LLNL-SM-674579

Lawrence Livermore National Laboratory

libROM User Guide and Design

Author(s)

William Arrighi, Geoffrey Oxberry, Tanya Vassilevska, Kyle Chand

Lawrence Livermore National Laboratory
P.O. Box 808, Livermore, CA 94551-0808

July 14, 2015

Disclaimer

Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344.

Table of Contents

Disclaimer 2

1.0 Introduction 4

2.0 Description of Incremental SVD Algorithms 4
- 2.1 Standard Incremental SVD Algorithm 4
- 2.2 Fast Update Incremental Algorithm 5
- 2.3 Computational Differences 6
- 2.4 Summary 6

3.0 Library Organization 6
- 3.1 Overview 7
- 3.2 Vector and Matrix 7
- 3.3 BasisReader and BasisWriter 7
- 3.4 Database and HDFDatabase 8
- 3.5 SVD 8
- 3.6 StaticSVD 8
- 3.7 IncrementalSVD 8
- 3.8 IncrementalSVDStandard and IncrementalSVDFastUpdate 8
- 3.9 SVDSampler 8
- 3.10 StaticSVDSampler and IncrementalSVDSampler 9
- 3.11 SVDBasisGenerator 9
- 3.12 StaticSVDBasisGenerator and IncrementalSVDBasisGenerator 9
- 3.13 Utilities and ParallelBuffer 9

4.0 Building the Library 9
- 4.1 Running configure 9
- 4.2 Building the Library 10
- 4.3 Running the Tests 10
  - 4.3.1 smoke_test 10
  - 4.3.2 uneven_dist 10
  - 4.3.3 random_test 11

5.0 Example Usage 11
- 5.1 Basis Vector Generation From Full Order Model Simulation 11
- 5.2 Reduced Order Model Construction and Simulation 12

6.0 Scaling 14

7.0 Application to Convection-Diffusion Simulation 14
- 7.1 Full Order Solution 14
- 7.2 Reduced Order Solution 15

References 18

1.0 Introduction

The libROM library is a collection of C++ classes that implements order reduction via singular value decomposition (SVD) of sampled state vectors. It implements the 2 parallel SVD algorithms described by Matthew Brand [1] as well as a serial "static" or non-incremental SVD algorithm. The library also provides a mechanism for adaptive sampling.

The library supports the construction of multiple sets of basis vectors, wherein each set of basis vectors corresponds to a different simulation time interval. Users have control over the maximum number of basis vectors in a set that, along with the adaptive sampling control, determines how many sets of basis vectors will be collected.

The library provides the means to read and write the basis vectors resulting from the order reduction. Construction of the reduced order model from the basis vectors must be performed by the application developer and is not a part of the library.

The library is configured with a configure script and built with the resulting generated GNU Makefile. There are 3 required external libraries: mpi, lapack, and hdf5. Details about configuring and building the library are supplied below.

This document is intended to provide an overview of the design of the library and the theory behind it. Details about the library API and individual classes are provided through doxygen documentation that may be built along with the library.

2.0 Description of Incremental SVD Algorithms

In this section we will give a brief description of the incremental SVD algorithms and discuss the advantages of each.

Both algorithms decompose the system, Q, into the product of three matrices: $Q = VsW^T$. V contains the basis vectors for the reduced order model, s contains the singular values, and W^T contains the basis vectors in the row space that we do not need to track. In the fast update algorithm the left matrix, V, is factored into two other matrices, U and U_p so that the basis vectors, V, are $V = UU_p$. The standard method does not perform this factorization.

2.1 Standard Incremental SVD Algorithm

For the initial sampled simulation state vector, u:

1. $s = \|u\|$
2. $V = u / \|u\|$
3. $n = 1$

For each subsequent sampled simulation state vector, u:

1. $l = V^T u$
2. $k = \|u - Vl\|$
3. $j = (u - Vl) / k$

4. $Q = \begin{bmatrix} diag(s) & l \\ 0 & k \end{bmatrix}$
5. Compute svd(Q) to form V', s', W'.
6. If k<linearity tolerance then u is not linearly independent and the system is updated as:
    a. $V = VV'_{1:n,1:n}$
    b. $s = s'_{1:n}$
    c. Reorthogonalize V if necessary.
7. Otherwise the system is updated as:
    a. $V = \begin{bmatrix} V & j \end{bmatrix}$
    b. $V = VV'$
    c. $s = s'$
    d. $n = n+1$
    e. Reorthogonalize V if necessary.

