

ERC CISST Software

Quick start for cisstNumerical

Anton Deguet

Date: 2006/01/28 05:59:25

Contents

1	Introduction	2
2	An illustrative example: SVD	2
2.1	Using <code>nmrSVD</code> with user allocated containers	2
2.2	Using <code>nmrSVD</code> without specifying a workspace	4
2.3	Using <code>nmrSVDDynamicData::WorkspaceSize</code>	5
2.4	Using <code>nmrSVDDynamicData</code> to allocate everything	5
2.5	Using <code>nmrSVDDynamicData::UpdateMatrixS</code>	6
2.6	Using fixed size matrices without a data object	7
2.7	Using fixed size matrices with <code>nmrSVDFixedSizeData</code>	8
3	Functions with data object	8
3.1	FORTRAN based functions	8
3.1.1	<code>nmrInverse</code>	8
3.1.2	<code>nmrLU</code>	9
3.1.3	<code>nmrPInverse</code>	10
3.2	Native <code>cisst</code> functions	10
3.2.1	<code>nmrIsOrthonormal</code>	10
4	Others	11
4.1	<code>nmrGaussJordanInverse</code>	11
5	FORTRAN specifics	11
5.1	Compilation	11
5.2	Common properties	12

1 Introduction

These examples provide a quick introduction to the features of `cisstNumerical`. The code is part of the CVS repository and can be found in `cisst/examples/numericalTutorial`. To compile your own code, remember to include `cisstNumerical.h`.

`cisstNumerical` contains native functions as well as wrappers for existing numerical routines. As many algorithms can be found in FORTRAN, a significant part of `cisstNumerical` interfaces with FORTRAN routines. This has numerous consequences listed in section 5.

2 An illustrative example: SVD

The Singular Value Decomposition is a common algorithm and the `cisst` implementation illustrates many features of the `cisstNumerical` FORTRAN wrappers. The goal of SVD is to find the decomposition of a matrix A such as $A = U * \Sigma * V$ where both U and V are orthonormal and S is a diagonal matrix composed of singular values.

Most of the FORTRAN routines we are using will not allocate any memory nor check that the parameters are valid (see also section 5). Our wrappers not only check the size of the parameters to verify that enough memory has been allocated but they also provide some flexible mechanisms to allocate the required memory.

For the SVD, the underlying FORTRAN routine requires an input matrix A , two matrices to store U and Vt , a vector for the singular values S and a workspace for temporary variables (a.k.a. scratch space). From now on, we will call the two matrices U and Vt and the vector S the *output*.

Finally, since `cisstVector` supports both fixed size and dynamic vectors and matrices, `cisstNumerical` provides different wrappers and classes for each type of memory allocation. The examples we are providing illustrate different possible configurations, i.e. who allocates memory for what and how.

Example	Type	Output allocation	Workspace allocation
2.1	Dynamic	User, manually	
2.2	Dynamic	User, manually	<code>nmrSVD</code> function
2.3	Dynamic	User, manually	User with method <code>WorkspaceSize</code>
2.4	Dynamic	User with <code>nmrSVDDynamicData</code>	
2.6	Fixed size	User, manually	
2.7	Fixed size	User with <code>nmrSVDFixedSizeData</code>	

2.1 Using `nmrSVD` with user allocated containers

This example shows how to use the function `nmrSVD` with a dynamic matrix.

```

1 void ExampleSVDUserOutputWorkspace(void) {
2     const unsigned int size = 6;
3     // fill a matrix with random numbers

