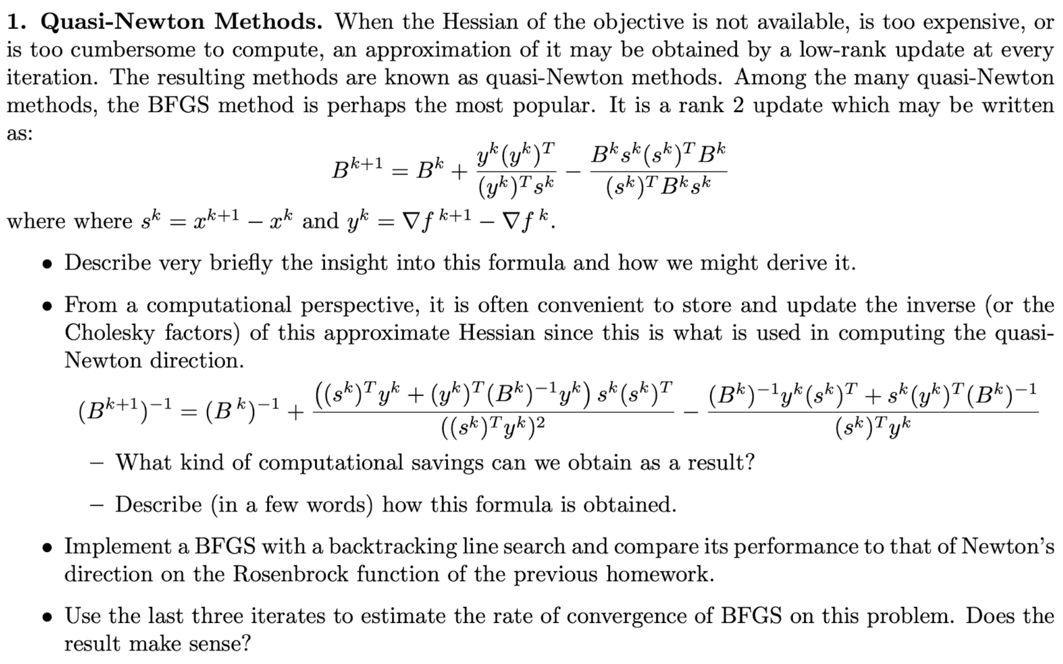
Numerical Optimization - Assignment 4

David Alvear(187594) - Nouf Farhoud(189731)

1. Quasi-Newton Methods



1. Quasi-Newton Methods. When the Hessian of the objective is not available, is too expensive, or is too cumbersome to compute, an approximation of it may be obtained by a low-rank update at every iteration. The resulting methods are known as quasi-Newton methods. Among the many quasi-Newton methods, the BFGS method is perhaps the most popular. It is a rank 2 update which may be written as:

$$B^{k+1} = B^k + \frac{y^k (y^k)^T}{(y^k)^T s^k} - \frac{B^k s^k (s^k)^T B^k}{(s^k)^T B^k s^k}$$

where where $s^k = x^{k+1} - x^k$ and $y^k = \nabla f^{\,k+1} - \nabla f^{\,k}$.

- Describe very briefly the insight into this formula and how we might derive it.
- From a computational perspective, it is often convenient to store and update the inverse (or the Cholesky factors) of this approximate Hessian since this is what is used in computing the quasi-Newton direction.

  $$(B^{k+1})^{-1} = (B^{\,k})^{-1} + \frac{\left((s^k)^T y^k + (y^k)^T (B^k)^{-1} y^k\right) s^k (s^k)^T}{((s^k)^T y^k)^2} - \frac{(B^k)^{-1} y^k (s^k)^T + s^k (y^k)^T (B^k)^{-1}}{(s^k)^T y^k}$$

  - What kind of computational savings can we obtain as a result?
  - Describe (in a few words) how this formula is obtained.
- Implement a BFGS with a backtracking line search and compare its performance to that of Newton's direction on the Rosenbrock function of the previous homework.
- Use the last three iterates to estimate the rate of convergence of BFGS on this problem. Does the result make sense?

- **a.**Describe very briefly the insight into this formula and how we might derive it.

  Quasi-Newton methods aim to approximate the Hessian matrix (or its inverse) without explicitly computing it, making them more efficient than methods that require calculating the exact Hessian. The goal is to update the inverse Hessian approximation B_k to obtain B_{k+1} using the information from s^k and y^k.

