The Portable Extendable Toolkit for Scientific Computing (PETSc)

Victor Eijkhout[*]

There are many libraries for scientific computing. Some are specialized to certain application areas, others are quite general. In this section we will take a brief look at the PETSc library for sparse matrix computations, and the BLAS/Lapack libraries for dense computations.

1 The Portable Extendable Toolkit for Scientific Computing

PETSc, the Portable Extendable Toolkit for Scientifc Computation [], is a large powerful library, mostly concerned with linear and nonlinear system of equations that arise from discretized Partial Diffential Equations (PDEs). Since it has many hundreds of routines (and a good manual already exists) we limit this tutorial to going through a few simple, yet representative, PETSc programs.

1.1 What is in PETSc?

PETSc can be used as a library in the traditional sense, where you use some high level functionality, such as solving a nonlinear system of equations, in your program. However, it can also be used as a toolbox, to compose your own numerical applications using low-level tools.

- Linear system solvers (sparse/dense, iterative/direct)
- Nonlinear system solvers
- Tools for distributed matrices
- Support for profiling, debugging, graphical output

The basic functionality of PETSc can be extended through external packages:

- Dense linear algebra: `Scalapack`, `Plapack`
- Grid partitioning software: `ParMetis`, `Jostle`, `Chaco`, `Party`
- ODE solvers: `PVODE`
- Eigenvalue solvers (including SVD): `SLEPc`
- Optimization: `TAO`

[*] This tutorial is taken from the book ‘Introduction to High Performance Scientific Computing’, `http://tinyurl.com/EijkhoutHPC`.

1.2 Design of PETSc

PETSc has an object-oriented design, even though it is implemented in C, a not object oriented language. PETSc is also parallel: all objects in Petsc are defined on an MPI communicator (section **??**). For an object such as a matrix this means that it is stored distributed over the processors in the communicator; for objects such as solvers it means that their operation is distributed over the communicator. PETSc objects can only interact if they are based on the same communicator. Most of the time objects will be defined on the `MPI_COMM_WORLD` communicator, but subcommunicators can be used too. This way, you can define a matrix or a linear system on all your available processors, or on a subset.

Parallelism is handled through MPI. You may need to have occasional MPI calls in a PETSc code, but the vast bulk of communication is done behind the scenes inside PETSc calls. Shared memory parallel (such as through OpenMP; section **??**) is not used explicitly, but the user can incorporate it without problems.

The object-oriented design means that a call such as matrix-vector multiplication

```
MATMult(A,x,y); // y <- A x
```

looks the same, regardless whether A is sparse or dense, sequential or parallel.

One implication of this is that the actual data are usually hidden from the user. Data can be accessed through routines such as

```
double *array;
VecGetArray(myvector,&array);
```

but most of the time the application does not explicitly maintain the data, only the PETSc objects containing the data. As the ultimate consequence of this, the user usually does not allocate data; rather, matrix and vector arrays get created through the PETSc create calls, and subsequent values are inserted through PETSc calls:

```
MatSetValue(A,i,j,v,INSERT_VALUES); // A[i,j] <- v
```

This may feel like the user is giving up control, but it actually makes programming a lot easier, since the user does not have to worry about details of storage schemes, especially in parallel.

1.3 Small examples

In this section we will go through a number of successively more complicated examples of the use of PETSc. The files can be downloaded from
`http://tinyurl.com/ISTC-petsc-tutorial`. While doing these examples it is a good idea to keep the manual page open:
`http://tinyurl.com/PETSc-man-page`. You can also download a manual in pdf form from
`http://tinyurl.com/PETSc-pdf-manual`.

When you do these examples, make sure to use a version of PETSc that has debug mode enabled!

