1 Theoretical problems

1.1 Periodic GD behaviour

To find the appropriate parameters which cause periodic behaviour, we use the GD update rules to define multiple equations and solve them.

1.1.1 Gradient descent

We assume

$$f(x) = \frac{x^2}{2},$$

which is strictly convex and its gradient equals

$$\nabla f(x) = x.$$

Since f is symmetric, we can compute parameters for which $x_2 = -x_1$ and thus $x_3 = x_1$. This means that the sequence is 2-periodic. The second point can be written as

$$x_2 = x_1 - \gamma \nabla f(x_1) = x_1 - \gamma x_1.$$

We write $x_1 = -x_2$ and compute the learning rate:

$$\begin{aligned} x_1 &= -x_2, \\ x_1 &= \gamma x_1 - x_1, \\ \gamma &= 2. \end{aligned}$$

Let $x_1 = 3$. We can confirm the sequence is indeed periodic:

$$\begin{aligned} x_1 &= 3, \\ x_2 &= x_1 - \gamma \nabla f(x_1) = -3, \\ x_3 &= x_2 - \gamma \nabla f(x_2) = 3 = x_1. \end{aligned}$$

1.1.2 Polyak gradient descent

On a function whose gradient equals

$$\nabla f(x) = \begin{cases} 25x, & x < 1, \\ x + 24, & 1 \leq x \leq 2, \\ 25x - 24, & 2 < x, \end{cases}$$

we can observe that the Polyak GD converges to the cycle $x_1 \approx 0.65$, $x_2 \approx -1.8$ and $x_3 \approx 2.12$ for the optimal parameters γ and μ.

1.1.3 Nesterov gradient descent

1.2 Polyak GD optimal learning rates

Our function is a quadratic, so we can use Theorem 5.2 to compute the optimal learning rates. First, we need to determine α and β. For this, we need to compute the eigenvalues of the Hessian. The gradient of f equals

$$\nabla f(x,y,z) = \begin{bmatrix} 2x+3 \\ 4y-2z-4 \\ -2y+8z+5 \end{bmatrix},$$

and from it, it follows that the Hessian is

$$\nabla^2 f(x,y,z) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 4 & -2 \\ 0 & -2 & 8 \end{bmatrix}.$$

Because this matrix is block diagonal, the first eigenvalue is $\lambda_1 = 2$ and the other two are eigenvalues of the 2×2 matrix in the bottom right corner:

$$\begin{vmatrix} 4-\lambda & -2 \\ -2 & 8-\lambda \end{vmatrix} = (4-\lambda)(8-\lambda) - 4 = \lambda^2 - 12\lambda + 28.$$

We compute the remaining eigenvalues using the quadratic formula:

$$\lambda_{2,3} = \frac{12 \pm \sqrt{12^2 - 4 \cdot 28}}{2} = 6 \pm 2\sqrt{2}.$$

This means that $\alpha = 2$ and $\beta = 6 + 2\sqrt{2}$. The learning rates from the theorem are then:

$$\gamma = \frac{2}{(\sqrt{\alpha} + \sqrt{\beta})^2} \doteq 0.104,$$
$$\mu = \left(\frac{\sqrt{\beta} - \sqrt{\alpha}}{\sqrt{\beta} + \sqrt{\alpha}} \right)^2 \doteq 0.081.$$

