Finding the largest eigenvalues of a real symmetric matrix, and corresponding eigenvectors

Nadav Har'El

Department of Mathematics
Technion − Israel Institute of Technology
Haifa 32000, Israel
E-Mail: nyh@gauss.technion.ac.il

ABSTRACT

This report describes my solution to the problem of finding the largest eigenvalues of a real symmetric matrix, and their corresponding eigenvectors, without finding all the eigenvalues of the given matrix.

1. Introduction.

Many problems require finding only a small part of the eigenvalues and eigenvectors of a large real symmetric matrix. For such problems, finding all the eigenvectors and eigenvalues, with algorithms such as Jacobi, or the QL method (cf. [NRC]), may be time consuming and wasteful. An example of such a problem is the Karhunen-Loeve expansion in pattern recognition: Given an ensemble of images, let say faces, one wants to find an optimal base for the vector space of images of the given size. The base has to be optimal in the sense that when an image similar to the ones in the ensemble (such as another face) is represented in that base, we can keep only, say, the first 40 base elements, and drop the others, while the error induced by that action is relatively small. The optimal base also has to be orthonormal, so representing an image in that base can be easily done. To find that optimal base, one has to define a certain real symmetric matrix, whose eigenvectors are the required base, where the most important vectors are the ones corresponding to the largest eigenvalues, and the ones corresponding to small eigenvalues can be dropped. For a more complete explanation of this problem see [KL1] and [KL2]. Another problem requiring only part of the eigenvalues and eigenvectors has to do with solving differential equations, which I will not get into in this report.

2. The basic idea of the algorithm

The basic idea of the algorithm is as follows: First we convert the given symmetric matrix to a similar tridiagonal matrix (i.e. has nonzero elements only on the diagonal and sub-diagonals), using the *Householder* algorithm (see below). We also get the appropriate transformation matrix. Now that we have reduced our problem to real symmetric tridiagonal matrices, we use a *Sturm Sequences* based algorithm (see below) to find the N largest eigenvalues, and their corresponding (orthonormal) eigenvectors, of the tridiagonal matrix. Finally, we use the transformation matrix to convert these eigenvectors to the ones of the original matrix. The orthonormality of the eigenvectors are preserved by this conversion, since the transformation matrix is orthogonal.

3. The Householder algorithm

This section describes the Householder method of reduction of a symmetric matrix to tridiagonal form, as defined in [NRC]. The following section is an extract from [NRC], pages 368-372.

The Householder algorithm reduces an $n \times n$ symmetric matrix A to tridiagonal form by $n-2$ orthogonal transformations. Each transformation annihilates the required part of a whole column and whole corresponding row. The basic ingredient is a Householder matrix P, which has the form

$$P = I - 2ww^T \tag{1}$$

where w is a real vector with $|w|^2 = 1$. (In the present notation, the *outer* or matrix product of two vectors, a and b is written ab^T, while the *inner* or scalar product of the vectors is written as $a^T b$.) The matrix P is orthogonal, because

$$P^2 = (1 - 2ww^T)(1 - 2ww^{T)} = 1 - 4ww^T + 4w(w^T w)w^T = 1 \tag{2}$$

Therefore $P = P^{-1}$. But $P^T = P$, and so $P^T = P^{-1}$, proving orthogonality.

Rewrite P as

$$P = 1 - \frac{uu^T}{H}, \tag{3}$$

where the scalar H is

$$H \equiv \frac{1}{2}|u|^2 \tag{4}$$

and u can now be any vector. Suppose x is the vector composed of the first column of A. Choose

$$u = x \pm |x|e_1 \tag{5}$$

where e_1 is the unit vector $(1, 0, \cdots, 0)^T$, and the choice of signs will be made later. Then

$$Px = x - \frac{u}{H}(x \pm |x|e_1)^T x = x - \frac{2u(|x|^2 \pm |x|x_1)}{2|x|^2 \pm 2|x|x_1} = x - u = -\pm e_1 \tag{6}$$

This shows that the Householder matrix P acts on a given vector x to zero all its elements except the first one.