1.3.1 Program structure

The first example (we only list C sources in this book; the download includes their Fortran equivalents) illustrates the basic structure of a PETSc program: the include file at the top and the calls to `PetscInitialize`, `PetscFinalize`. Further more it uses two routines:

- `PetscOptionsGetInt` is used to check if the program was invoked with a `-n` option and a numerical value: if you ran your program as `./init -n 45`, the variable `n` should have the value 45.
- `PetscPrintf` functions like the `printf` function, except that only one processor executes the print command. If you ran your program as `mpiexec -np 4 init` and you used `printf`, you would get four copies of the output.

init.c

```
#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **argv)
{
  MPI_Comm       comm;
  PetscInt       n = 20;
  PetscErrorCode ierr;

  PetscFunctionBegin;
  ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);
  comm = PETSC_COMM_WORLD;
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL); CHKERRQ(ierr);
  printf("n=%d\n",n);
  ierr = PetscPrintf(comm,"Input parameter: %d\n",n); CHKERRQ(ierr);
  ierr = PetscFinalize();CHKERRQ(ierr);
  PetscFunctionReturn(0);
}
```

Just about the only lines of MPI that you need to know when programming PETSc are:

```
C:
ierr = MPI_Comm_size(comm,&ntids);
ierr = MPI_Comm_rank(comm,&mytid);
F:
call MPI_Comm_size(comm,&ntids,ierr);
call MPI_Comm_rank(comm,&mytid,ierr);
```

The first line gives the size of the communicator, meaning how many processes there are; the second one gives the rank of the current process as a number from zero to the number of processes minus one.

Exercise 3.1. Add these two lines to your program. Look up the routine `PetscSynchronizedPrintf` in the documentation and use it to let each process print out a line like `Process 3 out of 7`. You may also need to use `PetscSynchronizedFlush`.

1.3.2 Vectors

Next we start making PETSc objects, first a vector.

vec.c

```
#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **argv)
{
  MPI_Comm       comm;
  int ntids,mytid,myfirst,mylast;
  Vec            x;
  PetscInt       n = 20;
  PetscReal      one = 1.0;
  PetscErrorCode ierr;

  PetscFunctionBegin;
  ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);
  comm = PETSC_COMM_WORLD;
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
      CHKERRQ(ierr);

  MPI_Comm_size(comm,&ntids); MPI_Comm_rank(comm,&mytid);

  ierr = VecCreate(comm,&x);CHKERRQ(ierr);
  ierr = VecSetSizes(x,PETSC_DECIDE,n); CHKERRQ(ierr);
  ierr = VecSetType(x,VECMPI); CHKERRQ(ierr);

  ierr = VecGetOwnershipRange(x,&myfirst,&mylast); CHKERRQ(ierr);
  ierr = PetscSynchronizedPrintf(comm,"Proc %d, range %d--%d\n",
                      mytid,myfirst,mylast); CHKERRQ(ierr);

  ierr = VecSet(x,one); CHKERRQ(ierr);
  ierr = VecAssemblyBegin(x); CHKERRQ(ierr);
  ierr = VecAssemblyEnd(x); CHKERRQ(ierr);
  ierr = VecView(x,0); CHKERRQ(ierr);

  ierr = VecDestroy(&x); CHKERRQ(ierr);
  ierr = PetscFinalize();CHKERRQ(ierr);
  PetscFunctionReturn(0);
}
```

Note how it takes several calls to fully create the vector object:

- The type `VECMPI` means that the vector will be distributed.
- The routine setting the size has two size parameters; the second specifies that the global size is `n`, and the first one says that you leave the distribution for PETSc to decide.

At the end of the program there is a to `VecDestroy`, which deallocates the memory for the vector. While this is strictly speaking not necessary for the functioning of the program, it is a good idea to issue `Destroy` calls for each `Create` call, since it can prevent potential *memory leaks*.

Exercise 3.2. Comment out the `VecDestroy` call, and run the program with the option `-malloc_dump`.

PETSc will now report on all memory that had not been freed at the time of the `PetscFinalize` call.

