IMPROVEMENTS TO HESSENBERG REDUCTION

SHANKAR, YANG, HAO

1. abstract

The problem of matrix reduction is critical to effectively solving linear algebra problems like the Standard Eigen Value problem. There are real world applications on precisely these kinds of problems like structural mechanics, computational fluid dynamics, web engine rank search etc. A triangular matrix is probably the best form to solve these problems and while the computational effort to transform a dense matrix into a triangular one is intense, it is possible to generate an alternate form called the Hessenberg matrix with considerably less effort. A Hessenberg matrix is a triangular matrix with a few more sub diagonals than a triangular matrix. By identifying transformation techniques to convert a general matrix into the Hessenberg form, called Hessenberg reduction, it is seen that a computationally efficient method to perform operations like Eigen value problem arise when the first step is converting a general matrix into Hessenberg form.

2. Introduction

Real world problems that require mathematical modeling are huge sparse matrices, most of which are non-symmetric. Be it fluid computations, weather forecasting, earthquake modeling, structural integrity analysis and so on, most of it deals with obtaining the eigen values of real life huge matrices that are non-symmetric in nature. This would mean dealing with complex numbers and we know that complex number computations are twice as slow as real number computations. Therefore, it makes sense to reduce complex number computations by delaying them as much as possible in the computational chain. Traditional solutions to the eigen value problems and linear solves rely on a triangular matrix transformation from which it is quite easy to extract the necessary results. However, it can be seen that it is possible to obtain near triangular matrix forms without using complex numbers.

Inferring from the above discussion, we can use similarity transforms to reduce a matrix to its triangular form in order to find the eigen values of the system. However, by Abel's theorem, it is clear that there exists no algorithm which can compute the mentioned transformation in a finite number of steps. But it is possible to bring a matrix close to its triangular form in a finite number of steps. This matrix form is called Hessenberg matrix and this reduction is the first step towards simplifying an eigen value problem by bringing it as close to a triangular matrix as possible through similarity transforms and is called Hessenberg Reduction.

An n x n matrix A is called upper Hessenberg if

$$a_{ij} = 0 \text{ whenever } i > j + 1 \text{ ie. } \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \end{bmatrix} \tag{1}$$

If the matrix is symmetric (hermitian for a complex matrix), then this reduction results in a tridiagonal form mentioned below.

$$\begin{bmatrix} * & * & & & \\ * & * & * & & \\ & * & * & * & \\ & & * & * & * \\ & & & * & * \end{bmatrix}$$

2.1. **Hessenberg Reduction.** For the step 1 in the Hessenberg reduction process, we use orthogonal transformations (reflections/rotations) to zero out the necessary elements in each column. Using similarity transforms, if we have an orthogonal transformation vector $\hat{P}$, applying similarity transform $\hat{P}A\hat{P}$ does not work because the zeroes obtained during the first multiplication are removed due to the second orthogonal transform. However, $\hat{P}^T A\hat{P}$ does work because of the implied inverse in the transpose of an orthogonal matrix. The next step is to remove the extra sub diagonals of a Hessenberg matrix to make it triangular. However the focus of this project is Step 1, namely the reduction of a general matrix to its Hessenberg form.

2.2. **Literature Review.** Various implementation notes from LAPACK were reviewed to find good implementations of the Hessenberg reduction through Householder Reflectors, Givens Rotators and Block Householder Reflectors. The column Householder reflector and the Givens rotation algorithms were picked from "Parallel Block Hessenberg Reduction using Algorithms-By-Tiles for Multicore Architectures Revisited LAPACK Working Note #208" [1]. The block Hessenberg Reduction method was picked from [2]. Background information was obtained from "Fundamentals of Matrix Computations" by David Watkins.

3. Algorithm

The algorithm to reduce a General matrix to upper Hessenberg form is simple. In the simplest implementation, assuming a matrix mxn, each column is taken and the lower n-i elements are zeroed out, where i=2....m for each column.