To reduce a symmetric matrix A to tridiagonal form, we choose the vector x for the first Householder matrix to be the lower $n-1$ elements of the first column. Then the the lower $n-2$ elements will be zeroed:

$$P_1 A = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & & & & \\ 0 & & {}^{(n-1)}P_1 & & \\ \cdots & & & & \\ 0 & & & & \end{bmatrix} \cdot \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & & & & \\ a_{31} & & * & & \\ \cdots & & & & \\ a_{n1} & & & & \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ k & & & & \\ 0 & & * & & \\ \cdots & & & & \\ 0 & & & & \end{bmatrix} \tag{7}$$

Here we have written the matrices in partitioned form, with ${}^{(n-1)}P$ denoting a Householder matrix with dimensions $(n-1)\times(n-1)$, and * denoting an irrelevant part of the matrix. The quantity k is simply plus or minus the magnitude of the vector $(a_{21}, \ldots, a_{n1})^T$.

The complete orthogonal transformation is now

$$A' = PAP = \begin{bmatrix} a_{11} & k & 0 & \cdots & 0 \\ k & & & & \\ 0 & & * & & \\ \cdots & & & & \\ 0 & & & & \end{bmatrix} \tag{8}$$

Now choose the vector x for the second Householder matrix to be the bottom $n-2$ elements of the second column, and from it construct

$$P_2 \equiv \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & & & \\ \cdots & \cdots & & {}^{(n-2)}P_2 & \\ 0 & 0 & & & \end{bmatrix} \tag{9}$$

The identity block in the upper left corner of P_2 insures that the tridiagonalization achieved in the first step will not be spoiled by this one while the $(n-2)$-dimensional Householder matrix at the lower corner of P_2 creates one additional row and column of the tridiagonal output. Clearly, a sequence of $n-2$ such transformations will reduce the matrix A to tridiagonal form.

Instead of actually carrying out the matrix multiplications in PAP, we compute a vector

$$p \equiv \frac{Au}{H} \tag{10}$$

Then

$$AP = A(1 - \frac{uu^T}{H}) = A - pu^T$$

$$A' = PAP = A - pu^T - up^T + 2Kuu^T$$

where the scalar K is defined by

$$K = \frac{u^T p}{2H} \tag{11}$$

If we write

$$q \equiv p - Ku \tag{12}$$

then we have

$$A' = A - qu^T - uq^T \tag{13}$$

This is the computationally useful formula.

Following Wilkinson and Reinsch, the routine for Householder reduction used by the algorithm described in this document actually starts in the n^{th} column of A, not the first as in the explanation above. In detail, the equations are as follows: At stage m ($m = 1, 2, \ldots, n-2$) the vector u has the form

$$u^T = (a_{i1}, \ldots, a_{i,i-2}, a_{i,i-1} \pm \sqrt{\sigma}, 0, \ldots, 0) \tag{14}$$

Here

$$i \equiv n - m + 1 = n, n-1, \ldots, 3 \tag{15}$$

and the quantity σ ($|x|^2$ in our earlier notation) is

$$\sigma = (a_{i1})^2 + \cdots + (a_{i,i-1})^2 \tag{16}$$

We choose the sign of σ in (9) to be the same as the sign of $a_{i,i-1}$ to lessen roundoff error.

Variables are thus computed in the following order: $\sigma, u, H, p, K, q, A'$. At any stage m, A is tridiagonal in it's last m-1 rows and columns.

If the eigenvectors of the final tridiagonal matrix are found, then the eigenvectors of A can be obtained by applying the accumulated transformation

$$Q = P_1 P_2 \cdots P_{n-2} \tag{17}$$

to those eigenvectors. We therefore form Q by recursion after all the P's have been determined:

$$Q_{n-2} = P_{n-2}$$

$$Q_j = P_j Q_{j+1}, \; j = n-3, \ldots, 1, \tag{18}$$

$$Q = Q_1$$

It has been shown that the Householder algorithm described above, as implemented In the routine I use, has the complexity of about $4/3n^3$ at the limit of large n.

4. The Sturm sequence algorithm

The following section describes the algorithm of finding the greatest eigenvalues of an irreducible real symmetric tridiagonal matrix. An irreducible tridiagonal matrix is a tridiagonal matrix with no zeros on the subdiagonal. See [INA], page 281 for farther discussion of Sturm Sequences and Bisection Methods.