  How it could be derived:

Updating our approximate Hessian at each iteration by adding two symmetric ranl
$B_{k+1} = B_k + U_k + V_k$
$B_{k+1} = B_k + auu^T + bvv^T$

Constraints: Linearly independent non-zero vectors

Quasi-Newton condition:
$B_{k+1}\Delta x_k = y_k$
$B_{k+1}\Delta x_k = B_k\Delta x_k + auu^T\Delta x_k + bvv^T\Delta x_k = y_k$

$u = y_k \quad v = B_k$

$B_{k+1}\Delta x_k + ay_ky_k^T\Delta x_k + bB_k\Delta x_k\Delta x_k^T B_k^T\Delta x_k = y_k$
$y_k(1 - ay_k^T\Delta x_k) = B_k\Delta x_k(1 + b\Delta x_k^T B_k^T\Delta x_k)$
$\rightarrow a = \dfrac{1}{y_k^T\Delta x_k}$
$\rightarrow b = -\dfrac{1}{\Delta x_k^T B_k\Delta x_k}$

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T\Delta x_k} - \frac{B_k\Delta x_k\Delta x_k^T B_k}{\Delta x_k^T B_k\Delta x_k}$$

Quasi Newton Method:

b. What kind of computational savings can we obtain?

- Reduced memory requirements
- Avoidance of Hessian calculations

In [ ]:
```python
import numpy as np

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

def rosenbrock_gradient(x):
    grad = np.zeros(2)
    grad[0] = -40 * x[0] * (x[1] - x[0]**2) - 2 * (1 - x[0])
    grad[1] = 20 * (x[1] - x[0]**2)
    return grad

def bfgs_rosenbrock(x0, max_iter=1000, tolerance=1e-6):
    n = len(x0)
    B = np.eye(n)  # Initial approximation of the Hessian matrix
    x = x0.copy()
    g = rosenbrock_gradient(x)

    objective_values = [rosenbrock(x)]
    for _ in range(max_iter):
        p = -np.linalg.solve(B, g)  # Compute search direction using the inv
        alpha = 1.0
        c = 0.9  # Backtracking line search parameters
        rho = 0.5

        while rosenbrock(x + alpha * p) > rosenbrock(x) + c * alpha * np.dot
            alpha *= rho

        x_new = x + alpha * p
        g_new = rosenbrock_gradient(x_new)
        s = x_new - x
        y = g_new - g

        if np.linalg.norm(y) < tolerance:
            break

        B = B + np.outer(y, y) / np.dot(y, s) - np.dot(B, np.outer(s, s)).dc
        x = x_new
        g = g_new

        objective_values.append(rosenbrock(x))

    return x, objective_values

# Test the BFGS method on the Rosenbrock function
x0 = np.array([-1.2, 1.0])
result_bfgs, objective_values = bfgs_rosenbrock(x0)

print("BFGS solution:", result_bfgs)
print("BFGS objective value:", rosenbrock(result_bfgs))

convergence_rate = np.diff(objective_values[-3:-1]) / np.diff(objective_valu
print("Convergence rate:", convergence_rate)
```

```
BFGS solution: [0.9999951  0.99998987]
BFGS objective value: 2.513437300362502e-11
Convergence rate: [1.30611695]
```

What are objective values?

The objective function measures the performance or quality of a solution or point in the optimization problem. By examining the sequence of objective values over iterations, you can get insights into the convergence behavior of the BFGS method. If the objective values decrease rapidly, it indicates that the algorithm is converging towards a solution. On the other hand, if the objective values remain relatively high or fluctuate, it may indicate convergence issues or the presence of local minima.

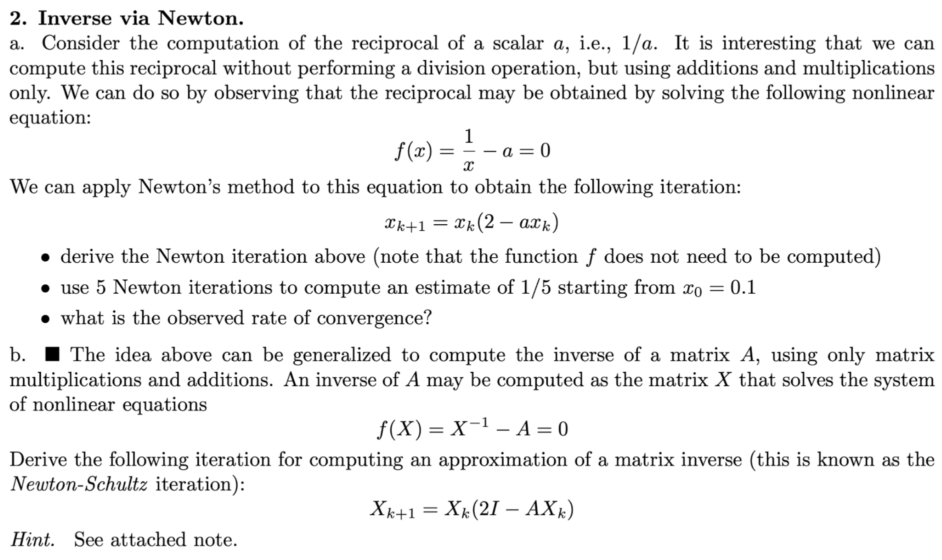
In [ ]:
```
from scipy.optimize import minimize

# Test Newton's method on the Rosenbrock function
result_newton = minimize(rosenbrock, x0, method='Newton-CG', jac=rosenbrock_
print("Newton's solution:", result_newton.x)
print("Newton's objective value:", result_newton.fun)
```

```
Newton's solution: [0.99999533 0.99999064]
Newton's objective value: 2.1849015139448183e-11
```

