SPIKE Documentation v1.0

Braegan Spring

October 31, 2018

Contents

1 Intro **1**

2 Compilation **2**

3 Environment Variables **3**

4 Examples **4**

5 Usage **7**
5.1 SPIKE parameter array entries . . . 7
5.2 A quick(-ish) note about partition sizes and spm . . . 9
5.3 Subroutine Listing . . . 9
5.3.1 spikeinit . . . 9
5.3.2 Xspike_tune . . . 10
5.3.3 Xspike_gbsv . . . 10
5.3.4 Xspike_gbtrf . . . 10
5.3.5 Xspike_gbtrs . . . 10
5.3.6 Xspike_gbtrsi . . . 10
5.3.7 Xspike_gbtrfp . . . 10
5.3.8 Xspike_gbtrsp . . . 11
5.4 Table of variables . . . 11
5.5 SPM entries . . . 12

A More Examples **13**
A.1 Split Factorize & Solve . . . 13
A.2 Split Factorize & Solve With Pivoting . . . 14
A.3 Solve With Iterative Refinement . . . 15
A.4 Transpose Solve . . . 16
A.5 C example . . . 17

1 Intro

This implementation of SPIKE v1.0 is intended for use with banded matrices, on shared memory machines. Overall we have attempted to follow the calling conventions of the LAPACK banded solvers as closely as possible. However, SPIKE needs additional arguments to describe the partitioning configuration, and some extra work space.

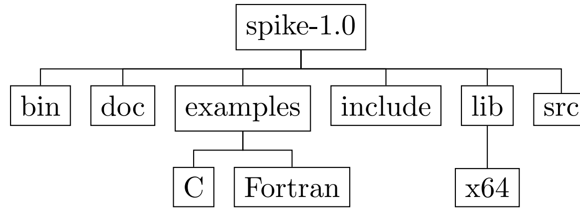

Figure 1: SPIKE directory tree

- Xspike_gbsv mimics Xgbsv (Combined factorize and solve)
- Xspike_gbtrf and Xspike_gbtrfp mimic Xgbtrf (Factorize)
- Xspike_gbtrs and Xspike_gbtrsp mimic Xgbtrs (Solve)

Where X may be D for double precision, Z for double complex, S for single precision (floating point), and C for single precision complex. There are also the subroutines Xspike_gbtrsi, which mimic Xgdbtrs but perform iterative refinement.

2 Compilation

The following instructions apply to a Linux/Unix environment.

1. Obtain the SPIKE package. It may be found at HTTP://WWW.SPIKE-SOLVER.ORG.
2. Put the package in your desired directory.
3. Extract the package: `tar -xzvf spike-1.0.tar.gz`. This should produce the file hierarchy shown in figure 1.
4. Navigate to the `src` directory to build the project.
   (a) Review the included `make.inc` file and make any necessary changes.
      - SPIKE is written primarily in Fortran90, with some Fortran77. As such, it should be possible to build the library with most Fortran compilers. `gfortran` and `ifort` have been tested and examples exist for both. However, this method of building will impose Fortran runtime dependencies. For example, if SPIKE is compiled using `ifort`, and the user code is compiled with a `C` compiler or `gfortran`, it will be necessary to link the Fortran runtime with `-lifcoremt`. Similarly, if `gfortran` is used to compile the library, any other compilers which wish to call the library will have to include `-lgfortran`. This is 'option 1' in `make.inc`.
      - If a more portable library is desired, it is possible to replace the Fortran runtime dependencies with included C functions. Currently, this option is only implemented with the Intel Fortran compiler. This is 'option 2' in `make.inc`.

   (b) Issue the make command. The build system is compatible with a parallel build process. Select one of the following:
      - The command `make -j all` may be issued to build the library with default options. The resulting library, `libspike.a`, will be placed in the directory `spike-1.0/lib/x64`.

- The variables ARCH and INSTALLDIR may also be passed to the makefile. The library will be placed in the directory <INSTALLDIR>/<ARCH>. For example issuing the command, `make -j all ARCH=x64 INSTALLDIR=../` will produce the library in the default directory.

