```
                ierr = MatColoringSetType(matcoloring, MATCOLORINGSL);
CHKERRV(ierr);
                ierr = MatColoringSetFromOptions(matcoloring);
CHKERRV(ierr);
                ierr = MatColoringApply(matcoloring, &iscoloring);
CHKERRV(ierr);
                ierr = MatColoringDestroy(&matcoloring); CHKERRV(ierr);
            }
            ierr = MatFDColoringCreate(J, iscoloring, &matfdcoloring);
CHKERRV(ierr);
            ierr = MatFDColoringSetFunction(matfdcoloring,(PetscErrorCode
(*)(void))SNESTSFormFunction,ts); CHKERRV(ierr);
            ierr = MatFDColoringSetFromOptions(matfdcoloring);
CHKERRV(ierr);
            ierr = MatFDColoringSetUp(J,iscoloring,matfdcoloring);
CHKERRV(ierr);
            SNES snes;
            ierr = TSGetSNES(ts, &snes); CHKERRV(ierr);
            ierr = SNESSetFromOptions(snes); CHKERRV(ierr);
            ierr =
SNESSetJacobian(snes,J,J,SNESComputeJacobianDefaultColor,matfdcoloring);
CHKERRV(ierr);
            ierr = ISColoringDestroy(&iscoloring); CHKERRV(ierr);
            /* ----- Set the preconditioner */
            KSP ksp;
            ierr = SNESGetKSP(snes,&ksp); CHKERRV(ierr);
            ierr = KSPSetFromOptions(ksp); CHKERRV(ierr);
            PC pc;
            ierr = KSPGetPC(ksp,&pc); CHKERRV(ierr);
            ierr = PCSetFromOptions(pc); CHKERRV(ierr);
        }
        /* ---- If the user does not want to use coloring */
        else {
            /* ----- Matrix-free representation of the jacobian for LHS
now */
            SNES           snes;
            DM             sdm;
            KSP            ksp;
            PC             pc;
            TSProblemType ptype;
            SNESType       snestype;
            Mat            A, B;
            Vec            tmp;
            PetscInt       local_size, global_size;
            /**/
            ierr = TSGetSNES(ts, &snes); CHKERRV(ierr);
            ierr = SNESGetType(snes, &snestype); CHKERRV(ierr);
            ierr = TSGetProblemType(ts, &ptype); CHKERRV(ierr);

            ierr = SNESGetDM(snes, &sdm); CHKERRV(ierr);
            ierr = DMGetGlobalVector(sdm, &tmp); CHKERRV(ierr);
            ierr = VecGetLocalSize(tmp, &local_size); CHKERRV(ierr);
            ierr = VecGetSize(tmp, &global_size); CHKERRV(ierr);
            ierr = DMRestoreGlobalVector(sdm, &tmp); CHKERRV(ierr);
            ierr = MatCreateShell(user->model->comm, local_size,
 local_size, global_size, global_size, user, &A); CHKERRV(ierr);
            if ((!strcmp(snestype, SNESKSPONLY)) || (ptype == TS_LINEAR))
 {
```

```
                // linear problem
                user->flag_is_linear = PETSC_TRUE;
                ierr = MatShellSetOperation(A, MATOP_MULT, (void
(*)(void))PetscJacobianFunction_Linear); CHKERRV(ierr);
                ierr = SNESSetType(snes, SNESKSPONLY); CHKERRV(ierr);
            } else {
                // nonlinear problem
                user->flag_is_linear = PETSC_FALSE;
                user->jfnk_eps = 1e-7;
                ierr = PetscOptionsGetReal(PETSC_NULL, PETSC_NULL, "-
jfnk_epsilon", &user->jfnk_eps, PETSC_NULL); CHKERRV(ierr);
                ierr = MatShellSetOperation(A, MATOP_MULT, (void
(*)(void))PetscJacobianFunction_JFNK); CHKERRV(ierr);
            }
            ierr = MatSetUp(A); CHKERRV(ierr);
            /* ------ Preconditioner */
            user->flag_use_precon = PETSC_FALSE;
