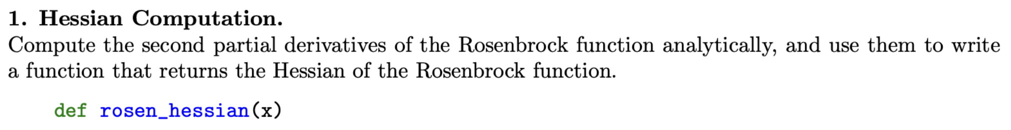
Numerical Optimization - Assignment 2

David Alvear(187594) - Nouf Farhoud(189731)

In [ ]:

```
import jax
import numpy as np
import jax.numpy as jnp
from numpy import linalg as la
import matplotlib.pyplot as plt
from numpy.random import default_rng
```

1. Hessian Computation



1. Hessian Computation.
Compute the second partial derivatives of the Rosenbrock function analytically, and use them to write a function that returns the Hessian of the Rosenbrock function.

```
def rosen_hessian(x)
```

In [ ]:

```
def rosen_hessian(x):
    y = x[1]
    x = x[0]
    hessian = [[(120*x**2 - 40*y) + 2, -40 * x],
               [-40 * x, 20]]
    return hessian

x = [1.0, 1.0]

hessian_matrix = rosen_hessian(x)
print("Hessian Matrix:")
print(hessian_matrix)
```

```
Hessian Matrix:
[[82.0, -40.0], [-40.0, 20]]
```

$$f(x,y) = (a-x)^2 + b(y-x^2)^2 \rightarrow 10(y-x^2)^2 + (1-x)^2$$

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial^2 y} \end{bmatrix}$$

$$\frac{\partial f}{\partial x} = -40x(y-x^2) - 2(1-x)$$

$$\frac{\partial f}{\partial y} = 20(y-x^2)$$

$$\frac{\partial^2 f}{\partial x^2} = 120x^2 - 40y + 2$$

$$\frac{\partial^2 f}{\partial y^2} = 20$$

$$\frac{\partial f}{\partial x \partial y} = -40x$$

2. Automatic Differentiation

2. Automatic differentiation.
In this exercise you will use Autograd functionality of the `jax` package to automatically differentiate native Python and Numpy code.

- Use `jax.grad` to generate a Python function that returns the gradient of the Rosenbrock function. Compare the automatically generated function to the Python function you wrote manually in Exercise 2 of the previous assignment.
- Use `jax.hessian` to generate a Python function that returns the Hessian of the Rosenbrock function. Compare it to the Python function you wrote manually Exercise 1 above.

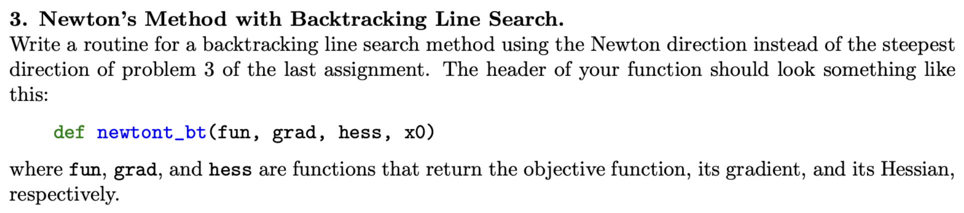
In [ ]:
```python
def myf(x):
    y = 10 * (x[1] - x[0]**2)**2 + (1 - x[0])**2
    return y

gf = jax.grad(myf)
Hf = jax.hessian(myf)

xk = np.array([1.0, 1.0])
```

```python
# print(xk)
print(gf(xk))
print(Hf(xk))
```

```
[0. 0.]
[[ 82. -40.]
 [-40.  20.]]
```

- The gradients of jax.grad and the one of the previous assignment both equate to [0. 0.].
- The jax.hessian also showed a result equal to that of question 1 in this assignment, as both equate to [[82. -40.] [-40. 20.]].

3. Newtons Method with Backtracking Line Search.



3. Newton's Method with Backtracking Line Search.
Write a routine for a backtracking line search method using the Newton direction instead of the steepest direction of problem 3 of the last assignment. The header of your function should look something like this:

```python
def newtont_bt(fun, grad, hess, x0)
```

where **fun**, **grad**, and **hess** are functions that return the objective function, its gradient, and its Hessian, respectively.