If you run the program in parallel the vector will be created distributed over the processors. The program contains a call to `VecGetOwnershipRange` to discover what part of the vector lives on what processor. You see that the `VecView` calls display for each processor precisely that part of the vector.

Exercise 3.3. The listing has a call to `VecSet` to set the vector elements to a constant value. Remove this line. Use the `myfirst,mylast` variables and the PETSc routine `VecSetValue` to let every process set its local vector elements to the processor number. (The last argument of `VecSetValue` should be `INSERT_VALUES`.) That is, if the vector is six elements long and there are three processors, the resulting vector should be $(0,0,1,1,2,2)$.

Exercise 3.4. Change the code from the previous exercise. Now every vector element should be the sum of the processor number and the previous processor numbers; in the above example the result should be $(0,0,1,1,3,3)$. Read the man page for `VecSetValue`!

Run the code from the previous two exercises again, adding a commandline argument `-log_summary`. Observe that the first code has no messages in the `VecScatterBegin/End` calls, but the second one does.

1.3.3 Matrices

Let's move up to matrices. Creating a matrix is similar to creating a vector. In this case the type is `MPIAIJ` which stands for a distributed sparse matrix.

mat.c

```
#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **argv)
{
  MPI_Comm       comm;
  int ntids,mytid,localsize,myfirst,mylast,i;
  Mat            A;
  PetscInt       n = 20;
  PetscErrorCode ierr;

  PetscFunctionBegin;
  ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);
  comm = PETSC_COMM_WORLD;
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
      CHKERRQ(ierr);

  MPI_Comm_size(comm,&ntids); MPI_Comm_rank(comm,&mytid);
  localsize = PETSC_DECIDE;
  ierr = PetscSplitOwnership(comm,&localsize,&n); CHKERRQ(ierr);

  ierr = MatCreate(comm,&A); CHKERRQ(ierr);
  ierr = MatSetType(A,MATMPIAIJ); CHKERRQ(ierr);
  ierr = MatSetSizes(A,localsize,localsize,
                  PETSC_DECIDE,PETSC_DECIDE); CHKERRQ(ierr);
  ierr = MatGetOwnershipRange(A,&myfirst,&mylast); CHKERRQ(ierr);

  for (i=myfirst; i<mylast; i++) {
    PetscReal v=1.0*mytid;
    ierr = MatSetValues(A,1,&i,1,&i,&v,INSERT_VALUES); CHKERRQ(ierr);
  }
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
  ierr = MatView(A,0); CHKERRQ(ierr);

  ierr = MatDestroy(&A); CHKERRQ(ierr);
  ierr = PetscFinalize();CHKERRQ(ierr);
  PetscFunctionReturn(0);
}
```

In this example a diagonal matrix is constructed, with each processor setting only its locally stored elements.

Exercise 3.5. In addition to setting the diagonal, also set the first subdiagonal and the first super-diagonal, that is, the elements $(i, i-1)$ and $(i, i+1)$. To set all three elements in a row with one call you can do this:

```
vm = 1.*mytid;
for (i=myfirst; i<mylast; i++) {
  PetscInt j[3]; PetscReal v[3];
  j[0] = i-1; j[1] = i; j[2] = i+1;
  v[0] = --vm-1; v[1] = 2*vm; v[2] = -vm+1;
  ierr = MatSetValues(A,1,&i,3,j,v,INSERT_VALUES); CHKERRQ(ierr);
}
```

However, this code is not entirely correct. Edit the program using this fragment and run it. Diagnose the problem and fix it.

1.3.4 Matrix-vector operations

Next we will create a file `mul.c` based on `mat.c` and multiply the matrix and the vector. Make sure that the size declarations of the matrix and the vector are compatible. You also need a second vector to store the result of the multiplication. This is easiest done by

```
ierr = VecDuplicate(x,&y); CHKERRQ(ierr);
```

Exercise 3.6. Look up the `MatMult` routine in the documentation and use it your program. Use `VecView` to inspect the result. Note that no size parameters or anything pertaining to parallelism appears in the calling sequence.