```

```

4     vctDynamicMatrix<double> A(size, size);
5     vctRandom(A, -10.0, 10.0);
6     // create matrices (U, Vt) and vector (S) for the result
7     vctDynamicMatrix<double> U(size, size);
8     vctDynamicMatrix<double> Vt(size, size);
9     vctDynamicVector<double> S(size);
10    // now, create a workspace of the right size
11    // we will explain later why this is needed
12    vctDynamicVector<double> workspace(size * 10);
13    // and we can finally call the nmrSVD function
14    // using a copy of A because nmrSVD modifies the input
15    vctDynamicMatrix<double> Acopy = A;
16    try {
17        nmrSVD(Acopy, U, S, Vt, workspace);
18    } catch(...) {
19        std::cout << "An exception occurred, check cisstLog.txt." << std::endl;
20    }
21    // display the result
22    std::cout << "U:\n" << U << "\nS:\n" << S << "\nV:\n"
23        << Vt.TransposeRef() << std::endl;
24 }
25
26 void ExampleSVDEconomyUserOutputWorkspace(void) {
27     const unsigned int sizerows = 20;
28     const unsigned int sizecols = 3;
29     // fill a matrix with random numbers
30     vctDynamicMatrix<double> A(sizerows, sizecols, VCT_COL_MAJOR);
31     vctRandom(A, -10.0, 10.0);
32     // create matrices (U, Vt) and vector (S) for the result
33     vctDynamicMatrix<double> U(sizerows, sizecols, VCT_COL_MAJOR);
34     vctDynamicMatrix<double> Vt(sizecols, sizecols, VCT_COL_MAJOR);
35     vctDynamicVector<double> S(sizecols);
36     // now, create a workspace of the right size
37     // we will explain later why this is needed
38     vctDynamicVector<double> workspace(sizerows * 10);
39     // and we can finally call the nmrSVD function
40     // using a copy of A because nmrSVD modifies the input
41     vctDynamicMatrix<double> Acopy = A;
42     try {
43         nmrSVDEconomy(Acopy, U, S, Vt, workspace);
44     } catch(...) {
45         std::cout << "An exception occurred, check cisstLog.txt." << std::endl;
46     }
47     // display the result
48     std::cout << "A:\n" << A
49         << "\nUeconomy:\n" << U << "\nS:\n" << S << "\nV:\n"
50         << Vt.TransposeRef() << std::endl;
51 }

```

In this example, we have used a workspace 10 times bigger than the initial matrix which is large enough. Since the size of the workspace can be determined automatically, cisstNumer-

ical also provides an overloaded version of `nmrSVD` which doesn't require a workspace.

2.2 Using `nmrSVD` without specifying a workspace

```

1 void ExampleSVDImplicitWorkspace(void) {
2     const unsigned int size = 6;
3     // fill a matrix with random numbers
4     vctDynamicMatrix<double> A(size, size);
5     vctRandom(A, -10.0, 10.0);
6     // create matrices (U, Vt) and vector (S) for the result
7     vctDynamicMatrix<double> U(size, size);
8     vctDynamicMatrix<double> Vt(size, size);
9     vctDynamicVector<double> S(size);
10    // and we can finally call the nmrSVD function
11    // using a copy of A because nmrSVD modifies the input
12    vctDynamicMatrix<double> Acopy = A;
13    try {
14        nmrSVD(Acopy, U, S, Vt);
15    } catch(...) {
16        std::cout << "An exception occurred, check cisstLog.txt." << std::endl;
17    }
18 }
19
20 void ExampleSVDEconomyImplicitWorkspace(void) {
21     const unsigned int sizerows = 20;
22     const unsigned int sizecols = 3;
23     // fill a matrix with random numbers
24     vctDynamicMatrix<double> A(sizerows, sizecols, VCT_COL_MAJOR);
25     vctRandom(A, -10.0, 10.0);
26     // create matrices (U, Vt) and vector (S) for the result
27     vctDynamicMatrix<double> U(sizerows, sizecols, VCT_COL_MAJOR);
28     vctDynamicMatrix<double> Vt(sizecols, sizecols, VCT_COL_MAJOR);
29     vctDynamicVector<double> S(sizecols);
30     // and we can finally call the nmrSVD function
31     // using a copy of A because nmrSVD modifies the input
32     vctDynamicMatrix<double> Acopy = A;
33     try {
34         nmrSVDEconomy(Acopy, U, S, Vt);
35     } catch(...) {
36         std::cout << "An exception occurred, check cisstLog.txt." << std::endl;
37     }
38 }

```

This is easier to use, but one has to remember that a workspace is created dynamically by `nmrSVD`, i.e. every time the function is called some memory is allocated and released.

This behavior might not suit everyone, therefore `cisstNumerical` provides a couple of methods to ease the allocation of the workspace and the output matrices and vectors. All these methods are declared within the scope of a class called “data”. For SVD, we have two different classes available, `nmrSVDDynamicData` and `nmrSVDFixedSizeData`.