Assume a starting matrix

$$A = \begin{bmatrix} a_{11} & \dots & \dots & \dots & \dots \\ \vdots & * & * & * & * \\ \vdots & * & * & * & * \\ \vdots & * & * & * & * \\ \vdots & * & * & * & * \end{bmatrix}$$

Let $\hat{Q}_1$ be a reflector matrix such that $\hat{Q}_1 b = [-\tau_1, 0, \dots, 0]^T, (|\tau_1| = \|b\|_2)$ That is $Q_1 = \begin{bmatrix} 1 & 0^T \\ 0 & \hat{Q}_1 \end{bmatrix}$

Now performing the transformation $A_{1/2} = Q_1 A = \left[\begin{array}{c|c} a_{11} & c^T \\ \hline -\tau_1 & \\ 0 & \hat{Q}_1 \hat{A} \\ \vdots & \\ 0 & \end{array}\right]$

it is seen that there are zeros in the required rows in the first column.

Now applying $A_1 = A_{1/2} Q_1 = \left[\begin{array}{c|c} a_{11} & c^T \hat{Q}_1 \\ \hline -\tau_1 & \\ 0 & \hat{Q}_1 \hat{A} \hat{Q}_1 \\ \vdots & \\ 0 & \end{array}\right] = \left[\begin{array}{c|c} a_{11} & * \ \dots \ \dots \ * \\ \hline -\tau_1 & \\ 0 & \hat{A}_1 \\ \vdots & \\ 0 & \end{array}\right]$

Because of the structure of Q_1, the zeros in the first column are preserved. The second step is to now create reflector $\hat{Q}_2$ for A_1 instead of A and so on. Note that we can replace reflector $\hat{Q}$ with a rotator $\hat{R}$ instead. The objective is to zero out elements in each column to reduce it to Hessenberg form without losing the Eigen values in them. This is possible through orthogonal transforms like reflections and rotations via similarity transforms.

3.1. **Householder Reflections.** Shown in Algorithm 1 is the algorithm for the householder reflector vector generation. It takes a matrix A and returns the householder vector v operating on its first column.

Algorithm 1 *house*(n)

Require: A matrix A.
Ensure: The householder reflector vector $\hat{V}$.

1: $N \leftarrow Size(A)$
2: Generate $e1 \leftarrow [1, zeros(nrows - 1, ncols)]$
3: $v \leftarrow sign(A(1)) * \|A\|_2 * e1 + A$
4: $v \leftarrow v/\|V\|_2$

3.2. **Givens Rotations.** Shown in Algorithm 2 is the algorithm for the Givens rotator matrix generator. It takes a matrix A and row to be zeroed j and row affected i and returns a Givens rotator matrix which zeros out the element in the 1st column of the j^{th} row, modifying only the i^{th} row and leaving other rows intact.

Algorithm 2 *givens*(X, j, i)

Require: A matrix A, row j to be zeroed, row i affected by the rotation
Ensure: The givens rotator matrix R that zeros out element $(j, 1)$ of A and modifies only row $(i, :)$.

$G \leftarrow I_x$
$x_i \leftarrow x(i, 1)$
$x_j \leftarrow x(j, 1)$
$r \leftarrow \sqrt{x_i^2 + x_j^2}$
$\cos\theta \leftarrow x_i/r$
$\sin\theta \leftarrow x_j/r$
$G(i, i) \leftarrow G(j, j) \leftarrow \cos\theta$
$G(i, j) \leftarrow -\sin\theta$
$G(j, i) \leftarrow \sin\theta$

3.3. **Column Householder Reduction.** Shown in Algorithm 3 is the algorithm to reduce a General matrix to upper Hessenberg form using Column Householder reduction. It takes a matrix A and returns matrix in upper Hessenberg form.

Algorithm 3 *hessen*(A)

Require: A matrix A
Ensure: The upper Hessenberg form R of matrix A.

for $j = 1$ to $n - 2$ **do**
    $x \leftarrow A_{j+1:n,j}$
    $v_j \leftarrow sign(x_1)\|x\|_2 e1 + x$
    $v_j \leftarrow v_j/\|v_j\|_2$
    $A_{j+1:n,j:n} \leftarrow A_{j+1:n,j:n} - 2v_j(v_j^* A_{j+1:n,j:n})$
    $A_{1:n,j+1:n} \leftarrow A_{1:n,j+1:n} - 2(A_{j+1:n,j:n}v_j)v_j^*$
end for

3.4. **Givens Hessenberg Reduction.** Shown in Algorithm 4 is the algorithm to reduce a General matrix to upper Hessenberg form using Givens rotations. It takes a matrix A and returns matrix in upper Hessenberg form.

