MAP202 Static and Dynamic Optimisation - Coursework

F233014

Abstract

This report details an analysis of several unconstrained optimisation algorithms, applied to two specific test functions. The algorithms covered are steepest descent, exact line search, Newton's line search method, and quasi-Newton Davidon-Fletcher-Powell (DFP) formula.

1 Introduction

Optimisation algorithms are utilised across a variety of fields to improve processes and systems by finding the extrema of many variable problems. Such algorithms use mathematical and computational techniques to search through a vast solution space; arriving at the optimal solution; often subject to constraining criteria. Optimisation is used throughout a broad range of domains, including engineering, finance, logistics, and data analysis amongst others. This report provides an overview of a selection of optimisation algorithms and their application to specific unconstrained test functions, as well as a discussion of their strengths and limitations. The algorithms covered can be split in to two categories: those used to calculate the direction taken for an iteration, and those used to calculate the magnitude taken in said direction. Steepest descent and the Davidon-Fletcher-Powell (DFP) formula were used for the former; exact line search and Newton's line searching method were used for the latter.

1.1 Steepest descent

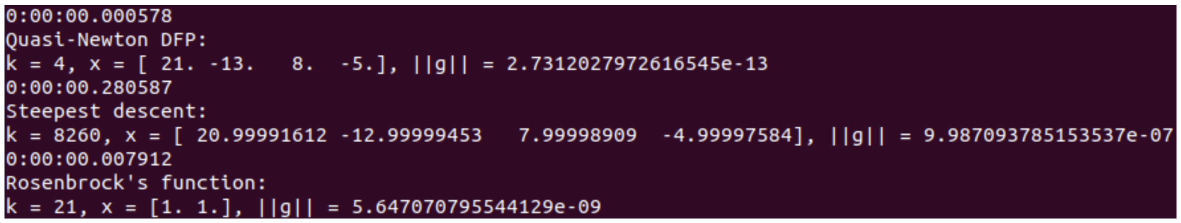
Steepest descent, also known as gradient descent, is an optimisation algorithm used to find the minimum of a function. In iteration $k+1$, the approximate solution, $x^{(k)}$, is updated in the direction of the negative gradient of the objective function, $f(x^{(k)})$. The negative gradient, $-\nabla f(x^{(k)})$, points in the direction of the steepest decrease of the objective function, hence the name steepest descent. The algorithm starts with an initial guess of the solution and then updates according to the following formula:

$$x^{(k+1)} = x^{(k)} - \alpha^{(k)} \nabla f(x^{(k)}) \,, \tag{1}$$

where $x^{(k+1)}$ is the updated solution, $x^{(k)}$ is the current solution, $\alpha^{(k)}$ is the step size, and $\nabla f(x^{(k)})$ is the gradient of the objective function evaluated at $x^{(k)}$. The step size $\alpha^{(k)}$ determines the size of the update at each iteration. A large step size can lead to overshooting the minimum, while a small step size can slow down the convergence. The magnitude of the step size is inherently a compromise between rate of convergence and stability.

1.2 Quasi-Newton methods

Quasi-Newton DFP [1] is an optimisation algorithm designed to solve problems where the objective function is smooth and continuous. It works by approximating the inverse Hessian matrix of the objective function, $C^{(k)} \approx H^{-1}(x^{(k)})$, which gives an estimate



```
0:00:00.000578
Quasi-Newton DFP:
k = 4, x = [ 21. -13.   8.  -5.], ||g|| = 2.7312027972616545e-13
0:00:00.280587
Steepest descent:
k = 8260, x = [ 20.99991612 -12.99999453   7.99998909  -4.99997584], ||g|| = 9.987093785153537e-07
0:00:00.007912
Rosenbrock's function:
k = 21, x = [1. 1.], ||g|| = 5.647070795544129e-09
```

Figure 1: Terminal output from running the python code given in appendix A.

of the objective function's curvature. At each iteration, the DFP formula updates the approximation of the inverse Hessian matrix by using the difference between the current and previous solutions, $\Delta x^{(k)} = x^{(k+1)} - x^{(k)}$, and the difference in gradient at these points, $\Delta g^{(k)} = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)})$.

$$C^{(k+1)} = C^{(k)} + \frac{\Delta x^{(k)}(\Delta x^{(k)})^T}{(\Delta x^{(k)})^T \Delta g^{(k)}} - \frac{C^{(k)}\Delta g^{(k)}(\Delta g^{(k)})^T C^{(k)}}{(\Delta g^{(k)})^T C^{(k)} \Delta g^{(k)}}\,. \tag{2}$$

