FENaPack Documentation

Release 2019.2.0.dev0

Jan Blechta, Martin Řehoř

May 31, 2019

Manual

FENaPack is a package implementing preconditioners for Navier-Stokes problem using FEniCS and PETSc packages. In particular, variants of PCD (pressure-convection-diffussion) preconditioner from[1],[2] are implemented.

[1] Elman H. C., Silvester D. J., Wathen A. J., *Finite Elements and Fast Iterative Solvers: With Application in Incompressible Fluid Dynamics*. Oxford University Press 2005. 2nd edition 2014.

[2] Olshanskii M. A., Vassilevski Y. V., *Pressure Schur complement preconditioners for the discrete Oseen problem*. SIAM J. Sci. Comput., 29(6), 2686-2704. 2007.

CHAPTER 1

Usage

To use FENaPack matching version of FEniCS (version 2019.2.0.dev0) compiled with PETSc, petsc4py and mpi4py is needed. Note that FENaPack uses same version numbering as FEniCS and follows its release schedule with a short lag.

To install FENaPack from source do:

```
pip3 install [--user|--prefix=...] [-e] .
```

in the source/repository root dir. Editable install using `-e` allows to use FENaPack directly from source directory while editing it which is suitable for development.

You can install latest FENaPack release form PyPI:

```
pip3 install [--user|--prefix=...] fenapack
```

or install latest development version from Github:

```
pip3 install [--user|--prefix=...] git+https://github.com/blechta/fenapack
```

To start experimenting:

```
cd demo/navier-stokes-pcd
python3 demo_navier-stokes-pcd.py --help
python3 demo_navier-stokes-pcd.py [opts]
mpirun -n 16 python3 demo_navier-stokes-pcd.py [opts]
```

Full documentation is available at https://fenapack.readthedocs.io/.

CHAPTER 2

Authors

- Jan Blechta <blechta@karlin.mff.cuni.cz>
- Martin Řehoř <rehor@karlin.mff.cuni.cz>

CHAPTER 3

License

FENaPack is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

FENaPack is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with FENaPack. If not, see <http://www.gnu.org/licenses/>.

CHAPTER 4

Acknowledgement

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project „IT4Innovations National Supercomputing Center – LM2015070“.

CHAPTER 5

Links

- Homepage https://github.com/blechta/fenapack
- Testing https://circleci.com/gh/blechta/fenapack
- Documentation https://fenapack.readthedocs.io/
- Bug reports https://github.com/blechta/fenapack/issues
- PyPI home https://pypi.org/project/fenapack

5.1 Mathematical background

Navier-Stokes equations

$$\begin{bmatrix} -\nu\Delta + \mathbf{v}\cdot\nabla & \nabla \\ -\operatorname{div} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ 0 \end{bmatrix}$$

solved by GMRES preconditioned from right by

$$\mathbb{P} := \begin{bmatrix} -\nu\Delta + \mathbf{v}\cdot\nabla & \nabla \\ & -\mathbb{S} \end{bmatrix}$$

with Schur complement $\mathbb{S} = -\operatorname{div}(-\nu\Delta + \mathbf{v}\cdot\nabla)^{-1}\nabla$ would converge in two iterations. Unfortunately $\mathbb{S}$ is dense. Possible trick is to approximate $\mathbb{S}$ by swapping the order of the operators

$$\mathbb{S} \approx \mathbb{X}_{\mathrm{BRM1}} := (-\Delta)(-\nu\Delta + \mathbf{v}\cdot\nabla)^{-1}$$

or

$$\mathbb{S} \approx \mathbb{X}_{\mathrm{BRM2}} := (-\nu\Delta + \mathbf{v}\cdot\nabla)^{-1}(-\Delta).$$

This gives rise to the action of 11-block of preconditioner $\mathbb{P}^{-1}$ given by

$$\mathbb{X}^{-1}_{\mathrm{BRM1}} := (-\nu\Delta + \mathbf{v}\cdot\nabla)(-\Delta)^{-1}.$$

or

$$\mathbb{X}_{\mathrm{BRM2}}^{-1} := (-\Delta)^{-1}\left(-\nu\Delta + \mathbf{v}\cdot\nabla\right).$$

Obviously additional artificial boundary condition for Laplacian solve $-\Delta^{-1}$ is needed in the action of preconditioner. Modifying the approach from[2] we implement $\mathbb{X}_{\mathrm{BRM1}}^{-1}$ as

$$\mathbb{X}_{\mathrm{BRM1}}^{-1} := \mathbb{M}_p^{-1}(\mathbb{I} + \mathbb{K}_p\mathbb{A}_p^{-1})$$

where $\mathbb{M}_p$ is ν^{-1}-multiple of mass matrix on pressure, $\mathbb{K}_p \approx \nu^{-1}\mathbf{v}\cdot\nabla$ is a pressure convection matrix, and $\mathbb{A}_p^{-1} \approx (-\Delta)^{-1}$ is a pressure Laplacian solve with *zero boundary condition on inlet*. This is implemented by `fenapack.preconditioners.PCDPC_BRM1` and *PCD preconditioner for Navier-Stokes equations*.

Analogically we prefer to express BRM2 approach as

$$\mathbb{X}_{\mathrm{BRM2}}^{-1} := (\mathbb{I} + \mathbb{A}_p^{-1}\mathbb{K}_p)\mathbb{M}_p^{-1}$$

now with *zero boundary condition on outlet for Laplacian solve* and additional Robin term in convection matrix $\mathbb{K}_p$ roughly as stated in[1], section 9.2.2. See also *PCD preconditioner for Navier-Stokes equations* and `fenapack.preconditioners.PCDPC_BRM2`.

5.1.1 Extension to time-dependent problems (PCDR preconditioners)

Time disretization applied in unsteady problems typically leads to the need to incorporate a reaction term into the preconditioner. Typically, we end up with

$$\mathbb{X}_{\mathrm{BRM1}}^{-1} := \left(\frac{1}{\tau} - \nu\Delta + \mathbf{v}\cdot\nabla\right)(-\Delta)^{-1},$$

or

$$\mathbb{X}_{\mathrm{BRM2}}^{-1} := (-\Delta)^{-1}\left(\frac{1}{\tau} - \nu\Delta + \mathbf{v}\cdot\nabla\right),$$

where τ denotes a fixed time step and the original PCD preconditioner thus becomes PCDR (pressure-convection-diffusion-reaction) preconditioner. A straightforward way of how to implement the above actions is to update the pressure convection matrix $\mathbb{K}_p$ by a contribution corresponding to the scaled pressure mass matrix, namely

$$\mathbb{X}_{\mathrm{BRM1}}^{-1} := \mathbb{M}_p^{-1}\left(\mathbb{I} + \left(\mathbb{K}_p + \tau^{-1}\mathbb{M}_p\right)\mathbb{A}_p^{-1}\right),$$

or

$$\mathbb{X}_{\mathrm{BRM2}}^{-1} := \left(\mathbb{I} + \mathbb{A}_p^{-1}\left(\mathbb{K}_p + \tau^{-1}\mathbb{M}_p\right)\right)\mathbb{M}_p^{-1}.$$

However, for unsteady problems we prefer to use the following elaborated implementation of PCDR preconditioners, namely

$$\mathbb{X}_{\mathrm{BRM1}}^{-1} := \mathbb{R}_p^{-1} + \mathbb{M}_p^{-1}(\mathbb{I} + \mathbb{K}_p\mathbb{A}_p^{-1}),$$

or

$$\mathbb{X}_{\mathrm{BRM2}}^{-1} := \mathbb{R}_p^{-1} + (\mathbb{I} + \mathbb{A}_p^{-1}\mathbb{K}_p)\mathbb{M}_p^{-1},$$

[2] Olshanskii M. A., Vassilevski Y. V., *Pressure Schur complement preconditioners for the discrete Oseen problem*. SIAM J. Sci. Comput., 29(6), 2686-2704. 2007.