1.3.5 Solvers

Copy your `mat.c` file to `sys.c`: we are going to explore linear system solving. First you need to create a solver object and give it the matrix as operator:

```
ierr = KSPCreate(comm,&solver); CHKERRQ(ierr);
ierr = KSPSetOperators(solver,A,A,0); CHKERRQ(ierr);
ierr = KSPSolve(solver,x,y); CHKERRQ(ierr);
ierr = VecView(y,0); CHKERRQ(ierr);
```

Exercise 3.7. Add these lines to your code. Make sure you know what the correct solution is by using `MatMult` to obtain the right hand side.

You have just used the default linear system solver. Run the program again, but with the option `-ksp_view`. This will tell you all the details of what solver was used.

Solving the linear system is a one line call to `KSPSolve`. The story would end there if it weren't for some complications:

- Iterative methods can fail, and the solve call does not tell us whether that happened.
- If the system was solved successfully, we would like to know in how many iterations.
- There can be other reason for the iterative method to halt, such as reaching its maximum number of iterations without converging.

Exercise 3.8. Use the routine `KSPGetConvergedReason` to inspect the status of the solution vector. Use `KSPGetIterationNumber` to see how many iterations it took.

Exercise 3.9. Add code to your program to compute the residual and its norm. For the residual, look up the routines `VecDuplicate` and `VecAXPY`; for computing its norm look up `VecNorm`.

Exercise 3.10. Add a call to `KSPSetFromOptions` to your code. Use the option `-ksp_monitor` to observe the convergence behaviour.

The workings of a PETSc program can be customized to a great degree through the use of commandline options. This includes setting the type of the solver. In order for such options to be obeyed, you first need to put a command `KSPSetFromOptions` before the `KSPSolve` call.

Exercise 3.11. The `-ksp_view` option told you what solver and preconditioner were used. Look up the routines `KSPSetType` and `PCSetType` and use those to change the iterative method to CG and the preconditioner to Jacobi. Do this first by using commandline options, and then by editing the code.

1.4 A realistic program

This section will give you some further help towards solving a realistic PDE problem.

1.4.1 Construction of the coefficient matrix

In the examples above you used a commandline argument to determine the matrix size directly. Here we construct the matrix of 5-point stencil for the Poisson operator (see section **??** and in particular figure **??**). Determining its size takes two steps: you need to read the domain size $n = 1/h - 1$ and compute the matrix size from it.

C:

```
  int domain_size,matrix_size;
  PetscOptionsGetInt
    (PETSC_NULL,"-n",&domain_size,&flag);
  matrix_size = domain_size*domain_size;
```

Fortran:

```
  integer :: domain_size,matrix_size
  call PetscOptionsGetInt(PETSC_NULL_CHARACTER,
 >     "-n",domain_size,flag)
  matrix_size = domain_size*domain_size;
```

Now you use the `matrix_size` parameter for constructing the matrix.

1.4.2 Filling in matrix elements

Just like in the examples above, you want each processor to set only its local rows. The easiest way to iterate over those is to iterate over all variables / matrix rows and select only the local ones.

We will now set matrix elements (refer to the full domain, but only inserting those elements that are in its matrix block row.

C:

```
MatGetOwnershipRange(A,&myfirst,&mylast);
for ( i=0; i<domain_size; i++ ) {
  for ( j=0; j<domain_size; j++ ) {
    I = j + matrix_size*i;
    if (I>=myfirst && I<mylast) {
      J = I; // for the diagonal element
      MatSetValues
          (A,1,&I,1,&J,&v,INSERT_VALUES);
      J = .... // for the other points
      J = ....
    }
  }
}
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

Fortran:

```
call MatGetOwnershipRange(A,myfirst,mylast)
do i=0,matrix_size-1
   do j=0,domain_size-1
      ii = j + domain_size*i
      if (ii>=myfirst .and. ii<mylast) then
        jj = ii ; for the diagonal element
        call MatSetValues
```