In [ ]:

```python
###################### Backtracking line search ######################
def backtracking_linesearch(f, gk, pk, xk, alpha=0.1, beta=0.8):
  t = 1
  while (f(xk + t*pk) > f(xk) + alpha * t * gk @ pk):
    t *= beta
  return t

###################### Newton Method function ######################
def newton_bt(fun, grad, hess, x0, tol=1e-5):
  x = x0
  history = np.array([x0])
  while (la.norm(grad(x))>tol):
    p = la.solve(hess(x), -grad(x))
    t = backtracking_linesearch(fun, grad(x), p, x)
    x += t*p
    history = np.vstack((history,x))
  return x, history
```

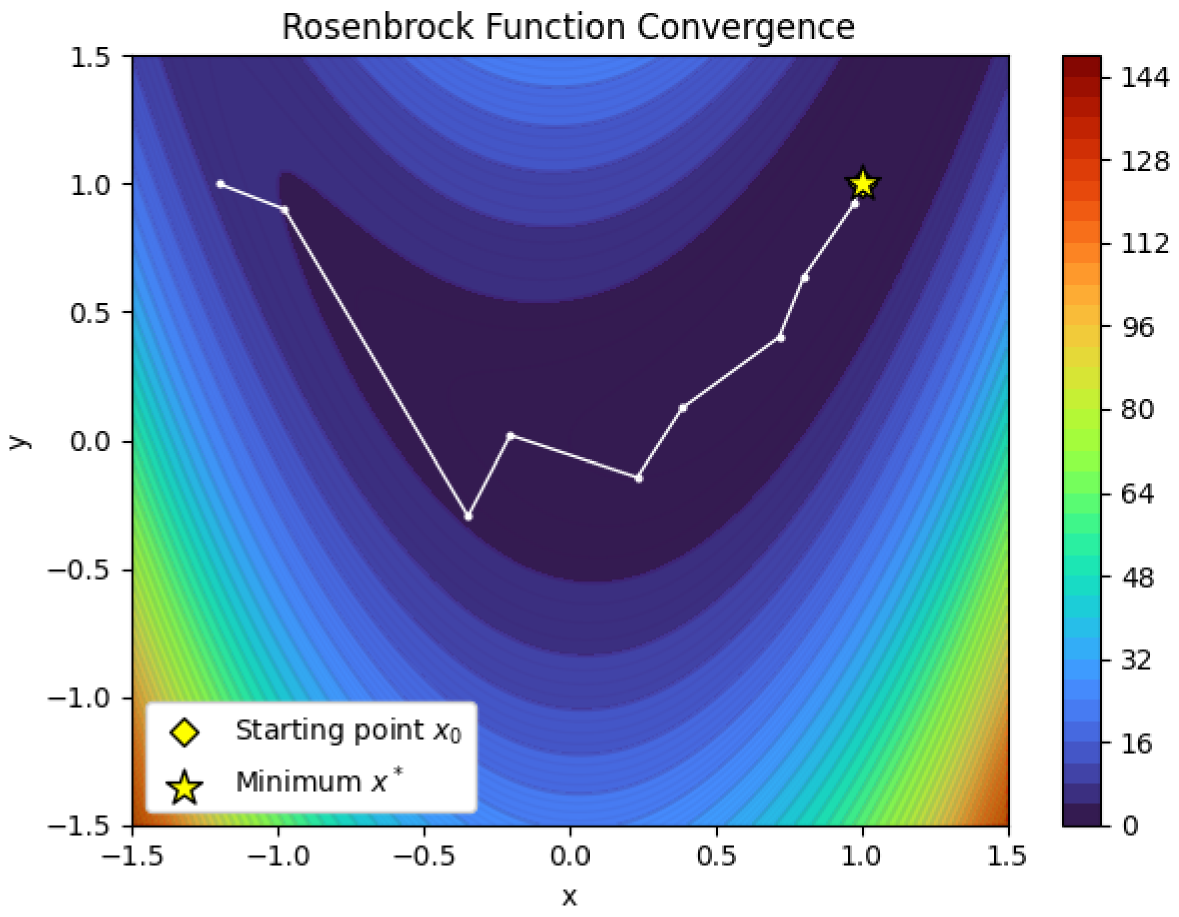
4. Performance of Newtons Method.

4. Performance of Newton's Method.
Test the performance of the algorithm on Rosenbrock's function $f(x) = 10(x_2 - x_1^2)^2 + (1 - x_1)^2$ starting at the point $[-1.2\ 1.0]^T$ by finding the number of iterations till convergence to a gradient norm of 10^{-5}.