Let $p(x)$ be a polynomial of degree n,

$$p(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_n\,, a_0 \neq 0.$$

It is possible to determine the number of real roots of $p(x)$ in a specified region by examining the number of sign changes $w(a)$ for certain points $x = a$ of a sequence of polynomials $p_i(x)\,, i = 0, 1, \ldots, m$, of descending degrees. Such a sign change happens whenever the sign of a polynomial value differs from that of its successor. Furthermore, if $p_i(a) = 0$, then this entry is to be removed from the sequence of polynomial values before the sign changes are counted. Suitable sequences of polynomials are so called *Sturm Sequences.*

Definition. The sequence

$$p(x) = p_0(x), p_1(x), \ldots, p_m(x)$$

of real polynomials is a Sturm sequence for the polynomial $p(x)$ if:

(a) All real roots of $p_0(x)$ are simple.

(b) sign $p_1(\zeta) = -$sign $p_0'(\zeta)$ if ζ is a real root of $p_0(x)$.

(c) For $i = 1, 2, \ldots, m-1,\ p_{i+1}(\zeta)p_{i-1}(\zeta) < 0$ if ζ is a real root of $p_i(x)$.

(d) The last polynomial $p_m(x)$ has no real roots.

For such Sturm sequences we have the following

Theorem. The number of real roots of $p(x) \equiv p_0(x)$ in the interval $a \leq x < b$ equals $w(b) - w(a)$, where $w(x)$ is the number of sign changes of a Sturm sequence

$$p_0(x), \ldots, p_m(x)$$

at location x.

See proof at [INA].

An important use of Sturm sequences, and indeed what I use in the algorithm I present in this report, is in *bisection methods* for determining the eigenvalues of real symmetric matrices which are tridiagonal. Recall the characteristic polynomials $p_i(x)$ of the principal minor formed by the first i rows and columns of the matrix $(J - xI)$, which, as can be seen, satisfy the recursion

$$p_0(x) \equiv 1, \tag{1}$$

$$p_1(x) \equiv \alpha_1 - x,$$

$$p_i(x) \equiv (\alpha_i - x)p_{i-1}(x) - \beta_i{}^2 p_{i-2}(x), \quad i = 2, 3, \ldots, n.$$

Where $\alpha_1, \ldots, \alpha_n$ are the diagonal of J, and $\beta_2, \ldots, \beta_n$ are the subdiagonal. The key observation is that the polynomials

$$p_n(x), p_{n-1}(x), \ldots, p_0(x)$$

are a Sturm sequence for the characteristic polynomial $p_n(x) = \det(J - xI)$ (note that the polynomials here are indexed in the opposite order from before, since this indexing seems more logical for a sequence generated by iteration), provided the off-diagonal elements $\beta_i, i = 2, \ldots, n$, of the tridiagonal matrix J are all nonzero. This is proven in [INA]. As a consequence of that proof we derive that $p_n(x)$ has simple real roots $\zeta_1 > \zeta_2 > \cdots > \zeta_n$, and that

$$\text{sign } p_{n-1}(\zeta_k) = (-1)^{n+k},$$

$$\text{sign } p_n'(\zeta_k) = (-1)^{n+k+1} = -\text{sign } p_{n-1}(\zeta_k),$$

for k = 1, 2 ,..., n.

For $x = -\infty$ the Sturm sequence above has the sign pattern

$$+, +, \ldots, +.$$

Thus $w(-\infty) = 0$. By the previous theorem, $w(\eta)$ indicates the number of roots ζ of $p_n(x)$ with $\zeta < \eta$: $w(\eta) \geq n + 1 - i$ holds if and only if $\zeta_i < \eta$.