3 Environment Variables

This implementation of SPIKE uses OpenMP and the system implementation of LAPACK. As a result, it should respond to the environment variables of both systems.

Specifically, when using ifort and MKL:

- OMP_NUM_THREADS
  Sets the number of threads that SPIKE can use. Any number of threads will be accepted.

- KMP_AFFINITY
  OpenMP affinity settings may increase performance. SPIKE is a domain decomposition algorithm, and this implementation attempts to match partitions to OpenMP threads where possible. So, locality may be increased when OpenMP threads are mapped consistently to hardware threads. Additionally, hyperthreads have been found to be slightly performance degrading for SPIKE. On a system with two hyperthreads per core, we have used the following setting to avoid hyperthreads:
  `KMP_AFFINITY=granularity=fine,compact,1,0`
  This sets a given pair of hyperthreads to be maximally 'far' from each other. When combined with `compact` this makes the OpenMP runtime avoid using both hyperthreads at the same time.

- MKL_NUM_THREADS & MKL_DYNAMIC
  For operations on each partition, SPIKE uses some LAPACK routines as well as the included 'lbprim' banded primitives. The latter call the BLAS routines.

  Although there are many LAPACK implementations, the examples provided assume that the popular Intel MKL version of the library. Because MKL itself also provides parallel functionality, it is possible to implement a hybrid configuration where each partition can be associated with multiple MKL threads. By default, MKL_DYNAMIC is set to TRUE, which means that MKL-LAPACK and MKL-BLAS will avoid using threading inside each partition.

  To override this default value and force nested openmp parallelism MKL_DYNAMIC must be set to FALSE. In general, it is more useful to allocate threads to SPIKE when the matrix has a narrow band, while MKL becomes more competitive when the band is wider. When using nested parallelism, MKL_NUM_THREADS×OMP_NUM_THREADS should be equal to the total number of threads you wish to use.

  In general, combining SPIKE and MKL parallelism can be tricky, and we haven't explored it fully.

4 Examples

A minimal Fortran90 'hello world' example is shown here. Additional examples (including a C example) are provided in appendix A. More detailed examples are distributed with the package.

If the following code is placed in a file called `example.f90`, and that file is placed in the root SPIKE directory, `spike-1.0`, it should be possible to compile the example by issuing the command:

```
ifort example.f90 -qopenmp -I$MKLROOT/include -Llib/x64/ -lspike -mkl -o example.exe
```

```
program main

integer :: info
integer, parameter :: n=600000, kl=1, ku=1, nrhs=1
double precision, dimension(kl+ku+1,n) :: A,oA
double precision, dimension(n,nrhs) :: f,of
integer, dimension(64) :: spm

! Define matrix and right-hand-side
A = -1.0d0
A(ku+1,:) = 4.0d0
f(:,1) = 1.0d0
oA=A
of=f

call spikeinit(spm,n,max(kl,ku)) ! Initialize spm array

spm(1) = 1 ! Instructs SPIKE to print timing and partition information

call dspike_gbsv(spm,n,kl,ku,nrhs,A,kl+ku+1,f,n,info)

! Check the residual
call dgbmv("N",n,n,kl,ku,1.0d0,oA,kl+ku+1,f(:,1),1,-1.0d0,of(:,1),1)
print *, "Max residual is: ", maxval(abs(of(:,1)))

end program
```

The resulting outputs using a multi-core processor are provided in Figure 2 and Figure 3 using 2 and 4 threads, respectively. The number of threads can be selected before running the code. In BASH shell, we can use the following command with n=2,4

```
>export OMP_NUM_THREADS=n
```

As mentioned above, a more detailed set of examples is distributed with the package. It may be found in the `examples` subdirectory of the `spike-1.0` directory. There are examples for Fortran using double precision, and C using double precision as well as single-complex precision. The directories are structured as `examples/<programming language>`. Each example directory contains a number of files:

