Computationally Finding Eigenvectors, Geometrically

Preface

I'm not quite familiar with many of the high-spec current algorithms that are used to find eigenvalues and eigenvectors with numeric stability, but I suppose they make great use of the characteristic polynomial and other math shenanigans that I probably am not all too familiar with nor sound all too interesting. Thus, it would seem fun to circumvent all that and create a highly scuffed algorithm to locate an eigenvector (potentially multiple?) using geometrical intuition, which seems far more easy to work with.

Ideas

Suppose we have a real matrix A (with non-zero determinant) which we wish to find an eigenvector of. Traditionally, an eigenvector of A is defined to be any vector $\mathbf{v}$ such that

$$A\mathbf{v} = \lambda\mathbf{v},$$

where λ is the corresponding eigenvalue. In geometric terms, the origin, the point described by $\mathbf{v}$, and the image of $\mathbf{v}$ under A must be collinear. Motivated by this, we can instead characterize an eigenvector of A differently. Observe that

$$\cos\theta = \frac{A\mathbf{v}\cdot\mathbf{v}}{\|A\mathbf{v}\|\|\mathbf{v}\|}.$$

Principally, this value is -1 or 1 whenever $\mathbf{v}$ is an eigenvector. To simplify this, we shall take

$$\cos^2\theta = \frac{(A\mathbf{v}\cdot\mathbf{v})^2}{\|A\mathbf{v}\|^2\|\mathbf{v}\|^2}$$

as our metric for how "close," $\mathbf{v}$ is to being an eigenvector, with 0 representing that $\mathbf{v}$ is orthogonal to an (all?) eigenvector and 1 representing that $\mathbf{v}$ is in fact an eigenvector.

In order to simplify matters slightly, we shall want to take $\|\mathbf{v}\| = 1$. This would guarantee that each $\mathbf{v}$ that we find such that $\cos^2\theta = 1$ is part of a different family (I wonder what the right term is for this) of eigenvectors. Since we want to travel around in space however and doing so is cumbersome on a sphere, we will leave the magnitude unfixed for now. Thus, our metric is given by

$$\Gamma_A(\mathbf{v}) := \frac{(A\mathbf{v}\cdot\mathbf{v})^2}{\|A\mathbf{v}\|^2\|\mathbf{v}\|^2},$$

and this is differentiable, meaning it naturally emits a gradient which we shall now hope to derive. Consider the partial derivative taken for some component v_k. Observe that (using shorthand notation for partial derivatives for typing convenience)

$$\partial_{v_k}\Gamma_A(\mathbf{v}) = \frac{\|A\mathbf{v}\|^2\|\mathbf{v}\|^2\partial_{v_k}(A\mathbf{v}\cdot\mathbf{v})^2 - (A\mathbf{v}\cdot\mathbf{v})^2\partial_{v_k}\|A\mathbf{v}\|^2\|\mathbf{v}\|^2}{\|A\mathbf{v}\|^4\|\mathbf{v}\|^4}.$$

So now we must find some expressions for the following

$$\partial_{v_k}(A\mathbf{v}\cdot\mathbf{v})^2, \quad \partial_{v_k}\|A\mathbf{v}\|^2\|\mathbf{v}\|^2.$$

We shall tackle these in parts. We first note that

$$A\mathbf{v} = \begin{bmatrix}\mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_n\end{bmatrix}\mathbf{v} = \begin{bmatrix}\mathbf{w}_1\\ \mathbf{w}_2\\ \vdots\\ \mathbf{w}_n\end{bmatrix}\mathbf{v} = \begin{bmatrix}\mathbf{w}_1\cdot\mathbf{v}\\ \mathbf{w}_2\cdot\mathbf{v}\\ \vdots\\ \mathbf{w}_n\cdot\mathbf{v}\end{bmatrix} = \begin{bmatrix}A_{11}v_1 + A_{12}v_2 + \cdots + A_{1n}v_n\\ A_{21}v_1 + A_{22}v_2 + \cdots + A_{2n}v_n\\ \vdots\\ A_{n1}v_1 + A_{n2}v_2 + \cdots + A_{nn}v_n\end{bmatrix}.$$

By basic rules of derivatives, we have that

$$\begin{aligned}\partial_{v_k}(A\mathbf{v}\cdot\mathbf{v})^2 &= 2(A\mathbf{v}\cdot\mathbf{v})\,\partial_{v_k}(A\mathbf{v}\cdot\mathbf{v})\\ &= 2(A\mathbf{v}\cdot\mathbf{v})(A\mathbf{v}\cdot\partial_{v_k}(\mathbf{v}) + \partial_{v_k}(A\mathbf{v})\cdot\mathbf{v})\\ &= 2(A\mathbf{v}\cdot\mathbf{v})(\mathbf{w}_k\cdot\mathbf{v} + \mathbf{u}_k\cdot\mathbf{v})\\ &= 2(A\mathbf{v}\cdot\mathbf{v})((\mathbf{w}_k+\mathbf{u}_k)\cdot\mathbf{v}).\end{aligned}$$

