RefinEig:
A Program to refine Eigensystems

A Final Project by Sunil Ramesh
For Prof. Kahan

Table of Contents

1. Introduction

One of the most interesting problems in Numerical Mathematics is that of finding the eigenvalues and eigenvectors of a matrix. Even more challenging is the generalized Eigenproblem which deals with finding the Eigensystem of a nonsymmetric matrix. Often, the results of the computation are far less accurate than can actually be achieved if simulated high-precision arithmetic had been used. RefinEig is a program that will attempt to clean up the results of an Eigensystem calculation using Iterative Refinement as if the whole computation had been done using more significant bits of precision. RefinEig is designed to best refine the most common cause of problems: an isolated pair of nearly coincident eigenvalues.

2. The need for more than eig

The function eig in Matlab is considered good software for computing eigensystems. Ideally, given a matrix B, eig is supposed to calculate a nonsingular matrix Q whose columns are the eigenvectors of the given square matrix B and a diagonal matrix V of its eigenvalues. These constitute the solution to the diagonalization problem. So, eig should find Q and V such that

$B = Q * V * Q-1$.

However, because of rounding errors, the eigensystem produced is that of a matrix close to B, namely B - dB, so:

$B - \Delta B = Q * V * Q-1$. The matrix ΔB is comparable to the roundoff in B.

Even when ΔB is very small, Q and V might be calculated very inaccurately because of the algorithms used by eig.

Consider the case of Defective Matrices. (Defective Matrices are those that have too few eigenvectors to permit diagonalization by any similarity transformation) James Demmel showed in 1988 that Defective matrices are almost never encountered at random, but it is possible to get matrices that are near-defective more often.
RefinEig is designed to improve computed values of an eigensystem when the matrix is nearly defective.

3. RefinEig : A first look

RefinEig starts with a nonsingular matrix Q of approximate eigenvectors and a diagonal matrix V of approximate eigenvalues. In MATLAB, we would be able to get Q and V from the following assignment:

```
[Q, V] = eig(B)
```

Here, B is the matrix for which we are trying to compute the eigensystem. Now we are ready to invoke RefinEig

```
[Q, V] = RefinEig(Q, V, B)
```

RefinEig performs one step of iterative refinement to replace the old Q and V with new values. The step can be repeated manually if desired: RefinEig converges very quickly.

4. Computation of Residuals

The effectiveness of Iterative Refinement is generally limited by the accuracy to which residuals like $\Delta R := B*Q - Q*V$ can be computed. If Q and V were exactly computed, then ΔR should become 0, so in order for that to happen, a massive cancellation must be incurred. Accuracy, however is not limited by cancellation, but rather by the precision to which scalar products are accumulated during matrix multiplications.

Today's computers tend to have doubles that are 64-bits wide. They have 53 bits of precision in which B, Q and V are stored in memory. In this case, the computed residual dR does not generally rise much higher than the roundoff accumulated during its computation. In these cases, RefinEig can improve Q and V greatly. These cases occur often enough to justify the existence as will be shown in the discussion about Werner Frank's matrices. RefinEig is most useful, however, when the bits of dR are accumulated with extra precision, so it can be well above the roundoff error accumulated.

Most popular processors today (Please see Appendix A for an update), the floating point arithmetic is performed in registers that are 10 bytes wide. Thus, access is available to 64 significant bits instead of the normal 53. These wide registers are used regardless of the widths of narrower operands loaded from memory. All subexpressions in MATLAB are stored in 8-byte memory cells except that MATLAB accumulates scalar products in 10-byte registers during matrix multiplications because doing so is faster than storing the expression in memory.

Q. Does my computer have extra-precision accumulation ?

*A. Try [(2 - 2^33), 2^33, -1] * [(1 + 2^32) ; 2^32 ; 1] ;
on MATLAB. If the answer is +1, then you have extra-precision accumulation. If it is -1, then you don't. As an alternative, try running mxmuleps.m from
Appendix B on your MATLAB.*

In order to make use of the extra-precision, RefinEig computes its residuals by performing a MATLAB matrix multiplication. ΔR, for example is computed in the following way:

$\Delta R := B*Q - Q*V$ is obtained from $\Delta R := [B, Q] * [Q; -V]$;

The residual:

$\Delta^2 R := Q*G - \Delta R$ is obtained from $\Delta^2 R = [Q, dR]*[G;-I]$;
for an identity matrix of appropriate size.

On machines that do not have extra-precise registers, the matrix multiplication would produce the same result as if the computation was done without using the matrix multiplication.

5. How RefinEig works for Non-Hermitian B

Assume that we are staring off with a diagonalizable non-Hermitian matrix B and an approximation [Q, V] to its eigensystem as given to us by invoking eig. Also assume that the residual dR := B*Q - Q*V is small and that Q is not too ill-conditioned.
Here V will represent a diagonal matrix of eigenvalues.
Our aim is to compute a small correction ΔQ and a small diagonal correction dV such that
$B * (Q + \Delta Q) = (Q + \Delta Q) * (V + \Delta V)$ exactly.

However, by this equation, ΔQ is not determined uniquely because we can postmultiply both sides of the equation by any diagonal matrix and still have equality. For more information on this, please refer to Appendix C.

So, our first task is to introduce a restriction on ΔQ to make it unique.

Let $\Delta Z := Q^{-1}\Delta Q$.
We will now choose $\Delta Q = Q * \Delta Z$ by imposing the condition that $\mathrm{diag}(\Delta Z) = o$.

Now we can define:
$\Delta C := Q^{-1}\, \Delta R = Q^{-1}\, B\, Q\ -\ V$

It is necessary to be able to calculate ΔC as accurately as possible in order to protect it from accuracies caused when eigenvalues and eigenvectors' elements span a very wide range of magnitudes.

It is possible to do iterative refinenement by the following equation:
$\Delta C_{new} := \Delta C\ -\ Q\text{-}1\ (Q * \Delta C - \Delta R)$

Here, $Q * \Delta C - \Delta R$ is the residual from the defining equation $\Delta C := Q\ \Delta C - \Delta R = 0$

After the iterative refinement process, we now have a good estimate of ΔC.

