A High Performance C Package for Tridiagonalization of Complex Symmetric Matrices

Guohong Liu and Sanzheng Qiao
Department of Computing and Software
McMaster University
Hamilton, Ontario L8S 4L7, Canada
qiao@mcmaster.ca

Abstract

Block algorithms have better performance than scalar and single vector algorithms due to their exploitation of memory hierarchy. This paper presents a high performance C implementation of a block Lanczos tridiagonalization algorithm for complex symmetric matrices. The design principles of the implementation and techniques used in the implementation are described. Our experiments show that this implementation has high performance.

Keywords: Complex symmetric matrix, block Lanczos algorithm, singular value decomposition (SVD), Takagi factorization, C language, LAPACK.

1 Introduction

This paper presents a high performance C package for tridiagonalization of complex symmetric matrices using block Lanczos method. We first briefly describe the block Lanczos algorithm for tridiagonalization of complex symmetric matrix presented in [5].

For any complex symmetric matrix A of order n, there exist a unitary $Q \in C^{n\times n}$ and an order n nonnegative diagonal $\Sigma = \mathrm{diag}(\sigma_1, ..., \sigma_n)$, $\sigma_1 \geq$

$\sigma_2 \geq \cdots \geq \sigma_n \geq 0$, such that

$$A = Q \Sigma Q^{\mathrm{T}} \quad \text{or} \quad Q^{\mathrm{H}} A \bar{Q} = \Sigma.$$

This symmetric form of singular value decomposition (SVD) is called Takagi factorization [3, 6].

The computation of the Takagi factorization consists of two stages: tridiagonalization and diagonalization [2]. A complex symmetric matrix is first reduced to complex symmetric and tridiagonal form. The second stage, diagonalization of the complex symmetric tridiagonal matrix computed in the first stage, can be implemented by the implicit QR method [2, 4], or the more efficient divide-and-conquer method [8].

Block algorithms in which blocks of vectors instead of single vectors are operated upon are rich in matrix-matrix (level 3 BLAS) operations. Performance is thus improved by exploiting memory hierarchies.

The block Lanczos tridiagonalization algorithm [5] consists of two stages: block tridiagonalization and tridiagonalization. A complex symmetric matrix is first reduced to complex symmetric and block tridiagonal form. Assume $n = k \times b$, then there exists the decomposition

$$Q^{\mathrm{H}} A \bar{Q} = J = \begin{bmatrix} M_1 & B_1^{\mathrm{T}} & & \dots & 0 \\ B_1 & M_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & B_{k-1}^{\mathrm{T}} \\ 0 & \dots & & B_{k-1} & M_k \end{bmatrix},$$

where

$$Q = [Q_1, Q_2, ..., Q_k], Q_i \in C^{n \times b},$$

is orthonormal, $M_i \in C^{b \times b}$ are symmetric, and $B_i \in C^{b \times b}$ upper triangular.

The second stage reduces the block tridiagonal complex symmetric matrix J resulted from the first stage to complex symmetric tridiagonal. Specifically, we find a unitary $P = [\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n]$, $\mathbf{p}_i \in C^{n \times 1}$, such that

$$P^{\mathrm{H}} J \bar{P} = T = \begin{bmatrix} \alpha_1 & \beta_1 & & \dots & 0 \\ \beta_1 & \alpha_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{n-1} \\ 0 & \dots & & \beta_{n-1} & \alpha_n \end{bmatrix}.$$

In this paper, we present a high performance C package for the block Lanczos tridiagonalization of complex symmetric matrices. Our goals are outlined in Section 2 and the general design philosophy is given in Section 3 followed by a description of the design of each module and the techniques used to achieve our goals in Section 4. In particular, the techniques used to achieve high performance is presented in Section 5. Finally, Section 6 shows our experiment results to demonstrate the high performance.

2 Overview

Our primary goals for this package are: high performance, user friendly interface, highly modularized, and portable.