Algorithm 4 *hessen*1(A)

Require: A matrix A
Ensure: The upper Hessenberg form R of matrix A.

$G \leftarrow Id_n$
for $j = 1, 2$ to $n - 2$ **do**
    for $i = n, n - 1$ to $j + 2$ **do**
        Build the local $g(i - 1, i)$ such that $A(i, j) = 0$
        Accumulate $G \leftarrow g(i - 1, i) * G$
        Update $A \leftarrow g^T(i - 1, i) * A$
        Update $A \leftarrow A * g(i, i - 1)$
    end for
end for

3.5. **Block Hessenberg Reduction.** Shown in Algorithm **??** is the algorithm to reduce a General matrix to upper Hessenberg form using Block Householder reflectors. It takes a matrix A and returns matrix in upper Hessenberg form. The algorithm works by splitting the matrix into a set of columns(panels) and performing column householder reductions on each of them. Each panel's householder vectors are accumulated into a matrix and a deferred updating takes place using Level 3 operations between the panels.

4. Code

4.1. **Householder Reflections.** Here's the code for the house() function that just gives the householder reflector vectors given a matrix.

Listing 1. `house.m`

```
function [v] = house(x)
%Solving Householder matrix H
[nrows, ncols]=size(x);
e1=[1;zeros(nrows-1, ncols)];
v=sign(x(1))*norm(x)*e1 + x;
v=v/norm(v);
end
```

4.2. **Givens Rotations.** Here's the code for the givens() function that generates the rotator matrix given the input matrix and row whose first column is to be zeroed out.

Listing 2. `givens.m`

```
function [g]=givens(x,j,i)
% Function of Givens Rotation

% x: Input matrix
```

```
% i: Row affected by the zeroing operation
% j: Row to be zeroed (column 1)
% G: Givens rotation matrix
g=eye(length(x)); %Initialize givens matrix
xi=x(i,1); %Identify the ordinate pair over which the rotation happens
xj=x(j,1);
r=sqrt(xi^2+xj^2); %Find length of vector r from origin to this point
%Populate rotation matrix with the necessary elements
cost=xi/r;
sint=xj/r;
g(i,i)=cost;
g(i,j)=-sint;
g(j,i)=sint;
g(j,j)=cost;
end
```

4.3. **Column Hessenberg Reduction.** Here's code for Hessenberg Reduction using Column Householder

LISTING 3. `hessen.m`

```
function [R]=hessen(A)
% Hessenberg Reduction by using Householder Method
n=size(A,1);
R=A;
for j=1:n-2
    x=R(j+1:n, j);
    vj=house(x);
    R(j+1:n, j:n) = R(j+1:n, j:n) - 2 * vj * (vj'*R(j+1:n, j:n));
    R(1:n, j+1:n) = R(1:n, j+1:n) - 2 * (R(1:n, j+1:n) * vj) * vj';
end
```

4.4. **Givens Hessenberg Reduction.** Here's code for Hessenberg Reduction using a Givens rotator.

LISTING 4. `hessen1.m`

```
function [R]=hessen1(A)
% Hessenberg Reduction by using Givens Method
n=size(A);
G=eye(size(A)); %Gives rotation matrix accumulator
R=A; %Copy A into R
for j=1:n-2 %Outer loop (determines columns being zeroed out)
    for i=n:-1:j+2 %Inner loop (successively zeroes jth column)
        giv=givens(R(j:n, j:n), i-j+1, i-j);
        giv=blkdiag(eye(j-1), giv); %Resize rotator to full size
        G=giv*G; %Accumulate G which give a Q in the end
        %Perform similarity transform
```

```
12          R=giv'*R;
13          R=R*giv;
14      end
15      figure; matrixplot(R);
16  end
17  end
```

4.5. **Block Hessenberg Reduction.** Here's the code for Block Hessenberg Reduction