Let us go back to our original equation that we are trying to solve:
$B * (Q + \Delta Q) = (Q + \Delta Q) * (V + \Delta V)$ which we are trying to solve for ΔQ and diagonal ΔV.

Consider the substitutions:
$B = Q\,(V+C)\,Q^{-1}$. and $\Delta Q = Q\, \Delta Z$

We can now transform the equation into a new one:
$(V + \Delta C) * (I + \Delta Z) = (I + \Delta Z) * (V + \Delta V)$

We now expand to get:
$V + \Delta C + V*\Delta Z + \Delta C * \Delta Z = V + \Delta V + \Delta Z * V + \Delta Z * \Delta V$

We now proceed to take the diagonal entries of both sides. The left-hand-sides's diagonal is 0 because V and ΔV are diagonal, and $\mathrm{diag}(\Delta Z) = 0$.

$\Delta V = \mathrm{Diag}(\Delta C) + O(\Delta)^2$

We can also recognize here that

$\Delta Z * (V + \Delta V) - V * \Delta Z = \Delta C * (I + \Delta Z) - \Delta V$

By performing only half the cancellations from last time.

We are now ready to describe the process for finding ΔZ and ΔV.

Step 1. We first proceed to solve ΔV using:
$\Delta V = \mathrm{Diag}(\Delta C) + O(\Delta)^2$

Step 2. Solve for ΔZ. Solving for ΔZ is trickier and involves what is known as the **Sylvester equation**. Please see Appendix E for more information on how this equation is applied for the following solution.

<u>Finding ΔZ</u>

Let u be the column vector whose elements are all 1. Then condense the diagonal matrices V and ΔV into column vectors $v = \mathrm{diag}(V)$ and $\Delta v = \mathrm{diag}(\Delta V)$. Now proceed to compute
$E := (u * v^T - v * u^T) + (u * \Delta v^T - \Delta V) + I$

Now proceed to calculate **ELEMENTWISE**
$\Delta Z = ((\Delta C + \Delta C*\Delta Z) - \Delta V) / E = (\Delta C - \Delta V) / E + O(\Delta)^2$.

Notice that the diagonal elements of ΔZ will be 0 automatically because of the method in which ΔV is computed.

The above-described process might best be described as a **process of relaxation**. Relaxation is the name given to a process in which two values are sought and are calculated one after the other, and the process repeated till desired levels of accuracy are obtained.

In keeping with relaxation, the above steps 1 and 2 can be repeated iteratively to refine guesses of ΔZ and ΔV.

As a starting point, $\Delta Z := 0$ can be chosen. After 1 pass through these steps, we will obtain a new ΔZ in error my terms of order O(d)^2. After a second pass, the error in ΔZ will be $O(\Delta)^3$. After ΔZ is refined to a level of suitable accuracy:
$\Delta Q = Q^*\Delta Z$ can by computed.

Once we have a suitable ΔQ, $Q+\Delta Q$ and $V+\Delta V$ can be computed.

Unfortunately, the above process actually fails to converge in situations close to those that cause eig(B) to fail in the first place. The most likely failure situation arises, as mentioned above, from pairs of nearly coincident eigenvalues. So, RefinEig computes a starting ΔZ to use in place of $\Delta Z =0$. It calculates this from a formula that would be perfect if ΔC were a permuted diagonal sum of 1x1 and 2x2 matrices, and is otherwise still correct in the first order terms.

Recall that we want to solve:
$(V + \Delta C) * (I + \Delta Z) = (I + \Delta Z) * (V + \Delta V)$ for ΔZ and diagonal ΔV with the constraint that all the diagonal entries of ΔZ are zero.

The following recipe can be followed to come up with an initial guess for ΔZ.

We will again make use of the Column vector u of all 1's.
We will define $\Delta v := \text{diag}(\Delta V)$ and $v = \text{diag}(V)$.

First, we will construct $\Delta C2 = \Delta C - \text{Diag}(\Delta v)$ with zeros on its diagonal.

Next the matrix complex skew matrix $S := ((u*v^T - v*u^T) + (u*\Delta v^T - \Delta v*u^T)) / 2$.

Then $T := \text{sqrt}(S * S + \Delta C2^T * \Delta C)$ **ELEMENTWISE**

Now, we will proceed to compute skew $K := \text{Re} \{ S * \text{conj}(T) \}$ **ELEMENTWISE**

(conj refers to the complex conjugate)

During this process, we must reverse the sign of every element of T for which the corresponding element of K is either negative or subdiagonal and zero.

Lastly, we will compute $H := S + Y$ and then $\Delta Z := \Delta C2 / (H + I)$ **ELEMENTWISE**

The above recipe can be used even if ΔC is an arbitrary matrix and not a permuted diagonal sum of 1x1 and 2x2 matrices. In that case, then, the recipe produces approximations $\Delta V = \text{Diag}(\Delta v)$ and ΔZ that satisfy the original we were solving to within errors of order $O(\Delta)^2$. Therefore, RefinEig is able to reduce the errors after just pass of the relaxation process. to just $O(\Delta)^3$.

The first-guess procedure is what distinguishes RefinEig from previous attempts at iterative refinement. RefinEig still works well when clustered pairs of eigenvalues are close to each other but far away from other such pairs of clustered eigenvalues. This situation is what causes other iterative refinement schemes to fail.

There is an important point to note. If the original matrix Q is very close to a singular matrix, or if $V+\Delta V$ has too many values that are close to each other, then RefinEig will fail. It is for this reason, that it is useul to check the value of the residual $\Delta R := B*Q - Q*V$ after every run through to make sure that RefinEig did not cause matters to get worse. If the residual is smaller but still too large, RefinEig can be invoked again.

6. How RefinEig works for Hermitian A

A Notational Convention that uses symmetric letters A, H, I, M, O, T, U, V, W, X and Y to represent Hermitian matrices to distinguish them for the non-Hertmitian matrices will be used. It is for this reason that the input for the non-Hermitian was called B and now will be renamed to A. O, S, and Z will represent skew-Hermitian matrices.