Starting with an initial guess $x^{(0)}$ and $C^{(0)} = I_D$, where D is the dimensionality of the solution space, iterative solutions are calculated as:

$$x^{(k+1)} = x^{(k)} - \alpha^{(k)} C^{(k)} \nabla f(x^{(k)})\,. \tag{3}$$

1.3 Exact line search

Exact line search is used to find the optimal step size along a given search direction, $d^{(k)}$. The premise behind exact line search is to solve a one-dimensional optimisation problem along a particular search direction: $\tilde{f}(\alpha) = f(x^{(k)} + \alpha d^{(k)})$. For a quadratic function q, $\alpha^{(k)}$ and be calculated in a single step as:

$$\alpha^{(k)} = -\frac{d^{(k)T}\nabla q(x^{(k)}}{d^{(k)T} A d^{(k)}} \tag{4}$$

where A is the Hessian of the quadratic function.

1.4 Newton's line search method

Newton's line search method is used to find the root of a double differentiable function. Newton's method works by starting with an initial guess of the root or critical point and then improving the guess using the first and second derivatives of the function. The formula for Newton's method is expressed as follows:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}\,, \tag{5}$$

where x_{n+1} is the next guess, x_n is the current guess, $f(x)$ is the function to be optimized, and $f'(x)$ is the derivative of the function.

2 Results

2.1 Quasi-Newton DFP and steepest descent

To compare the previously mentioned optimisation algorithms, a quadratic test function $f(x) = \frac{1}{2}x^T Ax + b^T x$, with

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}, \ A = \begin{pmatrix} 1 & 0 & -1 & 3 \\ 0 & 2 & 1 & 0 \\ -1 & 1 & 6 & -1 \\ 3 & 0 & -1 & 10 \end{pmatrix}, \ b = \begin{pmatrix} 2 \\ 18 \\ -19 \\ -5 \end{pmatrix},$$

starting from and initial condition $x^{(0)} = (0, 0, 0, 0)^T$ was utilised. With stopping condition $||\nabla f|| < 10^{-6}$, the DFP formula with exact line search found the minimum in 0.000578 seconds with 4 iterations; steepest decent with exact line search found the minimum in 0.280587 seconds with 8260 iterations. The utility of the DFP formula is clearly demonstrated, finding the minimum to an accuracy 6 orders of magnitude greater than steepest descent, 3 orders of magnitude quicker (see figure 1).

2.2 Rosenbrock's function

Rosenbrock's function is commonly used as a test problem for optimisation algorithms [2],

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \, . \tag{6}$$

The DFP forumla in conjunction with Newton's line searching method was chosen to find the minima of equation 6, subject to the initial condition $x^{(0)} = (-1, 2)^T$ and stopping condition $||\nabla f|| < 10^{-4}$. The minimum was found in 0.007912 seconds, taking 21 iterations. The initial guess of each search magnitude was taken such that $||\alpha^{(k)} d^{(k)}|| = 1$, shown on lines 70 and 80. The stopping criteria for Newton's line searching method was taken as $||\nabla f|| < 10^{-8}$; again shown on lines 70 and 80. Relaxing this stopping criteria to the same as that of the DFP formula resulted in no solution found in a reasonable time frame. Tightening this stopping criteria to $||\nabla f|| < 10^{-12}$ equally hindered finding a solution to the DFP formula, as the line search method was excessively computationally intensive. The solution trajectory taken by the code in appendix A is shown in figure 2.

A Python Code

```
 1 from pylab import *
 2 from typing import Callable
 3 from datetime import datetime
 4 
 5 
 6 '''
 7 Quadratic test function: f(x) = 1/2 * x^T * A * x + b^T x
 8 Rosenbrock's test function: f(x) = 100 * (y - x**2)**2 + (1 - x)**2
 9 '''
10 
11 
12 def main():
```

