Reduction of a symmetric matrix to a tridiagonal form – summary

Student Name: Yoav Nir
Faculty Name: Prof. Sivan Toledo

Given a symmetric matrix **A**, one would like to partition it into a simpler tridiagonal form of $\mathbf{A=LTL^t}$, where **L** is a lower unit diagonal matrix, and **T** is a tridiagonal matrix. That is, **L** and **T** are square matrices satisfying:
- $\mathbf{L_{i,i} = 1}$ for all **i**, and $\mathbf{L_{i,j} = 0}$ for all **i**,**j** with $\mathbf{i > j}$.
- $\mathbf{T_{i,j} = 0}$ for all **i**,**j** such that $\mathbf{|i - j| > 1}$.

Furthermore, given that **A** is a sparse matrix (almost all elements are zeros), one may like to have a relatively sparse **L** in the partitioning.

So, we'd shortly discuss about a recursive Parlett-Reid algorithm (for full details see **(*)**). An implementation written in MATLAB language is provided later, along with sample functions.
Here are a few ways to summarize the way the algorithm is working, using a recursive function:
- To begin with, we always assume that we know the first column of **L**; So along with **A**, we receive it as an input to the recursive function (excluding the first element of the column, which is always **1**).
- For a matrix of size **1**, there's one and unique trivial partitioning. In the case of size **2**, there's also one unique partitioning, for any choice of the first column of **L**. It can be formed as a simple exercise in algebra.

Both of these cases are base cases in the recursive algorithm.
- For a larger matrix **A**, we begin by splitting it into four blocks: Top left $\mathbf{A^{[11]}}$, top right $\mathbf{A^{[12]}}$, bottom left $\mathbf{A^{[21]}}$ and bottom right $\mathbf{A^{[22]}}$. Similarly, we also look at blocks of **L**, **T** and **H := LT**.

As for how would one split the matrix (that is, what's the size of $\mathbf{A^{[11]}}$), this can basically depend on the implementation. It'd be efficient to split into equal parts (see the end of description in **(*)**).
For any choice of splitting, all matrices should be split in the same way.
- Now, the first step of the algorithm calculates $\mathbf{L^{[11]}}$ and $\mathbf{T^{[11]}}$ recursively, using the formula $\mathbf{A^{[11]} = L^{[11]}T^{[11]}L^{[11]t}}$ (which must hold if there's a partitioning). The relevant elements of the first column of **L** are passed as well, of course.

- To the beginning of the second step: Again assuming a partitioning exists, it satisfies $\mathbf{A}^{[21]} = \mathbf{H}^{[21]}\mathbf{L}^{[11]T}$. At this stage, everything is known here except for $\mathbf{H}^{[21]}$. So we can find $\mathbf{H}^{[21]}$ using multiple triangular solves (see the given code for implementation).
- Next, we expand $\mathbf{H}^{[21]}$. During the progress of expanding, a few elements can be removed, as they are all known to be zeros. Going over all columns of $\mathbf{H}^{[21]}$ one by one (excluding the very last column), we get a series of equations. From these we can get the columns of $\mathbf{L}^{[21]}$ we haven't had before.
- Now, from a similar equation resulted by looking at the latest column of $\mathbf{H}^{[21]}$, we may calculate $\mathbf{T}_{k+1,k}=\mathbf{T}^{[21]}_{1,k}$ or $\mathbf{T}_{k,k+1}=\mathbf{T}^{[12]}_{k,1}$, taking advantage of the fact that $\mathbf{L}$ has values of $\mathbf{1}$ on the main diagonal. Right afterwards, we can find out the first column of $\mathbf{L}^{[22]}$.
- Almost done, getting to the third stage of the algorithm, we write the following statement: $\mathbf{A}^{[22]} = \mathbf{H}^{[21]}\mathbf{L}^{[21]T} + \mathbf{H}^{[22]}\mathbf{L}^{[22]T}$. Expanding $\mathbf{H}^{[22]}\mathbf{L}^{[22]T}$, we get an expression consisting of $\mathbf{L}^{[22]}\mathbf{T}^{[22]}\mathbf{L}^{[22]t}$. Now, all terms in the statement (after expanding) are known at this stage, excluding $\mathbf{X} := \mathbf{L}^{[22]}\mathbf{T}^{[22]}\mathbf{L}^{[22]t}$.
So we can easily solve $\mathbf{X}$.
Finally, we partition $\mathbf{X} = \mathbf{L}^{[22]}\mathbf{T}^{[22]}\mathbf{L}^{[22]t}$ recursively, taking advantage of the fact that we know the first column of $\mathbf{L}^{[22]}$ at this point.