There is a huge difference between the Hermitian and the Non-Hermitian case of the Eigenproblem. We expect eig to produce an orthogonal (unitary) eigenvector matrix Q. So it must be true that Q'Q + I. However, roundoff corrupts this equation. Using the RefinEig algorithm for the Non-Hermitian Q + ΔQ will also not satisfy the equation. The equation will fail even when the final normalization of Q -> Q * sqrt(Diag(Q'Q))-1. Even if the formulas push every column of Q closer to an eigenvector, they pull Q farther from unitary if some eigenvalues of A are nearly the same. It is for this reason that RefinEig must treat Hermitian matrices differently. In addition, Hermitian matrices cannot be defective, so RefinEig cannot fail.

Let us suppose that an eigenvector matrix Q and a real diagonal eigenvalue matrix V have been found approximately for a Hermitian matrix A (= A'). In MATLAB, it we could find it using [Q, V] = eig(A)

MATLAB will ensure that Q is unitary. However, since RefinEig does not assume that eig is being used, it must replace Q by the nearest unitary matrix
P := Q * sqrt((Q'Q)-1)
A singular value decomposition can be used for this purpose.

There is, however, a quicker way to calculate P, if the residual ΔY := Q'Q - I, computed making use of high-precision residuals in MATLAB (from ΔY := [Q', I] * [Q;-I]), is small enough. If that is the case, then P := Q - Q*ΔY / 2 to working accuracy, and ΔY is small enough when 1 - |ΔY|^2 rounds to 1. So RefinEig is able to obtain an accurate P := Q- dQ.

The next task is to refine P towards the eigenvectors of A. However, this refinement is different from the non-Hermitian case in that it must preserve the orthogonality of the columns of P. So it is has to be put into the form:

P -> P - 2 * P * (I+ΔZ)-1 * ΔZ. Here, ΔZ = -ΔZ' is a skew-Hermitian matrix. The Cayley Transform of ΔZ is

I - 2*(I+ΔZ)-1*ΔZ = (I + ΔZ)-1(I - ΔZ) is obviously unitary. This cannot have an eigenvalue of -1, however.

In this equation, because postmultiplication by any Unitary Diagonal matrix is allowed, just like the non-Hermitian case (where any diagonal matrix would have done), we will insist again that diag(ΔZ) = o.

Please refer to Appendix A of Prof Kahan's paper [1] for a proof of whether this constraint is always satisfiable.

Now, define a residual:
ΔH := P' * (A*P - P*V) = P' * A * P - V

This is computed from P' * ([A, P], * [P; -V]) in MATLAB, would be Hermitian if there were no roundoff errors. However, we can replace this residual with another:
ΔH <- (ΔH + ΔH') / 2. Now the residual is also Hermitian.

Then (I + ΔZ)-1(I - ΔZ) turns out to have to be a matrix of eigenvectors for V + ΔH. From here, by the diagonalization problem, it follows that :

(V + ΔH) * ((I + ΔZ)-1 * (I - ΔZ)) = ((I + ΔZ)-1 * (I - ΔZ)) * (V + ΔV) where ΔV is also diagonal like V is.

The above equation simplifies to:

ΔV - ΔZ ΔV ΔZ - ΔZ (2V - ΔV) + (2V + ΔV) ΔZ = ΔH + ΔZΔH - ΔHΔZ - ΔZΔHΔZ.

This equation now has to be solved, just like the non-Hermitian case, for a real diagonal ΔV and a skew-Hermitian ΔZ.

Let us consider the ΔV just like we did for the non-Hermitian case.
Remembering that diag(ΔZ) = o and defining |ΔZ|^2 := -ΔZ' * ΔZ **(ELEMENTWISE)**

We can now form an estimate of
$\Delta v := (I + |\Delta Z|^2)^{-1}$ * diag(ΔH + ΔZΔH - ΔHΔZ - ΔZΔHΔZ) = diag(ΔH) + $O(\Delta)^2$.
Here Δv must be exactly real.

Now we have found ΔV := Diag(Δv)

We will now use our column vector u of all 1's. The vectors v = diag(V) and Δv = diag(ΔV) will also be needed again.

We will construct a skew matrix S := $(2v + \Delta v)\, u^T - u\, (2v + \Delta v)^T = -S^T$

Just like the non-Hermitian case, we will produce an estimate to dZ

$\Delta Z := (\Delta H - \Delta V + \Delta Z\Delta H - \Delta H\Delta Z - \Delta Z(\Delta H - \Delta V)\, \Delta Z) / (S + iI) = (\Delta H - \Delta V) / (S + iI) + O(\Delta)^2$

ELEMENTWISE

Now we will automatically have diag(Δz) = o. Note here that i = sqrt(-1)

We add iI to prevent division by zero on the diagonal. If, for some reason, they occur elsewhere, we can replace any resulting <infinity> or NaN in ΔZ by 0.
This will limit the damage done to those eigenvectors of A belonging to eigenvalues that are almost the same.

Now, these formulas can be used to refine ΔV and ΔZ in a relaxation process quite like the non-Hermitian case. Starting with $\Delta Z = 0$ on the right-hand-side, they will quickly yield errors of $O(\Delta)^2$ after one pass and errors of $O(\Delta)^3$ after another.

Once ΔZ is refined to a desirable level of accuracy,
$\Delta P := -2 * P * (I + \Delta Z)-1 * \Delta Z$ can be computed and then $P + \Delta P$ and $v + \Delta V$ can be computed.

However, we encounter the same problem that we do in the non-Hermitian case: The iteration can converge too slow, or might not converge at all if eigenvalues are too nearly coincident. We will guess a first ΔZ better than O much the same way we did for the non-Hermitian case, ie, by extending a closed-form solution.

The formula is derived the same way as a formula attributed by Bodewig, in 1959, to Jacobi (in 1838), Jahn (in 1948) and to Magnier (in 1948). Their formula, however, is only valid for real ΔH.

Their formula has served successfully as a quadratically convergent iteration to compute eigenvalues of real symmetric matrices (for order <= 4) in a programmable shirt pocket calculator, the HP 15C in 1982.