2.2 Fast Update Incremental Algorithm

For the initial sampled simulation state vector, u:

1. $s = \|u\|$
2. $U = u / \|u\|$
3. $U_p = [1]$
4. $V = UU_p$
5. $n = 1$

For each subsequent sampled simulation state vector, u:

1. $l = V^T u$
2. $k = \|u - Vl\|$
3. $j = (u - Vl) / k$
4. $Q = \begin{bmatrix} diag(s) & l \\ 0 & k \end{bmatrix}$
5. Compute svd(Q) to form V', s', W'.
6. If k<linearity tolerance then u is not linearly independent and the system is updated as:
    a. $U_p = U_p V'_{1:n,1:n}$
    b. $s = s'_{1:n}$
    c. Reorthogonalize U_p if necessary.
    d. $V = UU_p$
7. Otherwise the system is updated as:
    a. $U = \begin{bmatrix} U & j \end{bmatrix}$
    b. $U_p = \begin{bmatrix} U_p & 0 \\ 0 & 1 \end{bmatrix} V'$
    c. $V = UU_p$
    d. $s = s'$

e. $n = n + 1$
f. Reorthogonalize U and U_p if necessary.

2.3 Computational Differences

From the descriptions above it is clear that much of the two algorithms are identical. The major differences occur in how the left singular matrix is updated in steps 6 and 7.

The basis vectors, V, consist of a matrix of d rows and n columns where d is the dimension of a state vector and n is the number of samples. The dimension of a state vector is large and n << d. Therefore V is a matrix of many rows and few columns the rows being distributed across all the processors running the full order model. Thus, any computations involving V are relatively expensive parallel operations.

The same is true of the matrix, U, of the fast update algorithm. However, the matrix U_p is a small square matrix with n rows and columns.

Comparing step 6a for each algorithm one can see that in the standard method this is a parallel operation on V while in the fast update method this is a simple, local operation. However the fast update method adds the computation of V in step 6d, which is a parallel operation. So computationally this is something of a wash.

Comparing the standard method's step 7a-b with the fast update method's steps 7a-c we see that 7b of the standard method involves a parallel operation on V as does 7c of the fast update method. The other steps are simple, local operations. So again computationally the two algorithms are something of a wash.

Where the fast update method has a potential advantage is in the number of reorthogonalization steps that it requires relative to the standard algorithm. This is especially relevant for the reorthogonalization of V and U both of which are distributed. As Brand[1] points out, U and U_p may maintain their orthogonality better than V. If this hold true then the fast update method may have a computational advantage in that it must reorthogonalize a large distributed matrix less often than the standard method.

In the later discussion on the application of this library to form a specific reduced order model we will show data indicating that this advantage may not always exist in practice.

2.4 Summary

The library provides both methods and users are free to use whichever one they wish. There may be a slight computational advantage to the fast update method but that advantage may not exist for all problems.

3.0 Library Organization

This section describes the classes in the library and their intended use. API specifics are addressed in the doxygen documentation.

3.1 Overview

Overall, one can think of the library as having 2 purposes: creation of basis vectors from a simulation, and consumption of those basis vectors to form a reduced order model. The bulk of the library is devoted the first of these tasks while only a small subset is needed to support the second.

The StaticSVD, IncrementalSVD, IncrementalSVDStandard, and IncrementalSVDFastUpdate classes are all directly related to the creation of basis vectors. They implement various SVD-centric basis vector creation algorithms. Applications feed these algorithms state vectors and basis vectors are the result.

The StaticSVDSampler and IncrementalSVDSampler classes are also directly related to creation of basis vectors. In both cases the algorithms compute the next simulation time at which a sample is needed and are able to answer the query of whether a sample should be taken at a requested simulation time.

