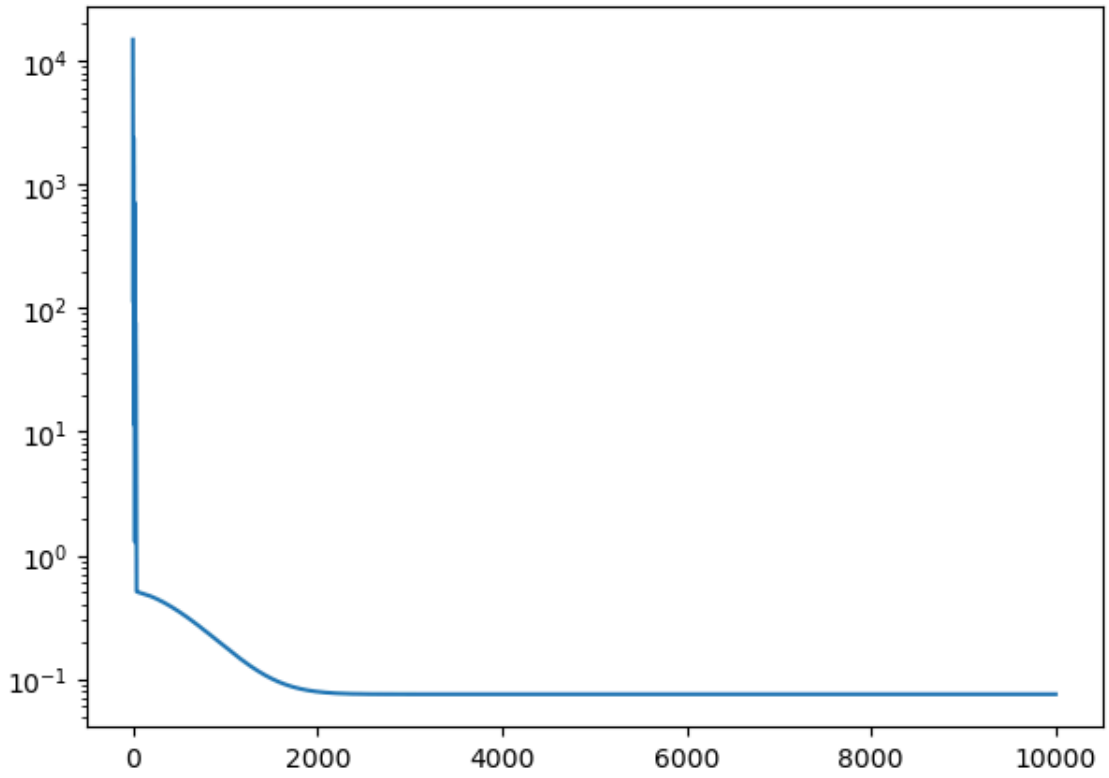
In [98]:
```
# sample data : 50 / n_inputs = 2; n_layer1 = 50; n_layer2 = 100; n_layer2 = 50; n_
fr, log_loss = get_trained_objfunc(50)
```

```
f_exact.abs().max() : 15278.9013671875
Training neural network of objective function
Epoch 0: Loss = 14916.75
Epoch 500: Loss = 0.3507004678249359
Epoch 1000: Loss = 0.18180952966213226
Epoch 1500: Loss = 0.10095685720443726
Epoch 2000: Loss = 0.07888087630271912
Epoch 2500: Loss = 0.07558285444974899
Epoch 3000: Loss = 0.07534201443195343
Epoch 3500: Loss = 0.07533477246761322
Epoch 4000: Loss = 0.07533470541238785
Epoch 4500: Loss = 0.07533469051122665
Epoch 5000: Loss = 0.07533469051122665
Epoch 5500: Loss = 0.07533470541238785
Epoch 6000: Loss = 0.07533470541238785
Epoch 6500: Loss = 0.07533469796180725
Epoch 7000: Loss = 0.07533469796180725
Epoch 7500: Loss = 0.07533469796180725
Epoch 8000: Loss = 0.07533469796180725
Epoch 8500: Loss = 0.07533469796180725
Epoch 9000: Loss = 0.07533469796180725
Epoch 9500: Loss = 0.07533469796180725
Finished training neural network of objective function
```