So far, we've shown an algorithm which does a partitioning of the form $\mathbf{A}=\mathbf{L}\mathbf{T}\mathbf{L}^{t}$, as previously mentioned. But how can $\mathbf{L}$ be relatively sparse?
Well, in the second step of the algorithm, we find the missing columns of $\mathbf{L}^{[21]}$ from a series of vector equations of the form $\mathbf{v} = \mathbf{a}\mathbf{x}$ (or equivalent).
Here, $\mathbf{v}$ is a known vector, $\mathbf{a}$ is a known scalar, and $\mathbf{x}$ is an unknown vector, being a column of $\mathbf{L}^{[21]}$ we're seeking.
If $\mathbf{a} \neq \mathbf{0}$, then there's one unique solution. If $\mathbf{a} = \mathbf{0}$ and $\mathbf{v} \neq \mathbf{0}$, then there's no solution and we halt.
Otherwise, $\mathbf{a} = \mathbf{0}$ and $\mathbf{v} = \mathbf{0}$, so any choice of $\mathbf{x}$ is a solution. It can be wise, therefore, to attempt and get a sparser $\mathbf{L}$, by choosing $\mathbf{x} = \mathbf{0}$ as a column of $\mathbf{L}^{[21]}$ satisfying the equation.

Implementation of the algorithm in MATLAB, including sample functions

```
% Recursive implementation of Parlett-Reid's algorithm for symmetric matrix
% partitioning into a tridiagonal form, as described in the following document:
% http://www.math.tau.ac.il/~shagil/pre-pubs/ShklarskiToledo-Aasen.pdf

% Given a symmetric matrix A, this function attempts to partition it into a
% tridiagonal form of A=LTL^t, where L is a lower unit triangular matrix
% and T is a tridiagonal matrix.
% This is a recursive function. L_Col1 is given to be the first column of L,
% excluding the value of 1 in its main diagonal, n is the size of A,
% and block_func is a pointer to a function used for determining
% how to partition matrices into blocks.
%
% Prototype of block_func:
% Given a natural number n >= 3, returns some natural k such that 2 <= k < n.
%
% Return value: A matrices pair (L,T) satisfying A=LTL^t on success,
% or (NaN,NaN) on failure.
function [L,T] = pr_recursive(A, L_Col1, n, block_func)
	L = eye(n);
	L(2:n, 1) = L_Col1;
	if (n == 1) % If n == 1 then we'd have T = A, L = I.
		T = A;
	% If n == 2, then we know L by the given L_Col1 input.
	% There's one unique T such that A = L*T*L^t, calculated here.
	elseif (n == 2)
		T(1,1) = A(1,1);
		T(1,2) = A(1,2) - T(1,1)*L(2,1);
		T(2,1) = T(1,2);
		T(2,2) = A(2,2) - A(2,1)*L(2,1) - L(2,1)*T(1,2);
	else	% n >= 3
		T = zeros(n);
		% Separate matrices to blocks of sizes kxk, (n-k)xk, kx(n-k)
		% and (n-k)x(n-k), according to given partitioning function.
		k = block_func(n);
		% First step of the algorithm, as described in the document
		% mentioned in the header, starting with a recursive call.
		[L_rec, T_rec] = pr_recursive(A(1:k,1:k), L_Col1(1:k-1), k, block_func);
		if (any(isnan(L_rec))) % Has it failed at some point?
			L = NaN; T = NaN;
			return;
		end
		L(1:k,1:k) = L_rec; T(1:k,1:k) = T_rec;
		% Second step of the algorithm, part 1.
		% H_block represents here the (2,1) block of H := LT,
		% with its size being (n-k)xk.
		H_block = mult_triangular_solve(A((k+1):n, 1:k), (L(1:k, 1:k))');

		% Second step of the algorithm, part 2,
		% Starting with the first column of the (2,1) block of H,
		% so the second column of block (2,1) of L can be calculated.
		if (T(2,1) == 0) % We can't divide by zero. Is there a solution?
			if (H_block(:, 1) - L((k+1):n,1)*T(1,1) == 0)
				L((k+1):n,2) = zeros(n-k, 1); % Yes
			else
				L = NaN; T = NaN; % No
				return;
```

```
                end
        else
                L((k+1):n,2) = (H_block(:, 1) - L((k+1):n,1)*T(1,1))/T(2,1);
        end
        % Now looping over columns 2 to (k-1) of the (2,1) block of H,
        % so we can find out the latest of block (2,1) of L.
        for i = 2:(k-1)
                if (T(i+1,i) == 0) % Again, can't divide. Any solution?