- One per precision to show the use of the 'gbsv' functionality

```
**********************************************
*********** SPIKE-SMP -BEGIN *****************
**********************************************
List of input parameters spm(1:64)-- if different from default
   spm(1)=1
Size system     600000
kl,ku           1       1
#Threads (Total) available           2
2 Partitions and 2 Threads
Partition ratios: 1.800000000000000e+00 3.500000000000000e+00
#Partitions with 1 and 2 threads
2       0
Partition sizes: 300000 300000
---------------------
|Factorization Time|
------------------------------
|   LU or SPIKE on blocks    |
--------------------------------------------------
| Thread |   Partition   |          Time         |
--------------------------------------------------
|    1   |       1       | 1.384186744689941e-02 |
|    2   |       2       | 1.330804824829102e-02 |
--------------------------------------------------
|Blocks Factorize |       1.385617256164551e-02  |
|Reduced System   |       3.123283386230469e-05  |
|Overall Factorize|       1.390504837036133e-02  |
--------------------------------------------------
------------
|Solve Time|
------------------------------
|  Solve Sweeps on blocks    |---------------------------------------
| Thread | Partition |         Time 1        |          Time 2       |
|    1   |     1     | 1.222529411315918e-01 | 1.232550144195557e-01 |
|    2   |     2     | 1.252839565277100e-01 | 1.235320568084717e-01 |
----------------------------------------------------------------------
|Blocks Solve        | 1.252870559692383e-01 | 1.235339641571045e-01 |
|Reduced System Solve| 7.295608520507812e-05                         |
|Overall Solve       | 2.488939762115479e-01                         |
----------------------------------------------------------------------
  Max residual is :   2.775557561562891E-016
```

Figure 2: Output of helloworld example using 2 threads.

```
*********************************************
*********** SPIKE-SMP -BEGIN *****************
*********************************************
List of input parameters spm(1:64)-- if different from default
   spm(1)=1
Size system     600000
kl,ku           1       1
#Threads (Total) available           4
4 Partitions and 4 Threads
Partition ratios: 1.800000000000000e+00 3.500000000000000e+00
#Partitions with 1 and 2 threads
4       0
Partition sizes: 234783 65217   65217   234783
---------------------
|Factorization Time|
------------------------------
|   LU or SPIKE on blocks    |
-------------------------------------------------
| Thread |   Partition   |          Time         |
-------------------------------------------------
|    1   |       1       | 9.718179702758789e-03 |
|    2   |       2       | 5.499386787414551e-02 |
|    3   |       3       | 6.807303428649902e-02 |
|    4   |       4       | 9.536981582641602e-03 |
-------------------------------------------------
|Blocks Factorize |       6.931304931640625e-02  |
|Reduced System   |       1.089572906494141e-04  |
|Overall Factorize|       6.950306892395020e-02  |
-------------------------------------------------
------------
|Solve Time|
------------------------------
|  Solve Sweeps on blocks    |--------------------------------------
| Thread | Partition |        Time 1         |         Time 2        |
|    1   |     1     | 9.419488906860352e-02 | 8.776593208312988e-02 |
|    2   |     2     | 4.948687553405762e-02 | 5.314803123474121e-02 |
|    3   |     3     | 7.169198989868164e-02 | 4.890084266662598e-02 |
|    4   |     4     | 9.282588958740234e-02 | 9.039402008056641e-02 |
--------------------------------------------------------------------
|Blocks Solve        | 9.419798851013184e-02 | 9.039592742919922e-02 |
|Reduced System Solve| 2.121925354003906e-05                         |
|Overall Solve       | 1.846210956573486e-01                         |
--------------------------------------------------------------------
  Max residual is :   2.775557561562891E-016
```

Figure 3: Output of helloworld example using 4 threads.

- One to show the pivoting 'trf/trs' functionality
- One to show the non-pivoting 'trf/trs' functionality

A Makefile has been included in each programming language. Issue the command `make all` in the appropriate directory to build the examples for your language of choice. The examples may also be configured with the included `make.inc` file, similarly to the build configuration of the library itself. If a custom install directory has been used when building the library, it may be necessary to set a `SPIKEROOT` environment variable to inform the Makefile of the location of the library. The Makefile will attempt to find `libspike.a` in the directory `SPIKEROOT/lib/ARCH`. By default this is located, starting from the directory where the package was unpacked, at `spike-1.0/lib/x64`.