In [99]:
```
fig, ax = plt.subplots(1, 1, figsize=(7, 5))
ax.plot(log_loss, label='loss')
ax.set_yscale('log')
fig.suptitle('Fig 8. Loss - sample data : 50 \n n_inputs = 2, n_layer1 = 50, n_laye
```

Out[99]:
```
Text(0.5, 0.98, 'Fig 8. Loss - sample data : 50 \n n_inputs = 2, n_layer1 = 50, n_
layer2 = 100, n_layer3 = 50, n_outputs = 1')
```

Fig 8. Loss - sample data : 50
n_inputs = 2, n_layer1 = 50, n_layer2 = 100, n_layer3 = 50, n_outputs = 1



(b) Build a Neural Network model to optimize the Brequet Range Equation.

In [100…

```
class Net_optimizationproblem(torch.nn.Module):
    def __init__(self,f_neural_net, constraints):
        super(Net_optimizationproblem, self).__init__()
        self.f_neural_net = f_neural_net; # NN of Objective function
        for param in self.f_neural_net.parameters(): # Keep the NN objective functi
            param.requires_grad = False; # Does not compute gradients wrt weights a
        self.constraints = constraints;
        self.linear = torch.nn.Linear(2,2); # Define NN of the Optimization Problem
        torch.nn.init.eye_(self.linear.weight);
        torch.nn.init.zeros_(self.linear.bias);


    def forward(self, x):
        x = self.linear(x);
        f_val = self.f_neural_net(x).squeeze();
        #f_val = obj_func(x); # Can also use the analytical objective function.
        constraints_val_1 = self.constraints[0](x);
        constraints_val_2 = self.constraints[1](x);
        constraints_val_3 = self.constraints[2](x);
        constraints_val_4 = self.constraints[3](x);

        output = f_val + 10*(torch.relu(-constraints_val_1) + torch.relu(-constrain
        return output;

#Custom loss function for optimization.
class CustomLoss(torch.nn.Module):
```

```
    def __init__(self):
        super(CustomLoss, self).__init__()

    def forward(self, predictions):
        return predictions;


def run_optimizer(objfunc_neural_net, constraints):
    optimization_problem = Net_optimizationproblem(objfunc_neural_net, constraints)
    loss_function = CustomLoss()
    optimizer = torch.optim.NAdam(optimization_problem.parameters(), lr=0.01);

    x_initial = torch.Tensor(1,2);
    x_initial[:,0] = 50.0;
    x_initial[:,1] = 10000.0;
    for epoch in range(10000):
        optimizer.zero_grad() # Gradient of wrt weights and biases are set to zero.
        outputs = optimization_problem(x_initial)
        loss = loss_function(outputs)
        loss.backward() # Backward propagation with automatic differentiation. Comp
        optimizer.step() # Updates weights and biases with specified learning rate
        if epoch % 1000 == 0:
            print("Epoch {}: Loss = {}".format(epoch, loss.detach().numpy()))

    x_optimal = optimization_problem.linear(x_initial);
    print("Optimal point = ", x_optimal);
    return x_optimal;
```

In [101…

```
def c1(x): # v >= 1 [m/s]
    return x[0][0] - 1.0

def c2(x): # v <= 540*1000/3600 [m/s]
    return -x[0][0] + 540*1000/3600

def c3(x): # v >= 1 [m]
    return x[0][1] - 1.0

def c4(x): # v <= 20000 [m]
    return -x[0][1] + 2*1e4
```

- Provide a plot of the convergence of the objective function as a number of epochs. Increase the number of epochs and determine the final required number of epochs.

- Compare the convergence against the SQP approach with respect to time per iteration.

Unfortunately I didn't have enough time to modify above functions for solving it. But the problem seems to be about the scaling issue.
Because I scaled training data when training fr, the order of fr might not be adeuqate to just handle by itself. And it would affect penalized loss function.

In [ ]:

```
run_optimizer(fr, [c1, c2, c3, c4])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[103], line 1
----> 1 run_optimizer(fr*15278.9013671875, [c1, c2, c3, c4])

TypeError: unsupported operand type(s) for *: 'Net' and 'float'
```