The bisection method for determining the i^{th} root ζ_i of $p_n(x)$ ($\zeta_1 > \zeta_2 > \cdots > \zeta_n$) now is as follows. Start with an interval $[a_0, b_0]$ which is known to contain ζ_1; e.g., choose $b_0 > \zeta_1, a_0 < \zeta_n$. This can be done by starting with $[-1, 1]$ and increasing the bounds if not all the roots are between them (i.e. increase bounds until $w(a_0) = 0$ and $w(b_0) = n$). Then divide this interval at its midpoint and check by means of the Sturm sequence which of the two subintervals contains ζ_i. The subinterval which contains ζ_i is again divided, and so on. More precisely, we form for $j = 0, 1, 2, \ldots$

$$u_j \equiv \frac{(a_j + b_j)}{2},$$

$$a_{j+1} \equiv \begin{cases} a_j \text{ if } w(\eta_j) \geq n + 1 - i, \\ \eta_j \text{ if } w(\eta_j) < n + 1 - i \end{cases}$$

$$b_{j+1} \equiv \begin{cases} \eta_j \text{ if } w(\eta_j) \geq n + 1 - i, \\ b_j \text{ if } w(\eta_j) < n + 1 - i \end{cases}$$

The quantities a_j increase, and the quantities b_j decrease, to the desired root ζ_i. The convergence process is linear with convergence rate 0.5. This method for determining the roots of the characteristic polynomial is relatively slow, but very accurate. It also has the feature, which makes it useful for our algorithm, that each root can be determined independently of the others.
Remember that we find this way only the eigenvalues of an irreducible tridiagonal matrix - part of the given tridiagonal matrix. See next sections for the algorithm to find the greatest eigenvalues of the whole matrix.

There is a problem with the algorithm mentioned above. The polynomials in (1), when evaluated at a certain point x, tend to grow large as the matrix grows larger (for example, for a $N \times N$ matrix, the last polynomial is of degree N. This causes problems on computers, since numbers of them have limited size, and numbers bigger than some number (about 10^{308} on a Sun computer) are treated like infinity, therefore no computations are possible). So, given a point x, we seek a way of finding a different sequence of values, $q_0, \ldots, q_n$ instead of $p_0(x), \ldots, p_n(x)$, which have the same signs, but low absolute values. This can be done using two methods I shall now describe:

4.1. The adaptive smoothing method

Let us look for a *smoothing sequence* σ_i of strictly positive numbers, which we will apply in the following manner:

$$q_0 \equiv p_0(x) \tag{2}$$

$$q_1 \equiv p_1(x)\sigma_1$$

$$q_2 \equiv p_2(x)\sigma_1\sigma_2$$

$$q_i \equiv p_i(x)\sigma_1\sigma_2\cdots\sigma_i, \quad i = 2, 3, \ldots, n.$$

We want the smoothing sequence to be such that each q_i will have the absolute value of 1, or 0. We shall then use the q_i sequence, instead of the $p_i(x)$ sequence, which is possible since they have the same signs, since σ_i are strictly positive. We start out by setting σ_1 to $\frac{1}{|p_1|}$. Then, as obvious from (2), q_1 is 0, 1, or –1.

Now, let say that we have found $q_0, \ldots, q_{i-1}$, and we are seeking q_i. By (1), we know that

$$p_i(x) \equiv (\alpha_i - x)p_{i-1}(x) - {\beta_i}^2 p_{i-2}(x)$$

So that

$$q_i \equiv p_i(x)\sigma_1\sigma_2\cdots\sigma_i = \sigma_1\sigma_1\cdots\sigma_i \cdot ((\alpha_i - x)p_{i-1}(x) - {\beta_i}^2 p_{i-2}(x))$$

$$= \sigma_i \cdot ((\alpha_i - x)q_{i-1} - {\beta_i}^2 q_{i-2}\sigma_{i-1})$$

We now define

$$\sigma_i \equiv \frac{1}{|(\alpha_i - x)q_{i-1} - {\beta_i}^2 q_{i-2}\sigma_{i-1}|}$$

Note that if the denominator has the value of zero, we set σ_i to 1. Now we have q_i which can be only 1, –1 or 0.

This method of smoothing the sequence is very good, and if fact works for all matrices I have tried. We shall now give an alternative smoothing method which makes the whole Sturm sequence algorithm about 30% faster, but unfortunately does not work good enough for very large (about 400×400) matrices. Note that one of these smoothing methods *must* be used, since without them the Sturm sequence algorithm will fail even for relatively small matrices.