StaticSVDBasisGenerator and IncrementalSVDBasisGenerator wrap a basis vector creation algorithm with its corresponding sampling algorithm. This provides an application with a single point of interaction should one want to use the library's sampling algorithms. It is possible for an application to write their own sampling algorithm in which case it is free to use the basis vector creation algorithm classes directly instead of one of these wrappers.

BasisReader is the one class directly related to the consumption of basis vectors. Its purpose is to allow applications constructing reduced order models from basis vectors to read the generated basis vectors.

The following sections give a thumbnail description of each class in the library. More detailed information about each class is provided in the doxygen documentation.

3.2 Vector and Matrix

These are the core classes that provide the necessary parallel linear algebra for the library. Vector encapsulates the notion of a vector distributed over multiple processors. Each processor owns some number of components of the vector. There is no requirement that the vector be distributed evenly among the processors. Matrix encapsulates the notion of a matrix distributed over multiple processors. Each processor owns some number of rows of the matrix. All processors contain the same number of columns. As with Vector, there is no requirement that the matrix be distributed evenly among the processors.

3.3 BasisReader and BasisWriter

Basis vectors are written to a Database file on disk by BasisWriter and read from a Database file on disk by BasisWriter. BasisReader provides the means for an application to read the basis vectors that have been created. Users request the basis vectors applicable to a specific simulation time. An application will directly use BasisReader. BasisWriter is essentially an internal class and users of the library do not need to interact with it directly. Whenever the basis vectors for an interval of simulation time is complete the library directs that they be written to the Database file via the BasisWriter.

BasisReader returns basis vectors in the form of a Matrix. Once an application has the basis vectors it is expected that it will form the reduced order model from this Matrix using Matrix's API. The only interaction with BasisReader is to get the basis vectors for the current time.

Each processor writes it's contribution to the set of basis vectors to it's own Database file. The user specifies the base or root name of the Database file. Each processors file is of the form root_name.pid where "pid" is a 6-digit processor ID. Hence processor 0 will produce the Database file root_name.000000.

3.4 Database and HDFDatabase

These classes are essentially internal and users of the library do not need to interact with them directly. They encapsulate the details of how the basis vectors are stored on disk. BasisReader and BasisWriter interact with these classes.

Database is an abstract base class that defines the interface to files containing basis vectors. It supports reading and writing of the types of data needed by the library and is not designed to support all possible data types.

HDFDatabase implements the interface of Database for HDF5 files. Currently the library only supports the HDF5 file format for basis vectors.

3.5 SVD

SVD is an abstract base class that defines the interface of a generic SVD-centric basis vector creation algorithm.

3.6 StaticSVD

StaticSVD implements what we call the "static" as opposed to the incremental SVD basis vector creation algorithm. By "static" we mean that the basis vectors are computed in one step after sampling of state vectors is complete. It is an inherently serial algorithm and is provided mainly for prototyping or for comparison with the parallel incremental algorithms.

3.7 IncrementalSVD

IncrementalSVD is an abstract base class that defines the internal interface of the incremental SVD basis vector creation algorithm. It implements the pure virtual member functions of class SVD in terms of the internal interface of the incremental algorithm.

3.8 IncrementalSVDStandard and IncrementalSVDFastUpdate

IncrementalSVDStandard and IncrementalSVDFastUpdate implement the interface defined by IncrementalSVD according to the 2 algorithms described by Matthew Brand [1]. Both algorithms are parallel.

3.9 SVDSampler

SVDSampler is an abstract base class defining the interface for sampling of state vectors.

3.10 StaticSVDSampler and IncrementalSVDSampler

StaticSVDSampler implements the interface of SVDSampler as a simple user controlled uniform sampling scheme for the Static SVD method of basis vector creation. IncrementalSVDSampler implements that interface as a user controlled adaptive sampling scheme for the Incremental SVD methods of basis vector creation.

3.11 SVDBasisGenerator

SVDBasisGenerator is an abstract base class defining the interface to a basis vector creation algorithm and corresponding sampling algorithm.