2 Programming problems

2.1 GD implementations

We implement ordinary GD and its modification in Python.

```
def gradient_descent(x_1, gradient, learning_rate, *, steps=0, timeout=-1):
    end = time() + timeout
    while time() < end or steps > 0:
        x_1 = x_1 - learning_rate * gradient(x_1)
        steps -= 1
    return x_1


def polyak_gd(x_1, gradient, learning_rate, momentum, *, steps=0, timeout=-1):
    end = time() + timeout
    prev, curr = x_1, x_1
    while time() < end or steps > 0:
        x_k = curr - learning_rate * gradient(curr) + momentum * (curr - prev)
        prev, curr = curr, x_k
        steps -= 1
    return curr


def nesterov_gd(x_1, gradient, learning_rate, momentum, *, steps=0, timeout=-1):
    end = time() + timeout
    prev, curr = x_1, x_1
    while steps > 0 or time() < end:
        offset = momentum * (curr - prev)
        x_k = curr - learning_rate * gradient(curr + offset) + offset
        prev, curr = curr, x_k
        steps -= 1
    return curr


def ada_grad(x_1, gradient, learning_rate, *, steps=0, timeout=-1):
    end = time() + timeout
    gradients_sum = np.zeros_like(x_1) + 1e-8
    while time() < end or steps > 0:
        grad = gradient(x_1)
        gradients_sum = gradients_sum + grad**2
        D_k = np.diag(np.power(gradients_sum, -0.5))
        x_1 = x_1 - learning_rate * D_k @ grad
        steps -= 1
    return x_1
```

2.2 Newton methods

Similarly to the previous problem, we implement the Newton method and the BFGS quasi-Newton method in Python.

```
def newton(x_1, gradient, hessian, *, steps=0, timeout=-1):
    end = time() + timeout
    while steps > 0 or time() < end:
        grad = gradient(x_1)
        hess = hessian(x_1)
        x_1 = x_1 - np.linalg.solve(hess, grad)
        steps -= 1
    return x_1


def bfgs(x_1, gradient, *, steps=0, timeout=-1):
    end = time() + timeout
    prev, curr = x_1, x_1
    prev_g, curr_g = gradient(prev), gradient(curr)
    B_k = np.eye(len(x_1))

    while steps > 0 or time() < end:
        # Clip the update step to prevent nan values
        x_k = curr - np.clip(B_k @ curr_g, -1e5, 1e5)
        prev, curr = curr, x_k
        prev_g, curr_g = curr_g, gradient(curr)

        gamma, delta = curr_g - prev_g, curr - prev
        dg_dot = np.dot(delta, gamma)

        if dg_dot > 1e-10:
            V = np.eye(len(x_1)) - np.outer(gamma, delta) / dg_dot
            B_k = V.T @ B_k @ V + np.outer(delta, delta) / dg_dot
        steps -= 1

    return curr
```

2.3 Comparison of methods

We compare the implemented methods on three functions with two starting points for each. To be able to use the GD methods and the BFGS, we need their gradients:

$$\nabla f_a(x,y,z) = \begin{bmatrix} 34x - 16y + 6z + 1 \\ -16x + 16y + 1 \\ 6x + 6z \end{bmatrix},$$
$$\nabla f_b(x,y,z) = \begin{bmatrix} 2(x-1) - 400x(y-x^2) \\ 2(y-1) + 200(y-x^2) - 400y(z-y^2) \\ 200(z-y^2) \end{bmatrix}.$$

For the Newton method we also need the Hessian:

$$\nabla^2 f_a(x,y,z) = \begin{bmatrix} 34 & -16 & 6 \\ -16 & 16 & 0 \\ 6 & 0 & 6 \end{bmatrix},$$
$$\nabla^2 f_b(x,y,z) = \begin{bmatrix} 2 - 400y + 1200x^2 & -400x & 0 \\ -400x & 202 - 400z + 1200y^2 & -400y \\ 0 & -400y & 200 \end{bmatrix}.$$

We omit the gradient f_c due to its complexity and compute the hessian $\nabla^2 f_c$ symbolically using `Sympy`. For each experiment, we test multiple learning rates (1e−5, 1e−4, 1e−2) and use the one which produces the lowest function value. For the momentum, we use 0.1 because it is the largest value for which the methods converged.

2.3.1 Function f_a

Observing results in Tables 1 and 2, we can see that all methods at some point converge to the minimum -0.198. In all cases, the Newton method works the best. Other methods converge slower, only catching up at $N = 100$ and $t \geq 0.1$ s. When stopping by time criterion, all methods perform as well as the Newton method. BFGS performs as well as Newton, for $N \geq 10$.
Looking at the results, we can assume that the first starting point is better because all methods return a lower functional value.