                        if (H_block(:, i) - L((k+1):n,i-1)*T(i-1,i) - L((k+1):n,i)*T(i,i) == 0)
                                L((k+1):n,i+1) = zeros(n-k, 1); % Yes
                        else
                                L = NaN; T = NaN; % No
                                return;
                        end
                else
                        L((k+1):n,i+1) = (H_block(:, i) - L((k+1):n,i-1)*T(i-1,i) - L((k+1):n,i)*T(i,i))/T(i+1,i);
                end
        end
        % Finally, take care of the latest column of the same H block,
        % so we can find out the first column of the (2,2) block of L,
        % as well as element T[k+1,k], or T[k,k+1].
        T(k,k+1) = H_block(1,k) - L(k+1,k-1)*T(k-1,k) - L(k+1,k)*T(k,k);
        T(k+1,k) = T(k,k+1);
        if (T(k+1,k) == 0) % Again, need to check if there's a solution.
                if (H_block(:,k) - L((k+1):n,k-1)*T(k-1,k) - L((k+1):n,k)*T(k,k) == 0)
                        L(k+1,k+1) = 1; % Yes
                        L((k+2):n,k+1) = zeros(n-k-1, 1);
                else
                        L = NaN; T = NaN; % No
                        return;
                end
        else
                L((k+1):n,k+1) = (H_block(:,k) - L((k+1):n,k-1)*T(k-1,k) - L((k+1):n,k)*T(k,k))/T(k+1,k);
        end
        % Third step of the algorithm, with one more recursive call.
        [L_rec, T_rec] = pr_recursive(A((k+1):n,(k+1):n) - H_block*(L((k+1):n, 1:k))' -
L((k+1):n,k)*T(k,k+1)*(L((k+1):n,k+1))', L((k+2):n, k+1), n-k, block_func);
        if (any(isnan(L_rec))) % It could fail at some point.
                L = NaN; T = NaN;
                return;
        end
        L((k+1):n,(k+1):n) = L_rec; T((k+1):n,(k+1):n) = T_rec;
end
```

```
% Recursive implementation of Parlett-Reid's algorithm for symmetric matrix
% partitioning into a tridiagonal form, as described in the following document:
% http://www.math.tau.ac.il/~shagil/pre-pubs/ShklarskiToledo-Aasen.pdf

% Given a symmetric matrix A, this function attempts to partition it into a
% tridiagonal form of A=LTL^t, where L is a lower unit triangular matrix
% and T is a tridiagonal matrix.
% It takes advantage of a recursive function, which partitions matrices into
% blocks in the way. The argument block_func is a pointer to a function used
% for determining how to partition matrices into blocks.
%
% Prototype of block_func:
% Given a natural number n >= 3, returns some natural k such that 2 <= k < n.
%
% Return value: A matrices pair (L,T) satisfying A=LTL^t on success,
% or (NaN,NaN) on failure.
function [L,T] = pr(A, part_func)
	[L,T] = pr_recursive(A, zeros(1, length(A)-1), length(A), part_func);

% A helper function. Given a matrix A and an upper unit triangular matrix L,
% solves the matrix equation A = XL.
%
% Return value: Some matrix X satisfying A = XL.
%
% Assumption: A's width is the same as L's size (L being a square matrix).
function X = mult_triangular_solve(A, L)
	[m,n] = size(A);
	X = zeros(m, n);
	for i=1:m
		for j=1:n
			X(i,j) = A(i,j);
			for k=1:(j-1)
				X(i,j) = X(i,j) - X(i,k)*L(k,j);
			end
		end
	end

% A sample function used for partitioning a matrix into blocks.
% Given a natural number n, returns the upper part of n/2.
function k = upper_half(n)
	k = ceil(n/2);

% A sample program.
function sample()
	disp(' ')
	disp('Random symmetric matrix created:')
	A = 5*rand(6); A = A + A'
	disp('Output of pr(A, @upper_half):')
	[L,T] = pr(A, @upper_half)
	if (not(any(isnan(L))))
		disp(['Norm of the difference A-LTL^t: ' num2str(norm(A-L*T*L'))])
	end
```

(*) Reference:
Gil Shklarski and Sivan Toledo.
Blocked and Recursive Algorithms for Triangular Tridiagonalization.
Accepted to *ACM Transactions on Mathematical Software*, Nov 2009.