LISTING 5. BlockHess.m

```
1  function [Q R] = BlockHess (A, r)
2  % Block Hessenberg Reduction
3  % p — number of blocks
4  % r — number of extra diagonals
5  % pr x r block structure
6  [numRows, numCols]  = size(A)
7  numParts = max(2, floor(numCols/r));
8
9  a = floor(numCols/numParts);
10 partition = ones(1, numParts)*a;
11
12 % Split matrix rows into partition, storing result in a cell array
13 R = mat2cell(A, numRows, partition)
14 Q=eye(numRows);
15 V=eye(numRows);
16 for k = 1:numParts
17     for j=1:r
18         s=j+(k-1)*r;
19         v=house(R{k}(s:numRows, j))
20         V(s:numRows,j)=v;
21         R{k}(s:numRows, j:r) = R{k}(s:numRows, ...
               j:r)-2*v*v'*R{k}(s:numRows, j:r);
22     end
23     Y=V(:,1);
24     W=-2*V(:,1);
25     for j=2:r
26         z=-2*(eye(length(W))+W*Y')*V(:,j);
27         W=[W z];
28         Y=[Y V(:,j)]
29     end
30     if k<numParts
31         R_tmp=horzcat(R{k+1:numParts});
32         R_tmp=(eye(length(Y))+Y*W')*R_tmp;
33         R_tmp=horzcat(R{1:k}, R_tmp);
34         R = mat2cell(R_tmp, numRows, partition);
35         Q=Q*(eye(length(Y))+W*Y');
36     end
37 end
```

```
38
39 R=cell2mat(R);
40 end
```

4.6. **Testing.** Here's the code for complete test function

LISTING 6. hr_test.m

```
1  clc
2  clear
3  format long; % increase precision
4  n=64;%Set matrix size 8
5  A=-1+2.0*rand(n,n);
6  R=hess(A); %Perform the built-in hessenberg reduction (reference)
7  R1=hessen(A); %Perform the hessenberg reduction (column householder)
8  R2=hessen1(A); %Perform the hessenberg reduction (givens rotation)
9  figure; matrixplot(R1);
10 figure; matrixplot(R2);
11 %Test definition of Hessenberg
12 norm1=norm((R-hess(A))/R) %Just for sanity, built-in is assumed to work
13 norm2=norm((R-R1)/R)
14 norm3=norm((R-R2)/R)
15 %Test eigen values
16 orig_eig=eig(A);
17 eval=eig(R); %E-val value of built-in hessenberg reduced matrix
18 eval1=eig(R1); %E-val of hessenberg reduced matrix(column householder)
19 eval2=eig(R2); %E-val of hessenberg reduced matrix(givens rotation)
20 eig_norm=norm((eval-eig(R))/eig(R))
21 eig_norm1=norm((eval1-eval)/eval)
22 eig_norm2=norm((eval2-eval)/eval)
23 tElapsed=zeros(1,50); %Create a time elapsed vector
24 tElapsed1=zeros(1,50); %Create a time elapsed vector
25 tElapsed2=zeros(1,50); %Create a time elapsed vector
26 tElapsed3=zeros(1,50); %Create a time elapsed vector
27 x=logspace(1, 2, 50); %create logspace for the x-axis
28 for i=1:50 %loop through the order of matrix
29     n=round(x(i)); %round the logspace to get the closest integer order
30     %Initialize coefficient matrix G with a random value
31     G=-1+2.0*rand(n,n); % Random matrix of order n between -1 and 1
32     tStart = tic; %Start the timer
33     result=hess(G); %Use the built-in hess function
34     tElapsed(i) = toc(tStart); %stop the timer and capture elapsed time
35     tStart = tic; %Start the timer
36     result=hessen(G); %Use the column householder function
37     tElapsed1(i) = toc(tStart); %stop the timer and capture elapsed time
38     tStart = tic; %Start the timer
39     result=hessen1(G);
40     tElapsed2(i) = toc(tStart); %stop the timer and capture elapsed time
41
```

```
42 end
43 figure; loglog(x, tElapsed, 'r-', x, tElapsed1, 'g-', x, tElapsed2, ...
       'b-'); %Plot the time vs time elapsed
44 grid on;
45 xlabel('Matrix size');
46 ylabel('Time of computation');
47 legend('Built-in', 'HR Column', 'Givens HR');
```

5. Results

This section lists the results obtained from the experiments done so far on various reduction methods. The test code first compares the norm of the reduced matrix with that of the built-in hess command, then compares the norm of the eigen values obtained through each of these methods to the eigen values of the original matrix. A timing is also performed to benchmark each of these functions to each other and to the built-in hess command.

