MATLAB test suite for SuiteSparse:GraphBLAS

Timothy A. Davis
davis@tamu.edu, Texas A&M University.
http://www.suitesparse.com and http://aldenmath.com

June 20, 2020

SuiteSparse:GraphBLAS includes a MATLAB implementation of nearly the entire GraphBLAS specification, including all built-in types and operators. The typecasting rules and integer operator rules from GraphBLAS are implemented in MATLAB via `mexFunctions` that call the GraphBLAS routines in C. All other functions are written purely in MATLAB `M`-files, and are given names of the form `GB_spec_*`. All of these MATLAB interfaces and `M`-file functions they are provided in the software distribution of SuiteSparse:GraphBLAS. The purpose of this is two-fold:

- **Illustration and documentation:** MATLAB is so expressive, and so beautiful to read and write, that the `GB_spec_*` functions read almost like the exact specifications from the GraphBLAS API. Excerpts and condensed versions of these functions appear in the User Guide. The reader can benefit from studying the `GB_spec_*` functions to understand what a GraphBLAS operation is computing. For example, in the User Guide, `GrB_mxm` includes a condensed and simplified version of `GB_spec_mxm`.

- **Testing:** Testing the C interface to SuiteSparse:GraphBLAS is a significant challenge since it supports so many different kinds of operations on a vast range of semirings. It is difficult to tell from looking at the result from a C function in GraphBLAS if the result is correct. Thus, each function has been written twice: once in a highly-optimized function in C, and again in a simple and elegant MATLAB function. The latter is almost a direct translation of all the mathematics behind the GraphBLAS API, so it is much easier to visually inspect the GB_spec_* version in MATLAB to ensure the correct mathematics are being computed.

The following functions are included in the SuiteSparse:GraphBLAS software distribution. Each has a name of the form GB_spec_*, and each of them is a "mimic" of a corresponding C function in GraphBLAS. Not all functions in the C API have a corresponding mimic; in particular, many of the vector functions can be computed directly with the corresponding matrix version in the MATLAB implementations. A list of these files is shown below:

MATLAB GB_spec function	corresponding GraphBLAS function or method
GB_spec_accum.m	$\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$
GB_spec_mask.m	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$
GB_spec_accum_mask.m	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$
GB_spec_Vector_extractElement.m	GrB_Vector_extractElement
GB_spec_build.m	GrB_Matrix_build
GB_spec_Matrix_extractElement.m	GrB_Matrix_extractElement
GB_spec_extractTuples.m	GrB_Matrix_extractTuples
GB_spec_mxm.m	GrB_mxm
GB_spec_vxm.m	GrB_vxm
GB_spec_mxv.m	GrB_mxv
GB_spec_Vector_eWiseMult.m	GrB_Vector_eWiseMult
GB_spec_Matrix_eWiseMult.m	GrB_Matrix_eWiseMult
GB_spec_Vector_eWiseAdd.m	GrB_Vector_eWiseAdd
GB_spec_Matrix_eWiseAdd.m	GrB_Matrix_eWiseAdd
GB_spec_Vector_extract.m	GrB_Vector_extract
GB_spec_Matrix_extract.m	GrB_Matrix_extract
GB_spec_Col_extract.m	GrB_Col_extract
GB_spec_subassign.m	GxB_subassign
GB_spec_assign.m	GrB_assign
GB_spec_apply.m	GrB_apply
GB_spec_select.m	GxB_select
GB_spec_reduce_to_vector.m	GrB_reduce (to vector)
GB_spec_reduce_to_scalar.m	GrB_reduce (to scalar)
GB_spec_transpose.m	GrB_transpose
GB_spec_kron.m	GrB_kronecker

Additional files are included for creating test problems and providing inputs to the above files, or supporting functions:

MATLAB `GB_spec` function	purpose
`GB_spec_compare.m`	Compares output of C and MATLAB functions
`GB_spec_random.m`	Generates a random matrix
`GB_spec_op.m`	MATLAB mimic of built-in operators
`GB_spec_operator.m`	Like `GrB_*Op_new`
`GB_spec_opsall.m`	List operators, types, and semirings
`GB_spec_semiring.m`	Like `GrB_Semiring_new`
`GB_spec_descriptor.m`	mimics a GraphBLAS descriptor
`GB_spec_identity.m`	returns the identity of a monoid
`GB_spec_matrix.m`	conforms a MATLAB sparse matrix to GraphBLAS
`GB_define.m`	creates draft of `GraphBLAS.h`

An intensive test suite has been written that generates test graphs in MATLAB, then computes the result in both the C version of the SuiteSparse:GraphBLAS and in the MATLAB `GB_spec_*` functions. Each C function in GraphBLAS has a direct `mexFunction` interface that allow the test suite in MATLAB to call both functions.