Using the same ideas as the non-Hermitian case, the derivation begins by considering 2x2 Hermitian matrices ΔH and extends the arguments to permuted diagonal sums ΔH of 1x1 and 2x2 matrices. It provides for these matrices exactly the skew solution ΔZ in a formula that approximates the same in general within an error of $O(\Delta)^2$.
The results of the derivation are given below:

Define signum(mu) as follows:
signum(μ) := μ / |μ| for all u except 0, and define signum(0) := 0

For all finite μ, define:
$\beta(\mu)$:= signum(μ) * tan(arctan(|μ| / 2)) = μ / (1 + sqrt (1 + |μ|^2)) and
f(μ) := signum(μ) * tan(arctan(|μ| / 4)) = β (β (μ))

Some computers can compute f(mu) faster using tan and arctan than from two square roots and divisions.

So : |β (μ)| < 1 and |f(μ)| < 1 / (1 + sqrt(2))

From vectors u := the column vector of 1's and y := (diag(V) + diag(ΔH)) / 2, compute skew-Hermetian

ΔS := ΔH / (yu' - uy' + iI) **ELEMENTWISE** and
ΔZ := f(ΔS) **ELEMENTWISE**

The exception is that wherever an element of Δz_{ij} of ΔZ is found to be infinity or NaN because of division by zero or because of overflow, replace it by
Δz_{ij} := signum((j-i) * Δh_{ij}) / (1 + sqrt(2)) using the corresponding element Δh_{ij} of ΔH.

Hence, all elements Δz_{ij} of ΔZ satisfy Δz_{ij} = - Δz_{ij}, signum(Δz_{ij}) = +/- signum(Δh_{ij}), and |Δz_{ij}| <= 1 / (1 + sqrt(2))

The formula above is perfect for ΔZ if ΔH is a permutation of a diagonal sum of 1x1 and 2x2 matrices, and correct to first order in ΔH otherwise. The first-guess provided, therefore, is better than Z := O.

This formula, however, also exposes a weakness of RefinEig.

If A has a cluster of too many eigenvalues that are too close to each other, ΔZ may have too many elements that are not very tiny, and then $(I + |\Delta Z|^2)$ may be close to a singular matrix or might be one itself. Then, while the iterative refinement of $(I + |\Delta Z|^2)^{-1}$ reduces the malfunction of RefinEig, it does not entirely eliminate it.

Also, if ΔZ has huge elements, the same problem arises with $(I + \Delta Z)^{-1} * \Delta Z$. A reasonable alternative is to clip the magnitudes of the excessively large elements of ΔZ.

7. Testing RefinEig: Werner Frank Matrices

There is a family of matrices discovered by Werner Frank in 1958. These matrices have been widely used to test eigensystem-computing software because some of the eigenvalues become extremely ill-conditioned as n increases.

Please look at Appendix B for an example of a Matlab program used to produce a Frank matrix of a given order.

The eigenvalues of a Frank matrix are all positive real numbers, and they occur in reciprocal pairs. That is, if f is an eigenvalue of a Frank Matrix, then so is 1/f. In the case of n being odd, then, the Frank Matrix will have an eigenvalue of 1.

Cf. Abramowitz and Stegun showed in 1964 that by reducing the eigenproblem to one of a tridiagonal matrix, that:
Every
(sqrt(f) – 1/sqrt(f)) is a zero of the Hermite Polynomial

Hen(x) := x * Hen-1(x) – (n-1)Hen-1(x).

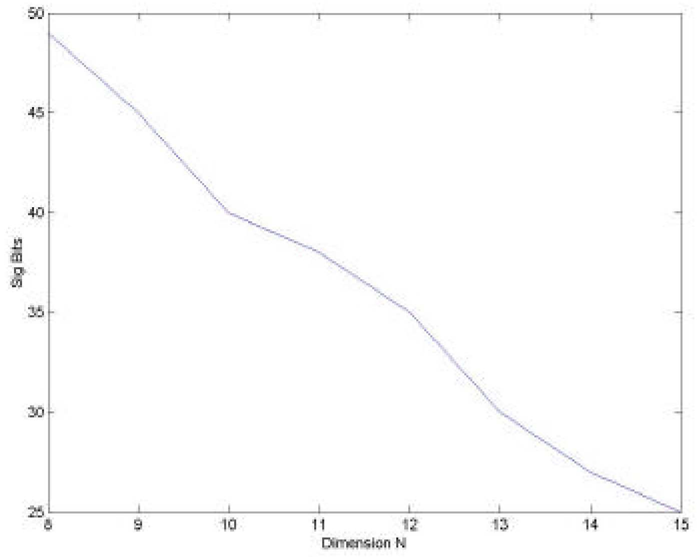
The smallest eigenvalues f of the Frank Matrices become very ill-conditioned as the order of the matrix increases.

As many as 16 sig bits might be lost for 8 <= n <= 19.

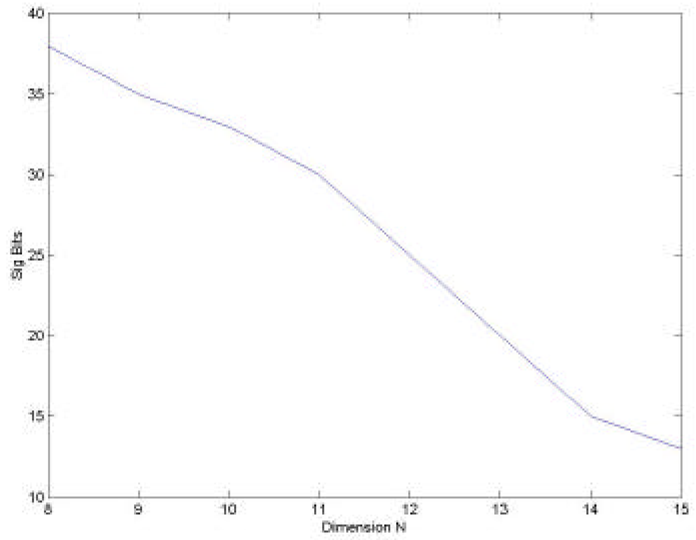
As a matter of fact, what we find is that eig refuses to return all real values, but returns values in conjugate pairs of complex numbers for n >= 15.