3.12 StaticSVDBasisGenerator and IncrementalSVDBasisGenerator

StaticSVDBasisGenerator implements the interface of SVDBasisGenerator for the "static" (serial, non-incremental) method and wraps StaticSVD and StaticSVDSampler. IncrementalSVDBasisGenerator implements this interface for the incremental methods and wraps IncrementalSVDStandard/IncrementalSVDFastUpdate and IncrementalSVDSampler. As mentioned above, this allows a single point of interaction for applications wishing to use both the library's basis vector creation and sampling algorithms.

3.13 Utilities and ParallelBuffer

These are both intended to be internal classes although Utilities contains some features that may be of use to an application. The Utilities class provides static methods for error reporting and string manipulation. Utilities.h contains macros for checking assertions and aborting due to an error that an application may find useful. ParallelBuffer is used by Utilities and is purely an internal class.

4.0 Building the Library

Building the library is accomplished in 2 steps. First one needs to run the supplied configure script that generates a GNU Makefile. Then one must run "make" on the generated Makefile to compile the code and create the library. Doxygen documentation may then be generated if desired. The supplied tests may be run to verify a successful build.

4.1 Running configure

There are several required pieces of information that must be supplied to configure. In addition there are several options that may be requested.

The required information is:

- The C++ compiler. It may be specified either by setting the CXX environment variable or with the –-with-CXX configure option.
- The location of the HDF5 installation. It is specified with the –-with-hdf5 configure option. Note that the location that you provide must be the directory where the lib and include directories for the hdf5 installation are found.
- The location of the lapack library. It is specified with the –-with-lapack configure option.

- If the C++ compiler wraps MPI then no further MPI related information is needed by configure. If this is not the case, then information about MPI must be supplied:
    - –-with-mpi-include defines the location of mpi.h
    - –-with-mpi-libs is a space delimited list of the necessary MPI libraries (e.g. “nsl socket mpi”).
    - –-with-mpi-lib-dirs is a space delimited list of the directories containing the libraries specified via –-with-mpi-libs (e.g. “/usr/lib /usr/local/mpi/lib”).
    - –-with-mpi-flags is a space delimited list of any other flags necessary to link with MPI.

Some of the more pertinent options are:

- --enable-opt builds an optimized version of the code.
- --enable-debug builds a version of the code with symbols (-g) that may be used by a debugger. Note that this is orthogonal to –-enable-opt. The code may be built with both –-enable-opt and –-enable-debug.
- --enable-check-assertions turns on assertion checking throughout the library. The default is for this to be off. It is on when –enable-debug is specified. This should remain off for an optimized build of the library.
- You may need to specify –-with-doxygen if your doxygen executable is in a non-standard location.

4.2 Building the Library

The configure script generates a GNU Makefile. The command “make” builds the library according to this Makefile. The command “make all” builds the library and the tests. The command “make dox” builds the doxygen documentation.

4.3 Running the Tests

There are 3 tests and baseline results supplied with the library. All tests are parallel and should be launched with the appropriate command for your platform (mpirun, srun, etc.). Redirect results to a file and diff the file against the appropriate baseline. For example:

```
mpirun –np 3 ./smoke_test >& test_out
diff test_out BASELINES/smoke_3proc.out
```

4.3.1 smoke_test

This test is a very simple test of IncrementalSVDFastUpdate and IncrementalSVDSampler. It enables the option to print debugging information from the basis generation algorithm. The test samples 2 hard coded state vectors, generates the basis vectors, and prints the matrix of singular values and the basis vectors. The total dimension of the system is 6 and the state vector is evenly distributed among the processors. Therefore this test may only be run on 1, 2, 3, or 6 processors.

4.3.2 uneven_dist

This test is very similar to smoke_test except that the state vector is not evenly distributed among the processors. As with smoke_test the total dimension of the system is 6. Therefore this test may be run on no more than 6 processors.

4.3.3 random_test

This is a test of both the static and incremental fast update algorithms and samplers. It also enables the option to print debugging information from the algorithms. In addition to the singular values and state vectors it also prints the product of the transpose of the basis vectors from the static algorithm with the basis vectors from the incremental algorithm. This product should be a unitary matrix.