Comparison of BFGS and Newton method:

Convergence behavior: Both methods have achieved very small objective values, indicating that they have found solutions that are very close to the true minimum of the Rosenbrock function. The BFGS method is a quasi-Newton method that updates an approximation of the Hessian matrix, while the Newton method uses the exact Hessian. In some cases, the Newton method can converge faster near the optimum due to its use of more accurate curvature information.

2. Inverse via Newton



2. Inverse via Newton.
a. Consider the computation of the reciprocal of a scalar a, i.e., $1/a$. It is interesting that we can compute this reciprocal without performing a division operation, but using additions and multiplications only. We can do so by observing that the reciprocal may be obtained by solving the following nonlinear equation:

$$f(x) = \frac{1}{x} - a = 0$$

We can apply Newton's method to this equation to obtain the following iteration:

$$x_{k+1} = x_k(2 - ax_k)$$

- derive the Newton iteration above (note that the function f does not need to be computed)
- use 5 Newton iterations to compute an estimate of $1/5$ starting from $x_0 = 0.1$
- what is the observed rate of convergence?

b. ■ The idea above can be generalized to compute the inverse of a matrix A, using only matrix multiplications and additions. An inverse of A may be computed as the matrix X that solves the system of nonlinear equations

$$f(X) = X^{-1} - A = 0$$

Derive the following iteration for computing an approximation of a matrix inverse (this is known as the *Newton-Schultz* iteration):

$$X_{k+1} = X_k(2I - AX_k)$$

Hint. See attached note.

- **a.** Derive the Newton iteration above (note that the function f does not need to be computed)

DERIVATION OF THE NEWTON METHOD

Given $f(x) = \frac{1}{x} - a$

Newton's method: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

$f(x) = \frac{1}{x} - a \Rightarrow f'(x) = -\frac{1}{x^2}$

So $x_{n+1} = x_n - \left(\frac{\frac{1}{x_n} - a}{-\frac{1}{x_n^2}}\right)$

$$x_{n+1} = x_n + ((1 - ax_n) \cdot x_n)$$

$$x_{n+1} = x_n + (x_n - ax_n^2)$$

$$x_{n+1} = 2x_n - ax_n^2$$

$$x_{n+1} = x_n(2 - ax_n)$$

Compute the estimate of 1/5 starting from $x_0 = 0.1$

In [ ]:
```
def reciprocal_newton(a, x0, num_iterations):
    x = x0 #setting initial guess for root
    for _ in range(num_iterations):
        x = x * (2 - a * x) #applying newton's method iteration formula
    return x

a = 5
x0 = 0.1
num_iterations = 5
estimate = reciprocal_newton(a, x0, num_iterations)
print("Estimate of 1/5:", estimate) #printing the estimated reciprocal of a
```

```
Estimate of 1/5: 0.1999999999534339
```

In [ ]:
```
errors = [abs(estimate - (1/5))] #list to store the errors
previous_estimate = estimate  # Initialize previous_estimate
for i in range(1, num_iterations):
    error_ratio = abs(estimate - (1/5)) / abs(previous_estimate - (1/5)) #co
    errors.append(error_ratio)
    previous_estimate = estimate #update estimate with current estimate

print("Observed rate of convergence:", errors)
```

```
Observed rate of convergence: [4.6566112077428556e-11, 1.0, 1.0, 1.0, 1.0]
```

- **b.** Derive the following iteration for computing an approximation of a matrix inverse (Newton-Shultz iteration).

