Richardson Eigenvector Purification Algorithms

- Wikipedia
- Claude 3.5 Sonnet
- Gemini Advanced 2.0
- DeepSeek V3
- OpenAI

Prompt: """

1. What is Richardson eigenvector purification algorithm?
2. can you implemented the algorithm in python?
3. prepare a test-case to validate your python code using the following input matrix

```
A = np.array([[2, 1, 0],
              [1, 2, 1],
              [0, 1, 2]])
```

"""

In [1]:
```
MAX_ITER = 10000
TOLERANCE = 1e-10
# Richardson relaxation parameter
ALPHA = 0.1
```

Answer by Claude

In [2]:
```
import numpy as np

def richardson_eigenvector_purification_claude(A, v0, mu, max_iter=MAX_ITER, tol=TOLERANCE):
    """
    Richardson eigenvector purification algorithm
```

```
    Parameters:
    A: numpy array - square matrix
    v0: numpy array - initial guess for eigenvector
    mu: float - shift parameter (approximate eigenvalue)
    max_iter: int - maximum number of iterations
    tol: float - convergence tolerance

    Returns:
    v: numpy array - purified eigenvector
    """
    v = v0 / np.linalg.norm(v0)  # Normalize initial vector

    for i in range(max_iter):
        v_old = v.copy()

        # Apply iteration: v = (A - μI)v
        v = A @ v - mu * v

        # Normalize
        v = v / np.linalg.norm(v)

        # Check convergence
        if np.linalg.norm(v - v_old) < tol:
            break

    return v
```

In [3]:

```
import numpy as np

# Test matrix
A = np.array([[2, 1, 0],
              [1, 2, 1],
              [0, 1, 2]])

# Initial guess for eigenvector
v0 = np.ones(3)

# Let's use μ = 0 to find the dominant eigenvector
mu = 0

# Run the algorithm
result = richardson_eigenvector_purification_claude(A, v0, mu)
```

```
# Validate the result
# 1. Check if it's actually an eigenvector (Av = λv)
eigenvalue = (A @ result) / result
print("Computed eigenvector:", result)
print("\nVerification:")
print("Av =", A @ result)
print("λv =", eigenvalue[0] * result)

# 2. Compare with numpy's built-in eigenvalue solver
eigenvalues, eigenvectors = np.linalg.eig(A)
dominant_idx = np.argmax(np.abs(eigenvalues))
true_eigenvector = eigenvectors[:, dominant_idx]

print("\nTrue eigenvalue:", eigenvalues[dominant_idx])
print("True eigenvector:", true_eigenvector)
print("Difference norm:", np.linalg.norm(np.abs(result) - np.abs(true_eigenvector)))
```

```
Computed eigenvector: [0.5        0.70710678 0.5       ]

Verification:
Av = [1.70710678 2.41421356 1.70710678]
λv = [1.70710678 2.41421356 1.70710678]

True eigenvalue: 3.4142135623730923
True eigenvector: [-0.5        -0.70710678 -0.5       ]
Difference norm: 1.9154907684079144e-11
```

Answer by Gemini

In [4]:

```
import numpy as np

def richardson_eigenvector_purification_gemini(A, tolerance=TOLERANCE, max_iterations=MAX_ITER):
    """
    Approximates the dominant eigenvector of a matrix using the Richardson method.

    Args:
        A: The input square matrix (NumPy array).
        tolerance: The convergence tolerance.
        max_iterations: The maximum number of iterations.
```

```
    Returns:
        A tuple containing:
            - The approximated dominant eigenvector (NumPy array).
            - The number of iterations performed.
    """

    n = A.shape[0]
    x = np.ones(n)  # Initialize with a vector of ones
    #x = np.random.rand(n) # Initialize with a random vector

    for iteration in range(max_iterations):
        y = A @ x  # Matrix-vector multiplication
        x_new = y / np.linalg.norm(y)  # Normalize using L2-norm

        # Check for convergence
        if np.linalg.norm(x_new - x) < tolerance:
            return x_new, iteration + 1

        x = x_new

    print("Maximum iterations reached without convergence.")
    return x, max_iterations
```