4.2. The power smoothing method

This method also works by defining q_i based on $p_i(x)$ but does not adapt the smoothing value - it has a constant smoothing value which is $\sigma = \frac{1}{|x|}$, and the q_i are defined as follows:

$$q_0 \equiv p_0(x)$$

$$q_1 \equiv p_1(x)\sigma$$

$$q_2 \equiv p_2(x)\sigma^2$$

$$q_i \equiv p_i(x)\sigma^i, \quad i = 3, \ldots, n.$$

This method works for modest sized matrices (less then about 400×400) since, ideally, a n^{th} degree polynomial evaluated at point x divided by x^n should be a *nice* number. We say ideally, since the polynomial coefficients themselves can grow very large (remember: we never actually find those coefficients - we only find the values at the point x by recursion) rendering the *power smoothing* method useless. This method has problems with large matrices (about 400×400), and may also not work very well with certain smaller matrices

4.3. Choosing the smoothing method

The routines I have written support both kinds of smoothing, and it is up to the user to decide which one to use. I would recommend for users who want to find eigenvalues or eigenvectors of a matrix of size 300×300 or smaller to try the power smoothing method, and if it fails (reports errors while finding eigenvectors for the incorrect eigenvalues it found) try the adaptive smoothing method. If you want to find eigenvalues and eigenvectors of a larger matrix, or want to take no risks of finding incorrect eigenvalues, then use the adaptive smoothing method. Remember that using the adaptive smoothing method makes the Sturm sequence part of the algorithm about 30% slower, but this is not real problem in large matrices, since the Householder overhead dominates the running time.

5. Finding eigenvectors by inverse iteration

The following section discusses the problem of finding an eigenvector corresponding to a given eigenvalue of an irreducible matrix. The algorithm is outlined in [NRC], page 394.

We first have to note that because of the discussion in the previous section, we have come to the conclusion that the characteristic polynomial of an irreducible tridiagonal real symmetric matrix has only real roots, and what is more important - all roots are simple. That saves us the trouble of looking for more than one linearly independent vectors. Also, different eigenvectors of such a matrix are automatically orthogonal, as they correspond to different eigenvalues.

The basic idea behind inverse iteration is quite simple. Let y be the solution of the linear system

$$(A - \tau I)y = b \tag{1}$$

where b is a random vector and τ is close to some eigenvalue of A. Then the solution y will be close to the eigenvector corresponding to τ. The procedure can be iterated: replace b by y and solve for the new y, which will be even closer to the true eigenvector. Note that if τ is too accurate, then equation (1) should not have a solution, so we change τ a little by adding some very small constant to it, such as 10^{-14}. Note that this causes the algorithm to have problems if two eigenvalues are about this close to each other, but that should almost never occur in practical uses (of course you can easily find a matrix which causes this problem, but typical matrices will not cause this problem).

We can see why this works by expanding both y and b as linear combinations of the eigenvectors x_j of A:

$$y = \sum_j \alpha_j x_j \qquad b = \sum_j \beta_j x_j \tag{2}$$

Then (1) gives

$$\sum_j \alpha_j (\lambda_j - \tau) x_j = \sum_j \beta_j x_j \tag{3}$$

so that

$$\alpha_j = \frac{\beta_j}{\lambda_j - \tau} \tag{4}$$

and

$$y = \sum_j \frac{\beta_j x_j}{\lambda_j - \tau} \tag{5}$$

If τ is close to λ_n, say, then provided β_n is not accidentally too small, y will be approximately x_n, up to normalization. Moreover, each iteration of this procedure gives another power of $\lambda_j - \tau$ in the denominator of (5). Thus the convergence is rapid for well separated eigenvalues (remember that we have already shown that all eigenvalues are simple).