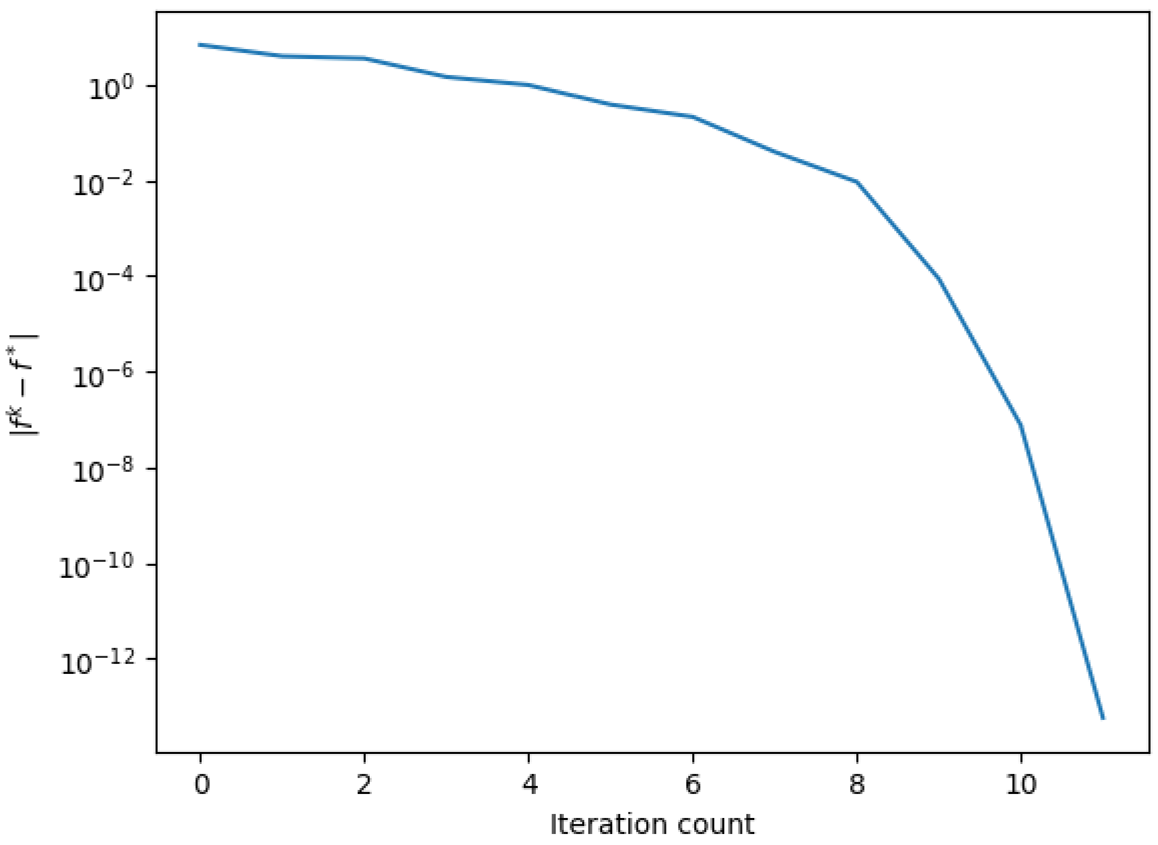
- Generate a semilog plot of $|f^k - f^*|$ vs. iteration count.
- Write an expression for the expected convergence behavior of the algorithm near the minimum
- Comment on the convergence plot, and compare it to that of the steepest descent direction

In [ ]:
```
############################################## Functions ###################
def rosenbrock(x):
    return 10 * (x[1] - x[0]**2)**2 + (1 - x[0])**2

grad_rosen = jax.grad(rosenbrock)
hessian_rosen = jax.hessian(rosenbrock)
############################################## Execution ###################

x0 = np.array([-1.2, 1])
x, history = newton_bt(rosenbrock, grad_rosen, hessian_rosen, x0)
```

In [ ]:
```
X, Y = np.meshgrid(np.linspace(-1.5, 1.5, 100), np.linspace(-1.5, 1.5, 100))
Z = rosenbrock([X, Y])
plt.contour(X, Y, Z, 40, alpha=0.2)
plt.contour(X, Y, Z, 160, alpha=0.08, linewidths=1)
im = plt.contourf(X, Y, Z, 40, cmap='turbo')
plt.scatter(*x0, marker='D', c='yellow', s=50, label='Starting point $x_0$',
plt.plot(history[::, 0], history[::, 1], 'white', lw=1)
plt.scatter(history[::, 0], history[::, 1], c='white', s=4)
plt.scatter(*history[-1], marker='*', c='yellow', s=180, label='Minimum $x^*
plt.xlabel('x')
plt.ylabel('y')
plt.legend(frameon=1, facecolor='white', framealpha=1, loc=3)
plt.title('Rosenbrock Function Convergence')
plt.colorbar(im);
plt.tight_layout()
```



```
In [ ]: nsteps = history.shape[0]
        fhist = np.zeros(nsteps)
        for i in range(nsteps):
          fhist[i] = rosenbrock(history[i,:])

        plt.figure('convergence')
        plt.semilogy(np.arange(0, nsteps), np.absolute(fhist))
        plt.xlabel('Iteration count')
        plt.ylabel(r'$|f^k - f^*|$')
        plt.show()
```