The total dimension of the system is 100 and the state vectors are evenly distributed among the processors. Therefore this test may be run on 1, 2, 4, 5, 10, 20, 25, 50, or 100 processors. There are 8 linearly independent state vectors and 2 that are linearly dependent. The state vectors are filled with randomly generated numbers but the numbers are generated in such a way that the distributed state vectors are identical for any number of processors. Therefore the singular values and state vectors should be the same for any number of processors except for differences arising from the parallel numerics.

5.0 Example Usage

The following 2 sections describe in general terms how to use libROM to create basis vectors from a simulation and how to use those basis vectors to create a reduced order model simulation.

5.1 Basis Vector Generation From Full Order Model Simulation

In order to generate basis vectors for a reduced order model from a full order simulation one must first construct the necessary SVDBasisGenerator. Then in the main simulation loop solution samples are taken as necessary. At the end of the main simulation loop the SVDBasisGenerator is told to end sampling. This example snippet demonstrates how this might be done.

```
// Solve du/dt = rhs
…

// Create the basis generator.
CAROM::IncrementalSVDBasisGenerator basis_generator(dim,
   linearity_tol,
   skip_linearly_dependent,
   do_fast_update,
   initial_dt,
   samples_per_time_interval,
   sampling_tol,
   max_time_between_samples,
   basis_file_name);

// Set initial conditions, etc.
…

// Simulation main loop.
while (simulation_not_done) {
   // Generate rhs.
   …

   // Sample the state vector if needed.
   if (basis_generator.isNextSample(simulation_time)) {
      basis_generator.takeSample(state_vector, simulation_time, dt);
      basis_generator.computeNextSampleTime(state_vector,
```

```
            rhs_vector,
            simulation_time);
    }

    // Advance the solution.
    …
}

// Tell basis generator that all samples have been collected.
basis_generator.endSampling();

// Remainder of full order simulation.
…
```

5.2 Reduced Order Model Construction and Simulation

The details of the construction of any specific reduced order model depend on the nature of the equation being solved and will therefore not be discussed here. This code snippet shows how one would use the BasisReader to obtain the basis vectors for different simulation times and use a reduced order model constructed from these vectors to execute a reduced order simulation.

```
// Construct the BasisReader and get the basis vectors for the initial
// simulation time.
CAROM::BasisReader reader(basis_file_name);
const CAROM::Mstrix* basis = reader.getBasis(simulation_time);

//Construct the reduced order model from the basis vectors.
…

// Set initial conditions on full order solution, u.
…

// Project the full order solution, u, into the reduced order solution,
// q, by q(i) = sum(j)(basis(j, i)*u(j)).
getProjectedSolution(basis, u, q);

// Reduced order model simulation main loop.
while (simulation_not_done) {
    // See if it is time to switch to a new set of basis vectors.
    if (reader.isNewBasis(simulation_time) {
        // Lift the current reduced order solution, q, back into the full
        // order solution, u, by u(i) = sum(j)(basis(i, j)*q(j)).
        getLiftedSolution(basis, q, u);

        // Get the basis vectors for this simulation time.
        delete basis;
        basis = reader.getBasis(simulation_time);

        // Construct the reduced order model from the new basis vectors.
        …

        // Project the full order solution, u, back into the reduced
        // order solution, q, by q(i) = sum(j)(basis(j, i)*u(j)).
        getProjectedSolution(basis, u, q);
    }

    // If it is time to write out the solution, lift the reduced order
    // solution, q, into the full order solution, u, and write it.
    if (time_to_write) {
        // Lift current reduced order solution, q, back into the full
```

```
      // order solution, u, by u(i) = sum(j)( basis(i, j)*q(j)).
      getLiftedSolution(basis, q, u);

      // Write u.
      …
   }

   // Advance the reduced order solution.
   …
}

// Remainder of reduced order simulation.
…

// Clean up.
delete basis;
…

void
getProjectedSolution(
   const Matrix* basis,
   const StateType& u,
   StateType& q)
{
   // Project the full order solution, u, back into the reduced order
   // solution, q, by q(i) = sum(j)(basis(j, i)*u(j)).
   for (int col = 0; col < basis->numColumns(); ++col) {
      double local_inner_product = 0.0;
      for (int row = 0; row < basis->numRows(); ++row) {
         local_inner_product += basis->item(row, col)*u(row);
      }
      double global_inner_product;
      MPI_Allreduce(&local_inner_product,
         &global_inner_product,
         1,
         MPI_DOUBLE,
         MPI_SUM,
         MPI_COMM_WORLD);
      q(col) = global_inner_product;
   }
}

void
getLiftedSolution(
   const Matrix* basis,
   const StateType& q,
   StateType& u)
{
   // Lift solution by u(i) = sum(j)( basis(i, j)*q(j)).
   for (int row = 0; row < basis->numRows(); ++row) {
      double local_inner_product = 0.0;
      for (int col = 0; col < basis->numColumns(); ++col) {
         local_inner_product += basis->item(row, col)*q(col);
      }
      double global_inner_product;
      MPI_Allreduce(&local_inner_product,
         &global_inner_product,
         1,
         MPI_DOUBLE,
         MPI_SUM,
         MPI_COMM_WORLD);
      u(row) = global_inner_product;
```

```
    }
}
```