[1] Elman H. C., Silvester D. J., Wathen A. J., *Finite Elements and Fast Iterative Solvers: With Application in Incompressible Fluid Dynamics*. Oxford University Press 2005. 2nd edition 2014.

where $\mathbb{R}_p^{-1} \approx \frac{1}{\tau}(-\Delta)^{-1}$, while $\mathbb{R}_p$ itself is approximated and implemented as

$$\mathbb{R}_p := \mathbb{B}\left(\tau^{-1}\mathbb{D}_{\mathrm{M}}\right)^{-1}\mathbb{B}^T,$$

Here, $\mathbb{D}_{\mathrm{M}}$ is the diagonal of the velocity mass matrix, $\mathbb{D}_{\mathrm{M}} = \mathrm{diag}(\mathbb{M}_{\mathbf{u}})$, and $\mathbb{B}^T$ corresponds to the discrete pressure gradient which is obtained as the 01-block of the original system matrix. Let us emphasize that this submatrix is extracted from **the system matrix with velocity Dirichlet boundary conditions being applied on it**.

The choice of $\mathbb{R}_p$ as above can be justified especially in the case of $\tau \to 0_+$, for which

$$\mathbb{S}^{-1} := \left(-\operatorname{div}\left(\frac{1}{\tau} - \nu\Delta + \mathbf{v}\cdot\nabla\right)^{-1}\nabla\right)^{-1} \approx \frac{1}{\tau}\left(\mathbb{B}\mathbb{M}_{\mathbf{u}}^{-1}\mathbb{B}^T\right)^{-1},$$

and simultaneously $\mathbb{X}^{-1} \approx \mathbb{R}_p^{-1} = \frac{1}{\tau}\left(\mathbb{B}\mathbb{D}_{\mathrm{M}}^{-1}\mathbb{B}^T\right)^{-1}$. The same approximation of the minus Laplacian operator was previously used also in[1], see Remark 9.6 therein.

5.2 PCD preconditioner for Navier-Stokes equations

This demo is implemented in a single Python file, `demo_navier-stokes-pcd.py`, which contains both the variational forms and the solver.

5.2.1 Underlying mathematics

See *Mathematical background*.

5.2.2 Implementation

Only features beyond standard FEniCS usage will be explained in this document.

Here comes an artificial boundary condition for PCD operators. Zero Dirichlet condition for Laplacian solve is applied either on inlet or outlet, depending on the variant of PCD. Note that it is defined on pressure subspace of the mixed space `W`.

```
# Artificial BC for PCD preconditioner
if args.pcd_variant == "BRM1":
    bc_pcd = DirichletBC(W.sub(1), 0.0, boundary_markers, 1)
elif args.pcd_variant == "BRM2":
    bc_pcd = DirichletBC(W.sub(1), 0.0, boundary_markers, 2)
```

Then comes standard formulation of the nonlinear equation

```
# Arguments and coefficients of the form
u, p = TrialFunctions(W)
v, q = TestFunctions(W)
w = Function(W)
u_, p_ = split(w)
nu = Constant(args.viscosity)

# Nonlinear equation
F = (
      nu*inner(grad(u_), grad(v))
    + inner(dot(grad(u_), u_), v)
```

(continues on next page)

(continued from previous page)

```
    - p_*div(v)
    - q*div(u_)
)*dx
```

We will provide a possibility to mock Newton solver into Picard iteration by passing Oseen linearization as Jacobian `J`

```
# Jacobian
if args.nls == "picard":
    J = (
          nu*inner(grad(u), grad(v))
        + inner(dot(grad(u), u_), v)
        - p*div(v)
        - q*div(u)
    )*dx
elif args.nls == "newton":
    J = derivative(F, w)
```

"Preconditioner" Jacobian `J_pc` features added streamline diffusion to stabilize 00-block if algebraic multigrid is used. Otherwise we can pass `None` as a precoditioner Jacobian to use the system matrix for preparing the preconditioner.

```
# Add stabilization for AMG 00-block
if args.ls == "iterative":
    delta = StabilizationParameterSD(w.sub(0), nu)
    J_pc = J + delta*inner(dot(grad(u), u_), dot(grad(v), u_))*dx
elif args.ls == "direct":
    J_pc = None
```

L^2 scalar product ("mass matrix") `mp`, convection operator `kp`, and Laplacian `ap` to be used by `PCD BRM preconditioner` are defined using pressure components `p`, `q` on the mixed space `W`. They are passed to the class *`fenapack.assembling.PCDAssembler`* which takes care of assembling the operators on demand.

```
# PCD operators
mp = Constant(1.0/nu)*p*q*dx
kp = Constant(1.0/nu)*dot(grad(p), u_)*q*dx
ap = inner(grad(p), grad(q))*dx
if args.pcd_variant == "BRM2":
    n = FacetNormal(mesh)
    ds = Measure("ds", subdomain_data=boundary_markers)
    kp -= Constant(1.0/nu)*dot(u_, n)*p*q*ds(1)

# Collect forms to define nonlinear problem
pcd_assembler = PCDAssembler(J, F, [bc0, bc1],
                             J_pc, ap=ap, kp=kp, mp=mp, bcs_pcd=bc_pcd)
problem = PCDNonlinearProblem(pcd_assembler)
```

Now we create GMRES preconditioned with PCD, set the tolerance, enable monitoring of residual during Krylov iterarations, and set the maximal dimension of Krylov subspaces.

```
# Set up linear solver (GMRES with right preconditioning using Schur fact)
linear_solver = PCDKrylovSolver(comm=mesh.mpi_comm())
linear_solver.parameters["relative_tolerance"] = 1e-6
PETScOptions.set("ksp_monitor")
PETScOptions.set("ksp_gmres_restart", 150)
```

Next we choose a variant of PCD according to a parameter value

```
# Set up subsolvers
PETScOptions.set("fieldsplit_p_pc_python_type", "fenapack.PCDPC_" + args.pcd_variant)
```

00-block solve and PCD Laplacian solve can be performed using algebraic multigrid

```
if args.ls == "iterative":
    PETScOptions.set("fieldsplit_u_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_u_ksp_max_it", 1)
    PETScOptions.set("fieldsplit_u_pc_type", "hypre")
    PETScOptions.set("fieldsplit_u_pc_hypre_type", "boomeramg")
    PETScOptions.set("fieldsplit_p_PCD_Ap_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_p_PCD_Ap_ksp_max_it", 2)
    PETScOptions.set("fieldsplit_p_PCD_Ap_pc_type", "hypre")
    PETScOptions.set("fieldsplit_p_PCD_Ap_pc_hypre_type", "boomeramg")
```

PCD mass matrix solve can be efficiently performed using Chebyshev iteration preconditioned by Jacobi method. The eigenvalue estimates come from[1], Lemma 4.3. **Don't forget to change them appropriately when changing dimension/element. Neglecting this can lead to substantially worse convergence rates.**

```
PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_type", "chebyshev")
PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_max_it", 5)
PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_chebyshev_eigenvalues", "0.5, 2.0")
PETScOptions.set("fieldsplit_p_PCD_Mp_pc_type", "jacobi")
```

The direct solver is used by default if the aforementioned blocks are not executed. FENaPack tries to pick MUMPS by default and following parameter enables very verbose output.

```
elif args.ls == "direct" and args.mumps_debug:
    # Debugging MUMPS
    PETScOptions.set("fieldsplit_u_mat_mumps_icntl_4", 2)
    PETScOptions.set("fieldsplit_p_PCD_Ap_mat_mumps_icntl_4", 2)
    PETScOptions.set("fieldsplit_p_PCD_Mp_mat_mumps_icntl_4", 2)
```

Let the linear solver use the options

```
# Apply options
linear_solver.set_from_options()
```

Finally we invoke a Newton solver modification suitable to be used used with PCD solver.

```
# Set up nonlinear solver
solver = PCDNewtonSolver(linear_solver)
solver.parameters["relative_tolerance"] = 1e-5

# Solve problem
solver.solve(problem, w.vector())
```

5.2.3 Complete code

Show/Hide Code

Download Code

[1] Elman H. C., Silvester D. J., Wathen A. J., *Finite Elements and Fast Iterative Solvers: With Application in Incompressible Fluid Dynamics*. Oxford University Press 2005. 2nd edition 2014.

```
from dolfin import *
from matplotlib import pyplot

from fenapack import PCDKrylovSolver
from fenapack import PCDAssembler
from fenapack import PCDNewtonSolver, PCDNonlinearProblem
from fenapack import StabilizationParameterSD

import argparse, sys, os

# Parse input arguments
parser = argparse.ArgumentParser(description=__doc__, formatter_class=
                                 argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument("-l", type=int, dest="level", default=4,
                    help="level of mesh refinement")
parser.add_argument("--nu", type=float, dest="viscosity", default=0.02,
                    help="kinematic viscosity")
parser.add_argument("--pcd", type=str, dest="pcd_variant", default="BRM2",
                    choices=["BRM1", "BRM2"], help="PCD variant")
parser.add_argument("--nls", type=str, dest="nls", default="picard",
                    choices=["picard", "newton"], help="nonlinear solver")
parser.add_argument("--ls", type=str, dest="ls", default="iterative",
                    choices=["direct", "iterative"], help="linear solvers")
parser.add_argument("--dm", action='store_true', dest="mumps_debug",
                    help="debug MUMPS")
args = parser.parse_args(sys.argv[1:])

# Load mesh from file and refine uniformly
mesh = Mesh(os.path.join(os.path.pardir, "data", "mesh_lshape.xml"))
for i in range(args.level):
    mesh = refine(mesh)

# Define and mark boundaries
class Gamma0(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary
class Gamma1(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], -1.0)
class Gamma2(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 5.0)
boundary_markers = MeshFunction("size_t", mesh, mesh.topology().dim()-1)
boundary_markers.set_all(3)        # interior facets
Gamma0().mark(boundary_markers, 0) # no-slip facets
Gamma1().mark(boundary_markers, 1) # inlet facets
Gamma2().mark(boundary_markers, 2) # outlet facets

# Build Taylor-Hood function space
P2 = VectorElement("Lagrange", mesh.ufl_cell(), 2)
P1 = FiniteElement("Lagrange", mesh.ufl_cell(), 1)
W = FunctionSpace(mesh, P2*P1)

# No-slip BC
bc0 = DirichletBC(W.sub(0), (0.0, 0.0), boundary_markers, 0)

# Parabolic inflow BC
```

(continues on next page)

(continued from previous page)

```
inflow = Expression(("4.0*x[1]*(1.0 - x[1])", "0.0"), degree=2)
bc1 = DirichletBC(W.sub(0), inflow, boundary_markers, 1)

# Artificial BC for PCD preconditioner
if args.pcd_variant == "BRM1":
    bc_pcd = DirichletBC(W.sub(1), 0.0, boundary_markers, 1)
elif args.pcd_variant == "BRM2":
    bc_pcd = DirichletBC(W.sub(1), 0.0, boundary_markers, 2)

# Provide some info about the current problem
info("Reynolds number: Re = %g" % (2.0/args.viscosity))
info("Dimension of the function space: %g" % W.dim())

# Arguments and coefficients of the form
u, p = TrialFunctions(W)
v, q = TestFunctions(W)
w = Function(W)
# FIXME: Which split is correct? Both work but one might use
# restrict_as_ufc_function
u_, p_ = split(w)
#u_, p_ = w.split()
nu = Constant(args.viscosity)

# Nonlinear equation
F = (
      nu*inner(grad(u_), grad(v))
    + inner(dot(grad(u_), u_), v)
    - p_*div(v)
    - q*div(u_)
)*dx

# Jacobian
if args.nls == "picard":
    J = (
          nu*inner(grad(u), grad(v))
        + inner(dot(grad(u), u_), v)
        - p*div(v)
        - q*div(u)
    )*dx
elif args.nls == "newton":
    J = derivative(F, w)

# Add stabilization for AMG 00-block
if args.ls == "iterative":
    delta = StabilizationParameterSD(w.sub(0), nu)
    J_pc = J + delta*inner(dot(grad(u), u_), dot(grad(v), u_))*dx
elif args.ls == "direct":
    J_pc = None

# PCD operators
mp = Constant(1.0/nu)*p*q*dx
kp = Constant(1.0/nu)*dot(grad(p), u_)*q*dx
ap = inner(grad(p), grad(q))*dx
if args.pcd_variant == "BRM2":
    n = FacetNormal(mesh)
    ds = Measure("ds", subdomain_data=boundary_markers)
    kp -= Constant(1.0/nu)*dot(u_, n)*p*q*ds(1)
```

(continues on next page)

(continued from previous page)

```
    #kp -= Constant(1.0/nu)*dot(u_, n)*p*q*ds(0)  # TODO: Is this beneficial?

# Collect forms to define nonlinear problem
pcd_assembler = PCDAssembler(J, F, [bc0, bc1],
                             J_pc, ap=ap, kp=kp, mp=mp, bcs_pcd=bc_pcd)
problem = PCDNonlinearProblem(pcd_assembler)

# Set up linear solver (GMRES with right preconditioning using Schur fact)
linear_solver = PCDKrylovSolver(comm=mesh.mpi_comm())
linear_solver.parameters["relative_tolerance"] = 1e-6
PETScOptions.set("ksp_monitor")
PETScOptions.set("ksp_gmres_restart", 150)

# Set up subsolvers
PETScOptions.set("fieldsplit_p_pc_python_type", "fenapack.PCDPC_" + args.pcd_variant)
if args.ls == "iterative":
    PETScOptions.set("fieldsplit_u_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_u_ksp_max_it", 1)
    PETScOptions.set("fieldsplit_u_pc_type", "hypre")
    PETScOptions.set("fieldsplit_u_pc_hypre_type", "boomeramg")
    PETScOptions.set("fieldsplit_p_PCD_Ap_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_p_PCD_Ap_ksp_max_it", 2)
    PETScOptions.set("fieldsplit_p_PCD_Ap_pc_type", "hypre")
    PETScOptions.set("fieldsplit_p_PCD_Ap_pc_hypre_type", "boomeramg")
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_type", "chebyshev")
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_max_it", 5)
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_chebyshev_eigenvalues", "0.5, 2.0")
    #PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_chebyshev_esteig", "1,0,0,1")  #␣
↪FIXME: What does it do?
    PETScOptions.set("fieldsplit_p_PCD_Mp_pc_type", "jacobi")
elif args.ls == "direct" and args.mumps_debug:
    # Debugging MUMPS
    PETScOptions.set("fieldsplit_u_mat_mumps_icntl_4", 2)
    PETScOptions.set("fieldsplit_p_PCD_Ap_mat_mumps_icntl_4", 2)
    PETScOptions.set("fieldsplit_p_PCD_Mp_mat_mumps_icntl_4", 2)

# Apply options
linear_solver.set_from_options()

# Set up nonlinear solver
solver = PCDNewtonSolver(linear_solver)
solver.parameters["relative_tolerance"] = 1e-5

# Solve problem
solver.solve(problem, w.vector())

# Report timings
list_timings(TimingClear.clear, [TimingType.wall, TimingType.user])

# Plot solution
u, p = w.split()
size = MPI.size(mesh.mpi_comm())
rank = MPI.rank(mesh.mpi_comm())
pyplot.figure()
pyplot.subplot(2, 1, 1)
plot(u, title="velocity")
pyplot.subplot(2, 1, 2)
```

(continues on next page)

(continued from previous page)

```
plot(p, title="pressure")
pyplot.savefig("figure_v_p_size{}_rank{}.pdf".format(size, rank))
pyplot.figure()
plot(p, title="pressure", mode="warp")
pyplot.savefig("figure_warp_size{}_rank{}.pdf".format(size, rank))
pyplot.show()
```

5.3 PCD(R) preconditioner for unsteady Navier-Stokes equations

This demo is implemented in two Python files, `demo_navier-stokes-pcd.py` and `demo_navier-stokes-pcdr.py`, so it is easier to make comparison of PCD vs. PCDR. Each python file contains both the variational forms and the solver. The differences between the two files are marked by **#PCDR-DIFF** and described below in detail.

5.3.1 Underlying mathematics

See *Mathematical background*, especially *Extension to time-dependent problems (PCDR preconditioners)*.

5.3.2 Implementation

This demo extends *PCD preconditioner for Navier-Stokes equations* to time-dependent problems. Only the differences will be explained.

The nonlinear equation now contains terms from the time derivative (we use backward implicit Euler).

```
# Arguments and coefficients of the form
u, p = TrialFunctions(W)
v, q = TestFunctions(W)
w = Function(W)
w0 = Function(W)
u_, p_ = split(w)
u0_, p0_ = split(w0)
nu = Constant(args.viscosity)
idt = Constant(1.0/args.dt)

# Nonlinear equation
F = (
      idt*inner(u_ - u0_, v)
    + nu*inner(grad(u_), grad(v))
    + inner(dot(grad(u_), u_), v)
    - p_*div(v)
    - q*div(u_)
)*dx
```

The same holds for the Jacobian `J` that can be used to mock Newton solver into Picard iteration.

```
# Jacobian
if args.nls == "picard":
    J = (
          idt*inner(u, v)
        + nu*inner(grad(u), grad(v))
```

(continues on next page)

(continued from previous page)

```
        + inner(dot(grad(u), u_), v)
        - p*div(v)
        - q*div(u)
    )*dx
elif args.nls == "newton":
    J = derivative(F, w)
```

If we wish to use any of the `PCD BRM preconditioners`, then we need to enrich the *convection* operator `kp` by the *reaction* term from the time derivative.

```
# PCD operators
mp = Constant(1.0/nu)*p*q*dx
kp = Constant(1.0/nu)*(idt*p + dot(grad(p), u_))*q*dx
ap = inner(grad(p), grad(q))*dx
if args.pcd_variant == "BRM2":
    n = FacetNormal(mesh)
    ds = Measure("ds", subdomain_data=boundary_markers)
    kp -= Constant(1.0/nu)*dot(u_, n)*p*q*ds(1)

# Collect forms to define nonlinear problem
pcd_assembler = PCDAssembler(J, F, [bc0, bc1],
                             J_pc, ap=ap, kp=kp, mp=mp, bcs_pcd=bc_pcd)
problem = PCDNonlinearProblem(pcd_assembler)
```

#PCDR-DIFF No. 1: If we wish to use any of the `PCDR BRM preconditioners`, then the *convection* operator `kp` remains unchanged, but we need to supply the velocity mass matrix `mu = idt*inner(u, v)*dx` to *`fenapack.assembling.PCDAssembler`*. The pressure gradient `gp = -p_*div(v)` does not have to be assembled as it can be extracted from the Jacobian `J`.

```
# Collect forms to define nonlinear problem
pcd_assembler = PCDAssembler(J, F, [bc0, bc1],
                             J_pc, ap=ap, kp=kp, mp=mp, mu=mu, bcs_pcd=bc_pcd)
assert pcd_assembler.get_pcd_form("gp").phantom # pressure grad obtained from J
```

#PCDR-DIFF No. 2: The fact that we want to use the PCDR preconditioner must be invoked from options.

```
# Set up subsolvers
PETScOptions.set("fieldsplit_p_pc_python_type", "fenapack.PCDRPC_" + args.pcd_variant)
```

#PCDR-DIFF No. 3: The Laplacian solve related to the *reaction* term can be performed using algebraic multigrid. (The direct solver is used by default if the following block is not executed.)

```
if args.ls == "iterative":
    PETScOptions.set("fieldsplit_p_PCD_Rp_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_p_PCD_Rp_ksp_max_it", 1)
    PETScOptions.set("fieldsplit_p_PCD_Rp_pc_type", "hypre")
    PETScOptions.set("fieldsplit_p_PCD_Rp_pc_hypre_type", "boomeramg")
```

Try to run

```
python3 demo_unsteady-navier-stokes-pcd.py --pcd BRM1
python3 demo_unsteady-navier-stokes-pcdr.py --pcd BRM1
```

to see that the results can look like this:

No. of DOF	Steps	Krylov its	Krylov its (p.t.s.)	Time (s)
25987	25	3157	126.3	99.21
25987	25	1686	67.4	67.65

Let us remark that the difference in case `--pcd BRM2` is not so striking.

5.3.3 Complete code

Show/Hide Code `demo_unsteady-navier-stokes-pcd.py`

`Download Code`

```
from dolfin import *
from matplotlib import pyplot

from fenapack import PCDKrylovSolver
from fenapack import PCDAssembler
from fenapack import PCDNewtonSolver, PCDNonlinearProblem
from fenapack import StabilizationParameterSD

import argparse, sys, os

# Parse input arguments
parser = argparse.ArgumentParser(description=__doc__, formatter_class=
                                 argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument("-l", type=int, dest="level", default=4,
                    help="level of mesh refinement")
parser.add_argument("--nu", type=float, dest="viscosity", default=0.02,
                    help="kinematic viscosity")
parser.add_argument("--pcd", type=str, dest="pcd_variant", default="BRM1",
                    choices=["BRM1", "BRM2"], help="PCD variant")
parser.add_argument("--nls", type=str, dest="nls", default="picard",
                    choices=["picard", "newton"], help="nonlinear solver")
parser.add_argument("--ls", type=str, dest="ls", default="direct",
                    choices=["direct", "iterative"], help="linear solvers")
parser.add_argument("--dm", action='store_true', dest="mumps_debug",
                    help="debug MUMPS")
parser.add_argument("--dt", type=float, dest="dt", default=0.2,
                    help="time step")
parser.add_argument("--t_end", type=float, dest="t_end", default=5.0,
                    help="termination time")
args = parser.parse_args(sys.argv[1:])

# Load mesh from file and refine uniformly
mesh = Mesh(os.path.join(os.path.pardir, "data", "mesh_lshape.xml"))
for i in range(args.level):
    mesh = refine(mesh)

# Define and mark boundaries
class Gamma0(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary
class Gamma1(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], -1.0)
class Gamma2(SubDomain):
```

(continues on next page)

(continued from previous page)

```
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 5.0)
boundary_markers = MeshFunction("size_t", mesh, mesh.topology().dim()-1)
boundary_markers.set_all(3)        # interior facets
Gamma0().mark(boundary_markers, 0) # no-slip facets
Gamma1().mark(boundary_markers, 1) # inlet facets
Gamma2().mark(boundary_markers, 2) # outlet facets

# Build Taylor-Hood function space
P2 = VectorElement("Lagrange", mesh.ufl_cell(), 2)
P1 = FiniteElement("Lagrange", mesh.ufl_cell(), 1)
W = FunctionSpace(mesh, P2*P1)

# No-slip BC
bc0 = DirichletBC(W.sub(0), (0.0, 0.0), boundary_markers, 0)

# Parabolic inflow BC
inflow = Expression(("(1.0 - exp(-5.0*t))*4.0*x[1]*(1.0 - x[1])", "0.0"),
                    t=0.0, degree=2)
bc1 = DirichletBC(W.sub(0), inflow, boundary_markers, 1)

# Artificial BC for PCD preconditioner
if args.pcd_variant == "BRM1":
    bc_pcd = DirichletBC(W.sub(1), 0.0, boundary_markers, 1)
elif args.pcd_variant == "BRM2":
    bc_pcd = DirichletBC(W.sub(1), 0.0, boundary_markers, 2)

# Provide some info about the current problem
info("Reynolds number: Re = %g" % (2.0/args.viscosity))
info("Dimension of the function space: %g" % W.dim())

# Arguments and coefficients of the form
u, p = TrialFunctions(W)
v, q = TestFunctions(W)
w = Function(W)
w0 = Function(W)
u_, p_ = split(w)
u0_, p0_ = split(w0)
nu = Constant(args.viscosity)
idt = Constant(1.0/args.dt)

# Nonlinear equation
F = (
      idt*inner(u_ - u0_, v)
    + nu*inner(grad(u_), grad(v))
    + inner(dot(grad(u_), u_), v)
    - p_*div(v)
    - q*div(u_)
)*dx

# Jacobian
if args.nls == "picard":
    J = (
          idt*inner(u, v)
        + nu*inner(grad(u), grad(v))
        + inner(dot(grad(u), u_), v)
        - p*div(v)
```

(continues on next page)

(continued from previous page)

```
        - q*div(u)
    )*dx
elif args.nls == "newton":
    J = derivative(F, w)

# Add stabilization for AMG 00-block
if args.ls == "iterative":
    delta = StabilizationParameterSD(w.sub(0), nu)
    J_pc = J + delta*inner(dot(grad(u), u_), dot(grad(v), u_))*dx
elif args.ls == "direct":
    J_pc = None

# PCD operators
mp = Constant(1.0/nu)*p*q*dx
kp = Constant(1.0/nu)*(idt*p + dot(grad(p), u_))*q*dx
ap = inner(grad(p), grad(q))*dx
if args.pcd_variant == "BRM2":
    n = FacetNormal(mesh)
    ds = Measure("ds", subdomain_data=boundary_markers)
    # TODO: What about the reaction term? Does it appear here?
    kp -= Constant(1.0/nu)*dot(u_, n)*p*q*ds(1)
    #kp -= Constant(1.0/nu)*dot(u_, n)*p*q*ds(0)  # TODO: Is this beneficial?

# Collect forms to define nonlinear problem
pcd_assembler = PCDAssembler(J, F, [bc0, bc1],
                             J_pc, ap=ap, kp=kp, mp=mp, bcs_pcd=bc_pcd)
problem = PCDNonlinearProblem(pcd_assembler)

# Set up linear solver (GMRES with right preconditioning using Schur fact)
linear_solver = PCDKrylovSolver(comm=mesh.mpi_comm())
linear_solver.parameters["relative_tolerance"] = 1e-6
#PETScOptions.set("ksp_monitor")
PETScOptions.set("ksp_gmres_restart", 150)

# Set up subsolvers
PETScOptions.set("fieldsplit_p_pc_python_type", "fenapack.PCDPC_" + args.pcd_variant)
if args.ls == "iterative":
    PETScOptions.set("fieldsplit_u_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_u_ksp_max_it", 1)
    PETScOptions.set("fieldsplit_u_pc_type", "hypre")
    PETScOptions.set("fieldsplit_u_pc_hypre_type", "boomeramg")
    PETScOptions.set("fieldsplit_p_PCD_Ap_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_p_PCD_Ap_ksp_max_it", 2)
    PETScOptions.set("fieldsplit_p_PCD_Ap_pc_type", "hypre")
    PETScOptions.set("fieldsplit_p_PCD_Ap_pc_hypre_type", "boomeramg")
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_type", "chebyshev")
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_max_it", 5)
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_chebyshev_eigenvalues", "0.5, 2.0")
    #PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_chebyshev_esteig", "1,0,0,1")  #␣
↪FIXME: What does it do?
    PETScOptions.set("fieldsplit_p_PCD_Mp_pc_type", "jacobi")
elif args.ls == "direct" and args.mumps_debug:
    # Debugging MUMPS
    PETScOptions.set("fieldsplit_u_mat_mumps_icntl_4", 2)
    PETScOptions.set("fieldsplit_p_PCD_Ap_mat_mumps_icntl_4", 2)
    PETScOptions.set("fieldsplit_p_PCD_Mp_mat_mumps_icntl_4", 2)
```

(continues on next page)

(continued from previous page)

```
# Apply options
linear_solver.set_from_options()

# Set up nonlinear solver
solver = PCDNewtonSolver(linear_solver)
solver.parameters["relative_tolerance"] = 1e-5

# Solve problem
t = 0.0
time_iters = 0
krylov_iters = 0
solution_time = 0.0
while t < args.t_end and not near(t, args.t_end, 0.1*args.dt):
    # Move to current time level
    t += args.dt
    time_iters += 1

    # Update boundary conditions
    inflow.t = t

    # Solve the nonlinear problem
    info("t = {:g}, step = {:g}, dt = {:g}".format(t, time_iters, args.dt))
    with Timer("Solve") as t_solve:
        newton_iters, converged = solver.solve(problem, w.vector())
    krylov_iters += solver.krylov_iterations()
    solution_time += t_solve.stop()

    # Update variables at previous time level
    w0.assign(w)

# Report timings
list_timings(TimingClear.clear, [TimingType.wall, TimingType.user])

# Get iteration counts
result = {
    "ndof": W.dim(), "time": solution_time, "steps": time_iters,
    "lin_its": krylov_iters, "lin_its_avg": float(krylov_iters)/time_iters}
tab = "{:^15} | {:^15} | {:^15} | {:^19} | {:^15}\n".format(
    "No. of DOF", "Steps", "Krylov its", "Krylov its (p.t.s.)", "Time (s)")
tab += "{ndof:>9}       | {steps:^15} | {lin_its:^15} | " \
       "{lin_its_avg:^19.1f} | {time:^15.2f}\n".format(**result)
print("\nSummary of iteration counts:")
print(tab)
#with open("table_pcd_{}.txt".format(args.pcd_variant), "w") as f:
#    f.write(tab)

# Plot solution
u, p = w.split()
size = MPI.size(mesh.mpi_comm())
rank = MPI.rank(mesh.mpi_comm())
pyplot.figure()
pyplot.subplot(2, 1, 1)
plot(u, title="velocity")
pyplot.subplot(2, 1, 2)
plot(p, title="pressure")
pyplot.savefig("figure_v_p_size{}_rank{}.pdf".format(size, rank))
pyplot.figure()
```

(continues on next page)

(continued from previous page)

```
plot(p, title="pressure", mode="warp")
pyplot.savefig("figure_warp_size{}_rank{}.pdf".format(size, rank))
pyplot.show()
```

Show/Hide Code demo_unsteady-navier-stokes-pcdr.py

Download Code

```
from dolfin import *
from matplotlib import pyplot

from fenapack import PCDKrylovSolver
from fenapack import PCDAssembler
from fenapack import PCDNewtonSolver, PCDNonlinearProblem
from fenapack import StabilizationParameterSD

import argparse, sys, os

# Parse input arguments
parser = argparse.ArgumentParser(description=__doc__, formatter_class=
                                 argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument("-l", type=int, dest="level", default=4,
                    help="level of mesh refinement")
parser.add_argument("--nu", type=float, dest="viscosity", default=0.02,
                    help="kinematic viscosity")
parser.add_argument("--pcd", type=str, dest="pcd_variant", default="BRM1",
                    choices=["BRM1", "BRM2"], help="PCD variant")
parser.add_argument("--nls", type=str, dest="nls", default="picard",
                    choices=["picard", "newton"], help="nonlinear solver")
parser.add_argument("--ls", type=str, dest="ls", default="direct",
                    choices=["direct", "iterative"], help="linear solvers")
parser.add_argument("--dm", action='store_true', dest="mumps_debug",
                    help="debug MUMPS")
parser.add_argument("--dt", type=float, dest="dt", default=0.2,
                    help="time step")
parser.add_argument("--t_end", type=float, dest="t_end", default=5.0,
                    help="termination time")
args = parser.parse_args(sys.argv[1:])

# Load mesh from file and refine uniformly
mesh = Mesh(os.path.join(os.path.pardir, "data", "mesh_lshape.xml"))
for i in range(args.level):
    mesh = refine(mesh)

# Define and mark boundaries
class Gamma0(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary
class Gamma1(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], -1.0)
class Gamma2(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 5.0)
boundary_markers = MeshFunction("size_t", mesh, mesh.topology().dim()-1)
boundary_markers.set_all(3)        # interior facets
Gamma0().mark(boundary_markers, 0) # no-slip facets
```

(continues on next page)