Stopping	**GD**	**Polyak**	**Nesterov**	**AdaGrad**	**Newton**	**BFGS**
$N = 2$	−0.035	−0.037	−0.037	−0.032	−0.198	30.3
$N = 5$	−0.073	−0.077	−0.077	−0.056	−0.198	47.6
$N = 10$	−0.113	−0.119	−0.119	−0.082	−0.198	−0.198
$N = 100$	−0.198	−0.198	−0.198	−0.180	−0.198	−0.198
$t = 0.1$ s	−0.198	−0.198	−0.198	−0.198	−0.198	−0.198
$t = 1$ s	−0.198	−0.198	−0.198	−0.198	−0.198	−0.198
$t = 2$ s	−0.198	−0.198	−0.198	−0.198	−0.198	−0.198

Table 1: Minimal function value obtained by running all methods on f_a with $\mathbf{x}_1 = [0,0,0]^\mathsf{T}$.

Stopping	**GD**	**Polyak**	**Nesterov**	**AdaGrad**	**Newton**	**BFGS**
$N = 2$	6.041	5.860	5.903	10.563	−0.198	430.9
$N = 5$	3.673	3.411	3.435	10.2	−0.198	48.6
$N = 10$	1.854	1.607	1.625	9.756	−0.198	−0.198
$N = 100$	−0.192	−0.195	−0.195	6.970	−0.198	−0.198
$t = 0.1$ s	−0.198	−0.198	−0.197	−0.198	−0.198	−0.198
$t = 1$ s	−0.198	−0.198	−0.198	−0.198	−0.198	−0.198
$t = 2$ s	−0.198	−0.198	−0.198	−0.198	−0.198	−0.198

Table 2: Minimal function value obtained by running all methods on f_a with $\mathbf{x}_1 = [1, 1, 0]^\mathsf{T}$.

2.3.2 Function f_b

The results in Table 3 and 4 are significantly different from the previous ones. First, we look at the first starting point. At $N = 2$ only the Newton method comes close to the minimum of 0.0, but other methods are far off. In other cases, the Newton method converges to the correct minimum, while the other methods come close at $N = 100$ and $t = 0.1$ s. For $t >= 1$ s, all methods obtain the same result. BFGS only converges after 0.1 s, otherwise it overflows.

Stopping	**GD**	**Polyak**	**Nesterov**	**AdaGrad**	**Newton**	**BFGS**
$N = 2$	8.091	7.926	7.941	8.155	0.035	/
$N = 5$	4.814	4.416	4.454	5.941	0.000	/
$N = 10$	2.113	1.755	1.787	4.056	0.000	/
$N = 100$	0.018	0.018	0.018	0.102	0.000	/
$t = 0.1$ s	0.0002	0.0007	0.0009	0.0076	0.0000	0.0000
$t = 1$ s	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
$t = 2$ s	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 3: Minimal function value obtained by running all methods on f_b with $\mathbf{x}_1 = [1.2, 1.2, 1.2]^\mathsf{T}$.

For the second starting point, the performance is worse. In the first two stopping criteria, all methods obtain similar results. At $N = 5$, the Newton method is the best, but at $N = 10$ its results are far worse. At that point, the Polyak GD is the best. At $N = 100$, the Newton method finally obtains the correct minimum. Other methods produce similar values. At $t = 0.1$ s, all methods except the AdaGrad, come very close to the minimum. For later times, all methods converge correctly.

Stopping	**GD**	**Polyak**	**Nesterov**	**AdaGrad**	**Newton**	**BFGS**
$N = 2$	8.831	8.612	8.649	9.752	9.062	/
$N = 5$	5.798	5.496	5.547	7.413	4.236	/
$N = 10$	4.490	4.386	4.402	5.748	1156.4	/
$N = 100$	4.208	4.207	4.207	4.258	0.000	/
$t = 0.1$ s	0.012	0.060	0.096	3.378	0.000	0.000
$t = 1$ s	0.0000	0.0000	0.0000	0.0002	0.0000	0.0000
$t = 2$ s	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 4: Minimal function value obtained by running all methods on f_b with $\mathbf{x}_1 = [-1, 1.2, 1.2]^\mathsf{T}$.