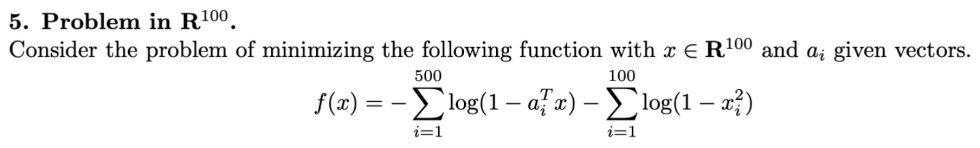
- Expected convergence of the algorithm near to the minimum

  $|f^{k+1} - f^{f*}| < C|f^k - f^*|^2$

  The newton's method converges quadratically when the function is near to the local minimum. The C is independent of the condition number.

- We can see that the algorithm converges to the minimum in 11 iterations. The fast convergence is due to the quadratic behaviour of the algorithm. In Assignment 1 we performed the same example with a steepest descent achieving the minimum in more than 400 iterations. The newton's method convergence is directly related with the use of the hessian information in finding the newton direction looking to the minimum of the quadratic approximation that the algorithm is based.

5. Problem in R^{100}


5. Problem in $\mathbf{R}^{100}$.
Consider the problem of minimizing the following function with $x \in \mathbf{R}^{100}$ and a_i given vectors.

$$f(x) = -\sum_{i=1}^{500} \log(1 - a_i^T x) - \sum_{i=1}^{100} \log(1 - x_i^2)$$

In [ ]:
```
def obj_func(x, A):
    sum1 = - jnp.log(jnp.ones(A.shape[1]) - jnp.dot(A.T, x)).sum()
```

```python
    sum2 = - jnp.log(jnp.ones(x.shape[0]) - x**2).sum()
    return sum1 + sum2
```

In [ ]:

```python
####################### Backtracking line search #######################
def backtracking_linesearch_multivar(f, gk, pk, xk, A, alpha=0.1, beta=0.8):
  t = 1
  # Feasibility condition
  # to no return step outside of of the region where the obj func is defined
  while True:
      # Check if the step is within the feasible region
      if jnp.any(A.T @ (xk + t*pk) >= 1) or jnp.any((xk + t*pk)**2 >= 1):
          t *= beta
          continue  # Skip the rest of the loop and check feasibility again

      # Check the Armijo condition (sufficient decrease condition)
      if f(xk + t*pk, A) > f(xk, A) + alpha * t * jnp.dot(gk, pk):
          t *= beta
      else:
          break  # Found acceptable t

  return t

####################### Newton Method function #######################
def newton_bt_multivar(fun, grad_fn, hess_fn, x0, A, tol=1e-5):
  x = x0
  k = 0
  history = [x0]
  f_values = [fun(x0, A)]  # Store the initial function value
  while True:
    grad = grad_fn(x, A)
    hess = hess_fn(x, A)

    if la.norm(la.solve(hess, -grad)) < tol:
        break

    p = la.solve(hess, -grad)
    t = backtracking_linesearch_multivar(fun, grad, p, x, A)
    x += t*p
    history.append(x)
    f_values.append(fun(x, A))
    k += 1

  print(f"Convergence in k:{k} steps")
  return x, np.array(history), np.array(f_values)
```

$$f(x) = -\sum_{i=1}^{500} \log(1-a_i^T x) - \sum_{i=1}^{100} \log(1-x_i^2)$$

Gradient $$\frac{df(x)}{dx} = \sum_{i=1}^{500} \frac{a_i}{1-a_i^T x} + \sum_{i=1}^{100} \frac{2x_i}{1-x_i^2}$$

Hessian

- if $J = K$

$$\frac{d^2 f(x)}{d x_J^2} = \sum_{i=1}^{500} \frac{a_{iJ}^2}{(1-a_i x)^2} + \frac{4x_J^2 + 2 - 2x_J^2}{(1-x_J^2)^2}$$