6.0 Scaling

In this section we will present the results of a weak scaling study of the fast update algorithm. For this study each processor generated 10 state vectors each containing 10,000 random doubles. Each processor constructed a different set of random values. The time to take all 10 samples and to compute the next sample time was measured. These 2 operations are exactly what one would need to insert into the main loop of a simulation in order to construct the basis vectors. Scaling results for up to 8192 processors on sierra, an Intel Xeon EP X5660 based Linux cluster with 12 cores per node,1856 nodes, and an InfiniBand QDR interconnect were obtained The results are shown in Figure 1 below.

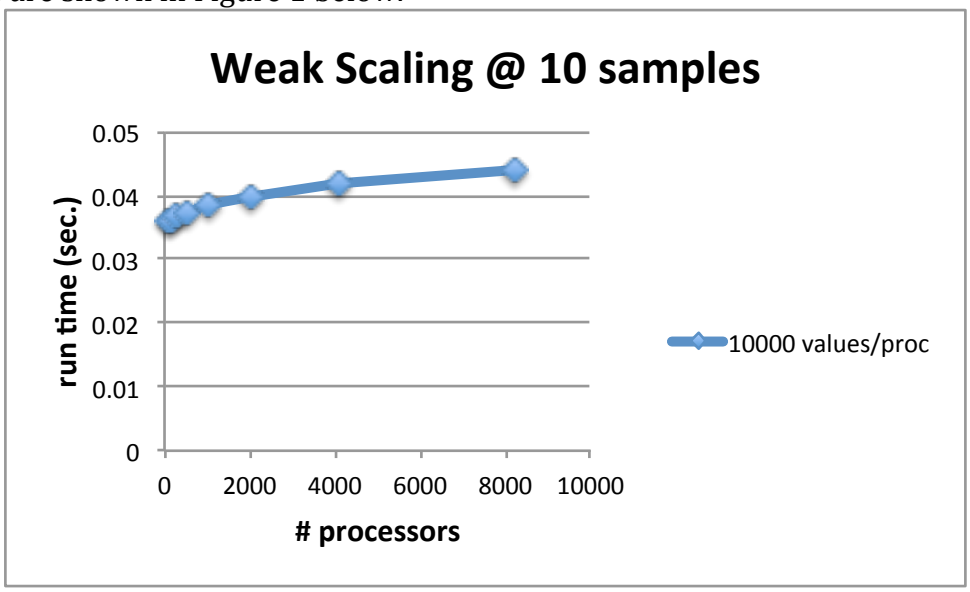

Figure 1

7.0 Application to Convection-Diffusion Simulation

In this section we will compare a full order simulation with a corresponding reduced order model simulation constructed from basis vectors generated by libROM. The simulation is of the convection-diffusion equation on a square. The governing equation is:

$$\frac{\partial u}{\partial t} = -a\frac{\partial u}{\partial x} - b\frac{\partial u}{\partial y} + v\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^y}\right)$$

In this example a=b=1.0 and v=0.1.