5 Usage

The general calling scheme is as follows:

- Initialize the spike parameter array. Call using the `spikeinit` subroutine.
- Optionally call `Xspike_tune` to discover a machine specific tuning constant.
- Optionally configure the spm array to your liking (detailed below).
- Choose one of the following:
  - If you would like to perform the factorization and solve in one call...
    * call `Xspike_gbsv` to finish the problem.
  - If you would like to perform the factorization and solve separately...
    * Allocate a work array of size $max(kl, ku)^2 \times spm(10)$ (The type of this work array should be the same as the matrix and vectors used).
    * If you would like to use the default non-pivoting operation...
      · Optionally save a copy of A into C if you might use iterative refinement.
      · call `Xspike_gbtrf` to perform the SPIKE factorization.
      · If you don't want to use iterative refinement, call `Xspike_gbtrs` to finish.
      · If you do want to use iterative refinement, call `Xspike_gbtrsi` to finish.
    * If you would like to use pivoting operation...
      · Call `Xspike_gbtrfp` to perform the SPIKE factorization.
      · Note — the layout of the A matrix is unusual in this case
      · Call `Xspike_gbtrsp` to finish the problem.

5.1 SPIKE parameter array entries

The SPM array controls some details of the SPIKE computation. Default values are in bold. Note: Array elements are given using Fortran notation, where the first element of the array is spm(1). If you are using C, the array indices should be reduced by one.

User input flags:

- spm(1): Print flag.
  - **0: Do not print spike partition and timing information**

- 1: Print spike partition and timing information

- spm(2): Partition size optimization flag. To improve load balancing, sizes of the submatrices into which A is broken vary in size. These sizes are described in terms of the ratio of the first partition size to the sizes of the two types of internal partitions. The best value depends on the hardware used, the bandwidth of the matrix, and the number of vectors in the solution B. For more details see section 5.2.
  - **0: Use partition ratio values stored in spm(4) and spm(5)**
  - 1: Use partition ratio values designed for many solution vectors (it turns out these are constant, so spm(4) and spm(5) are ignored in this case)
  - 2: Use number of solution vectors and matrix bandwidth to compromise (only for `Xspike_gbsv`)

- spm(3): Precision improvement flag. This feature is only availible when using Xspike_gbsv. SPIKE may either use partial pivoting or iterative refinement to improve numerical results, at the cost of performance.
  - **0: Do not attempt to improve precision**
  - 1: Use a pivoting solver
  - 2: Use iterative refinement

- spm(4): (expert option) Ratio of the first partition size to the size of the large inner partition (ratio is spm(4)/10). Defaults to **18** (Ratio 1.8), which we have found to provide good performance in many cases. May be modified by the `Xspike_tune` or `Xspike_gbsv` subroutines. This value may be ignored based on the setting for spm(2).

- spm(5): (expert option) Ratio of the first partition size to the size of the small inner partition (ration is spm(5)/10). Defaults to **35** (Ratio 3.5), which we have found to provide good performance in many cases. May be modified by the `Xspike_tune` or `Xspike_gbsv` subroutines. This value may be ignored based on the setting for spm(2).

- spm(7): (expert option) Tuning constant for partition ratio (expert option). This is ultimately used to set spm(4) and spm(5) in `Xspike_gbsv` if the spm(2) flag is set to 2 (automatic tuning). Defaults to **16**, which we have found to provide good performance in many cases.

- spm(11): Max number of iterations for iterative refinement. Default is **3**. May be ignored based on spm(3).

- spm(12): Exponent for residual tolerance for double precision (and double complex)– I.E, res tolerance is $10^{-spm(12)}$. Default is **12**. May be ignored based on spm(3).

- spm(13): Exponent for residual tolerance for single precision (and single complex)– I.E, res tolerance is $10^{-spm(13)}$. Default is **5**. May be ignored based on spm(3).

- spm(14) : Norm type used internally for various calculations (particularly, for finding the residual when performing iterative refinement).
  - 0: Norm ∞
  - **1: Norm 1**
  - 2: Norm 2

Information parameters (information obtained from `spikeinit`.