2.3 Using `nmrSVDDynamicData::WorkspaceSize`

```

1 void ExampleSVDWorkspaceSize(void) {
2     const unsigned int size = 6;
3     // create the input matrix with the correct size
4     vctDynamicMatrix<double> A(size, size);
5     // now, create a workspace of the right size
6     vctDynamicVector<double> workspace;
7     workspace.SetSize(nmrSVDDynamicData::WorkspaceSize(A));
8     // Allocate U, Vt, S and use the workspace for nmrSVD ...
9 }
10
11 void ExampleSVDEconomyWorkspaceSize(void) {
12     const unsigned int sizerows = 20;
13     const unsigned int sizecols = 6;
14     // create the input matrix with the correct size
15     vctDynamicMatrix<double> A(sizerows, sizecols);
16     // now, create a workspace of the right size
17     vctDynamicVector<double> workspace;
18     workspace.SetSize(nmrSVDEconomyDynamicData::WorkspaceSize(A));
19     // Allocate U, Vt, S and use the workspace for nmrSVD ...
20 }

```

This method simplifies the allocation of the workspace but doesn't solve another problem that we have ignored so far: If the input matrix is not square, the size of the different output containers are a bit trickier to determine, i.e. if the input matrix is $m \times n$ then U should be $m \times m$, Vt $n \times n$ and the vector S should be $\min(m, n)$ long. This is not awfully difficult but still requires some extra attention from the caller. To facilitate the user's work, it is possible the use the class `nmrSVDDynamicData` to allocate not only the workspace but the output as well.

2.4 Using `nmrSVDDynamicData` to allocate everything

```

1 void ExampleSVDDynamicData(void) {
2     // fill a matrix with random numbers
3     vctDynamicMatrix<double> A(10, 3, VCT_COL_MAJOR);
4     vctRandom(A, -10.0, 10.0);
5     // create a data object
6     nmrSVDDynamicData svdData(A);
7     // and we can finally call the nmrSVD function
8     vctDynamicMatrix<double> Acopy = A;
9     nmrSVD(Acopy, svdData);
10    // display the result
11    std::cout << "A:\n" << A
12              << "\nU:\n" << svdData.U()
13              << "\nS:\n" << svdData.S()
14              << "\nV:\n" << svdData.Vt().TransposeRef() << std::endl;
15 }
16

```


2.7 Using fixed size matrices with `nmrSVDFixedSizeData`

```

1 void ExampleSVDFixedSizeData(void) {
2     // fill a matrix with random numbers
3     vctFixedSizeMatrix<double, 5, 7, VCT_COL_MAJOR> A, Acopy;
4     vctRandom(A, -10.0, 10.0);
5     Acopy = A;
6     // create a data object
7     typedef nmrSVDFixedSizeData<5, 7, VCT_COL_MAJOR> SVDDataType;
8     SVDDataType svdData;
9     // and we can finally call the nmrSVD function
10    nmrSVD(Acopy, svdData);
11    // compute the matrix S
12    SVDDataType::MatrixTypeS S;
13    SVDDataType::UpdateMatrixS(svdData.S(), S);
14    // display the initial matrix as well as U * S * V
15    std::cout << "A:\n" << A
16              << "\nU_*_*S_*_*Vt:\n"
17              << svdData.U() * S * svdData.Vt() << std::endl;
18 }

```

The interface of the `nmrSVDFixedSizeData` is pretty much the same as `nmrSVDDynamicData` except that the size and storage order are now specified using template parameters. To simplify our example, we introduced a type for `DataType`. This approach is strongly recommended whenever one uses the `cisst` fixed size vectors and matrices.

3 Functions with data object

Besides the function `nmrSVD`, `cisstNumerical` includes more numerical functions which can be used with either a data object or some vectors and matrices provided by the caller.

3.1 FORTRAN based functions

The `cisstNumerical` FORTRAN wrappers are all written using the approach used for `nmrSVD` and share the different properties listed in section 5.