The tests that were carried out in Maple were done with Digits := 100

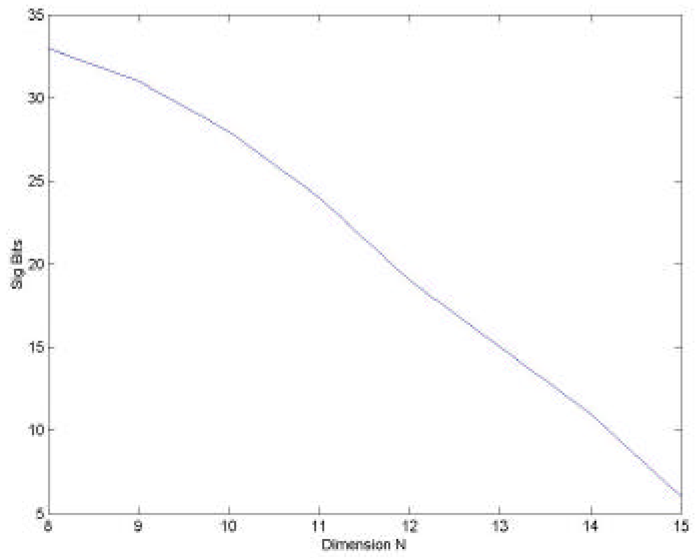
Correct Significant Bits obtained from RefinEig on 10-byte wide floating register machines for Frank Matrix F.



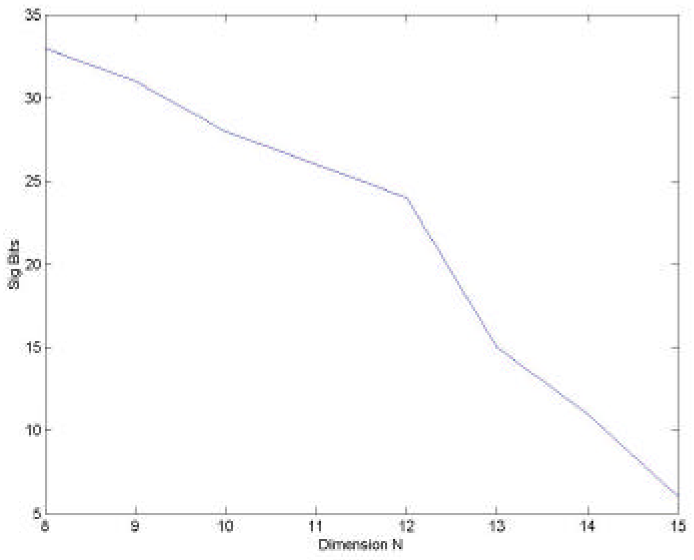
Correct Significant Bits obtained from RefinEig with 8-byte floating register calculations for Frank matrix F



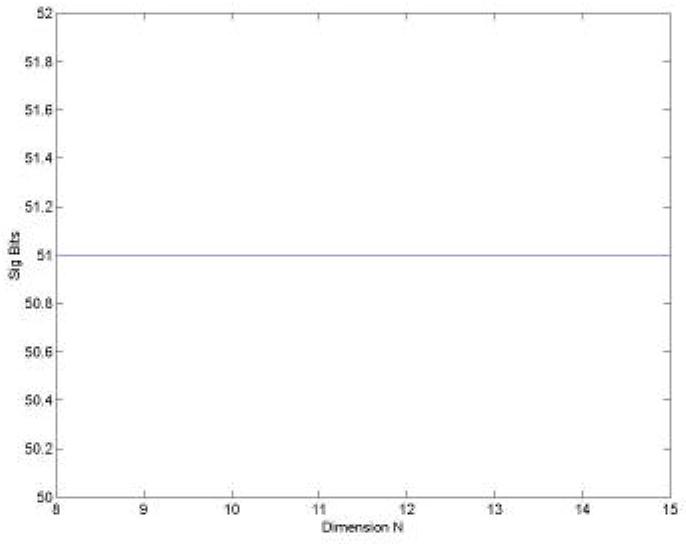
Correct Significant Bits obtained from Eig. For Frank Matrix F



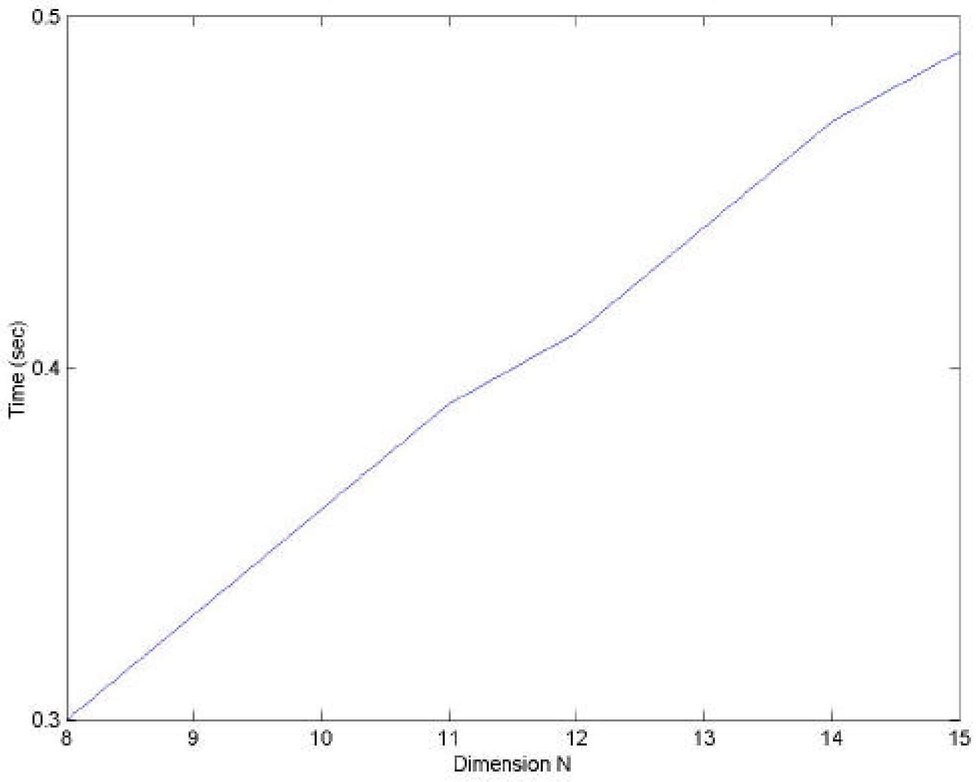
Correct Significant Bits obtained from Eig (nobalance) for Frank Matrix F