5.1. **Hessenberg Reduction using Householder Reflectors.** The output of Hessenberg reduction using Householder Reflectors is shown below. Relative Norm with default

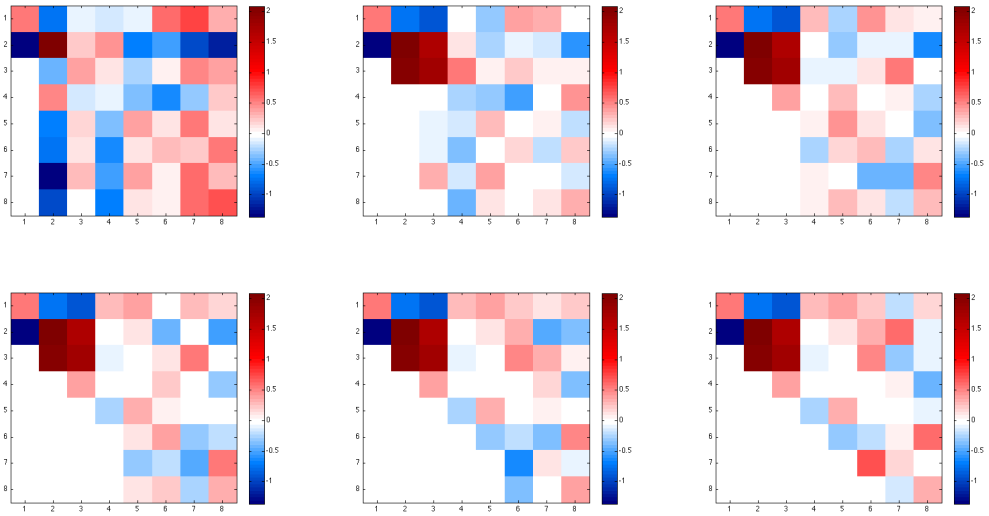

Figure 1. Matrix Plots of Hessenberg reduction using Householder reflectors for each iteration N=8

hess(): 6.743127020622670e-12 Relative Norm of eigen values compared to original matrix's eigenvalues: 2.179818782631296e-13

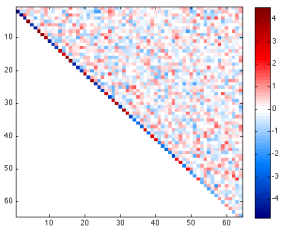

FIGURE 2. Matrix Plot results of a Hessenberg reduced matrix using householder reflectors for N=64

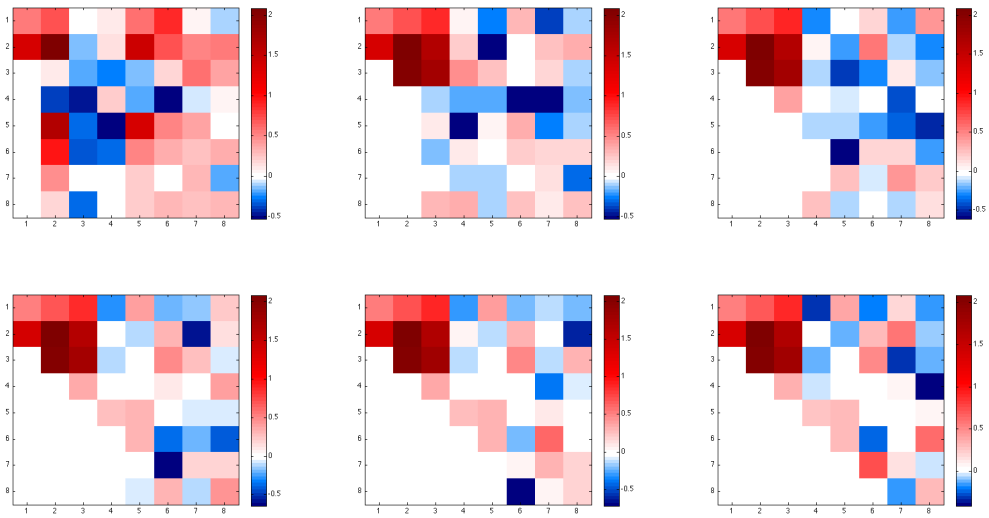

FIGURE 3. Matrix Plots of Hessenberg reduction using Householder reflectors for each iteration N=8