Suppose at the i^{th} stage of iteration we are solving the equation

$$(A - \lambda_i I)y = x_i \tag{6}$$

where x_i and λ_i are our current guesses for the eigenvector and eigenvalue of interest (we shall not be updating λ_i, since it is already accurate, but a variant of this algorithm also lets you improve eigenvalues previously found). Since y of (6) is an improved approximation to x, we normalize it and set

$$x_{i+1} = \frac{y}{|y|} \tag{7}$$

These formulas look simple enough, but in practice, the implementation is quite tricky

We begin by choosing a random normalized vector, as the initial guess for the eigenvector, x_0. We then solve (6). If the solving algorithm fails (I will not discuss here the algorithm to solve a tridiagonal system of equations, since it is a simple Gaussian elimination for tridiagonal system) then we choose a new random vector and start again. If the solution of (6) succeeded, we check if $|x_1 - x_0| < \varepsilon$, where ε is some tolerance, which I have chosen in my program to be 10^{-15}. If it is indeed smaller, then we have found our eigenvector x_1. If not we keep iterating. If after some number of iterations, say 10 iterations, $|x_{i+1} - x_i|$ has not decreased enough, then we begin again with a new random vector. If, after a new random vector has been chosen, let say, 3 times, we still did not find an eigenvector, we regretfully have to say that the

algorithm failed. This should not happen, but, unfortunately, I still have some problems with giant matrices (about $800x800$). (Of course, if you are unlucky enough to get 3 consecutive bad random vectors, then that's another problem. However, this seems very unlikely.)

This algorithm the complexity of $O(n^3)$, and it is much less efficient than the QL method (cf. [NRC]). However, when less then about 25 percent of the eigenvectors are required, this algorithm should be more efficient.

6. Putting the algorithms together

Now that I have explained all those algorithms, it's time to put them together for our final algorithm. Given A - a $n \times n$ real symmetric matrix, we want to find it's s biggest eigenvalues, and respective eigenvectors.

We begin by using the Householder algorithm to transform A into a tridiagonal matrix. Call the tridiagonal matrix T and the orthogonal transformation matrix P. We can't proceed with the Sturm sequence algorithm, and the inverse iteration algorithms because they only work on an irreducible tridiagonal matrix. So we break T into square tridiagonal blocks T_i, $i = 1, \ldots, m$, which are irreducible. Note that any eigenvalue of a block T_i is also an eigenvalue of T, and vice versa - each eigenvalue of T is an eigenvalue of some T_i, and if λ is non-simple eigenvalue of degree j of T then there are exactly j blocks with a (simple) eigenvalue λ. This is obvious from the block structure of T (the characteristic polynomial of T is the product of the characteristic polynomials of T_i). Also, if we have found an eigenvector of T_i, then to find a corresponding eigenvector of T all we have to do is add zeros to the vector to make it line up with the appropriate block T_i in the matrix multiplication $Tv = \lambda v$.

Now we make an array of size s to remember the eigenvalues. First we find the biggest eigenvalue of each irreducible block T_i with the Sturm sequence algorithm from section 4, and sort them into the array. Now, we find the second biggest eigenvalue of each block. We merge them into the array. If the array was filled with eigenvalues and T_i's second eigenvalue was smaller then the smallest eigenvalue in the array, then we can stop looking at T_i in the next iteration, since it's next eigenvalues are even smaller. We also drop a T_i if we have finished getting all it's eigenvalues. We repeat this process, until we ran out of T_i's. We are left with the s biggest eigenvalues of T in the array. Now, for each of those eigenvalues we find the eigenvector corresponding to it, in the appropriate irreducible block T_i. We normalize this vector. We then add zeros to make it an eigenvector of T. We still have to see that the eigenvectors we get this way are orthogonal, however this is obvious: Two eigenvectors which were gotten from the same irreducible block correspond to different eigenvalues, and therefore are automatically orthogonal. Two eigenvectors which were gotten from different irreducible blocks are obviously orthogonal since their non-zero elements are at a different place.

Now that we have the s biggest eigenvalues of T, and their corresponding orthonormal eigenvectors, all we have to do is to multiply them by the orthogonal transformation matrix P. We then finally have the s biggest eigenvalues of A, and their corresponding orthonormal eigenvectors.