7.1 Full Order Solution

The full order solution was run for 200 time steps with a fixed dt of 0.001. A Dirichlet boundary condition of u=0 was used. The incremental fast update algorithm and incremental sampler with the following parameters were used to generate the basis vectors:

linearity_tol=1.0e-6
skip_linearly_dependent=false
samples_per_time_interval=200
max_time_between_samples=100.0 (unlimited)
The sampling tolerance was varied from 1.0e-1 to 1.0e-6. The solution was saved at t=0, t=0.04, t=0.08, t=0.12, t=0.16, and t=0.199. The solution at t=0.199 is shown in Figure 2.

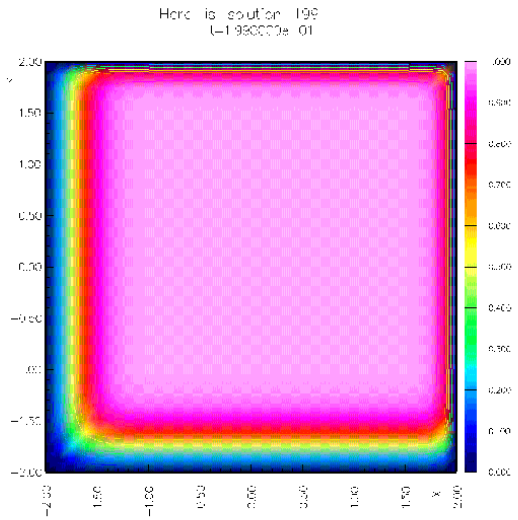

Figure 2

7.2 Reduced Order Solution

The reduced order solution was then constructed from the resulting basis vectors for each of the values of the sampling tolerance. The reduced order solution was saved at the same times as the full order solution. The L_2 and L_∞ norms were computed for the ensemble of saved solutions. The norms as a function of the sampling tolerance are presented in Figures 3 and 4 below.