  $$X_{k+1} = X_k(2I - AX_k).$$

Solve:
$f(X) = X^{-1} - A = 0$

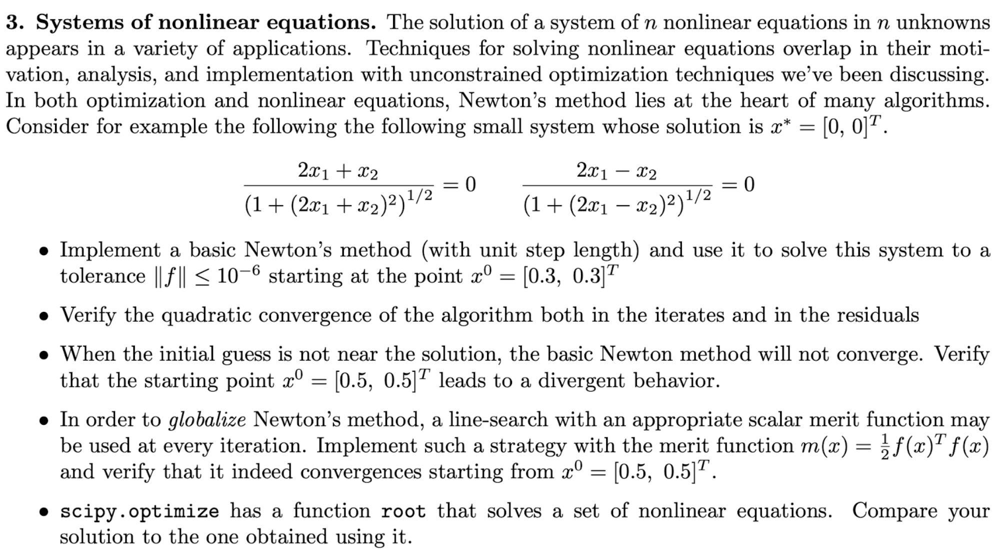
Derive the Newton-Schulz iteration:
$X_{k+1} = X_k(2I - AX_k)$

We have an $n \times n$ matrix. Using the Newton update we can find the roots of $f(X)$
$X_{k+1} = X_k - Df(X^k)^{-1} f(X^k)$
But: $Df(X^k) \rightarrow J(X^k)$.
We can treat f as a vector of n^2 with the Jacobian at X^k as an $n^2 \times n^2$ matrix. The
has the same shape of X to be updated in the iteration.
Using the identity: $Df(X) \backslash B = -X \backslash X$ we can find
$J(X^k) \backslash f(X^k)$

$Df(X^k)^{-1} \backslash f(X^k) = -X_k[X_k^{-1} - A]X_k$
$Df(X^k)^{-1} \backslash f(X^k) = [-X_k X_k^{-1} + X_k A]X_k = -X_k I + X_k A X_k, \text{ With: } X_k^{-1} X_k = I$
$Df(X^k)^{-1} \backslash f(X^k) = -X_k I + X_k A X_k$

Finally, plug into the Newton update iteration:
$X_{k+1} = X_k + X_k I - X_k A X_k$
$X_{k+1} = X_k[2I - AX_k]$

3. System of nonlinear equations



3. Systems of nonlinear equations. The solution of a system of n nonlinear equations in n unknowns appears in a variety of applications. Techniques for solving nonlinear equations overlap in their motivation, analysis, and implementation with unconstrained optimization techniques we've been discussing. In both optimization and nonlinear equations, Newton's method lies at the heart of many algorithms. Consider for example the following the following small system whose solution is $x^* = [0,\ 0]^T$.

$$\frac{2x_1 + x_2}{(1 + (2x_1 + x_2)^2)^{1/2}} = 0 \qquad \frac{2x_1 - x_2}{(1 + (2x_1 - x_2)^2)^{1/2}} = 0$$

- Implement a basic Newton's method (with unit step length) and use it to solve this system to a tolerance $\|f\| \leq 10^{-6}$ starting at the point $x^0 = [0.3,\ 0.3]^T$
- Verify the quadratic convergence of the algorithm both in the iterates and in the residuals
- When the initial guess is not near the solution, the basic Newton method will not converge. Verify that the starting point $x^0 = [0.5,\ 0.5]^T$ leads to a divergent behavior.
- In order to *globalize* Newton's method, a line-search with an appropriate scalar merit function may be used at every iteration. Implement such a strategy with the merit function $m(x) = \frac{1}{2} f(x)^T f(x)$ and verify that it indeed convergences starting from $x^0 = [0.5,\ 0.5]^T$.
- `scipy.optimize` has a function `root` that solves a set of nonlinear equations. Compare your solution to the one obtained using it.