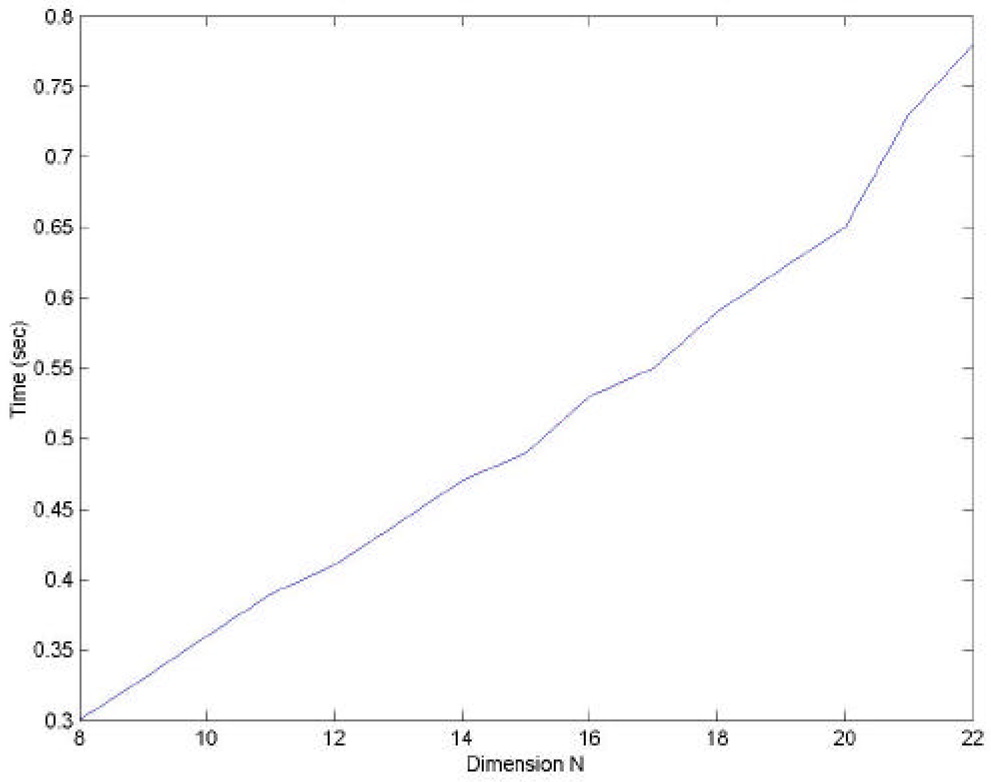
Correct Significant Bits obtained from Maple for Frank Matrix F



Graph of Dimension vs. Time taken to compute in Maple



Graph of Dimension vs. Time taken to compute RefinEig + Eig in MATLAB



8. Conclusions

As we can see from the last few graphs, Maple does an excellent job of computing the significant bits of the Frank matrices because it uses simulated high-precision arithmetic throughout the calculation. However, Maple takes a really long time to converge eve n for dimensions around n = 20. MATLAB did not converge, however for values for n greater than 15.

This brings us to a fundamental question for the use of high-precision mathematics. Sometimes it is more useful to let the brute-force approach take over, even if it does take a long time to compute. RefinEig works very well, as illustrated, but in a somewhat limited fashion, because it depends upon the fact that the first residual it calculates is valid.

Like everything else in Numerical Mathematics, the approach should be chosen appropriately to the situation and constraints at hand.

9. Acknowledgements

I would firstly like to thank Prof. William Kahan for the opportunity to work on this project with him. It has truly been a great learning experience for me, and I thank him for making me think! His guidance on this project has been truly inspirational.

I would also like to thank David Bindel, currently a CS Grad Student at UC Berkeley for his countless hours of helping me with trying to find a mex-file solution in Matlab 6, and also for putting up with many nagging questions from me over both email as well as in person.

Lastly, but not the least, I would like to thank Vishwanath Sankaran, currently a Mathematics Grad Student at UC Berkeley for helping me formulate the correct order in which to make my arguments in this paper.

10. References

[1] Kahan, William. RefinEig: a Program to Refine Eigensystems May 7, 1998

[2] Demmel, James. Applied Numerical Linear Algebra. Philadelphia © 1997

Appendix A

Update on Processors with 10-byte long floating point registers

The following processors have 10-byte long floating point registers

Intel x86/87	and (Cyrix and AMD) clones
Intel Pentium	and (Cyrix and AMD) clones
Intel Pentium 2	and (Cyrix and AMD) clones
Intel Pentium 3	and (Cyrix and AMD) clones
Intel Pentium 4	and (Cyrix and AMD) clones

Motorola 68040
Motorola 680x0 + 68881/2 processors

Matlab 5 and 6 are not currently supported. They will be supported as of August 2002.