2.3.3 Function f_c

For the last function, the BFGS again does not converge and the Newton method does not work for a difference. For the first starting point at $N = 2$, the Newton method does the best, but at $N = 5$, $N = 10$ and $N = 100$, the Polyak GD is the best. Still, none of the methods obtain the correct minimum. For time limits, all gradient descents converge correctly. The AdaGrad method performs slightly worse at $t = 0.1$ s.

Stopping	**GD**	**Polyak**	**Nesterov**	**AdaGrad**	**Newton**	**BFGS**
$N = 2$	6.75	6.49	6.56	13.7	2.15	/
$N = 5$	3.97	3.66	3.70	13.3	2.49	/
$N = 10$	2.04	1.82	1.83	12.8	1332.2	/
$N = 100$	0.08	0.07	0.07	9.3	9.9	/
$t = 0.1$ s	0.000	0.000	0.000	0.061	9.9	/
$t = 1$ s	0.000	0.000	0.000	0.000	9.9	/
$t = 2$ s	0.000	0.000	0.000	0.000	9.9	/

Table 5: Minimal function value obtained by running all methods on f_c with $\mathbf{x}_1 = [1, 1]^\mathsf{T}$.

The second starting point seems much worse because all methods have problems obtaining the correct minimum. For all $N \leq 100$, none of the methods come close. At time-stopping criteria, GD, Polyak and Nesterov obtain a value close to the correct minimum. AdaGrad is far off, the Newton method is stuck at around 14 and the BFGS diverges.

Stopping	**GD**	**Polyak**	**Nesterov**	**AdaGrad**	**Newton**	**BFGS**
$N = 2$	1557	774	826	169 646	15 056	/
$N = 5$	1118	565	605	165 200	423	/
$N = 10$	733	416	441	160 163	14.5	/
$N = 100$	57.1	44.7	45.5	127 388	14.2	/
$t = 0.1$ s	0.03	0.03	0.03	6973	14.2	/
$t = 1$ s	0.01	0.01	0.02	477	14.2	/
$t = 2$ s	0.002	0.004	0.006	208	14.2	/

Table 6: Minimal function value obtained by running all methods on f_c with $\mathbf{x}_1 = [4.5, 4.5]^\mathsf{T}$.

2.4 Linear regression

To fit a linear line to our data points, we add a column of ones to our data x. The parameters we are estimating are $\beta = [k, n]^\mathsf{T}$. We define the loss function

$$J(\beta) = \frac{1}{2N} \|\mathbf{X}\beta - \mathbf{y}\|^2$$

and its gradient

$$\frac{\partial J}{\partial \beta} = \frac{1}{N} (\mathbf{X}\beta - \mathbf{y})^\mathsf{T} \mathbf{X}.$$

For the SGD, we compute the gradient just using a single instance:

$$\frac{\partial J}{\partial \beta} = (\beta^\mathsf{T} x_i - y_i) x_i.$$

The optimal parameters for our data are $(k, n) = (1, 0.5)$. We use $(0.1, 0.1)$ as the starting point for all methods. For gradient descents, we decrease the learning rate for higher N. The results of our methods are shown in Table 7. We can see that all methods perform similarly. They all obtain the correct coefficient k. At $N = 50$, methods struggle to find the right n because the sample is small. The same behaviour is observed using an optimiser from a library.