```
 >           (A,1,ii,1,jj,v,INSERT_VALUES)
        jj = ii... ; for the other elements
        jj = ii...
      end if
   end do
end do
call MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY)
call MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY)
```

Exercise 3.12. Construct the matrix from equation (**??**) in section **??**. Compute and output the product with the identity vector (meaning that all elements are 1), and check that the result is correct. Make sure to test your program in parallel.

1.4.3 Finite Element Matrix assembly

PETSc's versatility in dealing with Finite Element matrices (see sections **??** and **??**), where elements are constructed by adding together contributions, sometimes from different processors. This is no problem in PETSc: any processor can set (or add to) any matrix element. The assembly calls will move data to their eventual location on the correct processors.

```
for (e=myfirstelement; e<mylastelement; e++) {
  for (i=0; i<nlocalnodes; i++) {
    I = localtoglobal(e,i);
    for (j=0; j<nlocalnodes; j++) {
      J = localtoglobal(e,j);
      v = integration(e,i,j);
      MatSetValues
            (mat,1,&I,1,&J,&v,ADD_VALUES);
      ....
    }
  }
}
MatAssemblyBegin(mat,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(mat,MAT_FINAL_ASSEMBLY);
```

1.4.4 Linear system solver

We have covered the basics of setting up the solver and solving the system above.

As an illustration of the toolbox nature of PETSc, you can now use routines you have already seen to compute the residual and its norm.

Exercise 3.13. Create a new vector `z` (use `VecDuplicate`) and store the product of `A` and the computed solution `y` (use `MatMult`) in it. If you solved the system accurately, `z` should now be equal to `x`. To see how close it is, use

```
PetscReal norm;
VecAXPY(z,-1,x);
VecNorm(z,NORM_2,&norm);
```
to subtract `x` from `z` and compute the norm of the result.

1.5 Quick experimentation

Reading a parameter from the commandline above is actually a special case of a general mechanism for influencing PETSc's workings through commandline options.

Here is an example of setting the iterative solver and preconditioner from the commandline:

```
yourprog -ksp_type gmres -ksp_gmres_restart 25
    -pc_type ilu -pc_factor_levels 3
```

In order for this to work, your code needs to call