(continued from previous page)

```
Gamma1().mark(boundary_markers, 1) # inlet facets
Gamma2().mark(boundary_markers, 2) # outlet facets

# Build Taylor-Hood function space
P2 = VectorElement("Lagrange", mesh.ufl_cell(), 2)
P1 = FiniteElement("Lagrange", mesh.ufl_cell(), 1)
W = FunctionSpace(mesh, P2*P1)

# No-slip BC
bc0 = DirichletBC(W.sub(0), (0.0, 0.0), boundary_markers, 0)

# Parabolic inflow BC
inflow = Expression(("(1.0 - exp(-5.0*t))*4.0*x[1]*(1.0 - x[1])", "0.0"),
                    t=0.0, degree=2)
bc1 = DirichletBC(W.sub(0), inflow, boundary_markers, 1)

# Artificial BC for PCD preconditioner
if args.pcd_variant == "BRM1":
    bc_pcd = DirichletBC(W.sub(1), 0.0, boundary_markers, 1)
elif args.pcd_variant == "BRM2":
    bc_pcd = DirichletBC(W.sub(1), 0.0, boundary_markers, 2)

# Provide some info about the current problem
info("Reynolds number: Re = %g" % (2.0/args.viscosity))
info("Dimension of the function space: %g" % W.dim())

# Arguments and coefficients of the form
u, p = TrialFunctions(W)
v, q = TestFunctions(W)
w = Function(W)
w0 = Function(W)
u_, p_ = split(w)
u0_, p0_ = split(w0)
nu = Constant(args.viscosity)
idt = Constant(1.0/args.dt)

# Nonlinear equation
F = (
      idt*inner(u_ - u0_, v)
    + nu*inner(grad(u_), grad(v))
    + inner(dot(grad(u_), u_), v)
    - p_*div(v)
    - q*div(u_)
)*dx

# Jacobian
if args.nls == "picard":
    J = (
          idt*inner(u, v)
        + nu*inner(grad(u), grad(v))
        + inner(dot(grad(u), u_), v)
        - p*div(v)
        - q*div(u)
    )*dx
elif args.nls == "newton":
    J = derivative(F, w)
```

(continues on next page)

(continued from previous page)

```
# Add stabilization for AMG 00-block
if args.ls == "iterative":
    delta = StabilizationParameterSD(w.sub(0), nu)
    J_pc = J + delta*inner(dot(grad(u), u_), dot(grad(v), u_))*dx
elif args.ls == "direct":
    J_pc = None

# PCD operators
mu = idt*inner(u, v)*dx
mp = Constant(1.0/nu)*p*q*dx
kp = Constant(1.0/nu)*dot(grad(p), u_)*q*dx
ap = inner(grad(p), grad(q))*dx
if args.pcd_variant == "BRM2":
    n = FacetNormal(mesh)
    ds = Measure("ds", subdomain_data=boundary_markers)
    # TODO: What about the reaction term? Does it appear here?
    kp -= Constant(1.0/nu)*dot(u_, n)*p*q*ds(1)
    #kp -= Constant(1.0/nu)*dot(u_, n)*p*q*ds(0)  # TODO: Is this beneficial?

# Collect forms to define nonlinear problem
pcd_assembler = PCDAssembler(J, F, [bc0, bc1],
                             J_pc, ap=ap, kp=kp, mp=mp, mu=mu, bcs_pcd=bc_pcd)
assert pcd_assembler.get_pcd_form("gp").phantom # pressure grad obtained from J
problem = PCDNonlinearProblem(pcd_assembler)

# Set up linear solver (GMRES with right preconditioning using Schur fact)
linear_solver = PCDKrylovSolver(comm=mesh.mpi_comm())
linear_solver.parameters["relative_tolerance"] = 1e-6
#PETScOptions.set("ksp_monitor")
PETScOptions.set("ksp_gmres_restart", 150)

# Set up subsolvers
PETScOptions.set("fieldsplit_p_pc_python_type", "fenapack.PCDRPC_" + args.pcd_variant)
if args.ls == "iterative":
    PETScOptions.set("fieldsplit_u_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_u_ksp_max_it", 1)
    PETScOptions.set("fieldsplit_u_pc_type", "hypre")
    PETScOptions.set("fieldsplit_u_pc_hypre_type", "boomeramg")
    PETScOptions.set("fieldsplit_p_PCD_Rp_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_p_PCD_Rp_ksp_max_it", 1)
    PETScOptions.set("fieldsplit_p_PCD_Rp_pc_type", "hypre")
    PETScOptions.set("fieldsplit_p_PCD_Rp_pc_hypre_type", "boomeramg")
    PETScOptions.set("fieldsplit_p_PCD_Ap_ksp_type", "richardson")
    PETScOptions.set("fieldsplit_p_PCD_Ap_ksp_max_it", 2)
    PETScOptions.set("fieldsplit_p_PCD_Ap_pc_type", "hypre")
    PETScOptions.set("fieldsplit_p_PCD_Ap_pc_hypre_type", "boomeramg")
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_type", "chebyshev")
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_max_it", 5)
    PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_chebyshev_eigenvalues", "0.5, 2.0")
    #PETScOptions.set("fieldsplit_p_PCD_Mp_ksp_chebyshev_esteig", "1,0,0,1")  #␣
↪FIXME: What does it do?
    PETScOptions.set("fieldsplit_p_PCD_Mp_pc_type", "jacobi")
elif args.ls == "direct" and args.mumps_debug:
    # Debugging MUMPS
    PETScOptions.set("fieldsplit_u_mat_mumps_icntl_4", 2)
    PETScOptions.set("fieldsplit_p_PCD_Ap_mat_mumps_icntl_4", 2)
    PETScOptions.set("fieldsplit_p_PCD_Mp_mat_mumps_icntl_4", 2)
```

(continues on next page)

(continued from previous page)

```
# Apply options
linear_solver.set_from_options()

# Set up nonlinear solver
solver = PCDNewtonSolver(linear_solver)
solver.parameters["relative_tolerance"] = 1e-5

# Solve problem
t = 0.0
time_iters = 0
krylov_iters = 0
solution_time = 0.0
while t < args.t_end and not near(t, args.t_end, 0.1*args.dt):
    # Move to current time level
    t += args.dt
    time_iters += 1

    # Update boundary conditions
    inflow.t = t

    # Solve the nonlinear problem
    info("t = {:g}, step = {:g}, dt = {:g}".format(t, time_iters, args.dt))
    with Timer("Solve") as t_solve:
        newton_iters, converged = solver.solve(problem, w.vector())
    krylov_iters += solver.krylov_iterations()
    solution_time += t_solve.stop()

    # Update variables at previous time level
    w0.assign(w)

# Report timings
list_timings(TimingClear.clear, [TimingType.wall, TimingType.user])

# Get iteration counts
result = {
    "ndof": W.dim(), "time": solution_time, "steps": time_iters,
    "lin_its": krylov_iters, "lin_its_avg": float(krylov_iters)/time_iters}
tab = "{:^15} | {:^15} | {:^15} | {:^19} | {:^15}\n".format(
    "No. of DOF", "Steps", "Krylov its", "Krylov its (p.t.s.)", "Time (s)")
tab += "{ndof:>9}       | {steps:^15} | {lin_its:^15} | " \
       "{lin_its_avg:^19.1f} | {time:^15.2f}\n".format(**result)
print("\nSummary of iteration counts:")
print(tab)
#with open("table_pcdr_{}.txt".format(args.pcd_variant), "w") as f:
#    f.write(tab)

# Plot solution
u, p = w.split()
size = MPI.size(mesh.mpi_comm())
rank = MPI.rank(mesh.mpi_comm())
pyplot.figure()
pyplot.subplot(2, 1, 1)
plot(u, title="velocity")
pyplot.subplot(2, 1, 2)
plot(p, title="pressure")
pyplot.savefig("figure_v_p_size{}_rank{}.pdf".format(size, rank))
```

(continues on next page)

(continued from previous page)

```
pyplot.figure()
plot(p, title="pressure", mode="warp")
pyplot.savefig("figure_warp_size{}_rank{}.pdf".format(size, rank))
pyplot.show()
```

5.4 CircleCI configuration

This page describes how to setup CircleCI version 2.0 for a project built on top of DOLFIN. Tests are run using FEniCS docker images, in particular using an image with the development version of FEniCS.