Appendix B

Code Listings

<u>RefinEig.m</u>

-- BEGIN REFINEIG.M ----------------------------------

```
function  [ Q1, V1 ] = RefinEig( Q, V, B )
%  [Q, V] = RefinEig( Q, V, B )  applies iterative refinement to try
%  to improve the accuracy of the eigenvectors  Q  and eigevalues  V ,
%  of a square matrix  B ,  previously delivered by  [Q  V] = eig(B) .
%  Prudence requires a check that the residual  [B, Q]*[Q; -V]  be not
%  much bigger after  RefinEig  than before,  since  RefinEig  could
%  worsen rather than improve accuracy if the eigenvalues are not well
%  separated or if the eigenvectors are too nearly linearly dependent.
%  Sometimes repetition of  [Q, V] = RefinEig( Q, V, B )  pays off.
%  This version of  RefinEig  works with  MATLAB  versions  3.5 & 4.2 .
%    RefinEig  works best on  ix86-based PCs  and  680x0-based Macs,
%    whose floating-point arithmetics accumulate matrix products with
%    11  extra bits of precision.         (C) W. Kahan,  18 Jan. 1996

I = eye(size(Q)) ;  u = diag(I) ;  n = length(u) ;
v = diag(V) ;  V = diag(v) ; % ...  ensures that  V  is diagonal.
if  all(all(B == B')) ,  % ...  deal first with  Hermitian  B .
   dQ = [Q', I]*[Q; -I]*0.5 ;  d = min([ norm(dQ,1), norm(dQ,inf)])^2 ;
   if  (0.25 - d) ~= 0.25 ,  % ... which is not uncommon, ...
                      %  ...  the likliest case  ...
                     if  (0.125 - d*d) == 0.125,
                     dQ = dQ - dQ*(1.5*dQ - 2.5*dQ*dQ) ;
                     % ...  Q  requires full re-orthogonalization:
                     else          [dQ, dZ, Q] = svd(Q) ;  Q = dQ*Q' ;
   dQ = [Q', I]*[Q; -I]*0.5 ; end, end % ... small dQ = (Q'*Q-I)/2 .
   dQ = Q*dQ ;  % ...  P = Q - dQ  is accurately unitary.
   dH = [B, Q, dQ, B]*[Q; -V; V; -dQ] ; % ... = B*(Q-dQ) - (Q-dQ)*V ..
   dH = [-dQ', Q']*[dH;dH] ;  dH = (dH+dH')*0.5 ; % ... = P'*(B*P-P*V)
   dh = diag(dH) ;  y = v*0.5 ;  yt = y' ;  dx = dh*0.5 ;  dxt = dx' ;
   dZ = (dH-diag(dh)) ./ ((y(:,u') - yt(u,:)) + (dx(:,u') - dxt(u,:)) +
I) ;
   dZ = dZ ./( sqrt(real(dZ.*conj(dZ)) + 1) + 1 ) ;
   dZ = dZ ./( sqrt(real(dZ.*conj(dZ)) + 1) + 1 ) ;  K = ~finite(dZ) ;
   if  any(any(K))
         a = 1/(1 + sqrt(2)) ;  A = triu(K)*a ;  A = A - A' ;
         dZ(K) = sign(dH(K)).*A(K) ;  end
   dX = dZ*dH ;  dX = ((dX' + dX) - dX*dZ) + dH ;  dx = real(diag(dX));
   A = real(conj(dZ).*dZ) ;  a = norm(A, 1) ; % ...  A = abs(dZ).^2 .
   if (8192 + a*a) == 8192 ,   % ...  the normal case
            dv = dx - A*dx ; % ...  dv = (I+A)\dx .
      else
            [L, U] = lu(I+A) ;
            dv = U\(L\dx) ;  dv = dv - U\(L\([I, -I, A]*[dv; dx; dv]));
            k = ~finite(dv) ;
            if any(k),  dv(k) = dx(k) ;  end,  end
                  dX = dX + dZ*diag(dv)*dZ ;  dX = (dX+dX')*0.5 ;
   y = 2*v ;  yt = y' ;  dvt = dv' ;
   dZ = ( y(:,u') - yt(u,:) ) + ( dv(:,u') - dvt(u,:) ) + I ;
   dZ = ( dX - diag(diag(dX))) ./ dZ ;   a = norm(dZ, 1) ;
   if (8192 + a*a) == 8192 ,   % ...  the normal case
          dX = (dZ - dZ*dZ)*2 ;  %  ...  = 2*((I+dZ)\dZ) .
      else
          K = ~( abs(dZ) < 1024 ) ; % Will this work on PC-Matlab 3.5 ?
          if  any(any(K))
               A = triu(K)*1024 ;  A = A - A' ;
               dZ(K) = sign(dX(K)).*A(K) ;  end
          dX = ( (I + dZ)\dZ )*2 ;  end
   dQ = [Q, -dQ, dQ]*[dX; dX; I] ;  Q1 = Q - dQ ;  V1 = diag(v + dv) ;
   return,  end  %  Subsequent lines deal with non-Hermitian  B .

dZ = [B, Q]*[Q; -V] ;  [L, U] = lu(Q) ;  dC = U\(L\dZ) ;  % residual.
dC = dC - U\(L\([Q, -I]*[dC; dZ])) ;  %  ...  iteratively refines  dC
dv = diag(dC) ;  % ...  1st order approx'n to eigenvalue corrections.
vt = v.' ;  dvt = dv.' ;
S = 0.5*( (vt(u,:) - v(:,u')) + (dvt(u,:) - dv(:,u')) ) ;  % ... skew.
dZ = dC - diag(dv) ;  Y = sqrt( dZ.*dZ.' + S.*S ) ;  % ... symmetric.

K = real(Y.*conj(S));
K = (K < 0) + triu(K==0, 1) ; % ... 0's & 1's .
            K = find(K);     % COMMENT THIS LINE BEFORE MATLAB 5.2
Y(K) = -Y(K) ;  % ... Turns  Y  skew.

S = S + Y ;
%  Because  MATLAB  bungles division of complex numbers by  Inf ,
% division
%  by zero cannot be prevented simply by setting  S = S + Inf*(~S) .
dZ = dZ ./ (S+I) ; % exact for diag-sum of 2x2 's,  else  1st order.
K = ~finite(dZ) ;  if any(any(K)),  dZ(K) = I(K) ;  end % NaN&Inf --> 0
dC = dC + dC*dZ ;  dvt = diag(dC).' ;  ...  2nd order approximation.
dV = diag(dvt) ;  S = ( vt(u,:) - v(:,u') ) + ( dvt(u,:) - dV ) + I ;
dZ =  (dC - dV) ./ S ;   % ...  2nd order approximation.
K = ~finite(dZ) ;  if any(any(K)) ,  dZ(K) = I(K) ;  end
dZ = Q*dZ ;
for  j = 1:n , % ... Normalize column  j  of  Q1  just as  eig  should.
  q = Q(:,j) ;  z = dZ(:,j) ; ...  q+z  is improved eigenvector.
  q1 = [q', -1]*[q; 1] ; % ... = q'*q - 1 .
  d = (z'*z + 2*real(q'*z)) + q1 ; % ... = (q+z)'*(q+z)^2 - 1.
  d = d/(1 + sqrt(1+d)) ;  dq = ( z - q*d )/(1 + d) ;
  if  dq'*dq*64 < 1 + q1 , % ... additive normalization is best:
        Q1(:,j) = q + dq ;
     else ,  % ... normalization by division may be more accurate:
        q = q+z ;  Q1(:,j) = q/sqrt( q'*q ) ;  end,  end  % ... dq, j
V1 = diag(v + dvt.') ;  %  End  RefinEig.m
```