In [5]:

```
import numpy as np

# ... (richardson_eigenvector_purification function from above) ...

# Input matrix
A = np.array([[2, 1, 0],
              [1, 2, 1],
              [0, 1, 2]])

# Run the Richardson algorithm
eigenvector, iterations = richardson_eigenvector_purification_gemini(A)

print("Approximated dominant eigenvector:", eigenvector)
print("Number of iterations:", iterations)

# Verification using NumPy's built-in function (for comparison)
eigenvalues, eigenvectors = np.linalg.eig(A)
dominant_eigenvalue_index = np.argmax(np.abs(eigenvalues))
true_dominant_eigenvector = eigenvectors[:, dominant_eigenvalue_index]
```

```
true_dominant_eigenvector = true_dominant_eigenvector/np.linalg.norm(true_dominant_eigenvector) #normalize

print("\nVerification using np.linalg.eig:")
print("True dominant eigenvector:", true_dominant_eigenvector)

# You can also check if the approximated eigenvector is close to the true one
# using np.allclose() with a certain tolerance.
#For example:
print(f"\nAre the vectors close: {np.allclose(eigenvector, true_dominant_eigenvector, atol=1e-05)}") #chec
```

```
Approximated dominant eigenvector: [0.5        0.70710678 0.5       ]
Number of iterations: 13

Verification using np.linalg.eig:
True dominant eigenvector: [-0.5        -0.70710678 -0.5       ]

Are the vectors close: False
```

Answer by DeepSeek

In [6]:

```
import numpy as np

def richardson_eigenvector_purification_deepseek(A, num_eigenvalues, max_iter=MAX_ITER, tol=TOLERANCE, alpl
    """
    Richardson eigenvector purification algorithm to compute eigenvalues and eigenvectors of a matrix A.

    Parameters:
        A (numpy.ndarray): The input matrix (symmetric or Hermitian).
        num_eigenvalues (int): Number of eigenvalues/eigenvectors to compute.
        max_iter (int): Maximum number of iterations.
        tol (float): Tolerance for convergence.
        alpha (float): Relaxation parameter.

    Returns:
        eigenvalues (numpy.ndarray): Computed eigenvalues.
        eigenvectors (numpy.ndarray): Computed eigenvectors (column-wise).
    """
    n = A.shape[0]
    eigenvectors = np.random.rand(n, num_eigenvalues)  # Random initial guess for eigenvectors
    eigenvalues = np.zeros(num_eigenvalues)
```

```
    for _ in range(max_iter):
        for i in range(num_eigenvalues):
            v = eigenvectors[:, i]
            Av = A @ v
            lambda_est = (v.T @ Av) / (v.T @ v)  # Rayleigh quotient for eigenvalue estimate
            eigenvectors[:, i] = v + alpha * (Av - lambda_est * v)  # Richardson update

            # Normalize the eigenvector
            eigenvectors[:, i] /= np.linalg.norm(eigenvectors[:, i])

            # Check for convergence
            if np.linalg.norm(A @ eigenvectors[:, i] - lambda_est * eigenvectors[:, i]) < tol:
                eigenvalues[i] = lambda_est

        # Break if all eigenvalues have converged
        if np.allclose(A @ eigenvectors, eigenvectors @ np.diag(eigenvalues), atol=tol):
            break

    return eigenvalues, eigenvectors
```