For details, see also the FEniCS Docker reference manual.

Use CircleCI 2.0. It has native support for Docker images.:

```
version: 2
```

Specify build jobs:

```
jobs:
  build:
```

`image` specifies a Docker image with the development version of FEniCS. Inside of a container we will be using a user `fenics` which has an installation of FEniCS. But we will work in a different directory derived from the name of the project. The `environment` needs to be adjusted to allow importing C++ DOLFIN libraries and finding necessary files for just-in-time compilation. This is done because CircleCI bypasses environment setup specified in the FEniCS image.:

```
docker:
  - image: quay.io/fenicsproject/dev
    user: fenics
    environment:
      LD_LIBRARY_PATH: /home/fenics/local/lib
      CMAKE_PREFIX_PATH: /home/fenics/local
      MPLBACKEND: Agg
working_directory: /home/fenics/fenapack
```

First step is checking out the source code into the working directory:

```
steps:
  - checkout
```

Then print some diagnostic information to a build log:

```
- run:
    name: Environment and FEniCS version info
    command: |
      echo $USER $HOME $PWD $PATH $LD_LIBRARY_PATH $CMAKE_PREFIX_PATH
      python3 -c'import ffc; print(ffc.git_commit_hash(), ffc.ufc_signature())'
      python3 -c'import dolfin; print(dolfin.git_commit_hash())'
```

Install the project from the working directory.:

```
- run:
    name: Install FENaPack
```

(continues on next page)

(continued from previous page)

```
    command: |
      pip3 install -v --user .
```

Try to import the project. That involves some just-in-time compilation which we test and measure as a separate build step.:

```
- run:
    name: Import FENaPack first time (JIT)
    command: python3 -c"import fenapack"
```

Run the unit tests using the pytest framework. By `-svl` options we make a test output more verbose and using `--junitxml` we save a test result in machine-readable format. In a later step we tell to CircleCI where the result is. CircleCI is able to provide various information based on the results on its web UI.:

```
- run:
    name: Unit tests
    command: py.test-3 test/unit -svl --junitxml /tmp/circle/unit.xml
```

Now we run parallel unit tests. This would normally be done by just prefixing a `py.test` command by an `mpirun` command. Here we wrap it in the `bash` instance to figure out an MPI rank number using the `${OMPI_COMM_WORLD_RANK:-$PMI_RANK}` variable, which should work both with MPICH and OpenMPI, and use it to generate separate test result files.:

```
- run:
    name: Unit tests MPI
    command: >
      mpirun -n 3 bash -c '
      py.test-3 test/unit -svl
      --junitxml /tmp/circle/unit-mpi-${OMPI_COMM_WORLD_RANK:-$PMI_RANK}.xml
      '
```

Now we run the benchmarking suite and store generated PDF files for later use. Note that we tell to a shell (note that every build step is run in a separate shell) not to exit on first failure by `set +e`. Instead we only want to eventually fail only on the `py.test` command by returning its exit code by `exit $rc`.:

```
- run:
    name: Bench
    command: |
      set +e
      py.test-3 test/bench -svl --junitxml /tmp/circle/bench.xml
      rc=$?
      mv *.pdf /tmp/circle
      exit $rc
```

Now we run the regression tests implemented by a homebrew Python script `test.py`. We copy resulting figures to directory where CircleCI collects artifacts.:

```
      - run:
          name: Regression tests
          command: |
            set +e
            cd test/regression
            NP=3 python3 -u test.py
            rc=$?
            cd ../../demo; find -name "*.pdf" -exec cp --parents {} /tmp/circle \;
            exit $rc
```

(continues on next page)

(continued from previous page)

```
# Install defcon and run test demo
      - run:
          name: Run defcon demo
          command: |
            CC=mpicc HDF5_MPI=ON pip3 install --no-cache-dir --user --no-binary=h5py␣
↪h5py
            pip3 install --no-cache-dir --user git+https://bitbucket.org/pefarrell/
↪defcon
            python3 -c"import defcon"
            python3 demo/defcon/mesh/genmesh.py 120
            cd demo/defcon && mpirun -n 2 python3 navier-stokes.py
            mkdir -p /tmp/circle/defcon && cp bifurcation.pdf /tmp/circle/defcon
```

Finally we tell to CircleCI to store build artifacts and test results, which both can be accessed on CircleCI website.:

```
- store_artifacts:
    path: /tmp/circle
    destination: build

- store_test_results:
    path: /tmp/circle
```

Download the complete configuration file `circle.yml`.

5.5 Releasing

```
# Bump version numbers
git ls-files | xargs grep --color 2019\.1
edit ...
git commit -am"Bump version numbers to 2019.2.0"

# Wait for CircleCI to report green and fix problems

# Create tag and push
git tag -m"Release version 2019.2.0" 2019.2.0
git push 2019.2.0

# Package and push to PyPI
git clean -fdx
python3 setup.py sdist
python3 setup.py bdist_wheel --universal
pip3 install --upgrade --user twine
twine upload dist/*

# Bump version numbers
edit ...
git commit -am"Bump version numbers to 2019.3.0.dev0"
```

5.6 fenapack package

5.6.1 Submodules

5.6.2 fenapack.assembling module

This module provides wrappers for assembling systems of linear algebraic equations to be solved with the use of PCD preconditioning strategy. These wrappers naturally provide routines for assembling preconditioning operators themselves.

class fenapack.assembling.**PCDAssembler**(*a*, *L*, *bcs*, *a_pc=None*, *mp=None*, *mu=None*, *ap=None*, *fp=None*, *kp=None*, *gp=None*, *bcs_pcd=[]*)

Bases: object

Base class for creating linear problems to be solved by application of the PCD preconditioning strategy. Users are encouraged to use this class for interfacing with fenapack.field_split.PCDKrylovSolver. On request it assembles not only the individual PCD operators but also the system matrix and the right hand side vector defining the linear problem.

__init__(*a*, *L*, *bcs*, *a_pc=None*, *mp=None*, *mu=None*, *ap=None*, *fp=None*, *kp=None*, *gp=None*, *bcs_pcd=[]*)

Collect individual variational forms and boundary conditions defining a linear problem (system matrix + RHS vector) on the one side and preconditioning operators on the other side.

Arguments

a (dolfin.Form or ufl.Form) Bilinear form representing a system matrix.

L (dolfin.Form or ufl.Form) Linear form representing a right hand side vector.

bcs (list of dolfin.DirichletBC) Boundary conditions applied to a, L, and a_pc.

a_pc (dolfin.Form or ufl.Form) Bilinear form representing a matrix optionally passed to preconditioner instead of a. In case of PCD, stabilized 00-block can be passed to 00-KSP solver.

mp, mu, ap, fp, kp, gp (dolfin.Form or ufl.Form) Bilinear forms which (some of them) might be used by a particular PCD(R) preconditioner. Typically they represent "mass matrix" on pressure, "mass matrix" on velocity, minus Laplacian operator on pressure, pressure convection-diffusion operator, pressure convection operator and pressure gradient respectively.

bcs_pcd (list of dolfin.DirichletBC) Artificial boundary conditions used by PCD preconditioner.

All the arguments should be given on the common mixed function space.

All the forms are wrapped using *PCDForm* so that each of them can be endowed with additional set of properties.

By default, mp, mu, ap and gp are assumed to be constant if the preconditioner is used repeatedly in some outer iterative process (e.g Newton-Raphson method, time-stepping). As such, the corresponding operators are assembled only once. On the other hand, fp and kp are updated in every outer iteration.

Also note that gp is the only form that is by default in a *phantom mode*. It means that the corresponding operator (if needed) is not obtained by assembling the form, but it is extracted as the 01-block of the system matrix.

The default setting can be modified by accessing a *PCDForm* instance via *PCDAssembler.get_pcd_form()* and changing the properties directly.

__weakref__
list of weak references to the object (if defined)

get_dolfin_form(*key*)
Return form as `dolfin.Form` or `ufl.Form`.

get_pcd_form(*key*)
Return form wrapped in *PCDForm*.

gp(*Bt*)
Assemble discrete pressure gradient. It is crucial to respect any constraints placed on the velocity test space by Dirichlet boundary conditions.

pc_matrix(*P*)
Assemble preconditioning matrix `P` whose relevant blocks can be passed to actual parts of the `KSP` solver.

rhs_vector(*b*, *x=None*)
Assemble right hand side vector `b`.

The version with `x` is suitable for use inside a (quasi)-Newton solver.

system_matrix(*A*)
Assemble system matrix `A`.

class fenapack.assembling.**PCDForm**(*form*, *const=False*, *phantom=False*)
Bases: `object`

Wrapper for PCD operators represented by `dolfin.Form` or `ufl.Form`. This class allows to record specific properties of the form that can be utilized later while setting up the preconditioner.

For example, we can specify which matrices remain constant during the outer iterative algorithm (e.g. Newton-Raphson method, time-stepping) and which matrices need to be updated in every outer iteration.

__init__(*form*, *const=False*, *phantom=False*)
The class is initialized by a single form with default properties.

Arguments

form (`dolfin.Form` or `ufl.Form`) A form to be wrapped.

const (*bool*) Whether the form remains constant in outer iterations.

phantom (*bool*) If *True*, then the corresponding operator will be obtained not by assembling the form, but in some different way. (For example, pressure gradient may be extracted directly from the system matrix.)

__weakref__
list of weak references to the object (if defined)

5.6.3 fenapack.field_split module

This module provides subclasses of DOLFIN and petsc4py Krylov solvers implementing PCD fieldsplit preconditioned GMRES

5.6.4 fenapack.field_split_backend module

Tools for extraction and management of fieldsplit submatrices, subvectors, subbcs, subksps intended to be hidden from user interface

class fenapack.field_split_backend.**PCDInterface**(*pcd_assembler*, *A*, *is_u*, *is_p*, *deep_submats=False*)

Bases: object

Wrapper of PCDAssembler for interfacing with PCD PC fieldsplit implementation. Convection fieldsplit submatrices are extracted as shallow or deep submatrices according to deep_submats parameter.

__init__(*pcd_assembler*, *A*, *is_u*, *is_p*, *deep_submats=False*)
Create PCDInterface instance given PCDAssembler instance, system matrix and velocity and pressure index sets

__weakref__
list of weak references to the object (if defined)

apply_bcs(*vec*, *bcs_getter*, *iset*)
Transform dolfin bcs obtained using bcs_getter function into fieldsplit subBCs and apply them to fieldsplit vector. SubBCs are cached.

apply_pcd_bcs(*vec*)
Apply bcs to intermediate pressure vector of PCD pc

get_work_dolfin_mat(*key*, *comm*, *can_be_destroyed=None*, *can_be_shared=None*)
Get working DOLFIN matrix by key. can_be_destroyed=True tells that it is probably favourable to not store the matrix unless it is shared as it will not be used ever again, None means that it can be destroyed but it is not probably favourable and False forbids the destruction. can_be_shared tells if a work matrix can be the same with work matrices for other keys.

get_work_mats(*M*, *num*)
Return num of work mats initially created from matrix B.

get_work_vecs_from_square_mat(*M*, *num*)
Return num of work vecs initially created from a square matrix M.

setup_ksp(*ksp*, *assemble_func*, *iset*, *spd=False*, *const=False*)
Assemble into operator of given ksp if not yet assembled

setup_ksp_Ap(*ksp*)
Setup pressure Laplacian ksp and assemble matrix

setup_ksp_Mp(*ksp*)
Setup pressure mass matrix ksp and assemble matrix

setup_ksp_Rp(*ksp*, *Mu*, *Bt*)
Setup pressure Laplacian ksp based on velocity mass matrix Mu and discrete gradient Bt and assemble matrix

setup_mat_Bt(*mat=None*)
Setup and assemble discrete pressure gradient and return it

setup_mat_Fp(*mat=None*)
Setup and assemble pressure convection-diffusion matrix and return it

setup_mat_Kp(*mat=None*)
Setup and assemble pressure convection matrix and return it

setup_mat_Mu(*mat=None*)
Setup and assemble velocity mass matrix and return it

5.6.5 fenapack.nonlinear_solvers module

This module provides subclasses of DOLFIN interface for solving non-linear problems suitable for use with fieldsplit preconditioned Krylov methods

5.6.6 fenapack.preconditioners module

class fenapack.preconditioners.**BasePCDPC**
Bases: object

Base python context for pressure convection diffusion (PCD) preconditioners.

__weakref__
list of weak references to the object (if defined)

static create_default_ksp(*comm*)
Return Cholesky factorization KSP

get_work_vecs(*v*, *num*)
Return num of work vecs initially duplicated from v

init_pcd(*pcd_interface*)
Initialize by PCDInterface instance

class fenapack.preconditioners.**BasePCDRPC**
Bases: *fenapack.preconditioners.BasePCDPC*

Base python context for pressure convection diffusion reaction (PCDR) preconditioners.

class fenapack.preconditioners.**PCDPC_BRM1**
Bases: *fenapack.preconditioners.BasePCDPC*

This class implements a modification of PCD variant similar to one by[1].

class fenapack.preconditioners.**PCDPC_BRM2**
Bases: *fenapack.preconditioners.BasePCDPC*

This class implements a modification of steady variant of PCD described in[2].

class fenapack.preconditioners.**PCDRPC_BRM1**
Bases: *fenapack.preconditioners.BasePCDRPC*

This class implements an extension of *PCDPC_BRM1*. Here we add a reaction term into the preconditioner, so that is becomes PCDR (pressure-convection-diffusion-reaction) preconditioner. This particular variant is suitable for time-dependent problems, where the reaction term arises from the time derivative in the balance of momentum.

class fenapack.preconditioners.**PCDRPC_BRM2**
Bases: *fenapack.preconditioners.BasePCDRPC*

This class implements an extension of *PCDPC_BRM2*. Here we add a reaction term into the preconditioner, so that is becomes PCDR (pressure-convection-diffusion-reaction) preconditioner. This particular variant is suitable for time-dependent problems, where the reaction term arises from the time derivative in the balance of momentum.

[1] Olshanskii M. A., Vassilevski Y. V., *Pressure Schur complement preconditioners for the discrete Oseen problem*. SIAM J. Sci. Comput., 29(6), 2686-2704. 2007.

[2] Elman H. C., Silvester D. J., Wathen A. J., *Finite Elements and Fast Iterative Solvers: With Application in Incompressible Fluid Dynamics*. Oxford University Press 2005. 2nd edition 2014.

5.6.7 fenapack.stabilization module

fenapack.stabilization.**StabilizationParameterSD**(*wind*, *viscosity*, *density=None*)

Returns a subclass of dolfin.Expression representing streamline diffusion stabilization parameter.

This kind of stabilization is convenient when a multigrid method is used for the convection term in the Navier-Stokes equation. The idea of the stabilization involves adding an additional term of the form:

```
delta_sd*inner(dot(grad(u), w), dot(grad(v), w))*dx
```

into the Navier-Stokes equation. Here u is a trial function, v is a test function and w defines so-called "wind" which is a known vector function. Regularization parameter delta_sd is determined by the local mesh Peclet number (PE), see the implementation below.

Arguments

wind (dolfin.GenericFunction) A vector field determining convective velocity.

viscosity (dolfin.GenericFunction) A scalar field determining dynamic viscosity.

density (dolfin.GenericFunction) A scalar field determining density (optional).

5.6.8 fenapack.utils module

fenapack.utils.**allow_only_one_call**(*func*)

Decorator allowing provided instancemethod to be called only once. Additional calls raise error.

fenapack.utils.**get_default_factor_solver_type**(*comm*)

Return first available factor solver type name. This is implemened using DOLFIN now.

5.6.9 Module contents

This is FENaPack, FEniCS Navier-Stokes preconditioning package.

5.7 Index

Python Module Index

f

Index

Symbols

A

B

C

F

G

I

P

R

S