- spm(10): Number of $klu \times klu$ blocks required for work array.
- spm(20): Number of partitions into which SPIKE is broken.
- spm(21): Number of recursive levels for reduced system.
- spm(22): Number of threads used by SPIKE.
- spm(23): Number of partitions using a SPIKE2×2 kernel, for two threaded factorize and solve.

5.2 A quick(-ish) note about partition sizes and spm

There are three types of partitions into which SPIKE breaks the A matrix. The top-left and bottom-right partitions of A take the least work per element, and thus are the largest. Depending on the number of threads, some of the remaining partitions will be worked on by two threads, and some will be worked on by one. Partitions worked on by two threads are the second-largest, and partitions worked on by one are the smallest.

The size of these partitions are described in terms of ratios between them. The factors that determine these ratios can be separated into those that depend on the machine used to run SPIKE, and those that depend on the matrix and vectors used. The subroutine `Xspike_tune` (detailed below) will calculate the hardware dependent factor and save it to spm.

To combine the problem dependent and hardware dependent factors, it is necessary to know the number of vectors in the solution before performing the factorization. We do not necessarily know this value when `Xspike_gbtrf` is called. Instead, the parameter spm(2) can be used to describe one of two limiting cases. If you expect that the number of vectors in the solution is much greater than the matrix bandwidth, use spm(2) = 1. Otherwise (when using `Xspike_gbtrf`) the default of spm(2) = 0 is usually better.

Finally, when using `Xspike_gbsv` (combined factorize and solve), the number of vectors in the solution is known before the factorization is performed. If spm(2)=2 is set, `Xspike_gbsv` will take both factors into account. Note that this computation does not take iterative refinement into account (as the number of iterative steps is not known beforehand), so if you would like to use iterative refinement with a poorly conditioned matrix it may may sense to use spm(2) = 1.

5.3 Subroutine Listing

Input parameters (User Configurable)

5.3.1 spikeinit

```
spikeinit(spm,n,max(kl,ku))
```

This subroutine initializes the spike parameter array (`spm`) to reasonable defaults. `spm` controls the behavior of SPIKE, primarily the partitioning scheme. This function requires to values of `n` and `max(kl,ku)` because it also calculates the size of the SPIKE workarray which the user will have to provide. The contents of `spm` are shown in section 5.1.

5.3.2 Xspike_tune

`Xspike_tune(spm)`

This subroutine will determine the hardware dependent tuning constant for used to determine the partition ratios. The tuning constant is stored in spm(7) (not documented). This subroutine will also fill spm(4) and spm(5), the partition ratios, to their large bandwidth matrix values.

Discovering this tuning constant requires a single threaded banded factorize and solve. So, it is probably inadvisable to perform this every time SPIKE is used. Ultimately the information found by `Xspike_tune` depends on the big-O run times of the factorize and solve code which will be called by SPIKE on the partitions of A. This should not vary for a given hardware/LAPACK pairing, so `Xspike_tune` could be called when your program first starts, or even found independently and saved as an environment variable. It is likely, however, to change if `MKL_NUM_THREADS` changes. It is also likely that this tuning constant is different for your pivoting and non-pivoting routines.

5.3.3 Xspike_gbsv

`Xspike_gbsv(spm,n,kl,ku,nrhs,A,lda,B,ldb,info)`

This is the do-it-all factorize and solve subroutine. A matrix and a collection of vectors are entered, and the operation $AY = B \rightarrow Y$ (solution in B) is performed. The residual value for B is checked, and simple iterative refinement is performed if it is above the given tolerance. Unlike Lapack, the matrix A is returned to the initial state upon return. Note that, even in the case when the pivoting option is enabled, the layout of the matrix does not change.

5.3.4 Xspike_gbtrf

`Xspike_gbtrf(spm,n,kl,ku,A,lda,work,info)`

This subroutine performs the SPIKE DS factorization. This subroutine naturally over-writes A. So if you might want to perform some refinement later, or use A for something else, you should save it elsewhere before calling this subroutine.