In [7]:

```
# Test case
if __name__ == "__main__":
    # Input matrix
    A = np.array([[2, 1, 0],
                  [1, 2, 1],
                  [0, 1, 2]])

    # Number of eigenvalues to compute
    num_eigenvalues = 2

    # Compute eigenvalues and eigenvectors using Richardson algorithm
    eigenvalues, eigenvectors = richardson_eigenvector_purification_deepseek(A, num_eigenvalues)

    # Compute eigenvalues and eigenvectors using NumPy for validation
    eigenvalues_np, eigenvectors_np = np.linalg.eigh(A)
    eigenvalues_np = eigenvalues_np[:num_eigenvalues]
    eigenvectors_np = eigenvectors_np[:, :num_eigenvalues]

    # Print results
    print("Matrix A:")
    print(A)
    print("\nEigenvalues (Richardson):", eigenvalues)
```

```
    print("Eigenvalues (NumPy):", eigenvalues_np)
    print("\nEigenvectors (Richardson):")
    print(eigenvectors)
    print("\nEigenvectors (NumPy):")
    print(eigenvectors_np)

    # Validate results
    assert np.allclose(eigenvalues, eigenvalues_np, atol=1e-4), "Eigenvalues do not match!"
    print("\nValidation: Eigenvalues match within tolerance!")
```

```
Matrix A:
[[2 1 0]
 [1 2 1]
 [0 1 2]]

Eigenvalues (Richardson): [3.41421356 3.41421356]
Eigenvalues (NumPy): [0.58578644 2.        ]

Eigenvectors (Richardson):
[[0.5        0.5       ]
 [0.70710678 0.70710678]
 [0.5        0.5       ]]

Eigenvectors (NumPy):
[[ 5.00000000e-01 -7.07106781e-01]
 [-7.07106781e-01 -2.73553622e-16]
 [ 5.00000000e-01  7.07106781e-01]]
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[7], line 30
     27 print(eigenvectors_np)
     29 # Validate results
---> 30 assert np.allclose(eigenvalues, eigenvalues_np, atol=1e-4), "Eigenvalues do not match!"
     31 print("\nValidation: Eigenvalues match within tolerance!")

AssertionError: Eigenvalues do not match!
```

Answer by OpenAI

In [8]:
```
import numpy as np
```

```
def richardson_eigenvector_purification_openai(A, v0, lambda_approx, alpha=ALPHA, tol=TOLERANCE, max_iter=M
    """
    Perform Richardson eigenvector purification.

    Parameters:
        A (numpy.ndarray): The matrix.
        v0 (numpy.ndarray): Initial guess for the eigenvector.
        lambda_approx (float): Approximate eigenvalue associated with the eigenvector.
        alpha (float): Richardson relaxation parameter.
        tol (float): Tolerance for convergence.
        max_iter (int): Maximum number of iterations.

    Returns:
        numpy.ndarray: Purified eigenvector.
        int: Number of iterations performed.
    """
    v = v0 / np.linalg.norm(v0)  # Normalize initial vector
    for k in range(max_iter):
        # Compute the residual
        r = A @ v - lambda_approx * v
        # Update the eigenvector
        v_new = v - alpha * r
        # Normalize
        v_new /= np.linalg.norm(v_new)
        # Check for convergence
        if np.linalg.norm(r) < tol:
            return v_new, k + 1
        v = v_new
    raise ValueError("Richardson purification did not converge within the maximum number of iterations.")
```

In [9]:

```
# Test matrix
A = np.array([[2, 1, 0],
              [1, 2, 1],
              [0, 1, 2]])

# Initial guess for eigenvector
v0 = np.array([1, 0, 0])

# Approximate eigenvalue (choose a known eigenvalue of A, e.g., 3)
lambda_approx = 3

# Richardson relaxation parameter
```

```
alpha = ALPHA

# Perform purification
try:
    eigenvector, iterations = richardson_eigenvector_purification_openai(A, v0, lambda_approx, alpha)
    print(f"Purified Eigenvector: {eigenvector}")
    print(f"Iterations: {iterations}")

    # Validation
    print("Validation:")
    print("Av = ", A @ eigenvector)
    print("lambda * v = ", lambda_approx * eigenvector)
except ValueError as e:
    print(e)
```

```
Richardson purification did not converge within the maximum number of iterations.
```

In [ ]: