An Introduction to PETSc

A user’s point of view

Serge Van Criekingen

1Maison de la Simulation, Saclay

Grenoble, January 22nd, 2013
Outline

1. Introduction
2. Hello World
3. PETSc Vectors (Vec)
4. PETSc Matrices (Mat)
5. PETSc Krylov Solvers (KSP)
6. Wrapping up
1 Introduction

2 Hello World

3 PETSc Vectors (Vec)

4 PETSc Matrices (Mat)

5 PETSc Krylov Solvers (KSP)

6 Wrapping up
What is PETSc?

- **Portable, Extensible Toolkit for Scientific Computation**
What is PETSc?

- Portable, Extensible Toolkit for Scientific Computation
- Open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability (~ 130,000 procs @ANL) specialized in large sparse iterative parallel solvers.
What is PETSc?

- Portable, Extensible Toolkit for Scientific Computation
- open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability (≈ 130,000 procs @ANL) specialized in large sparse iterative parallel solvers.
- Interface for C/C++, Fortran, Python (and Matlab)
What is PETSc?

- Portable, Extensible Toolkit for Scientific Computation
- open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability (∼130,000 procs @ANL) specialized in large sparse iterative parallel solvers.
- Interface for C/C++, Fortran, Python (and Matlab)
- supports MPI parallelism (i.e., distributed memory), but latest versions: pthreads (shared-memory) and GPU.
What is PETSc?

- Portable, Extensible Toolkit for Scientific Computation
- open-source set of C tools for the parallel solution of PDEs, with an emphasis on scalability (≈ 130,000 procs @ANL) specialized in large sparse iterative parallel solvers.
- Interface for C/C++, Fortran, Python (and Matlab)
- supports MPI parallelism (i.e., distributed memory), but latest versions: pthreads (shared-memory) and GPU.
- Features:
  - parallel vector and matrices
  - data and grid management tools
  - Krylov iterative solvers
  - parallel preconditioners
  - interfaces with external packages
  - Newton-based nonlinear solvers
  - time-stepping ODE solvers
Related Tools

- Employed in an impressive list of **scientific applications** and in other **packages** (a.o.):
  - eigenproblems: SLEPc
  - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++
Related Tools

Employed in an impressive list of scientific applications and in other packages (a.o.):

- eigenproblems: SLEPc
- FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++

Interfaces with (a.o.)

- direct solvers: PaStiX, MUMPS and SuperLU
- preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
- automatic differentiation tools: ADIC / ADIFOR
- graph partitioner: ParMeTiS, Party, PTScotch
Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++

2. Interfaces with (a.o.) automatic install
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch
Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++

2. Interfaces with (a.o.) automatic install
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch

3. (Sca)Lapack: for dense matrices
Related Tools

1. Employed in an impressive list of **scientific applications** and in other **packages** (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++

2. **Interfaces** with (a.o.) **automatic install**
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch

3. **(Sca)Lapack**: for **dense** matrices

4. **Trilinos**: C++, “bigger” package, less integrated
   (“loose confederation of interoperable packages”)
Related Tools

1. Employed in an impressive list of scientific applications and in other packages (a.o.):
   - eigenproblems: SLEPc
   - FEM and FVM: OpenFVM, OOFEM, DEAL.II, libMesh, Feel++

2. Interfaces with (a. o.) automatic install
   - direct solvers: PaStiX, MUMPS and SuperLU
   - preconditioner libraries: Hypre, Euclid, Trilinos/ML (multi-level)
   - automatic differentiation tools: ADIC / ADIFOR
   - graph partitioner: ParMeTiS, Party, PTScotch

3. (Sca)Lapack: for dense matrices

4. Trilinos: C++, “bigger” package, less integrated (“loose confederation of interoperable packages”)

5. ACTS (Advanced CompuTational Software) collection (U.S. DOE) includes PETSc, SuperLU, SLEPc, Hypre, ...
Extensive documentation on www.mcs.anl.gov/petsc/:

- Manual page for all routines
- Examples
- Introduction and tutorials by developers
  ⇒ This talk will focus on my user experience.
Documentation on PETSc

1. Extensive documentation on www.mcs.anl.gov/petsc/:
   - Manual page for all routines
   - Examples
   - Introduction and tutorials by developers
     ⇒ This talk will focus on my user experience.

2. Fast support from petsc-maint@mcs.anl.gov
Summary

1. Introduction
2. Hello World
3. PETSc Vectors (Vec)
4. PETSc Matrices (Mat)
5. PETSc Krylov Solvers (KSP)
6. Wrapping up
#include "petsc.h"

int main( int argc, char *argv[] ){
    PetscErrorCode ierr;
    PetscInitialize( &argc, &argv, 0, 0 );
    ierr = PetscPrintf( PETSC_COMM_WORLD, "Hello World\n" );
    CHKERRQ(ierr);
    PetscFinalize();
    return 0;
}
Hello World in Fortran (only from first processor)

```fortran
program test
  implicit none
#include "finclude/petsc.h"
  PetscErrorCode :: ierr
  call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
  call PetscPrintf(PETSC_COMM_WORLD,"Hello World \n",ierr);
    CHKERRQ(ierr)
  call PetscFinalize(ierr)
end program test
```
Hello World in C and Fortran (only from first processor)

```c
#include "petsc.h"
int main( int argc, char *argv[] ){
    PetscInitialize( &argc, &argv, 0, 0 );
    PetscPrintf( PETSC_COMM_WORLD, "Hello World\n" );
    PetscFinalize();
    return 0;
}
```
Hello World in C and Fortran (only from first processor)

```c
#include "petsc.h"
int main( int argc, char *argv[] ){
    PetscInitialize( &argc, &argv, 0, 0 );
    PetscPrintf( PETSC_COMM_WORLD, "Hello World\n" );
    PetscFinalize();
    return 0;
}
```

```fortran
program test
    implicit none
#include "finclude/petsc.h"
    PetscErrorCode :: ierr
    call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
    call PetscPrintf(PETSC_COMM_WORLD,"Hello World \n",ierr)
    call PetscFinalize(ierr)
end program test
```
Notes

- ierr (type PetscErrorCode) required at the end of each Fortran call
- PETSC_COMM_WORLD can be a subset of MPI_COMM_WORLD
Notes

- ierr (type PetscErrorCode) required at the end of each Fortran call
- PETSC_COMM_WORLD can be a subset of MPI_COMM_WORLD
- PetscInitialize contains MPI_Init
  PetscFinalize contains MPI_Finalize
Notes

- ierr (type PetscErrorCode) required at the end of each Fortran call
- PETSC_COMM_WORLD can be a subset of MPI_COMM_WORLD
- PetscInitialize contains MPI_Init
  PetscFinalize contains MPI_Finalize

In general: MPI calls are "hidden" by PETSc
Notes

- ierr (type PetscErrorCode) required at the end of each Fortran call
- PETSC_COMM_WORLD can be a subset of MPI_COMM_WORLD
- PetscInitialize contains MPI_Init
  PetscFinalize contains MPI_Finalize

In general: MPI calls are "hidden" by PETSc

Exceptions: MPI_Comm_size and MPI_Comm_rank
Back to **Hello World** (with parallel output)

```c
MPI_Comm_rank(PETSC_COMM_WORLD, &myrank);
PetscSynchronizedPrintf(PETSC_COMM_WORLD,
    "Rank %d says hello \n", myrank );
PetscSynchronizedFlush(PETSC_COMM_WORLD);
```

yields with `mpirun -np 3`:

- Rank 1 says hello
- Rank 2 says hello
- Rank 3 says hello
Back to **Hello World** (with parallel output)

```c
 [...] 
 MPI_Comm_rank(PETSC_COMM_WORLD, &myrank);
 PetscSynchronizedPrintf(PETSC_COMM_WORLD,  
       "Rank %d says hello \n", myrank );
 PetscSynchronizedFlush(PETSC_COMM_WORLD);
 [...] 
```

yields with `mpirun -np 3`:

```
Rank 1 says hello  
Rank 2 says hello  
Rank 3 says hello 
```

(More complicated in Fortran)
Introduction

Hello World

PETSc Vectors (Vec)

PETSc Matrices (Mat)

PETSc Krylov Solvers (KSP)

Wrapping up
PETSc Vectors

Create

VecCreateSeq(PETSC_COMM_SELF, int m, Vec *x);

VecCreateMPI(MPI_Comm comm, int m, int M, Vec *x);

where
- \( m \) = local size, or PETSC_DECIDE if \( M \) given
- \( M \) = global size, or PETSC_DETERMINE if \( m \) given for all ranks
PETSc Vectors

Create

VecCreateSeq(PETSC_COMM_SELF, int m, Vec *x);

VecCreateMPI(MPI_Comm comm, int m, int M, Vec *x);

In Fortran:

call VecCreateSeq(PETSC_COMM_SELF, int m, Vec x,
  PetscErrorCode ierr)

call VecCreateMPI(MPI_Comm comm, int m, int M, Vec x,
  PetscErrorCode ierr)
**PETSc Vectors**

1. **Create**

2. **Set**

```c
VecSetValues(Vec x, int n, int *indices,
             PetscScalar *values, INSERT_VALUES);
```

```c
VecSetValues(Vec x, int n, int *indices,
             PetscScalar *values, ADD_VALUES);
```
PETSc Vectors

Create

- Set

VecSetValues(Vec x, int n, int *indices,
    PetscScalar *values, INSERT_VALUES);

VecSetValues(Vec x, int n, int *indices,
    PetscScalar *values, ADD_VALUES);

Notes: • 0-based global indices in C and Fortran
PETSc Vectors

1. Create
2. Set

`VecSetValues(Vec x, int n, int *indices, PetscScalar *values, INSERT_VALUES);`

`VecSetValues(Vec x, int n, int *indices, PetscScalar *values, ADD_VALUES);`

Notes:
- 0-based global indices in C and Fortran
- In Fortran: int indices(n), PetscScalar values(n)
PETSc Vectors

1. **Create**
2. **Set**

```c
VecSetValues(Vec x, int n, int *indices, 
             PetscScalar *values, INSERT_VALUES);
```

```c
VecSetValues(Vec x, int n, int *indices, 
             PetscScalar *values, ADD_VALUES);
```

**Notes:**
- 0-based global indices in C and Fortran
- In Fortran: `int indices(n), PetscScalar values(n)`
- Faster if `n` large
PETSc Vectors

1. Create
2. Set

VecSetValues(Vec x, int n, int *indices,
             PetscScalar *values, INSERT_VALUES);

VecSetValues(Vec x, int n, int *indices,
             PetscScalar *values, ADD_VALUES);

3. Assemble (allow overlap of communication and calculation)

VecAssemblyBegin(Vec x);
VecAssemblyEnd(Vec x);
PETSc Vectors

1. Create

2. Set

   VecSetValues(Vec x, int n, int *indices,
               PetscScalar *values, INSERT_VALUES);

   VecSetValues(Vec x, int n, int *indices,
               PetscScalar *values, ADD_VALUES);

3. Assemble (allow overlap of communication and calculation)

   VecAssemblyBegin(Vec x);
   VecAssemblyEnd(Vec x);

**Caution**: INSERT_VALUES and ADD_VALUES can **not** be mixed
(call assembly routines inbetween).
PETSc Vectors

- **Get** (local only!)
  - Single Value: `VecGetValues` (use global numbering)
**PETSc Vectors**

- **Get** (local only!)
  1. Single Value: `VecGetValues` (use global numbering)
  2. All local elements: `VecGetArray` (no copy made, time-efficient)
Get (local only!)

1. Single Value: VecGetValues (use global numbering)
2. All local elements: VecGetArray (no copy made, time-efficient)

VecGetArray(Vec v, PetscScalar **array);
...
VecRestoreArray(Vec v, PetscScalar **array);
PETSc Vectors

### Get (local only!)

1. Single Value: `VecGetValues` (use global numbering)
2. All local elements: `VecGetArray` (no copy made, time-efficient)

```c
VecGetArray(Vec v, PetscScalar **array);
...
VecRestoreArray(Vec v, PetscScalar **array);
```

```c
call VecGetArray(Vec v, PetscScalar vv(1),
    PetscOffset offset, PetscErrorCode ierr)
... vv(offset + i) ...
call VecRestoreArray(...)
```
**PETSc Vectors**

### Get (local only!)

1. Single Value: `VecGetValues` (use global numbering)

2. All local elements: `VecGetArray` (no copy made, time-efficient)

```c
VecGetArray(Vec v, PetscScalar **array);
...
VecRestoreArray(Vec v, PetscScalar **array);
```

```fortran
call VecGetArray(Vec v, PetscScalar vv(1),
                 PetscOffset offset, PetscErrorCode ierr)
...  vv(offset + i) ...
call VecRestoreArray(...)  
```

```fortran
call VecGetArrayF90(Vec v, PetscScalar pointer vv,  
                    PetscErrorCode ierr)
...  
call VecRestoreArrayF90(...)  
```
PETSc Vectors

**Get** (local only!)

1. Single Value: `VecGetValues` (use global numbering)
2. All local elements: `VecGetArray` (no copy made, time-efficient)

**Operations**

- `VecAXPY` \[ y = a \times x + y, \]
- `VecDot` \[ x \cdot y, \]
- `VecNorm` \[ \|A\|... \]
- : : :
PETSc Vectors

4. Get (local only!)
   1. Single Value: VecGetValues (use global numbering)
   2. All local elements: VecGetArray (no copy made, time-efficient)

5. Operations

VecView, VecDuplicate, VecGetOwnershipRange, VecDestroy, ...
Summary

1. Introduction
2. Hello World
3. PETSc Vectors (Vec)
4. PETSc Matrices (Mat)
5. PETSc Krylov Solvers (KSP)
6. Wrapping up
PETSc Matrices

**Create**  
(sparse)

MatCreateAIJ(MPI_Comm comm, int m, int n, int M, int N,  
int d_nz, int d_nnz[], int o_nz, int o_nnz[],Mat *A);

where (Matrix stored by rows)

- m = local number of rows, or PETSC_DECIDE if M given
- M = global number of rows, or PETSC_DETERMINE if m given for all ranks
- n = m if square (in general: such that Mat-Vec product OK)
- N = global number of columns, or PETSC_DETERMINE if n given for all ranks
PETSc Matrices

Create (sparse)

MatCreateAIJ(MPI_Comm comm, int m, int n, int M, int N,
            int d_nz, int d_nnz[], int o_nz, int o_nnz[], Mat *A);

M = N = 8

\[
\begin{array}{cccc}
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\times & \times & \times & \times \\
\end{array}
\]

rank 0: m=n=3

rank 1: m=n=3

rank 2: m=n=2
**PETSc Matrices**

### Create (sparse)

MatCreateAIJ(MPI_Comm comm, int m, int n, int M, int N, int d_nz, int d_nnz[], int o_nz, int o_nnz[], Mat *A);

- **M = N = 8**

- d_nnz={2,2,2}, o_nnz={2,3,3}
- d_nnz={3,2,2}, o_nnz={1,1,1}
- d_nnz={1,1}, o_nnz={3,4}
PETSc Matrices

Create (sparse)

MatCreateAIJ(MPI_Comm comm, int m, int n, int M, int N,
    int d_nz, int d_nnz[], int o_nz, int o_nnz[], Mat *A);

M = N = 8

\[
\begin{array}{cccc|cccc}
\times & \times & | & \times & | & \times \\
\times & \times & x & | & x & | & x \\
\times & x & x & x & | & x \\
x & x & x & x & | & x \\
x & x & x & x & | & x \\
x & x & x & x & | & x \\
\end{array}
\]

\[d_{nnz} = \{2,2,2\}, \quad o_{nnz} = \{2,3,3\}\]

\[d_{nnz} = \{3,2,2\}, \quad o_{nnz} = \{1,1,1\}\]

\[d_{nnz} = \{1,1\}, \quad o_{nnz} = \{3,4\}\]

* d_nnz specified ⇒ d_nz ignored
* d_nnz = PETSC_NULL & d_nz ≥ max{ d_nnz }
PETSc Matrices

**Create**

MatCreateAIJ(MPI_Comm comm, int m, int n, int M, int N,
   int d_nz, int d_nnz[], int o_nz, int o_nnz[],Mat *A);

Instead of AIJ, one can also use Dense and BAIJ
**PETSc Matrices**

### Create

```c
MatCreateAIJ(MPI_Comm comm, int m, int n, int M, int N,
int d_nz, int d_nnz[], int o_nz, int o_nnz[],Mat *A);
```

Instead of **AIJ**, one can also use **Dense** and **BAIJ**

Sequential versions also available
PETSc Matrices

1. Create

2. Set

MatSetValues(..., INSERT_VALUES);

MatSetValues(..., ADD_VALUES);
PETSc Matrices

Create

Set

MatSetValues(..., INSERT_VALUES);

MatSetValues(..., ADD_VALUES);

If BAIJ, use MatSetValuesBlocked
PETSc Matrices

1. **Create**
   
   MatSetValues(..., INSERT_VALUES);

2. **Set**
   
   MatSetValues(..., ADD_VALUES);

3. **Assemble**
   
   MatAssemblyBegin(Mat A, MatAssemblyType maType);
   MatAssemblyEnd(Mat A, MatAssemblyType maType);

   where `maType` is
   
   MAT_FLUSH_ASSEMBLY between INSERT_VALUES and ADD_VALUES,
   MAT_FINAL_ASSEMBLY before using the matrix.
PETSc Matrices

**Operations**

- MatAXPY: $Y = a \times X + Y,$
- MatMult: $y = Ax,$
- MatMultAdd: $x_3 = x_2 + Ax_1,$
- MatNorm: $\|A\|...$
- MatTranspose: $A^T$

Additionally:

- MatView
- MatDuplicate
- MatGetOwnershipRange
- MatDestroy...
PETSc Matrices

- **Operations**

- **Matrix-Free**

```c
extern int mult(Mat, Vec, Vec);
MatCreateShell(comm, m, n, M, N, ctx, &mat);
MatShellSetOperation(mat, MATOP_MULT, (void(*)(void))mult);
```
PETSc Matrices

## Operations

Matrix-Free

```c
extern int mult(Mat, Vec, Vec);
MatCreateShell(comm, m, n, M, N, ctx, &mat);
MatShellSetOperation(mat, MATOP_MULT, (void(*)(void))mult);
```

- `MatView`, `MatDuplicate`, `MatGetOwnershipRange`, `MatDestroy`, ...
Summary

1. Introduction
2. Hello World
3. PETSc Vectors (Vec)
4. PETSc Matrices (Mat)
5. PETSc Krylov Solvers (KSP)
6. Wrapping up
Create

KSPCreate(MPI_Comm comm, KSP *ksp);
PETSc Krylov Solvers (KSP)

**Create**

KSPCreate(MPI_Comm comm, KSP *ksp);

**Set**

KSPSetType(KSP ksp, KSPType kspType);
KSPSetOperators(KSP ksp, Mat A, Mat PrecondBase,
                     MatStructure flag);
KSPSetUp(KSP ksp);

where kspType = KSPCG, KSPGMRES, ...
flag = SAME_PRECONDITIONER, SAME_NONZERO_PATTERN,
       DIFFERENT_NONZERO_PATTERN
**PETSc Krylov Solvers (KSP)**

1. **Create**

   ```c
   KSPCreate(MPI_Comm comm, KSP *ksp);
   ```

2. **Set**

   ```c
   KSPSetType(KSP ksp, KSPType kspType);
   KSPSetOperators(KSP ksp, Mat A, Mat PrecondBase, MatStructure flag);
   KSPSetUp(KSP ksp);
   ```

3. **Solve $A x = b$**

   ```c
   KSPSolve(KSP ksp, Vec b, Vec x);
   ```
Preconditioning

KSPgetPC(KSP ksp, PC *pc);
PCSetType(PC pc, PCType PCJACOBI);
PCSetUp(PC pc);

Also: KSPSetTolerances, KSPGetConvergedReason,
KSPGetIterationNumber, KSPGetResidualNorm, ...
PETSc Krylov Solvers (KSP) and Preconditioning (PC)

- **Preconditioning**

KSPgetPC(KSP ksp, PC *pc);
PCSetType(PC pc, PCType PCSOR);
PCSetUp(PC pc);

Also: KSPSetTolerances, KSPGetConvergedReason,
KSPGetIterationNumber, KSPGetResidualNorm, ...
Preconditioning

KSPgetPC(KSP ksp, PC *pc);
PCSetType(PC pc, PCTYPE PCILU);
PCFactorSetLevels(PC pc, int level_of_fill);
PCSetUp(PC pc);
Preconditioning

```c
KSPgetPC(KSP ksp, PC *pc);
PCSetType(PC pc, PCTYPE PCASM);
PCASMSetOverlap(PC pc, int overlap);
PCSetUp(PC pc);
PCASMGetSubKSP(PC pc, int *n_local, int *first_local,
                KSP *subKSP[]);
for (i=0; i<nlocal; i++){
    KSPSetType(subKSP(i), KSPtype GMRES);
    KSPGetPC(KSP subKSP(i), PC *subPC);
    PCSetType(PC subPC, PCTYPE PCSOR);
}
```
PETSc Krylov Solvers (KSP) and Preconditioning (PC)