7. Performance of the Algorithm

One of the goals of this project was to make the algorithm presented here as fast as possible. If time was not important, then one can use one of the many generic algorithms floating around which find all the eigenvalues and eigenvectors, such as the routines implemented in Matlab†: the *eig* command of Matlab finds all the eigenvalues and eigenvectors of any matrix. While this may be useful in some cases, it is not needed in our case, and as we shall see, is much slower than our algorithm when only a part of the eigenvalues and eigenvectors are sought. In this section we shall also see examples of the time used by our algorithm as function of n - the size of the given matrix and s - the number of eigenvalues and eigenvectors wanted.

All of the following examples were run on tx.technion.ac.il, which is a Sun Sparcstation, with 690MP CPU (four processors), with the SunOS 4.1.2 operation system (Note that although the computer has four processors, the current algorithm has no support for parallel execution. An enhancement to the algorithm

could be to calculate eigenvectors in parallel).

I have written a program incorporating the algorithm described in this report, and ran an example in which I find eigenvectors and eigenvalues of some fixed 200×200 random matrix (i.e. a 200×200 random matrix was chosen, and remained fixed through all the following tests). The following table shows the time it took this program to run, and the time it took matlab to run on the same matrix. The program 'main3' in the following table is the program which uses the algorithm described in this report.

Table 1 - a 200×200 matrix		
Program	**Eigs wanted**	**CPU seconds**
main3	200	21.80
	50	11.10
	30	9.50
	20	8.90
	10	8.15
	0	7.30
Matlab	200	22.00

One can easily see from table 1 that main3 is faster than matlab, even when all 200 eigenvalues and eigenvectors are wanted. This is because the algorithm employed by Matlab does not use the fact that the matrix is symmetric. We can also see from this table how decreasing the number of wanted eigenvalues and eigenvectors decreases the time the program takes to run. Asking for only 50 out of the 200 eigenvalues and eigenvectors decreases the run time to half. Also, by looking at this table we can see that the Householder routine took 7.3 seconds to run (because finding 0 eigenvalues means only do the Householder routine), and every eigenvalue and eigenvector wanted took about 0.072 second to find.

The following table shows how does the size of the matrix influence the running time of the algorithm. In the table, all times are in CPU seconds, i.e. the time the computer really worked on the program. The 'Sturm&Inverse + Convert' heading specifies the average time it took to find one eigenvalue and corresponding eigenvector of the $n \times n$ random matrix. The 'Sturm&Inverse' time includes the Sturm algorithm to find eigenvalues and the inverse iteration algorithm to find the corresponding eigenvector, and the 'Convert' time specifies the time of the the matrix multiplication needed to convert the eigenvector to the one of the original matrix.

Table 2 - run time of parts of the algorithm (in CPU seconds)		
Matrix size	**Householder**	**Sturm&Inverse + Convert**
10	0.00	$0.000 + 0.000 = 0.000$
20	0.00	$0.004 + 0.001 = 0.005$
30	0.02	$0.006 + 0.001 = 0.007$
40	0.05	$0.007 + 0.003 = 0.010$
50	0.10	$0.008 + 0.002 = 0.010$
60	0.18	$0.009 + 0.004 = 0.013$
80	0.37	$0.013 + 0.005 = 0.018$
100	0.77	$0.015 + 0.010 = 0.025$
150	2.83	$0.022 + 0.024 = 0.046$
200	7.02	$0.030 + 0.042 = 0.072$
300	29.67	$0.045 + 0.097 = 0.142$
400	111.88	$0.061 + 0.172 = 0.233$
500	167.75	$0.079 + 0.270 = 0.349$
600	344.83	$0.101 + 0.384 = 0.485$
700	555.98	$0.126 + 0.525 = 0.651$
800	989.75	$0.145 + 0.688 = 0.833$

As can be seen quite clearly from this table (for example, compare the time for a 800×800 matrix to the time of a 400×400 matrix), the complexity of the Householder algorithm is nearly $O(n^3)$ for $n \to \infty$, like I mentioned before. This is really bad for extremely large matrices, but unfortunately it is the best algorithm available ([NCR] says that the Householder algorithm together with a QL algorithm with implicit shifts is

the most efficient known technique for finding *all* the eigenvalues and eigenvectors of a real symmetric matrix). From table 2 we can also see that the Sturm sequence and inverse iteration algorithms combined are approximately $O(n)$. The conversion of an eigenvector of the tridiagonal matrix to the one of the original matrix looks like a simple operation, but unfortunately, as can be seen in table 2, it has the complexity of $O(n^2)$, so for large n, the conversion takes much more time then the Sturm and inverse iteration algorithms! This is because the conversion works with the whole matrix, which is $n \times n$, as opposed to working with a tridiagonal matrix which has only $2n - 1$ elements.