- if $J \neq K$

$$\frac{d^2 f(x)}{d x_J^2} = \sum_{i=1}^{500} \frac{a_{iJ}\, a_{iK}}{(1-a_i x)^2}$$

$$\nabla^2 f(x) = \begin{cases} \displaystyle\sum_{i=1}^{500} \frac{a_{iJ}^2}{(1-a_i x)^2} + \frac{4x_J^2 + 2 - 2x_J^2}{(1-x_J^2)^2} & \text{if } J = K \\ \displaystyle\sum_{i=1}^{500} \frac{a_{iJ}\, a_{iK}}{(1-a_i x)^2} & \text{if } J \neq K \end{cases}$$

For Solving the system of linear equations

$\rightarrow$ The search direction is the direction that minimize:

$$q(\Delta x) = f^K + \nabla f^{K^T} \Delta x + \frac{1}{2} \Delta x^T \nabla^2 f^K \Delta x$$

$\rightarrow$ when ∇q vanishes:

$$\nabla f^K + \nabla^2 f^K \Delta x = 0$$

$\rightarrow$ $\Delta x = -\nabla^2 f^K \setminus \nabla f^K$ $\}$ System of equations

$\rightarrow$ $\Delta x_K = -\nabla^2 f^{K^{-1}} \nabla f^K$

where:

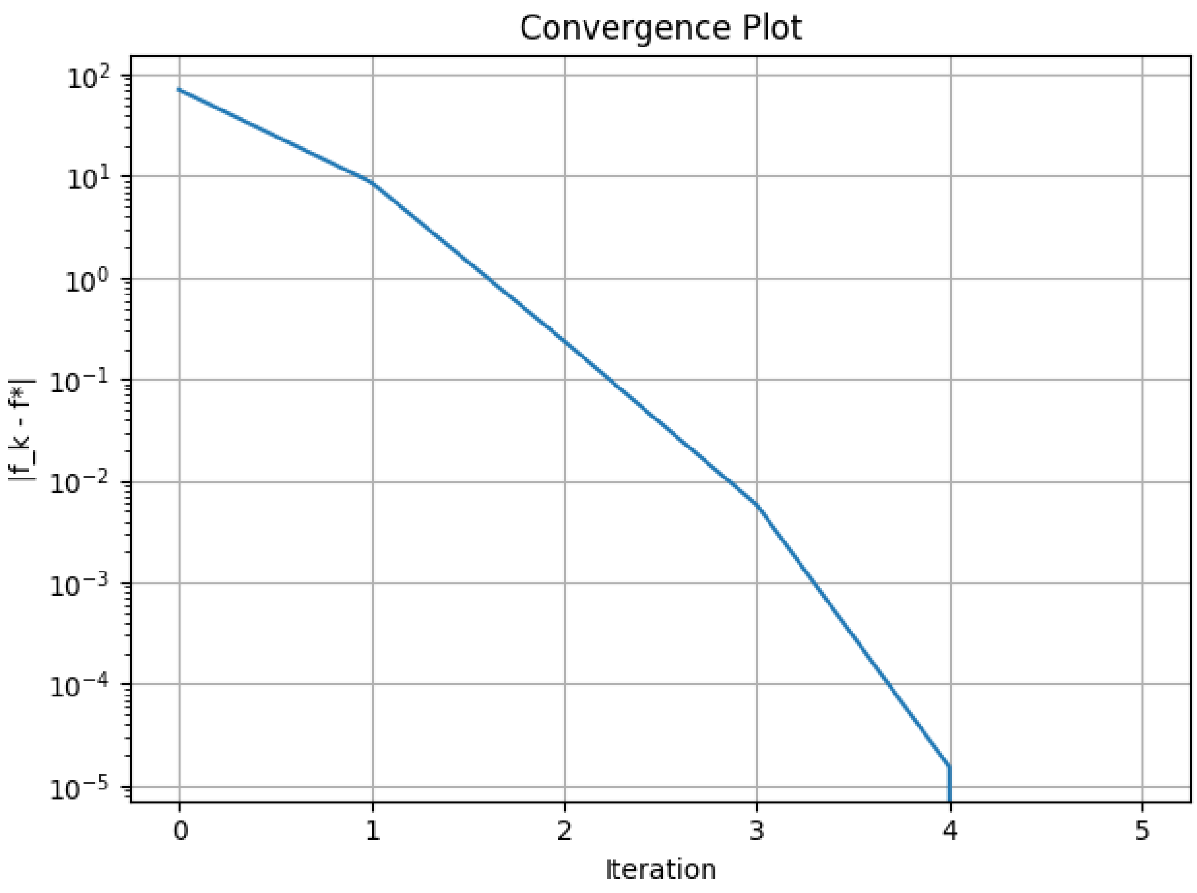
$$\nabla f_k = \sum^{500} \frac{a_{ik}}{1-a_i^T x} + \frac{2x_k}{1-x_k^2} \qquad \begin{cases} \nabla^2 f_k \underset{j=k}{} = \sum_{i=1}^{500} \frac{a_{ik}^2}{(1-a_i x)^2} + \frac{2x_k^2-2}{(1-x_k^2)^2} \\ \nabla^2 f_k \underset{j \neq k}{} = \sum_{i=1}^{500} \frac{a_{ij} a_{ik}}{(1-a_i x)^2} \end{cases}$$