This approach has its limitations:

- **matrix classes:** MATLAB only supports sparse double, sparse double complex, and sparse logical matrices. MATLAB can represent dense matrices in all 13 built-in GraphBLAS data types, so in all these specification `M`-files, the matrices are either in dense format in the corresponding MATLAB class, or they are held as sparse double or sparse logical, and the actual GraphBLAS type is held with it as a string member of a MATLAB `struct`. To ensure the correct typecasting is computed, most of the MATLAB scripts work on dense matrices, not sparse ones. As a result, the MATLAB `GB_spec_*` function are not meant for production use, but just for testing and illustration.

- **integer operations:** MATLAB and GraphBLAS handle integer operations differently. In MATLAB, an integer result outside the range of the integer is set to maximum or minimum integer. For example, `int8(127)+1` is `127`. This is useful for many computations such as image processing, but GraphBLAS follows the C rules instead, where integer values wrap, modulo style. For example, in GraphBLAS and in C, incrementing `(int8_t) 127` by one results in `-128`. Of course, an alternative would be for a MATLAB interface to create its own integer operators, each of which would follow the MATLAB integer rules of arithmetic. However, this would obscure the purpose of these `GB_spec_*` and `GB_mex_*` test functions, which is to test the C API of

GraphBLAS. When the `GB_spec_*` functions need to perform integer computations and typecasting, they call GraphBLAS to do the work, instead doing the work in MATLAB. This ensures that the `GB_spec_*` functions obtain the same results as their GraphBLAS counterparts.

- **elegance:** to simplify testing, each MATLAB `mexFunction` interface a GraphBLAS function is a direct translation of the C API. For example, `GB_mex_mxm` is a direct interface to the GraphBLAS `GrB_mxm`, even down the order of parameters. This approach abandons some of the potential features of MATLAB for creating elegant `M`-file interfaces in a highly usable form, such as the ability to provide fewer parameters when optional parameters are not in use. These `mexFunctions`, as written, are not meant to be usable in a user application. They are not highly documented. They are meant to be fast, and direct, to accomplish the goal of testing SuiteSparse:GraphBLAS in MATLAB and comparing their results with the corresponding `GB_spec_*` function. They are not recommended for use in general applications in MATLAB.

- **generality:** the MATLAB `mexFunction` interface needs to test the C API directly, so it must access content of SuiteSparse:GraphBLAS objects that are normally opaque to an end user application. As a result, these `mexFunctions` do not serve as a general interface to any conforming GraphBLAS implementation, but only to SuiteSparse:GraphBLAS.

In the MATLAB mimic functions, `GB_spec_*`, a GraphBLAS matrix `A` is represented as a MATLAB `struct` with the following components:

- `A.matrix`: the values of the matrix. If `A.matrix` is a sparse double matrix, it holds a typecasted copy of the values of a GraphBLAS matrix, unless the GraphBLAS matrix is also double (`GrB_FP64`).

- `A.pattern`: a logical matrix holding the pattern; `A.pattern(i,j)=true` if `(i,j)` is in the pattern of `A`, and `false` otherwise.

- `A.class`: the MATLAB class of the matrix corresponding to one of the 13 built-in types. Normally this is simply `class(A.matrix)`.

- `A.values`: most of the GraphBLAS test `mexFunctions` return their result as a MATLAB sparse matrix, in the `double` class. This works well for all types except for the 64-bit integer types, since a double has about 54 bits of mantissa which is less than the 64 bits available in a long integer. To ensure no bits are lots, these values are also returned as a vector. This enables `GB_spec_compare` to ensure the test results are identical down to the very last bit, and not just to within roundoff error. Nearly all tests, even in double precision, check for perfect equality, not just for results accurate to within round-off error.