Similarly, for the second expression, we have

$$\begin{aligned}\partial_{v_k}\|A\mathbf{v}\|^2\|\mathbf{v}\|^2 &= \|A\mathbf{v}\|^2\,\partial_{v_k}\left(\|\mathbf{v}\|^2\right) + \partial_{v_k}\left(\|A\mathbf{v}\|^2\right)\|\mathbf{v}\|^2\\ &= 2v_k\|A\mathbf{v}\|^2 + \|\mathbf{v}\|^2\sum_{i=1}^n \partial_{v_k}(\mathbf{w}_i\cdot\mathbf{v})^2\\ &= 2v_k\|A\mathbf{v}\|^2 + 2\|\mathbf{v}\|^2\sum_{i=1}^n(\mathbf{w}_i\cdot\mathbf{v})\,\partial_{v_k}(\mathbf{w}_i\cdot\mathbf{v})\\ &= 2v_k\|A\mathbf{v}\|^2 + 2\|\mathbf{v}\|^2\sum_{i=1}^n A_{ik}\,(\mathbf{w}_i\cdot\mathbf{v})\\ &= 2v_k\|A\mathbf{v}\|^2 + 2\|\mathbf{v}\|^2(A\mathbf{v}\cdot\mathbf{u}_k).\end{aligned}$$

There are no doubt other ways to arrive at these expressions, but I find it very nice that they aren't completely intractable.

With the general expression for the partial derivative determined, we can now calculate the gradient as per usual:

$$\mathsf{grad}\,\Gamma_A(\mathbf{v}) = \begin{bmatrix} \partial_{v_1}\Gamma_A(\mathbf{v}) \\ \partial_{v_2}\Gamma_A(\mathbf{v}) \\ \vdots \\ \partial_{v_n}\Gamma_A(\mathbf{v}) \end{bmatrix}.$$

This allows us to define a sequence of converging guesses $\mathbf{g}_1, \mathbf{g}_2, \cdots$ where $\mathbf{g}_1$ is determined by some intial choice and subsequent values in the sequence are determined by the recursion

$$\mathbf{g}_n = \mathsf{normalize}\big(\mathbf{g}_{n-1} + \gamma_n\,\mathsf{grad}\,\Gamma_A(\mathbf{g}_{n-1})\big),$$

where γ_n is some sequence that denotes a learning rate for steps of the gradient ascent process. In total, each calculation of $\mathsf{grad}\,\Gamma_A(\mathbf{g}_n)$ takes $O(n^2)$ time.

Remark. One must note that this doesn't really handle eigenvectors that are associated with complex eigenvalues, but that's probably fine (there's likely some complex conjugates one has to include). In addition it would be interesting to find and perhaps prove the convergence rate of the algorithm. Although this only finds one eigenvector/value pair at a time, one could likely place points around the unit sphere and see where they converge to.

Following this notion, actually, it would be interesting to see the probability that, given a uniformly randomly chosen point on the surface of unit sphere, the algorithm converges to a particular eigen-pair. Is it necessarily true that they would be equiprobable?

Code:

```
from math import log, sqrt
import numpy as np

# A = np.array([
#     [0.32, 0.53, 0.69],
#     [0.12, 0.33, 0.54],
#     [0.99, 0.32, 0.62]
# ])
A = np.random.rand(400, 400)
# print(A)

n = A.shape[0]

g0 = np.array([1 if i == 0 else 0 for i in range(n)], dtype=np.float64)

def eigenind(v):
    projected = np.dot(A, v)

   return np.dot(projected, v) ** 2 / (np.dot(projected, projected) * np.dot(v, v))

def numerical_gradient(v):
    grad = np.array([0 for i in range(n)], dtype=np.float64)
    h = 0.01

    for k in range(n):
        right = eigenind(np.array([
            v[j] + (h if j == k else 0)
            for j in range(n)
        ], dtype=np.float64))
        left = eigenind(np.array([
            v[j] - (h if j == k else 0)
            for j in range(n)
        ], dtype=np.float64))

        grad[k] = (right - left) / (2 * h)

    return grad

def gradient(v):
    projected = np.dot(A, v)

    norm_v = np.dot(v, v)
    norm_projected = np.dot(projected, projected)

    dotted = np.dot(projected, v)

    grad = np.array([0 for i in range(n)], dtype=np.float64)
    for k in range(n):
        row_k = A[k, :]
        col_k = A[:, k]

        cross = np.dot(row_k + col_k, v)

        left = 2 * norm_v * norm_projected * dotted * cross
        right = 2 * dotted ** 2 * (
            v[k] * norm_projected +
            norm_v * np.dot(projected, A[:, k])
        )

        denom = (norm_v * norm_projected) ** 2

        grad[k] = (left - right) / denom

    return grad

# Wow that matches up really nicely yay
# print("Numerical gradient: ", numerical_gradient(g0))
# print("Gradient: ", gradient(g0))

g = g0
gamma = 0.1

scores = []

for i in range(100):
    ascent = g + gamma * gradient(g)
    g = ascent / np.linalg.norm(ascent)
    score = eigenind(g)

    scores.append(score)

    print(f"Step {i + 1} score: {score}")

print("Eigenvector:", g)
print("Eigenvalue:", np.linalg.norm(np.dot(A, g)) / np.linalg.norm(g))
```

```
# Using https://en.wikipedia.org/wiki/Rate_of_convergence#Order_estimation,
# we may determine an approximation for the convergence rate of the algorithm

# Tbh I'm not quite sure how accurate this is in terms of actually giving the
# convergence rate (I might have to look at the actual math later) but it gives
# that the convergence rate is roughly linear (although some of the terms are
# far greater or lower)
"""
estimates = [
    log(abs(
        (scores[k] - scores[k-1]) /
        (scores[k-1] - scores[k-2]))) /
        log(abs((scores[k-1] - scores[k-2]) /
        (scores[k-2] - scores[k-3])
    ))
    for k in range(3, len(scores))
]

q = sum(estimates) / len(estimates)

print("Estimate convergence rates:", estimates)
print("Approximate convergence rate:", q)
"""
```