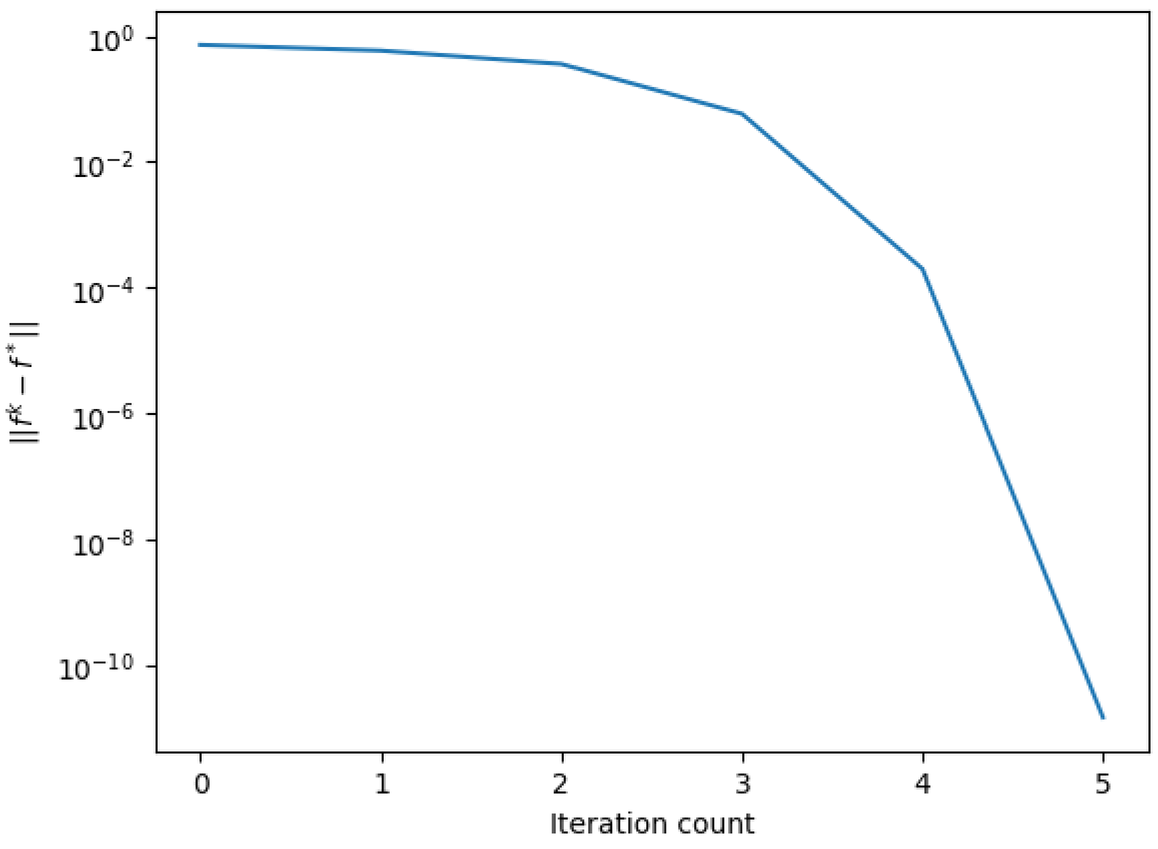
In [ ]:
```python
import numpy as np
import numpy.linalg as la
import jax
from scipy.optimize import root
import jax.numpy as jnp
import matplotlib.pyplot as plt
```

Solve nonlinear system using basic Newton's method

In [ ]:
```python
# Define the function
def nonlinear_function(x):
  x1, x2 = x
  f1 = (2*x1 + x2)/((1 + (2*x1 + x2)**2)**0.5)
  f2 = (2*x1 - x2)/((1 + (2*x1 - x2)**2)**0.5)
  return jnp.array([f1, f2])

####################### Newton Method function #######################
def newton_simple(fun, J, x0, tol=1e-5):
  x = x0
  history = np.array([x0])
  iterations = 0
  while (jnp.linalg.norm(fun(x))>tol):
    p = np.linalg.solve(J(x), -fun(x))
    # t = backtracking_linesearch(fun, grad(x), p, x)
    x += p
    history = np.vstack((history,x))
    iterations += 1

  print(f"Solved in {iterations} iterations")
  return x, history

# Get the Jacobian using Jax
jacobian_f = jax.jacfwd(nonlinear_function)
# Define initial condition
x0 = jnp.array([0.3, 0.3])

# Use newton method to solve
x, history = newton_simple(nonlinear_function, jacobian_f, x0, tol=10e-6)
```

```
Solved in 5 iterations
```

In [ ]:
```python
nsteps = history.shape[0]
fhist = np.zeros(nsteps)
for i in range(nsteps):
  fhist[i] = np.linalg.norm(nonlinear_function(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()
```