The performance of this package relies heavily upon the performance of the implementation of the block Lanczos tridiagonalization algorithm. The algorithm frequently calls basic matrix and vector computations, such as matrix-matrix multiplication and division, transpose or complex conjugate of a complex matrix, and QR factorization. Thus these routines are crucial for the performance of the package. Instead of reinventing wheels, whenever possible, we use the subroutines for these matrix and vector computations provided by the widely used LAPACK (Linear Algebra PACKage) [1], a Fortran library of routines for solving problems in numerical linear algebra. It is efficient on a wide range of high-performance computers if the hardware vendor has implemented an efficient set of BLAS (Basic Linear Algebra Subroutines).

Since LAPACK is widely used in scientific computing community, to provide a friendly user interface, we follow LAPACK naming conventions and interface style in designing our function names and interfaces. The users who are familiar with LAPACK will find that the use of our package for tridiagonalizing a complex symmetric matrix is almost identical to a LAPACK function for bidiagonalizing a general complex matrix. Our package, however, has better performance than its counterpart LAPACK function. Moreover, because of the similar interfaces, the application of the functions in our package can be smoothly mixed with the applications of LAPACK routines. The functions in this package can be called by a C++ program.

3 General Design

This package is divided into three layers:

- User interface
  The user simply invokes the only one C function defined in this layer to perform tridiagonalization of a complex symmetric matrix. User interface encapsulates the details of the block Lanczos algorithm implementation, and separates the application of the package from the algorithm implemented in the package.

- Block Lanczos algorithm
  This layer implements the block Lanczos tridiagonalization algorithm for complex symmetric matrices. It contains the block Lanczos algorithm modules.

- Support routines
  This layer supports the layer of block Lanczos algorithm by providing the lower level computing like QR factorization, and matrix-matrix, matrix-vector and vector-vector operations. This layer includes the wrapper of LAPACK and BLAS, LAPACK and BLAS modules, and modules for some low level computing routines unavailable in LAPACK or BLAS.

Following software design principles [7], we design the modules so that there is a weak coupling between modules and a strong cohesion within a module. Each module depends on as few other modules as possible. Consequently, changes of one module are transparent to other modules provided its interface with others is unchanged. For example, when we fine tuned the implementation of LAPACK and BLAS wrapper to improve the performance of the package, we did not have to touch other modules like block Lanczos algorithm modules. Moreover, the module structure helps the user understand the sizable block Lanczos algorithm implementation.

4 Design of Modules

In this section, we describe the details of the design of the modules and the techniques used to achieve our goals in this package described in the previous section. The hierarchy of the modules in this C package along with its application is shown in Figure 1.

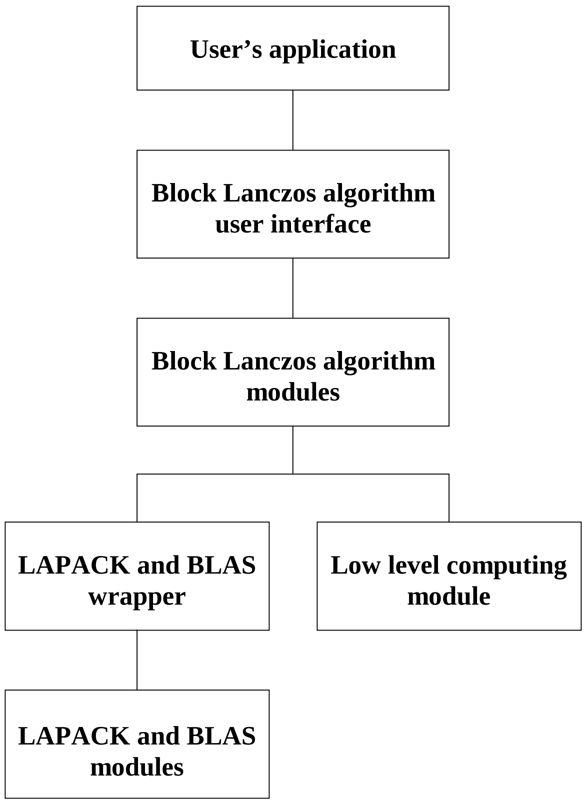
User's application
Block Lanczos algorithm user interface
Block Lanczos algorithm modules
LAPACK and BLAS wrapper
Low level computing module
LAPACK and BLAS modules

Figure 1: hierarchy of modules

4.1 User Interface

Normally the user of this package is most interested in only how to invoke the routine to perform tridiagonalization of a complex symmetric matrix. The block Lanczos algorithm user interface is what the user sees and uses. Using this package, the user needs to include only one header file containing the prototype of the user interface, `zcstrd`, and then link the block Lanczos tridiagonalization C package. The user interface module hides the secret of the implementation of the algorithm.