3.1.1 `nmrInverse`

This function computes the inverse of a matrix using an LU decomposition. It can be used for dynamic and fixed size matrices of any storage order. Nevertheless, for fixed size matrices of size 2, 3 or 4, we recommend to use `nmrGaussJordanInverse` (see 4.1).

```

1 void ExampleInverse(void) {
2     // Start with a fixed size matrix
3     vctFixedSizeMatrix<double, 6, 6> A, AInverse;
4     // Fill with random values
5     vctRandom(A, -10.0, 10.0);

```

```

6     AInverse = A;
7     // Compute inverse and check result
8     nmrInverse(AInverse);
9     std::cout << A * AInverse << std::endl;
10
11    // Continue with a dynamic matrix
12    vctDynamicMatrix<double> B, BInverse;
13    // Fill with random values
14    B.SetSize(8, 8, VCT_COL_MAJOR);
15    vctRandom(B, -10.0, 10.0);
16    BInverse = B;
17    // Compute inverse and check result
18    nmrInverse(BInverse);
19    std::cout << B * BInverse << std::endl;
20 }

```

In this example, we used the overloaded version of `nmrInverse` which doesn't require a data object. This is possible since the data object doesn't provide any useful information or result. As for most wrappers, using the function `nmrInverse` without a data object is not optimal if the function is going to be called multiple times. To optimize the memory allocation, one should use `nmrInverseFixedSizeData` or `nmrInverseDynamicData`.

3.1.2 nmrLU

The goal of LU is to find the factorization of a matrix A such as $A = L * U$ where U is an upper matrix and V is a lower matrix.

```

1 void ExampleLUDynamicData(void) {
2     // fill a matrix with random numbers
3     vctDynamicMatrix<double> A(5, 7, VCT_COL_MAJOR);
4     vctRandom(A, -10.0, 10.0);
5     // create a data object
6     nmrLUDynamicData luData(A);
7     // and we can finally call the nmrLU function
8     vctDynamicMatrix<double> Acopy = A;
9     nmrLU(Acopy, luData);
10    // LAPACK routine store the LU in input A and use
11    // a vector to store the permutations P
12    vctDynamicMatrix<double> P, L, U;
13    P.SetSize(nmrLUDynamicData::MatrixPSize(A));
14    L.SetSize(nmrLUDynamicData::MatrixLSize(A));
15    U.SetSize(nmrLUDynamicData::MatrixUSize(A));
16    nmrLUDynamicData::UpdateMatrixP(Acopy, luData.PivotIndices(), P);
17    nmrLUDynamicData::UpdateMatrixLU(Acopy, L, U);
18    std::cout << "A:\n" << A
19              << "\nP□□L□□U:\n" << (P * L * U) << std::endl;
20 }

```

It is important to notice that in this example we explicitly created the input using `VCT_COL_MAJOR`. As it is, `nmrLU` doesn't support the row first storage order.

Besides this constraint, the LU decomposition routine provided by LAPACK stores the result in one single matrix, replacing the input. This is perfectly good for most applications but one can also use the helper methods of `nmrLUDynamicData` to determine the size and compute the matrices P, L and U.

3.1.3 `nmrPInverse`

This function actually relies on `nmrSVD`. The corresponding data object `nmrPInverseDynamicData` and `nmrPInverseFixedSizeData` allocate a workspace large enough for `nmrSVD`.

3.2 Native *cisst* functions

The *cisst* native functions are more flexible than the FORTRAN wrappers mostly because the restrictions regarding the storage order and the compactness are lifted. The elements might also be different, i.e. one can use single precision floating point numbers if this makes any sense for his/her application.

3.2.1 `nmrIsOrthonormal`

In this example, we are using `nmrSVD` to create a couple of orthonormal matrices.

```

1 void ExampleIsOrthonormal(void) {
2     // fill a matrix with random numbers
3     vctDynamicMatrix<double> A(5, 7);
4     vctRandom(A, -10.0, 10.0);
5     // create a workspace and use it for the SVD data
6     vctDynamicVector<double>
7         workspace(nmrSVDDynamicData::WorkspaceSize(A));
8     nmrSVDDynamicData svdData(A, workspace);
9     // we can call the nmrSVD function
10    vctDynamicMatrix<double> Acopy = A;
11    nmrSVD(Acopy, svdData);
12    // check that the output is correct using our workspace
13    if (nmrIsOrthonormal(svdData.U(), workspace)) {
14        std::cout << "U is orthonormal" << std::endl;
15    }
16    // same with dynamic creation of a workspace
17    if (nmrIsOrthonormal(svdData.Vt())) {
18        std::cout << "Vt is orthonormal" << std::endl;
19    }
20 }
```

This example demonstrates two different ways to use the function `nmrIsOrthonormal`, one with a user defined workspace and one with no workspace at all (i.e. the function will allocate and free memory on the fly).