5.2. **Hessenberg Reduction using Givens Rotation.** The output of Hessenberg reduction using Householder Reflectors is shown below. Relative Norm with default hess(): 2.195456923554638e+02 Relative Norm of eigen values compared to original matrix's eigenvalues: 3.991088526709233e-13 Note that the relative norm with hess() is not right for Givens. This is acceptable because by observing the matrix visually, it is seen that the values are identical but several elements of the rotated matrix have inverted signs when compared to the reflected matrices. Since the eigen values do match the original, this is not a concern as there is no requirement that the reduced matrix should be identical through all methods.

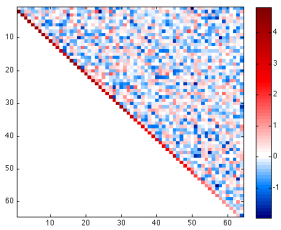

FIGURE 4. Matrix Plot results of a Hessenberg reduced matrix using givens rotators

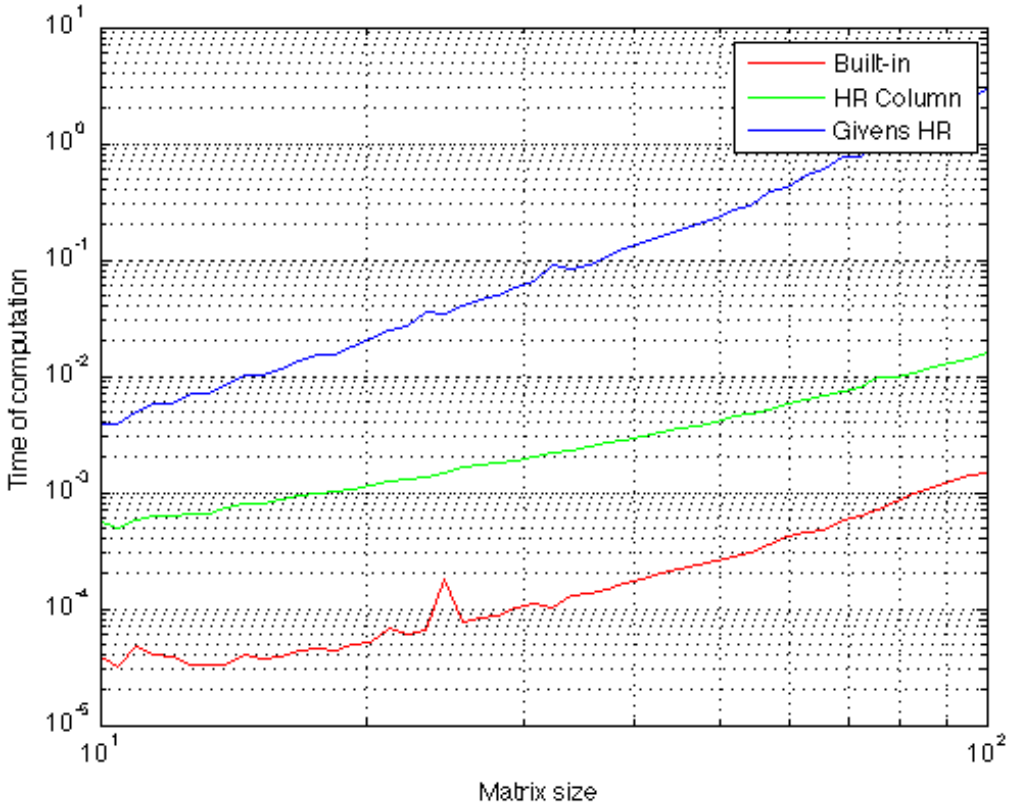

FIGURE 5. A log-log plot of timing of various Hessenberg Reduction functions (column householder, givens rotation and default hess())

5.3. **Timing.** This is the graphic of a log-log plot obtained while benchmarking each Hessenberg reduction variant varying the size of the input matrix.