```
KSPSetFromOptions(solver);
```

before the system solution. This mechanism is very powerful, and it obviates the need for much code recompilation.

1.6 Review questions

Exercise 3.14. Write a PETSc program that does the following:

- Construct the matrix
$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$
- Compute the sequence
$$x_0 = (1,0,\ldots,0)^t, \quad y_{i+1} = Ax_i, \quad x_i = y_i/\|y_i\|_2.$$
This is the power method (section **??**), which is expected to converge to the dominent eigenvector.
- In each iteration of this process, print out the norm of y_i and for $i > 0$ the norm of the difference $x_i - x_{i-1}$. Do this for some different problem sizes. What do you observe?
- The number of iterations and the size of the problem should be specified through commandline options. Use the routine `PetscOptionsGetInt`.

For a small problem (say, $n = 10$) print out the first couple x_i vectors. What do you observe? Explanation?

Exercise 3.15. Extend the previous exercise: if a commandline option `-inverse` is present, the sequence should be generated as $y_{i+1} = A^{-1}x_i$. Use the routine `PetscOptionsHasName`. What do you observe now about the norms of the y_i vectors?

2 Libraries for dense linear algebra: Lapack and Scalapack

Dense linear algebra, that is linear algebra on matrices that are stored as two-dimensional arrays (as opposed to sparse linear algebra; see section **??**, as well as the tutorial on PETSc **??**) has been standardized for a considerable time. The basic operations are defined by the three levels of *Basic Linear Algebra Subprograms (BLAS)*:

- Level 1 defines vector operations that are characterized by a single loop [5].
- Level 2 defines matrix vector operations, both explicit such as the matrix-vector product, and implicit such as the solution of triangular systems [4].
- Level 3 defines matrix-matrix operations, most notably the matrix-matrix product [3].

The name 'BLAS' suggests a certain amount of generality, but the original authors were clear [5] that these subprograms only covered dense linear algebra. Attempts to standardize sparse operations have never met with equal success.

Based on these building blocks libraries have been built that tackle the more sophisticated problems such as solving linear systems, or computing eigenvalues or singular values. *Linpack*[1] and *Eispack* were the first to formalize these operations involved, using Blas Level 1 and Blas Level 2 respectively. A later development, *Lapack* uses the blocked operations of Blas Level 3. As you saw in section **??**, this is needed to get high performance on cache-based CPUs. (Note: the reference implementation of the BLAS [1] will not give good performance with any compiler; most platforms have vendor-optimized implementations, such as the *MKL* library from Intel.)