In [ ]:
```
# Define A 100x500
x_0 = np.zeros(100)

seed = 211 # for reproducibility
rng = default_rng(seed)
A = rng.standard_normal((100, 500)) # A[i] are the vectors a_i

# Create the grad and hessian functions
grad_fn = jax.grad(obj_func)
hessian_fn = jax.hessian(obj_func)
```

In [ ]:
```
x, history, f_values = newton_bt_multivar(obj_func, grad_fn, hessian_fn, x_0
```

```
Convergence in k:5 steps
```

In [ ]:
```
f_star = f_values[-1]

# Now calculate the absolute differences
differences = np.abs(f_values - f_star)

iterations = np.arange(len(differences))
plt.semilogy(iterations, differences)
plt.xlabel('Iteration')
plt.ylabel('|f_k - f*|')
plt.title('Convergence Plot')
plt.tight_layout()
plt.grid(True)
plt.show()
```



6. Minimal Surface.

6. Minimal Surface. Consider the problem of generating a shape with minimal surface area. The shape is a surface of revolution about a vertical axis of length l (see figure below). Because of circular symmetry, the problem reduces to finding the radii of horizontal cross sections. The radii of the cross sections are fixed on the top and bottom at r_0. We can discretize the problem so the unknowns x represent the radii at n locations equally spaced along the vertical axis at a distance $h = l/(n+1)$. With this discretization the surface area may be approximated by the following function:

$$f(x) = 2\pi h x_1 \left[1 + \left(\frac{x_1 - r_0}{h}\right)^2\right]^{1/2} + 2\pi h \sum_{i=1}^{n-1} x_i \left[1 + \left(\frac{x_{i+1} - x_i}{h}\right)^2\right]^{1/2} + 2\pi h x_n \left[1 + \left(\frac{r_0 - x_n}{h}\right)^2\right]^{1/2}$$

Note the structure of the function: $f(x) = f_1(x_1) + f_2(x_1, x_2) + f_3(x_2, x_3) + \cdots f_n(x_{n-1}, x_n) + f_{n+1}(x_n)$.

Use Newton's method with a line search to find the optimal shape for $r_0 = 1$, $l = 0.75$, and $n = 20$.

- write a routine that computes the objective function.
- write a routine that computes the gradient.
- write a routine that computes the Hessian. Notice that the objective function consists of the sum of terms where each term contains only a pair of adjacent variables. Explain why this gives rise to a Hessian that has a tridiagonal structure.
- use your Newton routine with a backtracking line search to find the optimal shape starting from a cylindrical initial shape (i.e. $x_i = r_0$ for all i).
- plot the convergence behavior.