5.3.5 Xspike_gbtrs

`Xspike_gbtrs(spm,trans,n,kl,ku,nrhs,A,lda,work,B,ldb)`

This subroutine performs the solve stage for the matrix factorized by `Xspike_gbtrf`. Note that this subroutine does not write to A or the work array, so the factorization may be reused if you keep those arrays intact.

5.3.6 Xspike_gbtrsi

`Xspike_gbtrsi(spm,trans,n,kl,ku,nrhs,C,ldc,A,lda,work,B,ldb)`

This subroutine performs the solve stage, and then performs some iterative refinement if the solution is not within the desired tolerance. Similar to the previous subroutine, the matrix A and the work array are not written to in this subroutine, and so may be reused for later solves if you would like. Note that this subroutine does not accept a `ipiv` argument, because the iterative refinement and pivoting solvers are incompatible.

5.3.7 Xspike_gbtrfp

`Xspike_gbtrfp(spm,n,kl,ku,A,lda,work,ipiv,info)`

This subroutine performs the SPIKE DS factorization, using pivoting factorization and solve operation on the individual partitions. This subroutine naturally over-writes A. So if you might want to perform some refinement later, or use A for something else, you should save it elsewhere before calling this subroutine. Note that this function requires additional rows in the input array A. This is described in section 5.4.

5.3.8 Xspike_gbtrsp

`Xspike_gbtrsp(spm,n,kl,ku,nrhs,A,lda,work,ipiv,B,ldb)`

This subroutine performs the solve stage for the matrix factorized by `Xspike_gbtrfp`. Note that this subroutine does not write to A or the work array, so the factorization may be reused if you keep those arrays intact.

5.4 Table of variables

Name	Type	Size	Description
spm	int	64	The SPIKE parameter array.
trans	char		T for transpose solve, N for non-transpose solve, C for conjugate-transpose solve
n	int		Size of matrix A
kl	int		Lower bandwidth of matrix A
ku	int		Upper bandwidth of matrix A
nrhs	int		'Number of right hand sides' - the number of vectors in B
C	X	$ldc \times n$	A non-factorized copy of the matrix A
ldc	int		The leading dimension for the matrix C. ldc=kl+ku+1
A	X	$lda \times n$	The matrix A with which we will be performing $A^{-1}B \to B$ In most instances, A should be stored in the LAPACK/BLAS band storage format. *For most routines no extra-storage is needed, and the matrix can be stored in rows 1 to kl+ku+1 *For `Xspike_gbtrfp` the storage is similar to the format used by `Xgbtrf` but with the input matrix occupying rows 1+max(kl,ku) to max(kl,ku)+kl+ku,rather than rows kl+1 to 2kl+ku
lda	int		The leading dimension for the matrix A. Depends on the function used `Xspike_gbsv`, `Xspike_gbtrf`, `Xspike_gbtrs`, `Xspike_gbtrsi` : lda=kl+ku+1 `Xspike_gbtrfp`, `Xspike_gbtrsp` : lda=max(kl,ku)+kl+ku+1
work	X	$max(kl, ku)^2$ $\times spm(10)$	Work array used to store the SPIKE reduced system.
ipiv	int	n	Holds the permutation array for pivoting.
B	X	$ldb \times nrhs$	The vectors B on which we will be performing $A^{-1}B \to B$
ldb	int		Leading dimension of B. ldb $\geq$ n.
info	int		Info parameter info = 0 $\to$ Success. info = 1 $\to$ Boosting required. info = 2 $\to$ Illegal matrix entry.

Variables with no size are scalar. X may be double precision when using dspike, or double complex

when using zspike. Note that the size of the work array depends on the number of threads used. `spm(10)` is set by `spikeinit`.

5.5 SPM entries

Quick index of SPM entries. Details are shown in section 5.1. Note: Array elements are given following Fortran notation, where the first element of the array is spm(1).