For example, suppose we perform the tridiagonalization of a complex symmetric matrix of order 1024 specified by the memory pointer `doublecomplex *Mat`. The block size is set to 4. The starting matrix of orthonormal columns is specified by the pointer `doublecomplex *StartingMat`. After the execution of the block Lanczos algorithm, the diagonal and off-diagonal elements of the tridiagonal matrix are stored in memory `doublecomplex *a` and `doublereal *b` respectively. The matrices, which are obtained in block tridiaognalization stage and tridiagonalization stage of the block Lanczos algorithm, are specified by `doublecomplex *Q` and `doublecomplex *P` respectively. It is the user's responsibility to allocate the spaces for `doublecomplex *a`, `doublereal *b`, `doublecomplex *Q` and `doublecomplex *P` before calling the function. The following is a sample code:

```
#include "zcstrd.h"

void func ()
{
    /* Declarations of variables.              */
    int n;
    int bs;
    doublecomplex *Q = NULL;
    doublecomplex *P = NULL;
    doublecomplex *a = NULL;
    doublereal *b = NULL;
    int info;

    /* Set the input arguments.                */
    n = 1024;
    bs = 4;

    /* Allocate memories for *a, *b, *Q and *P. */
    ... ...

    /* Perform trdiagonalization.              */
    zcstrd_ (&n, Mat, &bs, StartingMat, a, b, Q, P, &info);
}
```

To provide the user with a friendly interface, we design the function most visible to the user following LAPACK style, since LAPACK is widely used in scientific computing community. Specifically, the LAPACK routine `zhetrd` reduces a complex Hermitian matrix to the real symmetric tridiagonal form by a unitary similarity transformation. The following is its prototype:

```
int zhetrd_ (char *uplo,
             integer *n, doublecomplex *a, integer *lda,
             doublereal *d, doublereal *e,
             doublecomplex *tau, doublecomplex *work,
             integer *lwork, integer *info)
```

Following the naming convention of LAPACK and the above prototype, we design our function, which reduces a complex symmetric matrix to the complex symmetric tridiagonal form by unitary transformations,

```
int zcstrd_ (integer *n, doublecomplex *A,
             integer *bs, doublecomplex *S,
             doublecomplex *a, doublereal *b,
             doublecomplex *Q, doublecomplex *P,
             integer *info)
```

LAPACK users can easily understand the name and the arguments. The first letter `z` indicates double complex. The next two letters `cs` indicate the complex symmetric matrix. The last three letters `trd` indicate the tridiagonalization reduction of the matrix. We adopt the data types defined in LAPACK, like integer, doublecomplex and doublereal, etc, for the arguments. The idea of the design is to allow users to seamlessly invoke the block Lanczos algorithm routine with other LAPACK routines.

The following are the descriptions of arguments of the user interface in LAPACK style:

```
n     (input) integer *
      The order of the matrix A. n >= 0.
A     (input) double complex array, dimension (n,n)
      The complex symmetric matrix A.
bs    (input) integer *
      The order of one block.  bs > 0.
S     (input) double complex array, dimension (n,bs)
      The starting block of orthonormal columns.
a     (output) double complex array, dimension (n)
      The diagonal elements of the tridiagonal matrix T.
```

```
b     (output) double real array, dimension (n-1)
      The subdiagonal elements of the tridiagonal matrix T.
Q     (output) double complex array, dimension (n,n)
      It is computed in block tridiagonalization stage.
P     (output) double complex array, dimension (n,n)
      It is computed in tridiagonalization stage.
info  (output) integer *
      = 0: successful exit
      < 0: if info = -i,
               the ith argument had an illegal value.
      > 0: Exception is thrown.
```

The matrix A is not stored in the packed form supported by LAPACK for the sake of performance. It would not be difficult to implement the tridiagonalization of complex symmetric matrices using the packed form for A. Also, note that this function returns two unitary matrices Q and P, not the product of them.