Steps N	**GD**	**SGD**	**Newton**	**BFGS**	**L-BFGS**
50	$(1.00, 0.38)$	$(1.00, 0.39)$	$(1.00, 0.38)$	$(1.00, 0.38)$	$(1.00, 0.38)$
100	$(1.00, 0.45)$	$(1.00, 0.46)$	$(1.00, 0.49)$	$(1.00, 0.49)$	$(1.00, 0.49)$
1 000	$(1.00, 0.10)$	$(1.00, 0.10)$	$(1.00, 0.51)$	$(1.00, 0.51)$	$(1.00, 0.51)$
10 000	$(1.00, 0.10)$	$(1.00, 0.10)$	$(1.00, 0.49)$	$(1.00, 0.49)$	$(1.00, 0.49)$
100 000	$(1.00, 0.10)$	$(1.00, 0.10)$	$(1.00, 0.50)$	$(1.00, 0.50)$	$(1.00, 0.50)$
1 000 000	$(1.00, 0.10)$	$(1.00, 0.10)$	$(1.00, 0.50)$	$(1.00, 0.50)$	$(1.00, 0.52)$

Table 7: Obtained linear regression parameters (k, n) using multiple methods and number of steps N.

For other numbers of instances, the Newton, BFGS and L-BFGS successfully obtain the correct result. GD and SGD on the other hand, do not obtain the correct value n. The reason for this is probably due to the size differences in components of the gradient. Increasing the learning rate would give a better estimate of n, but k would probably diverge.

2.5 Gradient descent improvement

Let $f(\mathbf{x}) = \mathbf{x}^\mathsf{T}\mathbf{H}\mathbf{x}$. We can write a block diagonal PD matrix $\mathbf{H}$ as

$$\mathbf{H} = \begin{bmatrix} \mathbf{A}_1 & & & \\ & \mathbf{A}_2 & & \\ & & \ddots & \\ & & & \mathbf{A}_m \end{bmatrix},$$

where $\mathbf{A}_i$ are PD matrices. Because $\mathbf{H}\mathbf{x}$ is the gradient of f, the GD update rule is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma \nabla f(\mathbf{x}_k) = \mathbf{x}_k - \gamma \mathbf{H}\mathbf{x}_k.$$

Since $\mathbf{H}$ is block diagonal, the gradient can also be written in blocks:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \mathbf{A}_1\mathbf{x}^{(1)} \\ \mathbf{A}_2\mathbf{x}^{(2)} \\ \vdots \\ \mathbf{A}_m\mathbf{x}^{(m)} \end{bmatrix}.$$

For a quadratic function, we can compute the optimal learning rate as

$$\gamma = \frac{2}{\alpha + \beta}.$$

We can compute a learning rate using α_i and β_i for each block. This should improve the convergence because a high eigenvalue of some block will not affect the convergence of other components. The improved update rule is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \begin{bmatrix} \frac{2}{\alpha_1+\beta_1}\mathbf{A}_1\mathbf{x}_k^{(1)} \\ \frac{2}{\alpha_2+\beta_2}\mathbf{A}_2\mathbf{x}_k^{(2)} \\ \vdots \\ \frac{2}{\alpha_m+\beta_m}\mathbf{A}_m\mathbf{x}_k^{(m)} \end{bmatrix}.$$

To test this approach we use a matrix $\mathbf{H}$ of two blocks. The first one has eigenvalues 0.2 and 0.8, and the other one 72 and 80. In Figure 1, we can see that the improved GD converges much faster than the original one.

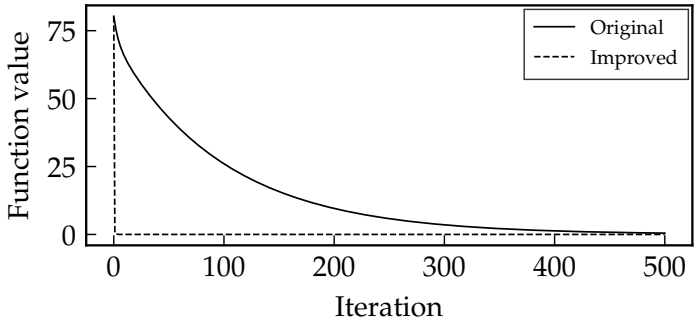

Figure 1: Convergence of both variants of the GD.