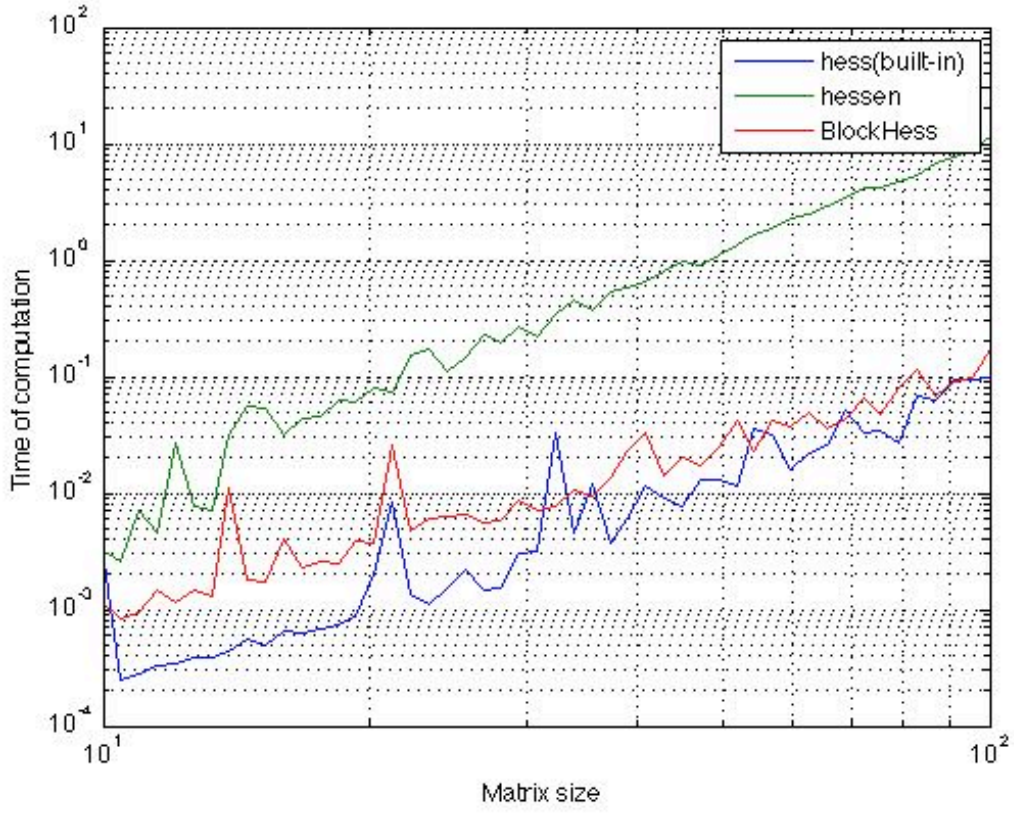

FIGURE 6. A log-log plot of timing of various Hessenberg Reduction functions (column householder, givens rotation and default hess())

6. Future Work

LAPACK suggests improvements to the Hessenberg reduction through several blocking methods, one with fast givens rotations and GPU variants of these. It would be quite interesting to benchmark the GPU variant with the CPU variant for various sizes of N (matrix size) to determine the break-in point at which the GPU starts to outperform the CPU. Lastly, parallelized algorithms for Hessenberg reduction would help to understand hardware implementations of the algorithm for possible implementation in an ASIC/FPGA.

7. References

1. Parallel Block Hessenberg Reduction using Algorithms-By-Tiles for Multicore Architectures Revisited LAPACK Working Note #208" 2. "Fundamentals of Matrix Computations" by David Watkins.

8. Appendix

8.1. **Matrix Plot code.** The following is the matrix plot code by Nathan Childress from matlabcentral. This was used to plot the matrices of various Hessenberg reduction forms.

