Reflection

D V Anantha Padmanabh
ME23B012

April 2024

Task 1

Breaking the Spectre

The task is to create a script, which takes a YAML serialized file as argument containing a Matrix object and output among other things, the *Spectral Decomposition*.

1.1 Introduction

1.1.1 Spectral Theorem

The spectral theorem states that any symmetric matrix M with $M_{ij} \in \mathbb{R}$ can be diagonalized by an orthogonal matrix. By this it means the matrix M can be written in the form

$$M = PDP^T$$

where P is an orthogonal matrix whose columns are the eigenvectors of M, D is a diagonal matrix whose entries are the eigenvalues of M, and P^T is the transpose of P. We can extend this to Normal Matrices too as seen in the task.

1.1.2 Spectral Decomposition

We know,

$$M = PDP^T$$

Let $\vec{u_1}, \vec{u_2}, \ldots, \vec{u_n}$ be the eigenvectors and $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of a matrix M. The spectral decomposition of M can be written as:

$$M = \begin{bmatrix} \vec{u_1} & \vec{u_2} & \dots & \vec{u_n} \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} \begin{bmatrix} \vec{u_1}^T \\ \vec{u_2}^T \\ \vdots \\ \vec{u_n}^T \end{bmatrix}$$

$$= \begin{bmatrix} \lambda_1\vec{u_1} & \lambda_2\vec{u_2} & \dots & \lambda_n\vec{u_n} \end{bmatrix} \begin{bmatrix} \vec{u_1}^T \\ \vec{u_2}^T \\ \vdots \\ \vec{u_n}^T \end{bmatrix}$$

$$M = \sum_{i=1}^{n} \lambda_i u_i u_i^T$$

1.1.3 Gram-Schmidt Process

The Gram-Schmidt process is a method for orthonormalizing a set of vectors. This process takes a finite, linearly independent set $S = \{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k\}$ and generates an orthogonal set $S' = \{\vec{u}_1, \vec{u}_2, \dots, \vec{u}_k\}$

In our case if eigenvalues are repeating, the corresponding eigenvectors will not be perpendicular. Here we can use the Gram-Schmidt process to orthogonalize the set of eigenvectors corresponding to the same eigenvalue.

Given a set of eigenvectors $\{\vec{v}_1, \vec{v}_2, \dots, \vec{v}_k\}$ corresponding to the same eigenvalue, the Gram-Schmidt process proceeds as follows:

$$\begin{aligned}
\vec{u}_1 &= \vec{v}_1, \\
\vec{u}_2 &= \vec{v}_2 - \frac{\langle \vec{v}_2, \vec{u}_1 \rangle}{\langle \vec{u}_1, \vec{u}_1 \rangle} \vec{u}_1, \\
\vec{u}_3 &= \vec{v}_3 - \frac{\langle \vec{v}_3, \vec{u}_1 \rangle}{\langle \vec{u}_1, \vec{u}_1 \rangle} \vec{u}_1 - \frac{\langle \vec{v}_3, \vec{u}_2 \rangle}{\langle \vec{u}_2, \vec{u}_2 \rangle} \vec{u}_2, \\
&\vdots \\
\vec{u}_k &= \vec{v}_k - \sum_{i=1}^{k-1} \frac{\langle \vec{v}_k, \vec{u}_i \rangle}{\langle \vec{u}_i, \vec{u}_i \rangle} \vec{u}_i.
\end{aligned}$$

The resulting set $\{\vec{u}_1, \vec{u}_2, \dots, \vec{u}_k\}$ is an orthogonal set of vectors. If we normalize each vector, we obtain an orthonormal set of eigenvectors.

1.2 Implementation

The script is written in Python and uses the SymPy library to perform operations on a 3x3 normal matrix.

The eigenvalues of the matrix are computed and the eigenvectors corresponding to these eigenvalues are then orthonormalized using the Gram-Schmidt process. This is done to handle the case of repeated eigenvalues (For distinct eigenvalues it just normalizes the eigenvectors).

```
## REMARK: Format of matrix.eigenvects() is a list of tuples,
# where each tuple is of the form (eigenvalue, multiplicity, eigenvectors)

## Eigenvalues of the matrix
lev=[]
for i in matrix.eigenvects():
    for j in range(i[1]):
        lev.append(i[0])

## Orthonormalize the eigenvectors using the Gram-Schmidt process
eigenvectors = []
for i in matrix.eigenvects():
    for j in range(i[1]):
        eigenvectors.append(i[2][j])
P_orthonormal = sympy.GramSchmidt(eigenvectors, True)
P_orthonormal = sympy.Matrix.hstack(*P_orthonormal)
```

The spectral decomposition of the matrix is computed by taking product of the orthogonalized eigenvectors adn their tranpose and multiplying each term by the corresponding eigenvalue.

```
## Spectral decomposition of the matrix
expr = sympy.matrices.expressions.matadd.MatAdd()
for i in range(len(lev)):
    vvT = P_orthonormal[:, i] * P_orthonormal[:, i].H
    term = sympy.MatMul(lev[i], vvT, evaluate=False)
    expr = sympy.MatAdd(expr, term, evaluate=False)
```

1.3 Observations and Conclusion

The spectral decomposition gives a way to split a matrix into a sort of orthogonal basis.

We need to use Gram-Schmidt Orthogonalization process for matrices with repeated eigenvalues because the corresponding eigenvectors wont be orthogonal.

Task 2

Optimaze

The task is to write a script to find the best fit plane for the points given as argument in a CSV file.

2.1 Introduction

2.1.1 Newton's Method of Optimization

Newton's method is a powerful iterative optimization algorithm to find the minimum (or maximum) of a function. It uses second order derivatives information for it.