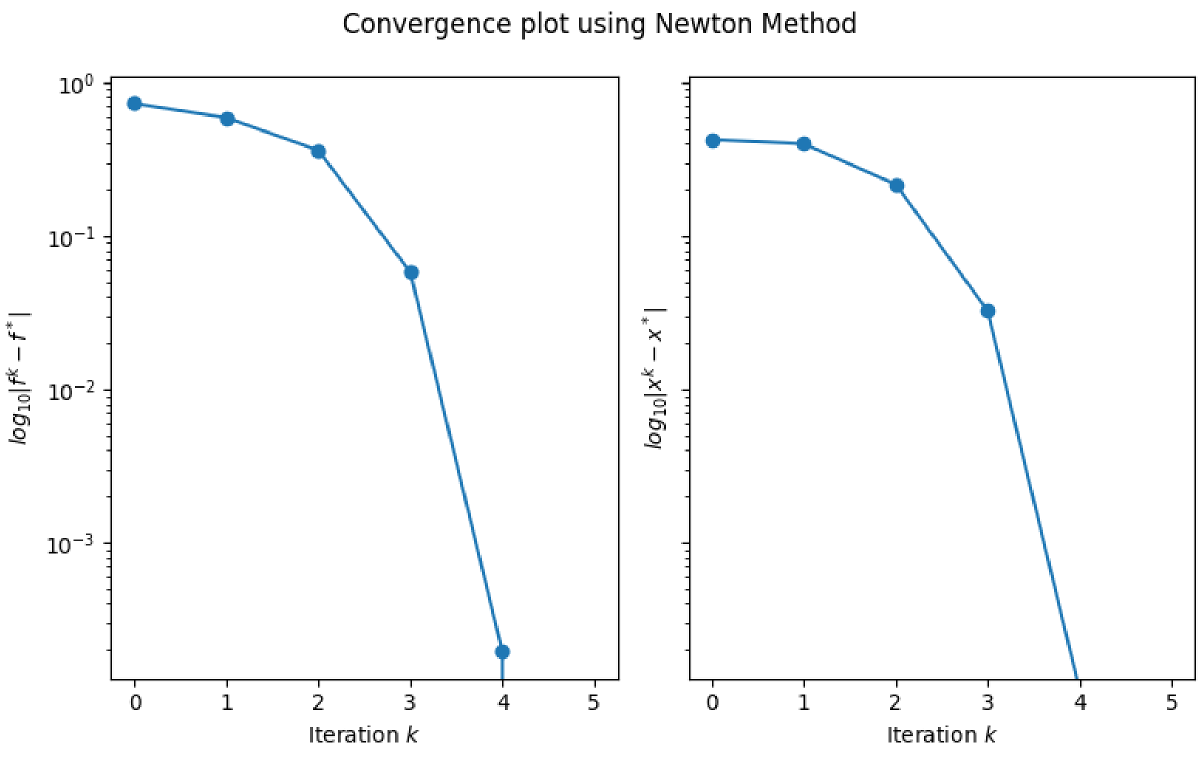


Verify Quadratic Convergence in the iterates and in the residuals

In [ ]:
```
def calculate_convergence_rate(fun, xk1, xk, x_k1, x_k2):
  # |f_(k+1) - f_k|
  fk1 = np.linalg.norm(fun(xk1) - fun(xk))
  norm_xk1 = np.linalg.norm(xk1 -xk)
  # |f_k - f_k(k-1)|
  fk = np.linalg.norm(fun(xk) - fun(x_k1))
  norm_xk = np.linalg.norm(xk- x_k1)
  # |f_(k-1) - f_(k-2)|
  f_k1 = np.linalg.norm(fun(x_k1) - fun(x_k2))
  norm_x_k1 = np.linalg.norm(x_k1 - x_k2)
  # get reciprocal rate
  # rec_rate = (np.log(fk1) - np.log(fk)) / (np.log(fk) - np.log(f_k1))
  rec_rate = (np.log(fk) - np.log(fk1)) / (np.log(f_k1) - np.log(fk))
  # get the iters rate
  iter_rate = (np.log(norm_xk1) - np.log(norm_xk)) / (np.log(norm_xk) - np.l
  return rec_rate, iter_rate

rec_rate, iter_rate = calculate_convergence_rate(nonlinear_function, history
print(f"Iterates convergence alpha: {iter_rate}")
print(f"Residuals convergence alpha: {rec_rate}")
```

```
Iterates convergence alpha: 2.8003129959106445
Residuals convergence alpha: 2.8832192420959473
```

In [ ]:
```
conver = list(map(lambda y: la.norm(nonlinear_function(y)-nonlinear_functior
conver2 = list(map(lambda y: la.norm(y-x), history))

fig, axs = plt.subplots(1, 2, sharey=True, figsize=(8, 5))

axs[0].plot(conver, 'o-')
axs[1].plot(conver2, 'o-')

axs[0].set_xlabel('Iteration $k$')
axs[1].set_xlabel('Iteration $k$')

axs[1].set_ylabel('$log_{10} |x^k-x^*|$')
axs[0].set_ylabel('$log_{10} |f^k-f^*|$')

plt.yscale('log')
fig.suptitle('Convergence plot using Newton Method')
plt.tight_layout()
```