4.2 Block Lanczos Algorithm Modules

Block Lanzcos algorithm modules implement the functions in the algorithm itself. They do not directly invoke any LAPACK or BLAS routine. The performance of these modules determine the performance of the whole package. These modules make frequent calls to LAPACK and BLAS wrapper functions and low level routines. In the following two subsections, we discuss LAPACK and BLAS wrapper modules and low level routine modules.

4.3 Lapack and BLAS Wrapper

LAPACK and BLAS wrapper modules implement such computations as QR factorization, matrix-matrix operations and matrix-vector operations, which block Lanzcos algorithm modules require, by calling LAPACK and BLAS routines. Why is a wrapper inserted between the algorithm modules and LAPACK and BLAS Modules? LAPACK and BLAS are designed for general purpose. To simplify the interfaces, we customize some LAPACK and BLAS routines and introduce the wrapper. We define some data structures for matrices and vectors in the wrapper. The following is the data structure for a general complex matrix,

```
typedef struct doubleComplexMat {
    doublecomplex *mat;
    int m;
    int n;
} DoubleComplexMat;
```

This data structure wraps the LAPACK data type, doublecomplex, with the dimensions of a matrix. It is used between modules to simplify the arguments of the routines. The wrapper of level 3 BLAS function `zgemm` is a good example. BLAS function `zgemm` implements matrix-matrix multiplication $\alpha \text{op}(A)\text{op}(B) + \beta \text{op}(C)$, where $\text{op}(X) = X, X^T, X^H$. Its prototype is

```
int zgemm_(char *transa, char *transb,
           integer *m, integer *n,
           integer *k, doublecomplex *alpha,
           doublecomplex *a, integer *lda,
           doublecomplex *b, integer *ldb,
           doublecomplex *beta, doublecomplex *c,
           integer *ldc)
```

Computation $R = \pm\text{op}(A)\text{op}(B)+C$ is frequently performed in block Lanczos algorithm. The wrapper function `mmult` sets $\alpha = 1$ and $\beta = 1$ and calls BLAS function `zgemm` to perform this computation. Thus, the parameters α and β are invisible in algorithm modules. The prototype of `mmult` is

```
int mmult (char *signa, DoubleComplexMat *A, char *transa,
           DoubleComplexMat *B, char *transb,
           DoubleComplexMat *C, DoubleComplexMat *R)


Input
    signa   "+",  R = op(A)*op(B) + C
            "-",  R = -op(A)*op(B) + C
    A       Matrix. Size of A = m*k
    transa  "N",  op(A) = A
            "T",  op(A) = A transpose
            "H",  op(A) = A Hermitian
    B       Matrix. Size of B = k*n
    transb  "N",  op(B) = B
            "T",  op(B) = B transpose
```

```
                "H",  op(B) = B Hermitian
                "C",  op(B) = B complex conjugate
        C       Matrix. Size of C = m*n
                if C = NULL, then R = +/- op(A)*op(B)
    Output
        R       Matrix. Size of R = m*n
                R = +/- op(A)*op(B) + C
```

Compared with the BLAS routine `zgemm`, which has 13 arguments, this wrapper function, which has only seven arguments, is simpler, thus more understandable. For example, the following is a segment of the Matlab code of the block Lanczos algorithm, where A, Q, R, M are four matrices.

```
R = A*conj(Q);
M = Q'*R;
R = R - Q*M;
```

The corresponding C implementation is:

```
mmult("+", A, "N", Q, "C", NULL, R);
mmult("+", Q, "H", R, "N", NULL, M);
mmult("-", Q, "N", M, "N", R, R);
```

Of course, reducing the number of arguments not only makes the code more understandable but also reduces the possibility of making errors in passing arguments.

We adopt the bottom-up strategy in the development of the wrapper. We thoroughly tested the correctness and accuracy of the wrapper.

4.4 Low Level Subroutine Module

Block Lanczos algorithm requires some low level subroutines unavailable in LAPACK and BLAS. These subroutines are implemented in a separate module, low level subroutine module. The low level subroutine module and the LAPACK and BLAS wrapper are at the same layer in the layering structure. The subroutines in this module include componentwise product of two complex matrices, and multiplication and addition of complex numbers. These subroutines are actually code segments extracted from LAPACK and BLAS. These subroutines may be replaced by alternative implementations without modifying other modules.