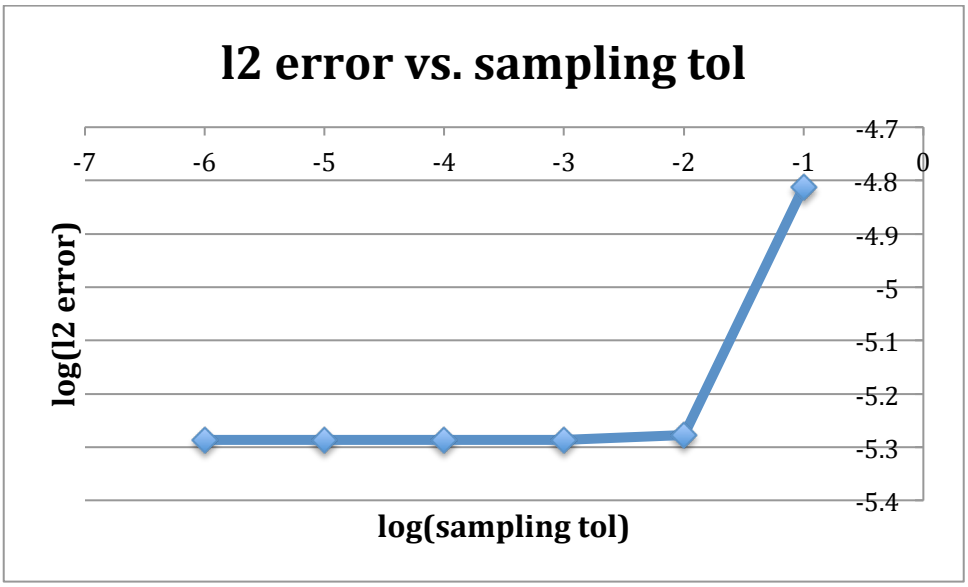

Figure 3

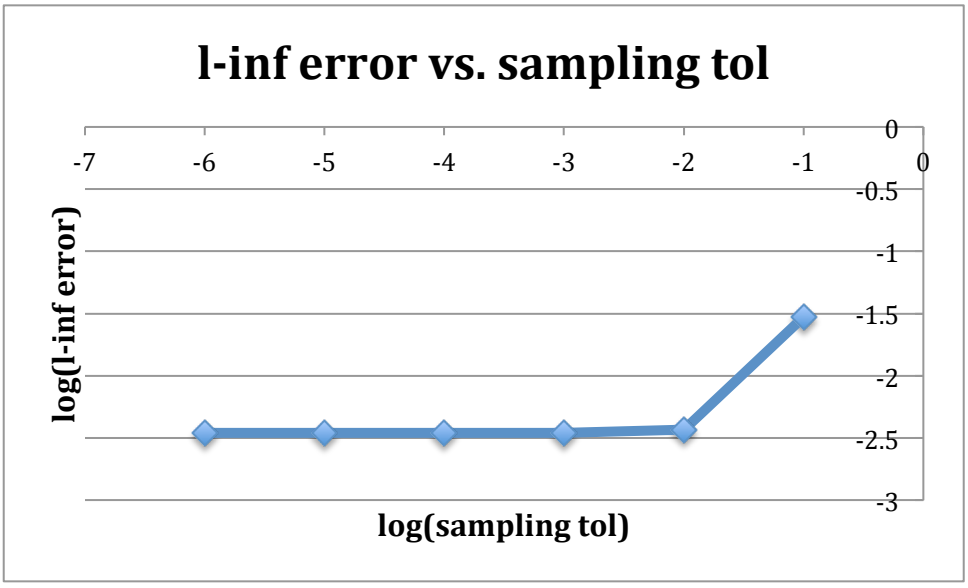

Figure 4

One can see that the asymptotic limit of the error is reached when the sampling tolerance is $\sim 10^{-2}$. Further decreasing the tolerance provides no improvement in the error.

The number of samples as a function of the sampling tolerance was also measured. In addition, the number of reorthogonalizations as a function of sampling tolerance was measured. These results are show in Figure 5.

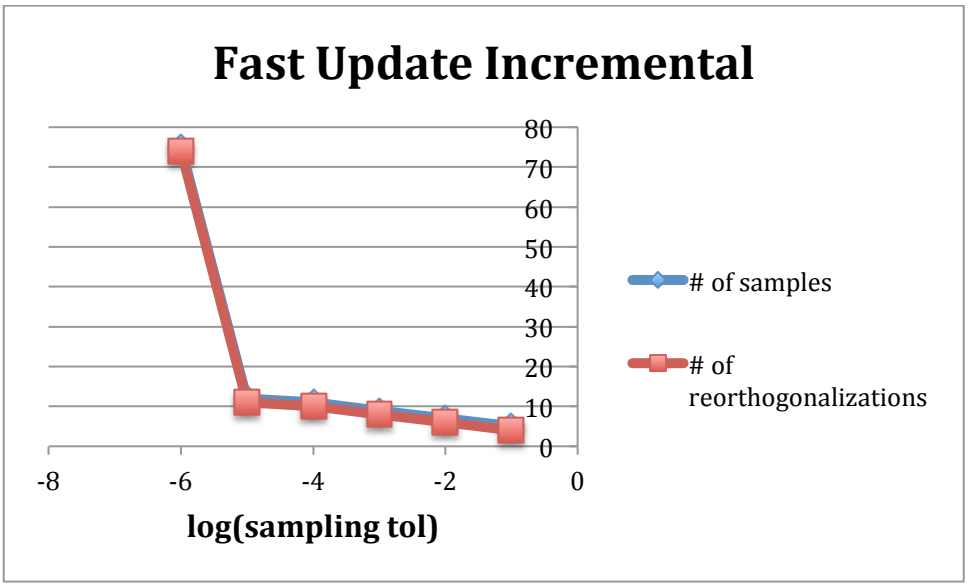

Figure 5

The number of reorthogonalizations is consistently one less than the number of samples. Only the first sample, which by definition does not incur a renormalization, is not renormalized.

For comparison, the standard incremental method was used with the same parameters as were used for the fast update method. There was no difference in the L_2 or L_∞ norms. In addition, the number of samples and number of reorthogonalizations was the same as for the fast update method. Hence, at least for this problem, there is no performance advantage to the fast update method when compared to the standard method.

References

[1] MATTHEW BRAND, *Incremental singular value decomposition of uncertain data with missing values,* in Computer VisionECCV 2002, Springer, 2002, pp. 707-720.