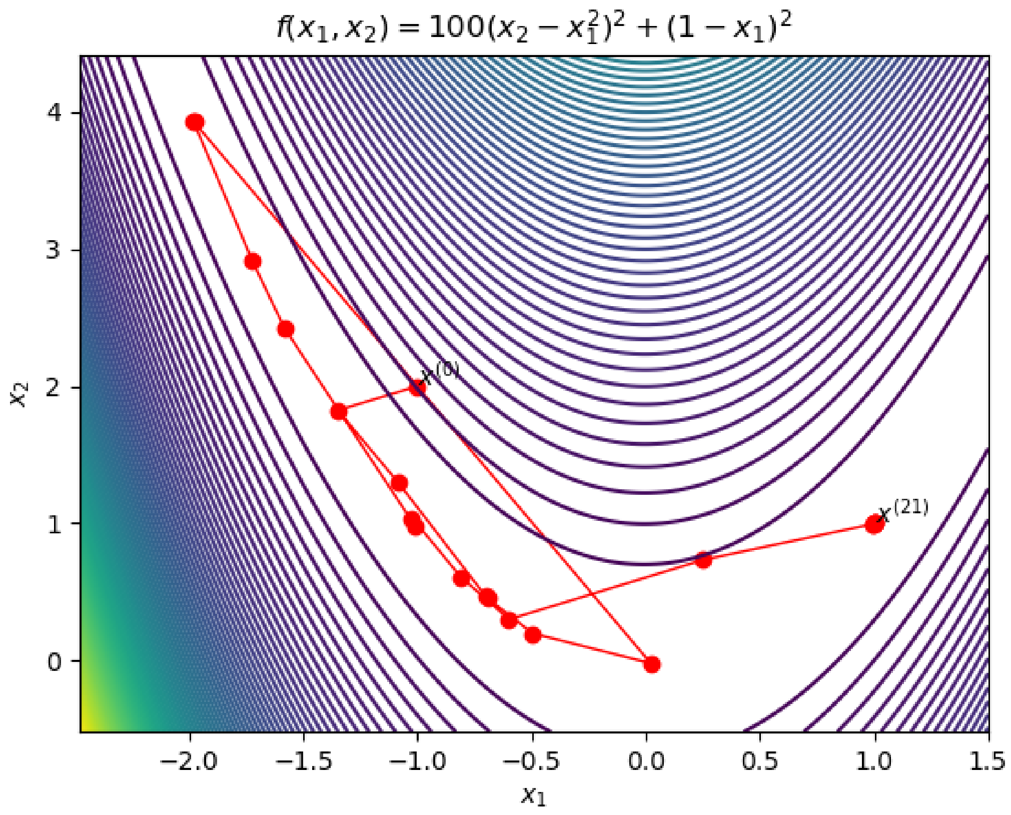


Figure 2: Rosenbrock's function and the path taken by the algorithm detailed in appendix A.

```python
    start = datetime.now()
    # 1.a) quasi-Newton DFP, exact line search
    error = 10**-6
    A = np.array([[1, 0, -1, 3], [0, 2, 1, 0], [-1, 1, 6, -1], [3, 0,
   -1, 10]])
    b = np.array([[2], [18], [-19], [-5]])
    x0 = np.array([[0], [0], [0], [0]])
    f = lambda x: (
        0.5 * np.linalg.multi_dot([np.transpose(x), A, x])
        + np.dot(np.transpose(b), x)
    )
    g = lambda x: np.dot(A, x) + b
    C0 = np.identity(np.size(x0))
    g0 = g(x0)
    d0 = -np.dot(C0, g0)
    alpha0 = (
        -np.dot(np.transpose(d0), g0)
        / np.linalg.multi_dot([np.transpose(d0), A, d0])
    )
    x = x0 + alpha0 * d0
    k = 1

    while np.linalg.norm(g(x)) >= error:
        gx = g(x)
        delta_x = x - x0
        delta_g = gx - g(x0)
        C = (
            C0 + np.dot(delta_x, np.transpose(delta_x))
            / np.dot(np.transpose(delta_x), delta_g)
            - np.linalg.multi_dot([C0, delta_g, np.transpose(delta_g), C0])
            / np.linalg.multi_dot([np.transpose(delta_g), C0, delta_g])
        )
        C0 = np.copy(C)
        d = -np.dot(C, gx)
        alpha = (
            -np.dot(np.transpose(d), gx)
            / np.linalg.multi_dot([np.transpose(d), A, d])
        )
        x0 = np.copy(x)
        x += alpha * d
        k += 1

    print(datetime.now() - start)
    print("Quasi-Newton DFP:")
    print(f"k = {k}, x = {x.flatten()}, ||g|| = {np.linalg.norm(g(x))}")

    start = datetime.now()
    # 1.b) steepest descent, exact line search
    x = np.array([[0.], [0.], [0.], [0.]])
    gx = g(x)
    dx = -gx
    alpha = (
        -np.dot(np.transpose(dx), gx)
        / np.linalg.multi_dot([np.transpose(dx), A, dx])
    )
    x += alpha * dx
    k = 1

    while np.linalg.norm(g(x)) >= error:
        gx = g(x)
        dx = -gx
        alpha = (
            -np.dot(np.transpose(dx), gx)
            / np.linalg.multi_dot([np.transpose(dx), A, dx])
        )
        x += alpha * dx
        k += 1

    print(datetime.now() - start)
    print("Steepest descent:")
    print(f"k = {k}, x = {x.flatten()}, ||g|| = {np.linalg.norm(g(x))}")

    start = datetime.now()
    # (2) quasi-Newton DFP, Newton's root finding method; Rosenbrock's
    function
    error = 10 ** -4
    x = np.array([[-1.], [2.]])
    x_list = [np.copy(x)]
    f = lambda x: 100 * (x[1] - x[0]**2)**2 + (1 - x[0])**2
    g = lambda x: np.array([
        2 * (200 * x[0]**3 - 200 * x[0] * x[1] + x[0] - 1),
        200 * (x[1] - x[0]**2)
    ])