4.5 LAPACK and BLAS Modules

At the bottom of the hierarchy of the modules are the LAPACK and BLAS modules. These modules may be replaced by other implementations. The only module affected by such changes is the LAPACK and BLAS wrapper.

5 Performance

High performance is one of the primary goals of this package. To achieve this goal, we

- streamline and customize some LAPACK and BLAS functions to fit the block Lanczos method;
- avoid dynamic memory allocation;
- reduce function calls without sacrificing the readability of the code.

LAPACK and BLAS are designed for general use. As described in Section 4.3, the wrapper function `mmult` simplifies the interface by reducing the number of arguments. Still, some cases never occur in our program. For example, $\text{op}(A)$ is never $\bar{A}$. Thus checking the value of `transa` for "C" is unnecessary and wasting of time. This case is eliminated for efficiency. The matrix division function `mdiv` is another example. It computes $R = A/B$. Our initial implementation checked whether the quotient matrix to be stored in matrix R or overwrites B, i.e. $B = A/B$. In our program, however, the overwriting case never occurs. We examined every wrapper function and eliminated unnecessary cases. The functions in the wrapper are frequently called by the block algorithm. Their performance have significant impact on the total performance.

Dynamic memory allocation costs run time. We avoid dynamic memory allocation by allocating static work buffers. For example, when `mmult` is called, the value of the parameter `transb` can be "C", that is, $\bar{B}$ is required in the matrix multiplication. The LAPACK function `zlacgv` calculates the complex conjugate of a matrix, however, the the input matrix is overwritten by its conjugate. Thus it is necessary to save the input matrix in a work buffer before calling `zlacgv`. There are other functions in the wrapper requiring work buffers. As we know, the functions in the wrapper are frequently called. To avoid frequent dynamic memory allocation, after careful inspection, we figured out work buffers necessary for the functions in the wrapper, and created a function `wrapperInit`, which allocates memory and assigns the pointers to the work buffers to some global variables. So, the functions in the wrapper can use the work buffers. During the initialization of the block Lanczos algorithm, `wrapperInit` is called and work buffers are allocated to be used by the functions in the wrapper. At the end of the block Lanczos algorithm, a function `freeWrapper` is called to release the memory.

Function calls require significant overhead. We avoid functions which are short and frequently called by using macros or simply inserting their implementations inside the caller functions. For example, the matrix multiplication function `mmult` requires the calculation of the complex conjugate of a matrix. The code for calculating complex conjugate is short. So, instead of calling a complex conjugate function inside `mmult`, we inserted the code for calculating conjugate in `mmult`. Since the code is short, the readability is not sacrificed, whereas the performance can be improved.

We compared the performance of the package before and after the adoption of the above strategies. The following is the run time of tridiagonalization of a random complex symmetric matrix of size 1024 before the adoption of these strategies:

```
Block Lanczos tridiagonalization: 285.22 seconds
```

Next is the run time of the improved implementation for another random complex symmetric matrix of size 1024:

```
Block Lanczos tridiagonalization: 231.87 seconds
```

6 Experiments

The package was developed in MS Visual C/C++ V6.0 environment. The experiments were carried out on a PC with Intel 846M processor and 256M memory running Windows 2000.

To test the accuracy, a random complex symmetric matrix generator is included in the package. The generator produces complex symmetric matrices with uniformly distributed random complex entries. The starting block S of orthonormal columns is initialized from the QR factorization of a random complex n-by-b matrix. The error in the orthogonality of Q was measured by: $\|I - Q^H Q\|_{\mathrm{F}}/n^2$, and the error in the tridiagonalization $T = Q^H A \bar{Q}$ was measured by: $\|Q^H A\bar{Q} - T\|_F/n^2$. Table 1 lists the results of our numerical experiments on accuracy.