In [ ]:
```
######################################## Objective Function ################
def objective_function(x, r_0, h):
  pi = jnp.pi
  f_x1 = 2 * pi * h * x[0] * jnp.sqrt(1 + ((x[0] - r_0) / h) ** 2)
  f_xn = 2 * pi * h * x[-1] * jnp.sqrt(1 + ((r_0 - x[-1]) / h) ** 2)
  sum_f = 0
  for i in range(len(x) - 1):
      xi = x[i]  # xi or x_n-1
      xi1 = x[i + 1]  # xi+1 or x_n
      sum_f += 2 * pi * h * xi * jnp.sqrt(1 + ((xi1 - xi) / h) ** 2)

  return f_x1 + sum_f + f_xn

######################################## Gradient Function #################
def objective_grad(x, r_0, h):
  grad = np.zeros_like(x)
  pi = jnp.pi
  grad[0] = 2*pi*h*jnp.sqrt(1 + ((x[0] - r_0) / h) ** 2) + 2 * pi * x[0] * (
             + 2*pi*h*jnp.sqrt(1 + ((x[1] - x[0]) / h) ** 2) - 2 * pi * x[0]

  grad[-1] = 2*pi*h*jnp.sqrt(1 + ((r_0 - x[-1]) / h) ** 2) - 2 * pi * x[-1]
             + 2 * pi * x[-2] * ((x[-1] - x[-2]) / h) * (1 + ((x[-1] - x[-2]

  for i in range(1, len(x)-1):
    grad[i] = 2*pi*x[i-1]*((x[i] - x[i-1]) / h) * (1 + ((x[i] - x[i-1]) / h)
              - 2 * pi * x[i] * ((x[i+1] - x[i]) / h) * (1 + ((x[i+1] - x[i]

  return grad

######################################## Hessian Function ##################
def objective_hessian(x, r_0, h):
    hess = np.zeros([n, n])
    hess[0,0] = 2*np.pi*(((x[0]-r0)/h)/(np.sqrt(1+((x[0]-r0)/h)**2)))  \
                +(2*np.pi*(x[0]**3-3*r0*x[0]**2+(3*r0**2+2*h**2)*x[0]-r0**3-
                +(2*np.pi*(x[0]-x[1])/(h*np.sqrt(1+((x[1]-x[0])/(h**2))**2))
                +(2*np.pi*(x[0]**3-3*x[1]*x[0]**2+(3*x[1]**2+2*h**2)*x[0]-x[

    hess[0,1] = (2*np.pi*(x[1]-x[0]))/(h*np.sqrt(1+(x[1]-x[0])**2/(h**2)))
                -(2*np.pi*x[0])/(h*(1+(x[1]-x[0])**2/(h**2))**(3/2))
    hess[1,0] = hess[0,1]

    hess[n-1,n-1] = (2*np.pi*x[n-2])/(h*(1+(x[n-1]-x[n-2])**2/(h**2))**(3/2)
                    + (2*np.pi*(x[n-1]-r0))/(h*np.sqrt(1+(r0-x[n-1])**2/(h**
                    +(2*np.pi*(x[n-1]**3-3*r0*x[n-1]**2+(3*r0**2+2*h**2)*x[n

    hess[n-1,n-2] = (-2*np.pi*(x[n-2]**3-3*x[n-1]*x[n-2]**2+(3*x[n-1]**2+2*h
    hess[n-2,n-1] = hess[n-1,n-2]

    for i in range(1,n-1):
        hess[i,i] = (2*np.pi*x[i-1])/(h*(1+(x[i]-x[i-1])**2/(h**2))**(3/2))
                    + (2*np.pi*(x[i]-x[i+1]))/(h*np.sqrt(1+(x[i+1]-x[i])**2/
                     +(2*np.pi*(x[i]**3-3*x[i+1]*x[i]**2+(3*x[i+1]**2+2*h**2

        hess[i,i+1] = (2*np.pi*(x[i+1]-x[i]))/(h*np.sqrt(1+(x[i+1]-x[i])**2/
                      -(2*np.pi*x[i])/(h*(1+(x[i+1]-x[i])**2/(h**2))**(3/2))

        hess[i,i-1] = -(2*np.pi*(x[i-1]**3-3*x[i]*x[i-1]**2+(3*x[i]**2+2*h**

    return hess
```

The defined hessian contains a tritriangular diagonal due to the relation of contiguos variables in the objective function. This structure shows the influence of the interaction of inmediate neighbors in the surface area calculation.

In [ ]:
```
######################################## Problem Definitions ################
r0 = 1
l = 0.75
n = 20
h = l/(n+1)
x0 = jnp.ones(n) * r0
```