Listing 7. `matrixplot.m`

```
function matrixplot(n)
%Colormap code reused from code for bluewhitered by Nathan Childress
%from matlabcentral
imagesc(n);
m = size(get(gcf,'colormap'),1);
bottom = [0 0 0.5];
botmiddle = [0 0.5 1];
middle = [1 1 1];
topmiddle = [1 0 0];
top = [0.5 0 0];

% Find middle
lims = get(gca, 'CLim');

% Find ratio of negative to positive
if (lims(1) < 0) & (lims(2) > 0)
    % It has both negative and positive
    % Find ratio of negative to positive
    ratio = abs(lims(1)) / (abs(lims(1)) + lims(2));
    neglen = round(m*ratio);
    poslen = m - neglen;

    % Just negative
    new = [bottom; botmiddle; middle];
    len = length(new);
    oldsteps = linspace(0, 1, len);
    newsteps = linspace(0, 1, neglen);
    newmap1 = zeros(neglen, 3);

    for i=1:3
        % Interpolate over RGB spaces of colormap
        newmap1(:,i) = min(max(interp1(oldsteps, new(:,i), newsteps)', ...
            0), 1);
    end

    % Just positive
    new = [middle; topmiddle; top];
    len = length(new);
    oldsteps = linspace(0, 1, len);
    newsteps = linspace(0, 1, poslen);
    newmap = zeros(poslen, 3);

    for i=1:3
```

```
43          % Interpolate over RGB spaces of colormap
44          newmap(:,i) = min(max(interp1(oldsteps, new(:,i), newsteps)', ...
                0), 1);
45      end
46
47      % And put 'em together
48      newmap = [newmap1; newmap];
49
50  elseif lims(1) ≥ 0
51      % Just positive
52      new = [middle; topmiddle; top];
53      len = length(new);
54      oldsteps = linspace(0, 1, len);
55      newsteps = linspace(0, 1, m);
56      newmap = zeros(m, 3);
57
58      for i=1:3
59          % Interpolate over RGB spaces of colormap
60          newmap(:,i) = min(max(interp1(oldsteps, new(:,i), newsteps)', ...
                0), 1);
61      end
62
63  else
64      % Just negative
65      new = [bottom; botmiddle; middle];
66      len = length(new);
67      oldsteps = linspace(0, 1, len);
68      newsteps = linspace(0, 1, m);
69      newmap = zeros(m, 3);
70
71      for i=1:3
72          % Interpolate over RGB spaces of colormap
73          newmap(:,i) = min(max(interp1(oldsteps, new(:,i), newsteps)', ...
                0), 1);
74      end
75
76  end
77
78  colormap(newmap);
79  colorbar;
80  end
```

8.2. **Performance Improvement Criteria.** To improve the performance of numerical linear algebra computations on a computing environment, a few factors need to be taken into account. Knowing there factors would help mathematicians and engineers get a better appreciation of proper algorithmic design and using techniques to improve performance of computing manyfold. Traditional computing architectures had a CPU (central processing unit) whose role was to fetch instructions and data sequentially and execute them one by one. This slowly evolved into SIMD (Single Instruction Multiple Data). In other words,

vectorized computations on the processor. Modern computing environments have multiple cores in the CPU and also use the GPU (Graphics Processing Unit) to perform massively parallel floating point computations.

All computing environments have a fast, small intermediate storage space called Cache whose only purpose is to fetch and store a chunk of data from primary/secondary memory. Since the cache is an extremely fast component, the fetch itself could be slow, but once fetched any data within the chunk can be accessed extremely fast. Cache locality implies that keeping data close together in memory would help the cache bring the entire block in one go from the primary memory. However, since the size of the cache is very limited, a complete matrix / vector may not be able to fit into the cache. The general rule is to improve cache locality as much as possible by storing consecutive elements as close to each other as possible.

Another factor to consider is the memory bandwidth. Higher the memory bandwidth, faster the flow of data into and out of the CPU and faster the computations. Since there is a finite limit to this bandwidth, this barrier is usually broken by using faster memories, faster bus etc.

Optimizing the number of calculations to be performed on a given set of data also improves performance if pipeline bubbles can be avoided by grouping instructions together, predictive branching in pipelines etc. The size of data also matters and bigger the data to be fed to the processor, the longer it takes, adding to the cache flushing every time a new chunk of data is brought in. And of course, in desktops, the pre-emptive multi tasking operating systems steal cpu cycles as well in-between computations.

In the case of a GPU, the algorithms can be parallelized to utilize the massively parallel computing elements it offers. A GPU has a large initial latency overhead but this is compensated for when the computation starts if the problem is sufficiently large.

The general rules when it comes to cache locality is that Vector-Vector (Level 1) operations are slower than Vector-Matrix (Level 2) operations which are slower than Matrix-Matrix (Level 3) operations. So blocking (grouping vectors to matrices) whenever possible increases the performance of the system.