With the advent of parallel computers, several projects arose that extended the Lapack functionality to distributed computing, most notably *Scalapack* [2] and *PLapack* [7, 6]. These packages are considerably harder to use than Lapack[2] because of the need for the two-dimensional block cyclic distribution; sections **??** and **??**. We will not go into the details here.

2.1 BLAS matrix storage

There are a few points to bear in mind about the way matrices are stored in the BLAS and LAPACK[3]:

2.1.1 Array indexing

Since these libraries originated in a Fortran environment, they use 1-based indexing. Users of languages such as C/C++ are only affected by this when routines use index arrays, such as the location of pivots in LU factorizations.

2.1.2 Fortran column-major ordering

Since computer memory is one-dimensional, some conversion is needed from two-dimensional matrix coordinates to memory locations. The *Fortran* language uses *column-major* storage, that is, elements in a column are stored consecutively; see figure 1. This is also described informally as 'the leftmost index varies quickest'.

1. The linear system solver from this package later became the *Linpack benchmark*; see section **??**.
2. PLapack is probably the easier to use of the two.
3. We are not going into band storage here.

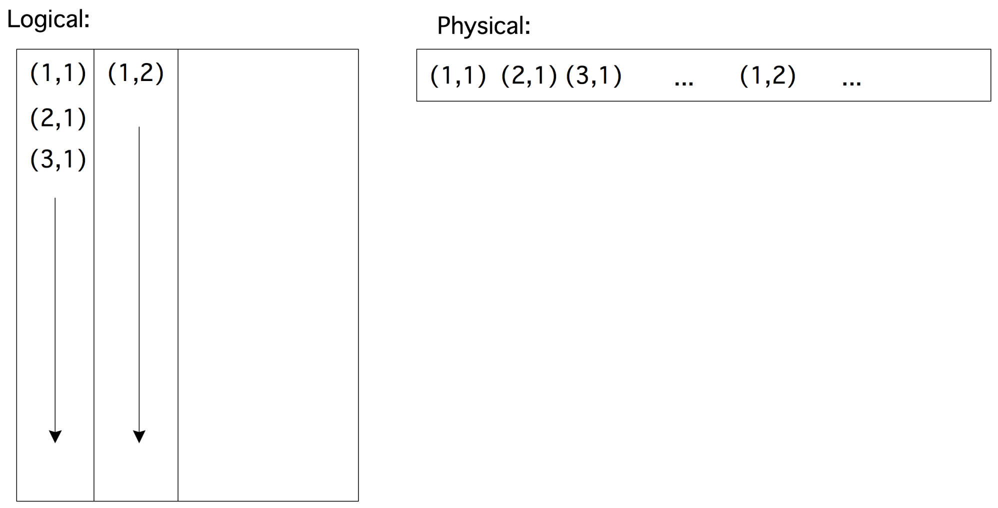

Figure 1: Column-major storage of an array in Fortran

2.1.3 Submatrices and the `LDA` parameter

Using the storage scheme described above, it is clear how to store an $m \times n$ matrix in mn memory locations. However, there are many cases where software needs access to a matrix that is a subblock of another, larger, matrix. As you see in figure 2 such a subblock is no longer contiguous in memory. The way to describe this

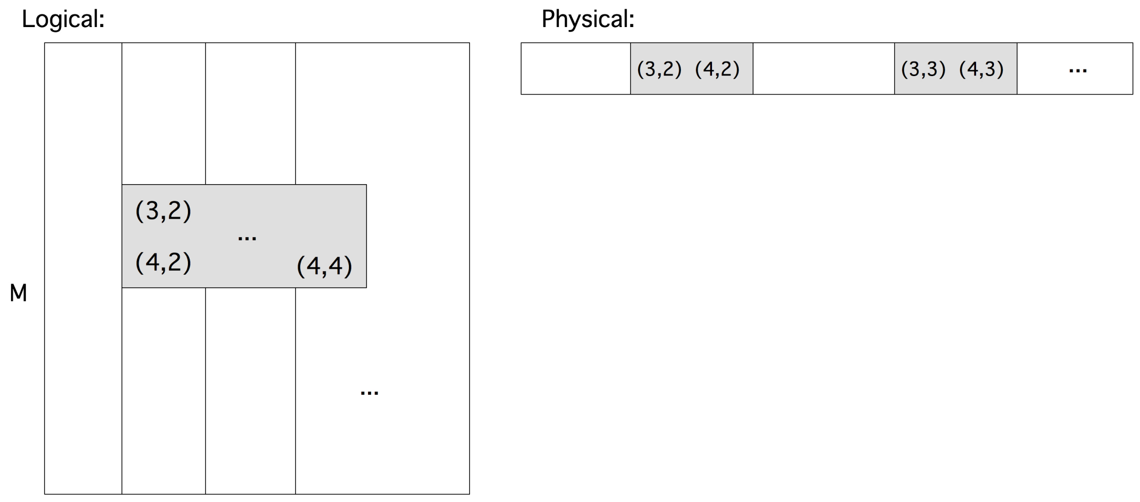

Figure 2: A subblock out of a larger matrix

is by introducing a third parameter in addition to `M,N`: we let `LDA` be the 'leading dimension of `A`', that is, the allocated first dimension of the surrounding array. This is illustrated in figure 3.

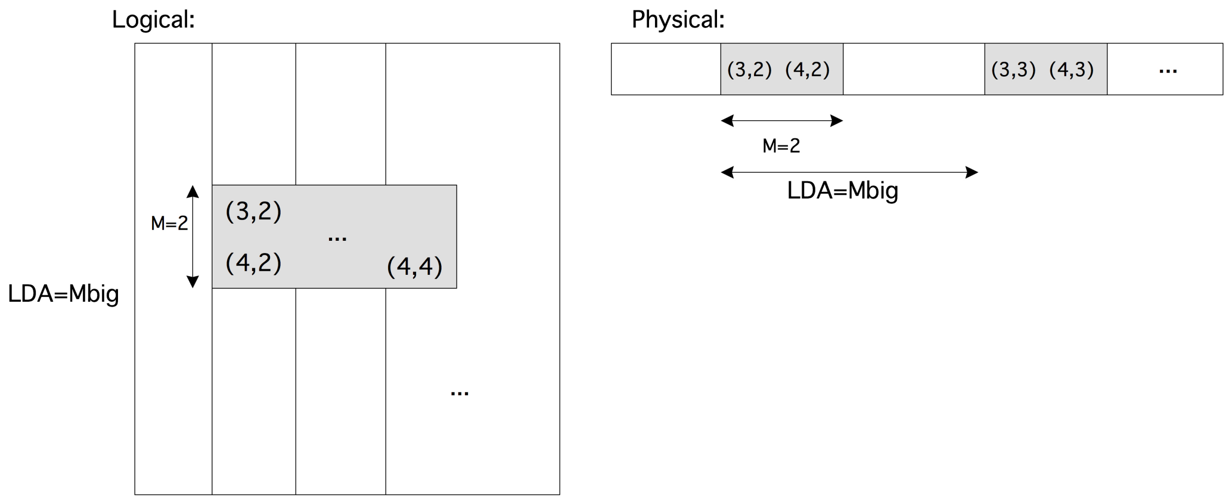

Figure 3: A subblock out of a larger matrix, using `LDA`

2.2 Organisation of routines

Lapack is organized with three levels of routines:

- Drivers. These are powerful top level routine for problems such as solving linear systems or computing an SVD. There are simple and expert drivers; the expert ones have more numerical sophistication.
- Computational routines. These are the routines that drivers are built up out of[4]. A user may have occasion to call them by themselves.
