1 Usage

```
gcc eigeng.c eigeng_tests.c -o eigeng_tests -O3 -fno-stack-protector
.\eigeng_tests
```

2 Analysis

From a quick analysis, the eigenvalue algorithm in eigeng.c is atleast an $O(n^3)$ algorithm. To prove that this is the case, consider the balance() function in eigeng.c, balance() goes through the matrix, cell by cell - which takes $O(n^2)$ time-, and swaps zero rows[1] with the last known non-zero row. By performing this swap, it is possible that the last non-zero row became a zero row so if a swap is performed, the algorithm repeats the step of looking for zero rows. In balance(), the algorithm potentially looks through the entire matrix n times, so the overall runtime is $O(n^3)$.

Of course, balance() is not efficient. Instead of looking through the entire matrix if a swap was performed, we can instead just look at the swapped rows and check if they are zeroes. This reduces the runtime of balance() to $O(n^2)$ because there would be n checks each needing to check n cells. Although this in theory is an improvement to the speed of the algorithm, it's not that simple. The runtime of other units of the algorithm are also $O(n^3)$ so, firstly, this change will not dramatically improve the efficiency because balance() is not a bottleneck function. Secondly, this change will result in more complicated code.

3 Design

In eigeng_tests.c, I've written a function called assert_matrix() which compares the output of eigen() from eigeng.c to correct eigenvalues calculated with wolfram alpha. There will always be n eigenvalues for an $n \times n$ matrix, but the ordering of the eigenvalues are unknown, so assert_matrix() checks if each correct eigenvalue has been seen in the output. This algorithm takes $O(n^2)$, but the process can be sped up to $O(n \log n)$. To do this, the correct eigenvalues are first sorted then we can find the right eigenvalue by binary searching which takes $O(\log n)$. Doing this n times for each returned eigenvalue, this process takes $O(n \log n)$. Since eigen() runs in $O(n^3)$, an $O(n^2)$

[1] a zero row is a row with all zeros in non-diagonal cells

time is negligible so the more complicated binary search method was not employed.

3.1 System Tests

Bugs exist in the edges of the input, so the tests are focused around special matrices.

Test 1: Normal matrix. Nothing special about test 1. Test 1 is a random collection of integers.

Test 2: Symmetric matrix. The matrix is symmetrical in it's main diagonal.

Test 3: Zero matrix. All cells in the matrix is zero. All eigenvalues and vectors are 0.

Test 4: Identity matrix. The matrix is the identity. All eigenvalues are real.

Test 5: Orthogonal matrix. Every row/column is orthogonal to every other row/column respectively.

3.2 Unit Tests

TODO

4 Suggestions and Improvements

4.1 Failed Idea: Binary Search Using Determinant

Initially when tackling this problem, I recalled the linear algebra definition of eigenvalues and eigenvectors: $M\vec{v} = \lambda\vec{v}$ where λ is an eigenvalue of M and $\vec{v}$ is an eigenvector. This equation can be re-written as $(M-\lambda I)\vec{v} = \vec{0}$. A valid eigenvalue for M is a λ that satisfies this equation so, we just have to check if the determinant of $(M - \lambda I) = 0$. This can be done using binary search, by guessing a value for λ and calculating the resulting determinant. The problem is the fastest algorithm to calculate the determinant is Gaussian Elimination which takes $O(n^3)$ time. When accounting for the binary search part, the overall runtime of this idea would be $O(n^3) \log n$ which is slower than the $O(n^3)$ runtime of eigeng.c.

4.2 Potential Idea: Stack Memory Instead of Heap Memory

When arrays are allocated using malloc()/calloc(), space on the heap is reserved on the heap for the array and a pointer to the space is written to the stack. When accessing heap memory, the pointer on the stack is first read, then the appropriate memory can be read - arrays on the heap use two read operations. If the array is on the stack, the array can be accessed directly with one read operation. Of course, caching makes it so that accessing heap memory isn't that much slower but stack memory is still accessed faster than heap memory.

Problems: the stack is much smaller than the heap. In C, I believe that you can only allocate around a 200×200 array before you run out of stack space. This potentially isn't a problem, however, since the algorithm runs in $O(n^3)$. If n is over 1000, the algorithm would take several hours on modern machines so n has to be a small integer.

4.3 Potential Idea: Global Variables

Similar to using stack memory, the idea to use global variables stems from the idea that passing objects through parameters is slow. When objects such as matrices are passed through parameters, a pointer to the object is pushed on to the stack. The pointer, used to find the location of the object, is read once the object needs to be accessed. Accessing objects through parameters, therefore, take two memory reads. Global objects are directly on the stack, so accessing them only takes one memory read.

Problems: It's much easier to make mistakes when dealing with global variables.