Configuration Entries	
1	Print flag
2	Partition size optimization flag
3	Pivoting/Iterative refinement flag
4	Large inner partition ratio
5	Small inner partition ratio
7	Tuning parameter
11	Iterative refinement iteration limit
12	Iterative refinement double precision residual tolerance
13	Iterative refinement single precision residual tolerance
14	Norm type
Information Entries	
10	Number of $klu \times klu$ blocks needed for the work matrix
20	Number of partitions into which SPIKE is broken
21	Number of recursive levels for reduced system
22	Number of threads used by SPIKE
23	Number of partitions using SPIKE 2×2 kernel

A More Examples

If the following code is placed in a file called `example.f90`, and that file is placed in the root SPIKE directory, `spike-1.0`, it should be possible to compile the example by issuing the command:

`ifort example.f90 -qopenmp -I$MKLROOT/include -Llib/x64/ -lspike -mkl -o example.exe`

A.1 Split Factorize & Solve

```
program main

integer :: info
integer, parameter :: n=60, kl=1, ku=1, nrhs=1
double precision, dimension(kl+ku+1,n) :: A,oA
double precision, dimension(n,nrhs) :: f,of
double precision, allocatable, dimension(:) :: work
integer, dimension(64) :: spm


! Define matrix and right-hand-side
A = -1.0d0
A(ku+1,:) = 4.0d0
f(:,1) = 1.0d0
oA=A
of=f

call spikeinit(spm,n,max(kl,ku)) ! Initialize spm array

allocate(work(max(kl,ku)*max(kl,ku)*spm(10))) ! Prepare work array

spm(1) = 1 ! Instructs SPIKE to print timing and partition information

call dspike_gbtrf(spm,n,kl,ku,A,kl+ku+1,work,info)
call dspike_gbtrs(spm,'n',n,kl,ku,nrhs,A,kl+ku+1,work,f,n)

! Check the residual
call dgbmv('N',n,n,kl,ku,1.0d0,oA,kl+ku+1,f(:,1),1,-1.0d0,of(:,1),1)
print *, "Max residual is: ", maxval(abs(of(:,1)))

end program
```

A.2 Split Factorize & Solve With Pivoting

```
program main

integer :: info, klu
integer, parameter :: n=60, kl=1, ku=1, nrhs=1
double precision, dimension(max(kl,ku)+kl+ku+1,n) :: A
double precision, dimension(kl+ku+1,n) :: oA
double precision, dimension(n,nrhs) :: f,of
double precision, allocatable, dimension(:) :: work
integer, dimension(64) :: spm
integer, dimension(n) :: ipiv

klu=max(kl,ku)

! Define matrix and right-hand-side
oA = -1.0d0
oA(ku+1,:) = 4.0d0
f(:,1) = 1.0d0
A(klu+1:klu+kl+ku+1,:) = oA(1:kl+ku+1,:) ! A requires extra pivoting rows
of=f

call spikeinit(spm,n,max(kl,ku)) ! Initialize spm array

allocate(work(klu*klu*spm(10))) ! Prepare work array

spm(1) = 1 ! Instructs SPIKE to print timing and partition information

call dspike_gbtrfp(spm,n,kl,ku,A,klu+kl+ku+1,work,ipiv,info)
call dspike_gbtrsp(spm,n,kl,ku,nrhs,A,klu+kl+ku+1,work,ipiv,f,n)

! Check the residual
call dgbmv('N',n,n,kl,ku,1.0d0,oA,kl+ku+1,f(:,1),1,-1.0d0,of(:,1),1)
print *, "Max residual is: ", maxval(abs(of(:,1)))

end program
```

A.3 Solve With Iterative Refinement

```
program main

integer :: info
integer, parameter :: n=60, kl=1, ku=1, nrhs=1
double precision, dimension(kl+ku+1,n) :: A,oA
double precision, dimension(n,nrhs) :: f,of
double precision, allocatable, dimension(:) :: work
integer, dimension(64) :: spm


! Define matrix and right-hand-side
A = 1.0d0
A(ku+1,:) = 1.0001d0 ! Value selected for force one round of refinement
f(:,1) = 1.0d0
oA=A
of=f

call spikeinit(spm,n,max(kl,ku)) ! Initialize spm array

allocate(work(max(kl,ku)*max(kl,ku)*spm(10))) ! Prepare work array

spm(1) = 1 ! Instructs SPIKE to print timing and partition information

call dspike_gbtrf(spm,n,kl,ku,A,kl+ku+1,work,info)
call dspike_gbtrsi(spm,'n',n,kl,ku,nrhs,oA,kl+ku+1,A,kl+ku+1,work,f,n)

! Check the residual
call dgbmv('N',n,n,kl,ku,1.0d0,oA,kl+ku+1,f(:,1),1,-1.0d0,of(:,1),1)
print *, "Max residual is: ", maxval(abs(of(:,1)))

end program
```