Please note in this example how we created a single workspace used by different routines. This is very convenient to avoid any unnecessary memory allocation but one must make sure that this workspace is not being used by two different threads.

It is also possible to create a data object for this problem (see `nmrIsOrthonormalDynamicData` and `nmrIsOrthonormalFixedSizeData`).

4 Others

4.1 nmrGaussJordanInverse

The Gauss Jordan inverse methods are implemented for fixed size matrices 2×2 , 3×3 and 4×4 . These functions, compiled in release mode, are faster than their FORTRAN counterparts.

```

1 void ExampleGaussJordanInverse(void) {
2     vctFixedSizeMatrix<double, 4, 4> A, AInverse;
3     vctRandom(A, -10.0, 10.0);
4     bool nonSingular;
5     double tolerance = 10E-6;
6     // call nmrGaussJordanInverse4x4
7     nmrGaussJordanInverse4x4(A, nonSingular, AInverse, tolerance);
8     if (nonSingular) {
9         std::cout << "A * AInverse:\n" << A * AInverse << std::endl;
10    } else {
11        std::cout << "A is a singular matrix" << std::endl;
12    }
13 }

```

Note that since these functions are fully implemented using the `cisst` package, any storage order or stride can be used (i.e. the matrices don't need to be compact).

5 FORTRAN specifics

5.1 Compilation

Most of the FORTRAN routines we are using come from the on-line code repository netlib.org. There is no standard binary distribution of these routines therefore we decided to provide a binary version of these routines (library and header files). We have two different versions for historical reasons:

- CNetlib: This is the oldest version, soon to be deprecated. The main default of this version is that it is not thread safe.
- `cisstNetlib`: This version is based on LAPACK3E routines and is thread safe. We strongly recommend to use this version.

The `cisstNumerical` API is the same (i.e. your code will be the same) for both binary distributions but you will need to configure your build differently using CMake: You will have to activate either `CISST_HAS_CISSTNETLIB` or `CISST_HAS_CNETLIB`.

For more details and to download these libraries, see www.cisst.org/cnetlib .

5.2 Common properties

All our wrappers for FORTRAN routines share the following properties:

1. The default storage order for matrices is column first in FORTRAN while it is row first in C/C++. Since `cisstVector` supports both formats, the user has to remember to create his matrices column first (using `VCT_COL_MAJOR`).

This is the default but there are some exceptions. For example, `nmrSVD` can be used with any storage order. In the case, `cisstNumerical` uses the fact that changing the storage order is similar to a transpose. For SVD, the problem $A^t = (U * \Sigma * V)^t = V^t * \Sigma^t * U^t$ is basically the same as $A = U * \Sigma * T$. Whenever this is possible, classes and functions of `cisstNumerical` are implemented to support both storage orders (either row or column major).

2. Most FORTRAN routines were not written with the concept of stride in mind. This means that all matrices and vectors which are finally used by a FORTRAN routine must be compact (i.e. use a contiguous block of memory).
3. Most LAPACK routines, will modify the input to avoid unnecessary memory allocation. Since `cisstNumerical` has been designed to avoid implicit memory allocation and copies as well, it is the caller's responsibility to create a copy of the input for future use.
4. These functions can only operate on matrices and vectors of `doubles`. This is because this function is actually a wrapper to a LAPACK routine which requires double precision floating point numbers. For the integers (e.g. vector of pivot indices), FORTRAN uses the equivalent of a C/C++ `long int`. To enforce this and remind the caller of this subtlety, the `cisstNumerical` interface defines and uses `F_INTEGER`.
5. If the matrices or vectors provided by the user are not correct (size, storage order, compact), an exception will occur (`std::runtime_error`). Since these exceptions are logged, the user might want to look at the file `cisstLog.txt` if his/her application quits unexpectedly.