LAPACK does not support complex symmetric matrix data structure. The SVD of a complex symmetric matrix has to be treated as the SVD of a general complex matrix. Thus we compared our package for tridiagonalization of complex symmetric matrices with the LAPACK routine `zgebrd`, bidiagonalization of general complex matrices, since both are the first stage in the computation of the SVD. Figure 2 shows the performance comparison with LAPACK's bidiagonalization routine, where the block size used in our package is four.

matrix order	block size	error in orthogonality	error in factorization
512	4	$4.750E-013$	$1.612E-011$
1024	4	$9.185E-014$	$4.142E-012$
1024	8	$7.473E-014$	$3.334E-012$
1024	16	$8.338E-014$	$3.891E-012$
1280	4	$7.774E-014$	$2.737E-012$

Table 1: Accuracy of block Lanczos tridiagonalization implementation.

7 Conclusion

In this paper, we have presented a high performance C package for the tridiagonalization of a complex symmetric matrix using block Lanczos algorithm. We have described our design ideas and techniques used in the package to achieve our goals: high performance, well structured, user friendly interface, and portable. Our experimental results show that our package performs better than a comparable bidiagonalization routine in LAPACK.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Third edition*. SIAM Publications, Philadelphia, 1999.

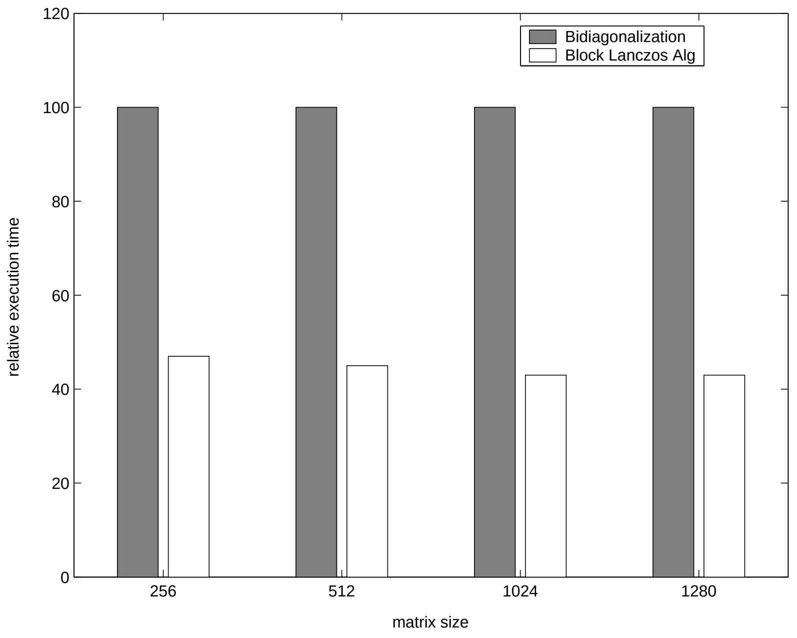

Figure 2: Comparison of the efficiency of block Lanczos tridiagonalization implementation. The y axis shows the execution times normalized to the execution time of LAPACK's bidiagonalization of complex matrices

[2] A. Bunse-Gerstner and W. B. Gragg. Singular value decompositions of complex symmetric matrices. *Journal of Computational and Applied Mathematics*, **21** (1988) 41–54.

[3] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[4] F. T. Luk and S. Qiao. A fast singular value algorithm for Hankel matrices. *Fast Algorithms for Structured Matrices: Theory and Applications, Contemporary Mathematics 323*, Editor V. Olshevsky, American Mathematical Society. 2003. 169–177.

[5] Sanzheng Qiao, Guohong Liu, and Wei Xu. Block Lanczos Tridiagonalization of Complex Symmetric Matrices. To appear in *Advanced Signal Processing Algorithms, Architectures, and Implementations XV*, edited by Franklin T. Luk. Proceedings of SPIE. Vol. 5910, 2005.

[6] T. Takagi. On an algebraic problem related to an analytic Theorem of Carathédory and Fejér and on an allied theorem of Landau. *Japan J. Math.* **1** (1924) 82–93.

[7] J.C. Van Vliet. *Software Engineering: Principles and Practice*, 2nd Edition. Wiley, 2000.

[8] Wei Xu and Sanzheng Qiao. A Divide-and-Conquer Method for the Takagi Factorization. *Technical Report No. CAS 05-01-SQ* , Department of Computing and Software, McMaster University. February 2005.