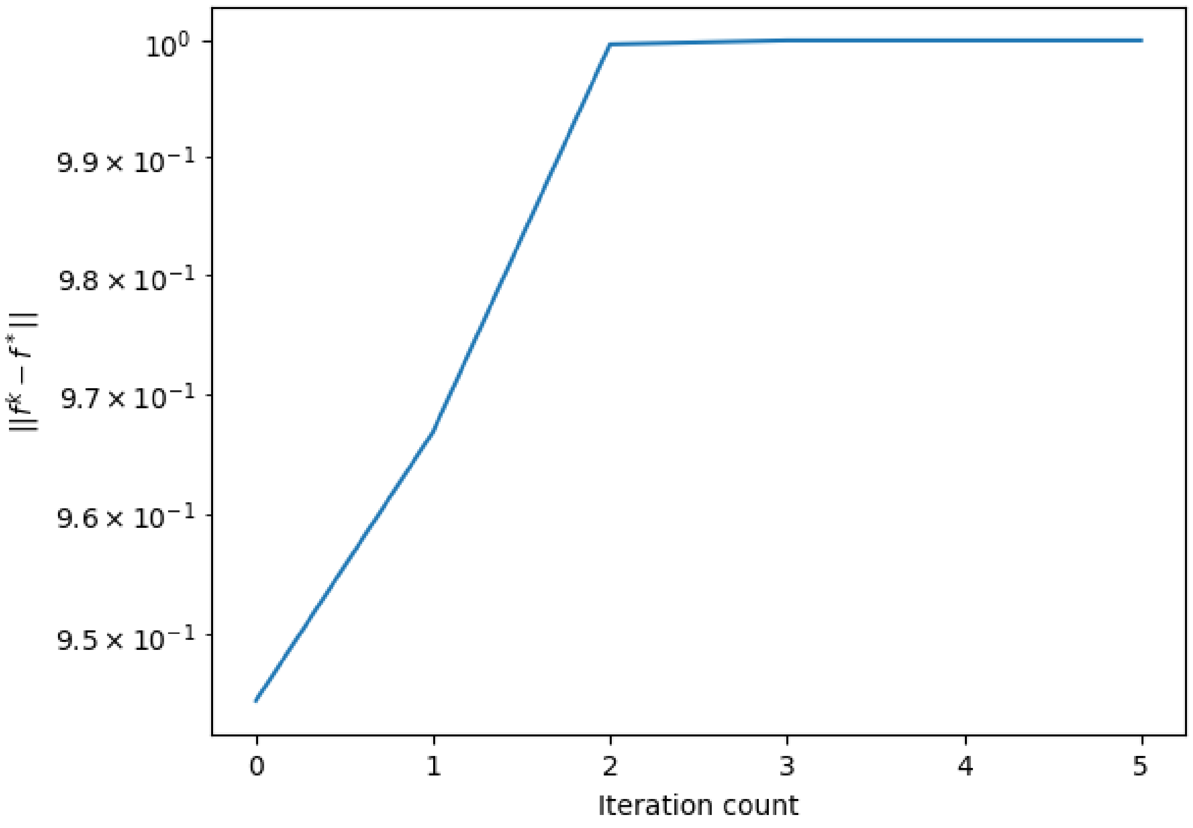
Verify that for the starting point $x_0 = [0.5, 0.5]^T$ leads to a divergent behaviour.

In [ ]:
```
x0 = jnp.asarray([0.5, 0.5])
x, history = newton_simple(nonlinear_function, jacobian_f, x0, tol=10e-6)
print(f"Solution x: {x}")
```

```
Solved in 5 iterations
Solution x: [-2.2707962e+20 -4.5415924e+20]
```

In [ ]:
```
nsteps = history.shape[0]
fhist = np.zeros(nsteps)
```

```
for i in range(nsteps):
  fhist[i] = np.linalg.norm(nonlinear_function(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()
```