A.4 Transpose Solve

```
program main

integer :: info
integer, parameter :: n=60, kl=1, ku=1, nrhs=1
double precision, dimension(kl+ku+1,n) :: A,oA
double precision, dimension(n,nrhs) :: f,of
double precision, allocatable, dimension(:) :: work
integer, dimension(64) :: spm


! Define matrix and right-hand-side
A = -1.0d0
A(ku+1,:) = 4.0d0
f(:,1) = 1.0d0
oA=A
of=f

call spikeinit(spm,n,max(kl,ku)) ! Initialize spm array

allocate(work(max(kl,ku)*max(kl,ku)*spm(10))) ! Prepare work array

spm(1) = 1 ! Instructs SPIKE to print timing and partition information

call dspike_gbtrf(spm,n,kl,ku,A,kl+ku+1,work,info)
call dspike_gbtrs(spm,'T',n,kl,ku,nrhs,A,kl+ku+1,work,f,n)

! Check the residual
call dgbmv('T',n,n,kl,ku,1.0d0,oA,kl+ku+1,f(:,1),1,-1.0d0,of(:,1),1)
print *, "Max residual is: ", maxval(abs(of(:,1)))

end program
```

A.5 C example

If the following code is placed in a file called `example.c`, and that file is placed in the root SPIKE directory, `spike-1.0`, it should be possible to compile the example by issuing the command:

```
gcc example.c -Iinclude -Llib/x64/ -lspike -L$MKLROOT/lib/intel64
-Wl,--no-as-needed -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5
-lpthread -lm -ldl -m64 -I$MKLROOT/include
```

```
#include <stdio.h>
#include <mkl.h>
#include "spike.h"
#define M(p,j,i) p[j+ ld##p *(i)] // FORTRAN-style matrix macro.

int main() {
        int spikeparam[64];
        const char trans='N';
        int n=60, kl=1, ku=1, nrhs=1, klu;
        int ldf=n, ldof=n, ldA=kl+ku+1, ldoA=kl+ku+1;
        double res, d_one  = 1.0,  d_mone =-1.0;
        double A[(kl+ku+1)*n], oA[(kl+ku+1)*n], f[n*nrhs], of[n*nrhs];
        int i, j, resi, i_one=1;
        int info;
        klu = (kl >= ku) ? kl : ku;

        // Generate A and copy it to oA
        for(j=0;j<ldA;j++)
                for(i=0;i<n;i++)
                {       M(A,j,i)  = (j==ku) ? 4.0 : -1.0;
                        M(oA,j,i) = M(A,j,i);}
        // Generate B (all ones) and copy it to oB.
        for(j=0;j<ldf;j++)
                for(i=0;i<nrhs;i++)
                {       M(f,j,i)  = 1.0;
                        M(of,j,i) = M(f,j,i);}

        // Initalize the spike params array
        spikeinit(spikeparam,&n,&klu);

        // Instruct SPIKE to print timing and partitioning info
        // Note that the index is off by one, because we're using C.
        spikeparam[0] = 1;
        dspike_gbsv(spikeparam,&n,&kl,&ku,&nrhs,A,&ldA,f,&ldf,&info);

        // Let's just check the residual of the first vector,
        // since we only have one vector by default.
        DGBMV(&trans, &n, &n, &kl, &ku,
              &d_mone, oA, &ldA, &M(f,0,0),&i_one,
              &d_one, &M(of,0,0), &i_one);
        res = M(of,idamax(&n, &M(of,0,0), &i_one)-1, 0);

        printf("n, \t kl, \t ku, \t nrhs, \t residual\n");
        printf("%d, \t %d, \t %d, \t %d, \t %.3E \n",n,kl,ku,nrhs,res);
        return 0; }
```