We have come to the conclusion, that the whole algorithm is $O(n^3)$ when n is large enough. Because I said the most efficient algorithm to find eigenvalues and eigenvectors of real symmetric matrices uses the Householder algorithm, then even the most efficient algorithm must be $O(n^3)$. Knowing that we can't improve the Householder algorithm, we look for ways to increase the speed of the second part of the algorithm (Sturm sequence bisection, inverse iteration, and matrix multiplication). Suppose, for example, that we have a $n \times n$ matrix, and we want to find half its eigenvalues and eigenvectors. Then, we use a $O(n^2)$ algorithm $\frac{n}{2}$ times, so overall we get a $O(n^3)$ complexity. Improving small details of the algorithm (see next section for planned improvements) will not make it much faster. We can really make that part of the algorithm much faster if we are working on a computer with several CPU's, such as a supercomputer. As we previously saw, an important benefit of this algorithm is that each eigenvalue and eigenvector can be found independently of the others. So we can use different processors to find different eigenvalues and their corresponding eigenvectors. For example, let us say we want to find the 200 biggest eigenvalues of the 800×800 matrix mentioned in table 2. Using the regular sequential algorithm will take us $989.75 + 200 \times 0.833 = 1156.35$ seconds. Using 100 CPU's, each finding two eigenvalues and the two corresponding eigenvectors, will take us only $989.75 + 2 * 0.833 = 991.416$ seconds, so we saved 164.934 seconds. If the matrix is bigger, or we want more eigenvalues and eigenvectors, then the absolute time difference will be even greater. This way we decrease the running time of the second part of the algorithm to $\frac{1}{100}$ of the original time (don't forget that we still have the $O(n^3)$ Householder overhead, which isn't decreased by using a computer with several processors. This is because the Householder algorithm is sequential by nature, and each step of it requires the results of the previous step).

8. Summary

As can be seen from this report, finding the biggest eigenvalues and corresponding eigenvectors can be a tough job. It is even harder trying to code the algorithm with a computer language, as you run into many unexpected problems. The most problematic algorithm to code in a machine with finite accuracy are the inverse iteration algorithm, together with it's algorithm for solving a tridiagonal system of equations, and the Sturm sequence algorithm. As I said before, this algorithm is not perfect yet, and runs into trouble in some cases of giant matrices.

Another thing that can be done to improve the final algorithm is to improve the convergence speed of the Sturm sequence algorithm. One way to do this is instead of always using the bounds $[a_0, b_0]$ which are good for all the eigenvalues, use more accurate bounds: when checking the place of one eigenvalue we automatically improve our guess on other eigenvalues' whereabouts. The other, more dramatic, change would be to completely discard the bisection method, and instead use a newton-related method, which is perhaps less accurate, but faster then the bisection method.

9. Acknowledgments

The project described in this report was supported by the US-Israel Binational Science Foundation, under supervision of Prof. Koby Rubinstein, Department of Mathematics, Technion - Israel Institute of Technology.

I would also like to thank my father, Zvi Har'El, for giving me very useful ideas.

10. References.

[NRC]

Press, H. W. & Flannery, B. P. & Teukolsky A. T. & Vetterling W. T., *Numerical Recipes in C, The Art of Scientific Computing*, Chapter 11. Cambridge University Press

[KL1]

Sirovich, L. & Kirby, M., *Low-dimensional procedure for the characterization of human faces*, Journal of the Optical Society of America, March 1987, Vol 4, No. 3, Page 519.

[KL2]

Sirovich, L. & Kirby, M., *Application of the Karhunen-Loeve Procedure for the Characterization of Human Faces*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 12, No 1, January 1990, Page 103.

[INA]

Stoer, J. & Bulirsch, R., *Introduction to Numerical Analysis*, Chapter 5.6. Springer-Verlag New York, 1980.