The fundamental approach is initiated with an initial guess x_0 and refined iteratively via the formula:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{2.1}$$

Here, $f'(x_k)$ denotes the first derivative (gradient) of the function at x_k, while $f''(x_k)$ signifies the second derivative (Hessian) at x_k. This updating principle pivots the current estimate in the opposite direction of the function's slope, modulated by its curvature at that point.

For multidimensional optimization, the gradient vector $\mathbf{g}$ is defined as:

$$\mathbf{g} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

Each element captures the partial derivative of the function concerning the respective variable.

The Hessian matrix $\mathbf{H}$ is represented by:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

This matrix contains the second-order partial derivatives of the function.

The update direction $\Delta \mathbf{v}_t$ at iteration t is determined by:

$$\Delta \mathbf{v}_t = (\mathbf{H}^{-1})_t \mathbf{g}_t$$

Here, $(\mathbf{H}^{-1})_t$ denotes the inverse of the Hessian matrix at iteration t, and $\mathbf{g}_t$ represents the gradient at iteration t.

Finally, the subsequent iteration x_{k+1} is computed as:

$$x_{k+1} = x_k - \Delta \mathbf{v}$$

This iterative refinement continues until specific convergence criteria are satisfied.

Newton's method is indeed a powerful optimization algorithm, particularly for functions with smooth and well-behaved derivatives. But calculation of the second derivative can be computationally expensive.

- It does not work if the Hessian is not invertible. This is clear from the very definition of Newton's method, which requires taking the inverse of the Hessian.
- It may not converge at all, but can enter a cycle having more than 1 point or:
  - It could have a bad starting point for example where double derivative is zero i.e denominator becomes zero.
  - It could have a discontinuous derivative.
- It can converge to a saddle point instead of to a local minimum.

2.1.2 Fitting a plane using Newton's method

Given a set of points in 3D space we get find the best fit plane using Newton's Method

The perpendicular distance of a point (x_0, y_0, z_0) from a plane of the form $ax + by + cz = 1$ is

$$d_\perp = \frac{|ax_0 + by_0 + cz_0 - 1|}{\sqrt{a^2 + b^2 + c^2}}$$

since this is not differentiable directly, we can square it to differentiate

$$d_\perp^2 = \frac{(ax_0 + by_0 + cz_0 - 1)^2}{a^2 + b^2 + c^2}$$

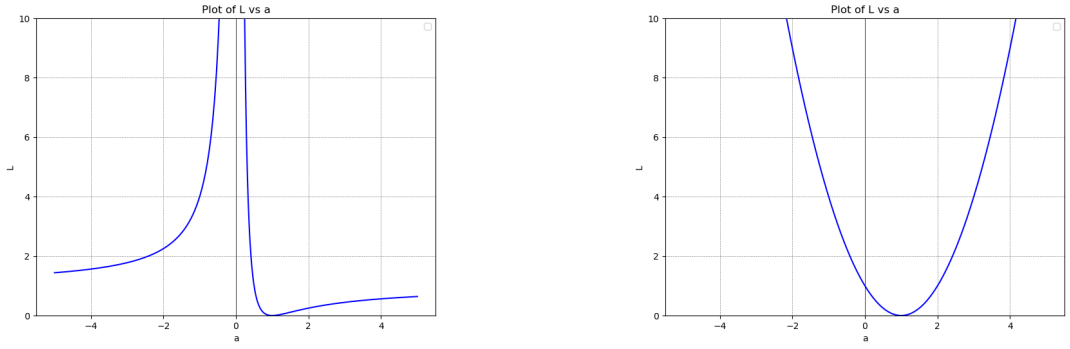
But we see that the double derivative of this is not simple. For example

$$\frac{\partial^2 (d_\perp)^2}{\partial a^2} = \frac{8(ax_0 + by_0 + cz_0 - 1)^2}{(a^2+b^2+c^2)^3} \cdot a^2 - \frac{8(ax + by + cz - 1)x}{(a^2+b^2+c^2)^2} \cdot a + \frac{2x^2}{a^2+b^2+c^2} - \frac{2(ax + by + cz - 1)^2}{(a^2+b^2+c^2)^2}$$

This will take a lot of computational time therefore we will the square of the absolute distance and use it as the loss function:

$$L = (ax_0 + by_0 + cz_0 - 1)^2$$

Also from the plots below we can see that the perpendicular distance has a minima at $-\infty$



(a) $d_\perp^2$ (b) d_{abs}^2

Figure 2.1: Plots

2.2 Implementation

```
## Define and initialise vector v
v = vector([0, 0, 0])
print(f"Initialised: {v[0]} {v[1]} {v[2]}")

count = error = 0

while True:
    # Calculate the gradient and Hessian at the current point
    grad_val = vector([0, 0, 0])
    H_val = matrix(3, 3, [0] * 9)  # 3x3 zero matrix
    for point in points:
        grad_val += grad.subs(
            {a: v[0], b: v[1], c: v[2], x: point[0], y: point[1], z: point[2]}
        )
        H_val += H.subs(
```

```
            {a: v[0], b: v[1], c: v[2], x: point[0], y: point[1], z: point[2]}
        )
        # Error calculation
        error += (v[0]*point[0] + v[1]*point[1] + v[2]*point[2] - 1)**2

    # Calculate the step size
    step = H_val.inverse() * grad_val

    # Check for convergence
    if step.norm() < 1e-6:
        break

    # Update the vector v
    v = v - step

    count += 1
```

2.3 Observation and Conclusion

The absolute distance as objective function is very efficient to do plane fitting.

Newton's method work good for easily differentiable objective function but since it is not the case most of the time, the method is not commonly used.