Preconditioning

```c
KSPgetPC(KSP ksp, PC *pc);
PCSetType(PC pc, PCTYPE_PCASM);
PCASMSetOverlap(PC pc, int overlap);
PCSetUp(PC pc);
PCASMSGetSubKSP(PC pc, int *n_local, int *first_local,
    KSP *subKSP[]);
for (i=0; i<nlocal; i++){
    KSPSetType(subKSP(i), KSPType KSPPREONLY);
    KSPGetPC(KSP subKSP(i), PC *subPC);
    PCSetType(PC subPC, PCTYPE PCSOR);
}
```

NB: KSPPREONLY = default "sub-type"
**Direct Method**

KSPSetType(KSP ksp, KSPType KSPPREONLY);
KSPGetPC(KSP ksp, PC *pc);
PCSetType(PC pc, PCTYPE PCLU);
**PETSc Krylov Solvers (KSP) and Preconditioning (PC)**

- **Direct Method**

  ```c
  KSPSetType(KSP ksp, KSPType KSPPREONLY);
  KSPgetPC(KSP ksp, PC *pc);
  PCSetType(PC pc, PCTYPE PCLU);
  ```

- **External Solvers**

  ```c
  PCFactorSetMatSolverPackage(PC pc,
                              MatSolverPackage MATSOLVERPASTIX);
  ```
PETSc Krylov Solvers (KSP) and Preconditioning (PC)

- **Direct Method**
  
  ```c
  KSPSetType(KSP ksp, KSPType KSPPREONLY);
  KSPgetPC(KSP ksp, PC *pc);
  PCSetType(PC pc, PCType PCLU);
  ```

- **External Solvers**
  
  ```c
  PCFactorSetMatSolverPackage(PC pc, MatSolverPackage MATSOLVERMUMPS);
  ```
PETSc Krylov Solvers (KSP) and Preconditioning (PC)

- Direct Method
- External Solvers
- For run-time specification:

  KSPSetFromOptions(KSP ksp);
PETSc Krylov Solvers (KSP) and Preconditioning (PC)

- Direct Method
- External Solvers
- For run-time specification:

```c
KSPSetFromOptions(KSP ksp);
```

e.g., `-ksp_rtol <rtol> -ksp_type <method>`
PETSc Krylov Solvers (KSP) and Preconditioning (PC)

- Direct Method
- External Solvers
- For run-time specification:

  KSPSetFromOptions(KSP ksp);

  PCSetFromOptions(PC pc);
Summary

1. Introduction
2. Hello World
3. PETSc Vectors (Vec)
4. PETSc Matrices (Mat)
5. PETSc Krylov Solvers (KSP)
6. Wrapping up
PETSc main “categories”

- Vec: vectors
- Mat: matrices
- KSP: Krylov subspace iterative methods
- PC: preconditioners
PETSc main “categories”

- **Vec**: vectors
- **Mat**: matrices
- **KSP**: Krylov subspace iterative methods
- **PC**: preconditioners
- **DM**: Data Management between meshes and vectors/matrices, used to build grids, also unstructured.
PETSc main “categories”

- Vec: vectors
- Mat: matrices
- KSP: Krylov subspace iterative methods
- PC: preconditioners
- DM: Data Management between meshes and vectors/matrices, used to build grids, also unstructured.
- SNES: Scalable Nonlinear Equations Solvers (Newton-like methods)
PETSc main “categories”

- **Vec**: vectors
- **Mat**: matrices
- **KSP**: Krylov subspace iterative methods
- **PC**: preconditioners
- **DM**: Data Management between meshes and vectors/matrices, used to build grids, also unstructured.
- **SNES**: Scalable Nonlinear Equations Solvers (Newton-like methods)
- **TS**: Time-Stepping (for ODE)
Conclusions

- PETSc is powerful and widely used.
Conclusions

- PETSc is powerful and widely used.

- Once set-up, PETSc enables to try a lot of different solution methods.
Conclusions

- PETSc is powerful and widely used.

- Once set-up, PETSc enables to try a lot of different solution methods.

- Want to know more: come to the training!

By Jérémy Foulon, Loïc Gouarin, Serge Van Criekingen

- **In March**: from 18 to 20 at the Maison de la Simulation (Saclay)
- **In May-June**: in Lyon area
Conclusions

- PETSc is powerful and widely used.
- Once set-up, PETSc enables to try a lot of different solution methods.
- Want to know more: come to the training!

By Jérémy Foulon, Loïc Gouarin, Serge Van Criekingen

- In March: from 18 to 20 at the Maison de la Simulation (Saclay)
- In May-June: in Lyon area

“Bilingual” training: exercises in C or Fortran