-- END REFINEIG.M --

-- BEGIN MXMULEPS.M --------------------------------

```
function  y = mxmuleps

%  mxmuleps  =  the roundoff threshold for  MATLAB's  matrix 
multiplication
%  =  eps     if no  Extra-Precise Accumulation occurs    ... (1)
%  =  eps/2048   if  Extra-Precise Accumulation occurs    ... (2)
%  is  NaN   if a Fused Multiply-Accumulate is enabled    ... (3)
%  on three important classes of computers conforming to  IEEE Standard
%  754
%  for  Binary Floating-Point Arithmetic.  These three classes include
% 1)Sun SPARC;  H-P PA RISC-1;  SGI MIPS;  DEC Alpha;  and some others;
%   and  Matlab 5.x  running on  Intel x86-based  PCs  and clones.
% 2)  Old Apple Macintoshes based upon the  Motorola 680x0;  and
%    Matlab 3.5 and 4.x  running on  Intel x86-based PCs and clones.
% 3) IBM RS/6000 & Apple Power Mac;  HAL SPARC;  SGI R8000;  H-P PA
%     RISC-2.
%  Matlab 3.5  does not run on computers in class  3)  whereon later
%   versions
%  of  Matlab  use  Fused Multiply-Accumulation  only during non-sparse
%    matrix multiplication so far as I have been able to determine.
% On computers in class  2)  MATLAB  accumulates  Extra-Precisely  only
% non-sparse matrix multiplication and perhaps exponentiation  ( y^n )
% so far as I have been able to determine. C)  W. Kahan,  2 Aug. 1998.

e = eps ;  %  eps = 1/2^52  on all machines listed above.
z = 4/3 ;  u = (1-z)*3 + 1 ; % |u| = 1 ULP of  1.xxxx ,  z = 4/3 - u/3
if  abs(u) ~= e,  % ...  Question whether arithmetic is anomalous:
 disp(' Has precision been altered?  Why do the following differ? ...')
      AnULPofOne = abs(u) ,  Eps = e
      disp(' Now  mxmuleps  cannot be trusted.'),  end % if |u| ~= e .
z = (z-1)*4 ;  zzz = [z; z; z] ; % ...  z = (1 - u)*4/3
y = max(abs([-u, 1+u, -1 ; -1, 1+u, -u]*zzz))*3 ;
                        % = 0 for Fused Mult-Acc.
if y == 0 ,  y = NaN ;  else  y = u/round(u/y) ;  end  %End mxmuleps
```

-- END MXMULEPS.M --------------------------------

--- BEGIN FRANK.M ----------------------------------

```
function F = Frank(n, k)
% F = Frank(n, k) is the Frank matrix of order n . The default
% is k = 0 ; otherwise the elements of F get reflected about the
% anti-diagonal (1,n)--(n,1) . Anyway, Frank is upper-Hessenberg.
if nargin == 1, k = 0; end
if n < 2, % ... necessitated by MATLAB’s faulty diag([], -1) .
F = eye(n) ; % ... with error message if n < 0 .
else
p = [n:-1:1] ;
F = triu( p( ones(n,1), : ) - diag( ones(n-1,1), -1 ), -1 ) ;
if k ~= 0 , F = F(p,p); end, end
```

--- END FRANK.M ----------------------------------

Appendix C

Non-Uniqueness of DQ

Consider the equation:

$B (Q + \Delta Q) = (Q + \Delta Q) (V + \Delta V)$

We are trying to solve this equation for ΔQ and for ΔV.

Where $V + dV$ is a diagonal matrix. Now consider multiplying both sides of the equation by a Diagonal matrix, D.

We now obtain:

$B (Q + \Delta Q) D = (Q + \Delta Q) (V + \Delta V) D$

Because $V + \Delta V$ and D are both diagonal matrices, they commute, that is:

$(V + \Delta V) D = D (V + \Delta V)$

Substituting this back into the original equation, we get:

$B (Q + \Delta Q) D = (Q + \Delta Q) D (V + \Delta V)$

Now, by defining $E = (Q + \Delta Q) D$, we get:

$B E = EV$,

for a matrix E that is not equal to Q. Moreover, a choice of any diagonal matrix D yields a new E.

So it must be the case that ΔQ is not unique.

Appendix D

Sylvester Equation and solution

Sylvester Equation (or Lyapunov Equation) is one of the form:

$AX - XB = C$, where X and C are mxn, A is mxm and B is nxn.

There is a systematic method for solving these equations.

A Sylvester Equation $AX - XB = F$ is equivalent to :

$(I_n \otimes A - B^T \otimes I_m)\,\mathrm{vec}(X) = \mathrm{vec}(F)$.

In our case, we are dealing with the equation:

$\Delta Z\,(V + \Delta V) - V\,\Delta Z = \Delta C\,(I + \Delta Z) - \Delta V$

…that we are trying to solve for ΔZ. Here, V, ΔV and ΔC are all known values, so we can put it into a Sylvester equation form.

In our case, $A = -V - \Delta C$, $B = (V + \Delta V)$ and $F = \Delta C - \Delta V$, and $X = \Delta Z$

In this case, B is a symmetric matrix, so $B = B^T$. Now, we have for our equations,

$(A - B)\,\mathrm{vec}(X) = \mathrm{vec}(F)$, which in our notation means, which we are solving for

The solution to this equation is the E in the exposition about non-Hermitian matrices (only done in a smart way) !!