- Auxiliary routines.

Routines conform to a general naming scheme: `XYYZZZ` where

X precision: `S,D,C,Z` stand for single and double, single complex and double complex, respectively.
YY storage scheme: general rectangular, triangular, banded.
ZZZ operation. See the manual for a list.

Expert driver names end on 'X'.

2.2.1 Lapack data formats

Lapack and Blas use a number of data formats, including

GE General matrix: stored two-dimensionally as `A(LDA,*)`
SY/HE Symmetric/Hermitian: general storage; `UPLO` parameter to indicate upper or lower (e.g. `SPOTRF`)
GB/SB/HB General/symmetric/Hermitian band; these formats use column-major storage; in `SGBTRF` overallocation needed because of pivoting
PB Symmetric of Hermitian positive definite band; no overallocation in `SPDTRF`

4. Ha! Take that, Winston.

2.2.2 Lapack operations

- Linear system solving. Simple drivers: `-SV` (e.g., `DGESV`) Solve $AX = B$, overwrite A with LU (with pivoting), overwrite B with X.
  Expert driver: `-SVX` Also transpose solve, condition estimation, refinement, equilibration
- Least squares problems. Drivers:
  `xGELS` using QR or LQ under full-rank assumption
  `xGELSY` ”complete orthogonal factorisation”
  `xGELSS` using SVD
  `xGELSD` using divide-conquer SVD (faster, but more workspace than `xGELSS`)
  Also: LSE & GLM linear equality constraint & general linear model
- Eigenvalue routines. Symmetric/Hermitian: `xSY` or `xHE` (also `SP`, `SB`, `ST`) simple driver `-EV` expert driver `-EVX` divide and conquer `-EVD` relative robust representation `-EVR`
  General (only `xGE`) Schur decomposition `-ES` and `-ESX` eigenvalues `-EV` and `-EVX`
  SVD (only `xGE`) simple driver `-SVD` divide and conquer `SDD`
  Generalized symmetric (`SY` and `HE`; `SP`, `SB`) simple driver `GV` expert `GVX` divide-conquer `GVD`
  Nonsymmetric Schur: simple `GGES`, expert `GGESX` eigen: simple `GGEV`, expert `GGEVX`
  svd: `GGSVD`

References

[1] Netlib.org BLAS reference implementation. `http://www.netlib.org/blas`.

[2] Yaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the fourth symposium on the frontiers of massively parallel computation (Frontiers ’92), McLean, Virginia, Oct 19–21, 1992*, pages 120–127, 1992.

[3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[5] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:308–323, 1979. (Algorithm 539).

[6] R. van de Geijn, Philip Alpatov, Greg Baker, Almadena Chtchelkanova, Joe Eaton, Carter Edwards, Murthy Guddati, John Gunnels, Sam Guyer, Ken Klimkowski, Calvin Lin, Greg Morrow, Peter Nagel, James Overfelt, and Michelle Pal. Parallel linear algebra package (PLAPACK): Release r0.1 (beta) users’ guide. 1996.

[7] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.