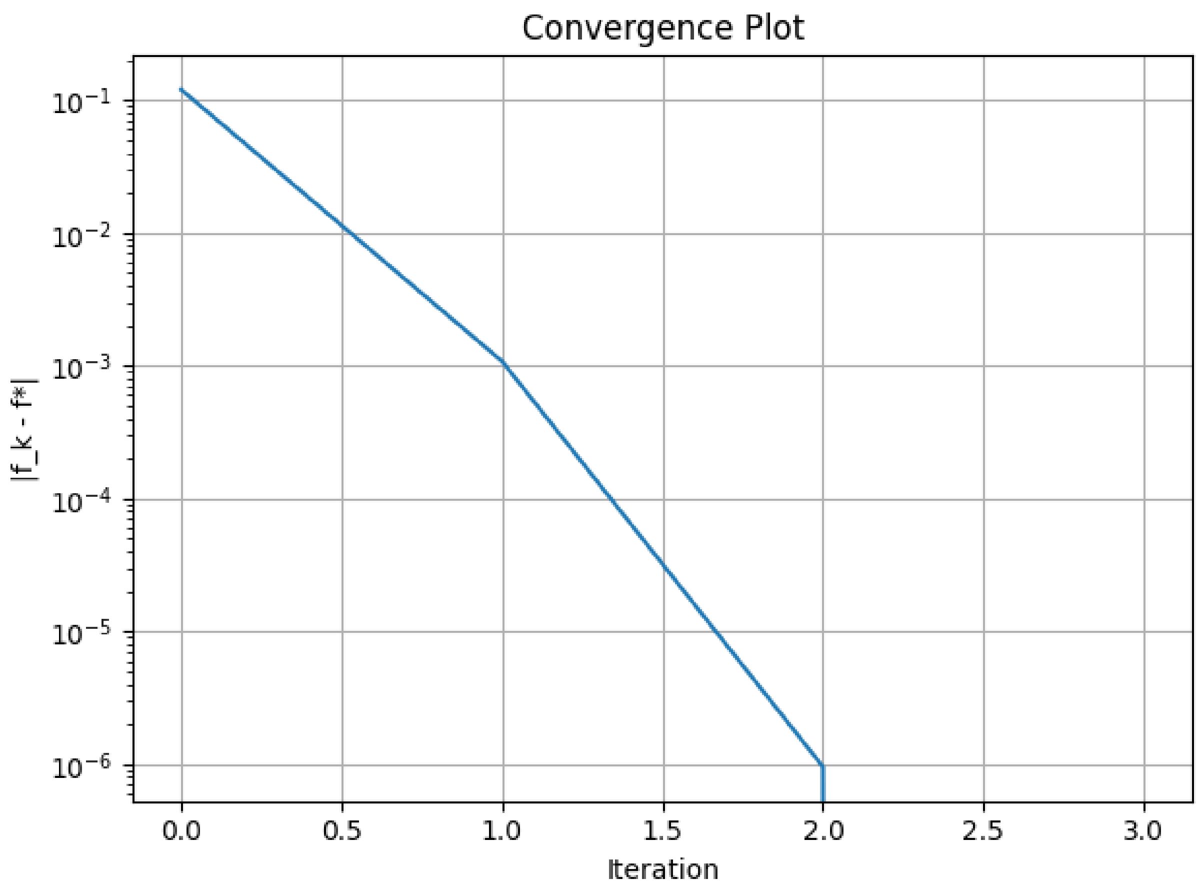
In [ ]:
```
######################################## Validate with Jax ##################
grad_co = objective_grad(x0, r0, h)
hess_co = objective_hessian(x0, r0, h)
gradf = jax.grad(objective_function)
hessf = jax.hessian(objective_function)
res = gradf(x0, r0, h)
hess_jax = hessf(x0, r0, h)
```

In [ ]:
```
####################### Backtracking line search #######################
def backtracking_linesearch_multivar(f, gk, pk, xk, r0, h, alpha=0.1, beta=0
  t = 1
  # Feasibility condition
  # to no return step outside of the region where the obj func is defined
  while True:

      if f(xk + t*pk, r0, h) > f(xk, r0, h) + alpha * t * jnp.dot(gk, pk):
          t *= beta
      else:
          break  # Found acceptable t

  return t

####################### Newton Method function #######################
def newton_bt_multivar(fun, grad_fn, hess_fn, x0, r0, h, tol=1e-5):
  x = x0
  k = 0
  history = [x0]
  f_values = [fun(x0, r0, h)]  # Store the initial function value
  while True:
    grad = grad_fn(x, r0, h)
    hess = hess_fn(x, r0, h)

    if la.norm(la.solve(hess, -grad)) < tol:
        break

    p = la.solve(hess, -grad)
    t = backtracking_linesearch_multivar(fun, grad, p, x, r0, h)
    x += t*p
    history.append(x)
    f_values.append(fun(x, r0, h))
    k += 1
  print(f"Convergence in k:{k} steps")
  return x, np.array(history), np.array(f_values)
```

In [ ]:
```
######################################## Solve Newton ######################
x, history, f_values = newton_bt_multivar(objective_function, objective_grad
```

```
Convergence in k:3 steps
```

In [ ]:
```
######################################## Convergence Plot #################
f_star = f_values[-1]

# Now calculate the absolute differences
differences = np.abs(f_values - f_star)

iterations = np.arange(len(differences))
plt.semilogy(iterations, differences)
plt.xlabel('Iteration')
plt.ylabel('|f_k - f*|')
plt.title('Convergence Plot')
plt.tight_layout()
plt.grid(True)
plt.show()
```



The optimization took 3 iterations to find the minimum of the objective function given the problem definitions.