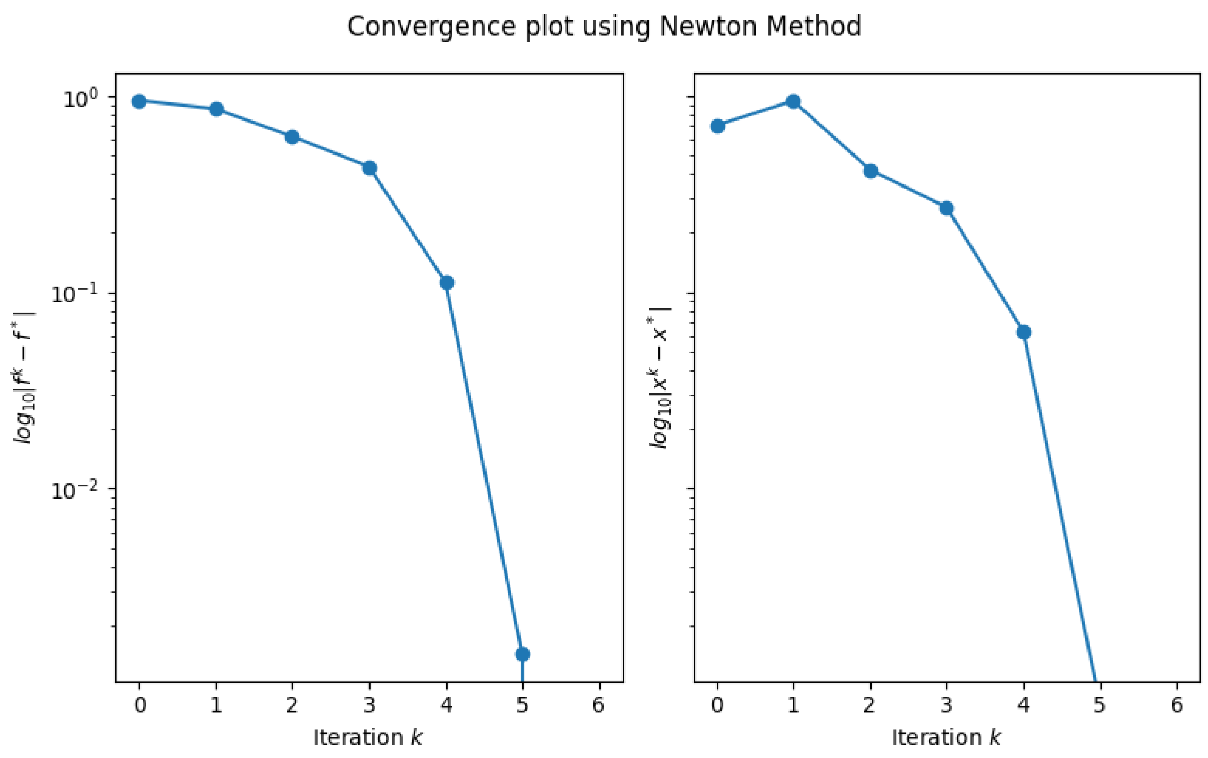
We can see that the error exploded having a divergence behaviour when the initial position is far from the optimal solution.

In [ ]:
```
# Newton method generalized
####################### Backtracking line search #######################
def backtracking_linesearch(f, pk, xk, gradf, alpha=0.1, beta=0.8):
  # gradient
  gk = gradf(xk)
  t = 1
  while (f(xk + t*pk) > f(xk) + alpha * t * gk @ pk):
    t *= beta
  return t

def merit_function(x):
  return 0.5 * nonlinear_function(x).T @ nonlinear_function(x)

####################### Newton Method function #######################
def newton_globalized(fun, J, m_func, m_gradf, x0, tol=1e-5):
  x = x0
  history = np.array([x0])
  iterations = 0
  while (jnp.linalg.norm(fun(x))>tol):
    p = np.linalg.solve(J(x), -fun(x))
    t = backtracking_linesearch(m_func, p, x, m_gradf)
    x += t*p
    history = np.vstack((history,x))
    iterations += 1

  print(f"Solved in {iterations} iterations")
  return x, history
```

In [ ]:
```
x0 = jnp.asarray([0.5, 0.5])
grad_merit = jax.grad(merit_function)
x_g, history_g = newton_globalized(nonlinear_function, jacobian_f, merit_fur
print(f"Solution x: {x_g}")
```

```
Solved in 6 iterations
Solution x: [7.5669959e-10 1.5133992e-09]
```

In [ ]:
```
conver = list(map(lambda y: la.norm(nonlinear_function(y)-nonlinear_functior
conver2 = list(map(lambda y: la.norm(y-x_g), history_g))

fig, axs = plt.subplots(1, 2, sharey=True, figsize=(8, 5))

axs[0].plot(conver, 'o-')
axs[1].plot(conver2, 'o-')

axs[0].set_xlabel('Iteration $k$')
axs[1].set_xlabel('Iteration $k$')

axs[1].set_ylabel('$log_{10} |x^k-x^*|$')
axs[0].set_ylabel('$log_{10} |f^k-f^*|$')

plt.yscale('log')
fig.suptitle('Convergence plot using Newton Method')
plt.tight_layout()
```



We can check that using backtracking line search with a scalar merit function we can find the optimal solution from a far initial guess.

Compare the solution of the globalized newton method for nonlinear equations with the root function of scipy.optimize.

In [ ]:
```
def nonlinear_function(x):
  x1, x2 = x
  f1 = (2*x1 + x2)/((1 + (2*x1 + x2)**2)**0.5)
  f2 = (2*x1 - x2)/((1 + (2*x1 - x2)**2)**0.5)
  return jnp.array([f1, f2])

# Initial guess
x0 = np.array([0.5, 0.5])

solution = root(nonlinear_function, x0)

# Check if the solver found a solution
if solution.success:
    print('Solution x:', solution.x)
```

```
Solution x: [ 2.35572888e-46 -1.32594889e-46]
```

Using the Scipy solver we achieve a a solution with a really low error, about four times the order of magnitud of the solution that we found with the globalized newton method.

Solution with roots: [2.35572888e-46 -1.32594889e-46]

```
Solution with Newton method: [7.5669959e-10 1.5133992e-09]
```

Apart from this difference we can still use the solution computed with the globalized newton's method with high accuracy.