    C0 = np.identity(np.size(x))
    g0 = g(x)
    d = -np.dot(C0, g0)
    df_tilde = lambda a, x, d: (
        200 * (d[1] - 2 * d[0] * (x[0] + a * d[0]))
        * ((x[1] + a * d[1]) - (x[0] + a * d[0])**2)
        - 2 * d[0]*(1 - (x[0] + a * d[0]))
    )
    d2f_tilde = lambda a, x, d: (
        -400 * (d[0]**2) * ((x[1] + a * d[1]) - (x[0] + a * d[0])**2)
        + 200 * (d[1] - 2 * d[0] * (x[0] + a * d[0]))
        * (d[1] - 2 * d[0] * (x[0] + a * d[0])) + 2 * d[0]**2
    )
    x += newton(x, d, df_tilde, d2f_tilde, 1 / np.linalg.norm(d),
   error**2) * d
    x_list.append(np.copy(x))

    while np.linalg.norm(g(x)) >= error:
        gx = g(x)
        delta_x = x - x_list[-2]
        delta_g = gx - g(x_list[-2])
        C = (
            C0 + np.dot(delta_x, np.transpose(delta_x))
            / np.dot(np.transpose(delta_x), delta_g)
            - np.linalg.multi_dot([C0, delta_g, np.transpose(delta_g),
    C0])
            / np.linalg.multi_dot([np.transpose(delta_g), C0, delta_g
   ])
        )
        C0 = np.copy(C)
        d = -np.dot(C, gx)
        x += newton(
            x,
            d,
            df_tilde,
            d2f_tilde,
            1 / np.linalg.norm(d),
            error**2
        ) * d
        x_list.append(np.copy(x))

    print(datetime.now() - start)
    print("Rosenbrock's function:")
    print(
        f"k = {len(x_list) - 1},"
        f"x = {x.flatten()},"
```

```python
        f"||g|| = {np.linalg.norm(g(x))}"
    )
    xvals, yvals = np.split(np.array([i.flatten() for i in x_list]),
    2, axis=1)
    plt.plot(xvals, yvals, 'ro-', linewidth=1.0)
    plt.text(xvals[0], yvals[0], r"$x^{(0)}$")
    plt.text(xvals[-1], yvals[-1], fr"$x^{{({len(xvals)-1})}}$")
    plt.title(r"$f(x_1,x_2)=100(x_2-x_1^2)^2+(1-x_1)^2$")
    plt.xlabel(r"$x_1$")
    plt.ylabel(r"$x_2$")
    X, Y = np.meshgrid(
        np.arange(xvals.min() - .5, xvals.max() + .5, .01),
        np.arange(yvals.min() - .5, yvals.max() + .5, .01)
    )
    Z = f([X, Y])
    plt.contour(X, Y, Z, 100)
    plt.show()


def newton(
    x: np.ndarray,
    d: np.ndarray,
    df_tilde: callable,
    d2f_tilde: callable,
    a: np.ndarray,
    error: float
) -> float:
    a -= df_tilde(a, x, d) / d2f_tilde(a, x, d)
    while abs(df_tilde(a, x, d)) > error:
        a -= df_tilde(a, x, d) / d2f_tilde(a, x, d)
    return a

if __name__ == "__main__":
    main()
```

References

[1] R. Fletcher, ”Practical methods of optimization”, Wiley, New York, p54, 1987.

[2] H. H. Rosenbrock, ”An Automatic Method for Finding the Greatest or Least Value of a Function”, The Computer Journal, Volume 3, Issue 3, 1960, p175-184