            ierr = PetscOptionsGetBool(PETSC_NULL, PETSC_NULL, "-
with_pc", (PetscBool*)&user->flag_use_precon, PETSC_NULL); CHKERRV(ierr);
            if (user->flag_use_precon) {
                /* ------- Set up preconditioner matrix */
                PetscInt dim;
                ierr = DMGetDimension(dm, &dim); CHKERRV(ierr);
                ierr = DMSetMatType(dm, MATAIJ); CHKERRV(ierr);
                ierr = DMCreateMatrix(dm, &B); CHKERRV(ierr);
                /* Set the RHSJacobian function for TS */
                ierr = TSSetIJacobian(ts, A, B, PetscIJacobian, user);
CHKERRV(ierr);
            } else {
                /* ------- Just roll without any preconditioner */
                /* -------- Set the RHSJacobian function for TS */
                ierr = TSSetIJacobian(ts, A, A, PetscIJacobian, user);
CHKERRV(ierr);
                /* -------- Set PC (preconditioner) to none */
                ierr = SNESGetKSP(snes, &ksp); CHKERRV(ierr);
                ierr = KSPSetFromOptions(ksp); CHKERRV(ierr);
                ierr = KSPGetPC(ksp, &pc);     CHKERRV(ierr);
                ierr = PCSetFromOptions(pc);   CHKERRV(ierr);
                ierr = PCSetType(pc, PCNONE);  CHKERRV(ierr);
            }
        }
    }

    /* -- Add the CFL condition on the time step */
    TSAdapt adapt;
    ierr = TSAdaptRegister("my-cfl-adapt", TSAdaptCreate_MyCFLAdapt);
CHKERRV(ierr);
    ierr = TSGetAdapt(ts, &adapt);
CHKERRV(ierr);
    ierr = TSAdaptSetType(adapt, "my-cfl-adapt");
CHKERRV(ierr);

    /* -- Finalize the time stepper set-up */
    ierr = TSSetPostStep(ts, PetscResidualEvaluation); CHKERRV(ierr);
    ierr = TSSetUp(ts);                                CHKERRV(ierr);
    ierr = TSSetApplicationContext(ts, user);          CHKERRV(ierr);

    /* Handle the parameters for a stationary computation */
```

```
                loc_residual[idof] += cg->volume * pointrhs[idof] *
pointrhs[idof];
            }
            loc_totalvolume += cg->volume;
        }
        /* - Gather everything to get the total values */
        PetscScalar residual[user->model->physics->dof]; for (idof = 0;
idof < user->model->physics->dof; ++idof) residual[idof] = 0.0;
        PetscScalar totalvolume = 0.0;
        ierr = MPI_Allreduce(loc_residual,     residual,     user->model-
>physics->dof, MPI_DOUBLE, MPI_SUM, user->model->comm); CHKERRQ(ierr);
        ierr = MPI_Allreduce(&loc_totalvolume, &totalvolume, 1,
MPI_DOUBLE, MPI_SUM, user->model->comm); CHKERRQ(ierr);
        for (idof = 0; idof < user->model->physics->dof; ++idof)
residual[idof] = sqrt(residual[idof] / totalvolume);

        /* Restore accesses */
        ierr = VecRestoreArrayRead(user->RHS_ref, &rhs); CHKERRQ(ierr);
        ierr = DMRestoreLocalVector(plex, &loc_RHS); CHKERRQ(ierr);
        ierr = VecRestoreArrayRead(cellGeometryFVM, &cellGeom);
CHKERRQ(ierr);
        ierr = ISRestorePointRange(cellIS, &cStart, &cEnd, &cells);
CHKERRQ(ierr);
        ierr = DMDestroy(&plex); CHKERRQ(ierr);

        /* Handle the stopping criterion and display */
        ierr = TSGetStepNumber(ts, &stepnum); CHKERRQ(ierr);
        if ((stepnum >= 0) && (stepnum % user->screenInterval == 0)) {
            ierr = PetscPrintf(user->model->comm, "| Iter %06d residual
[%.6e, %.6e, %.6e, %.6e] \n",
                               stepnum, residual[0], residual[1],
residual[2], residual[3]); CHKERRQ(ierr);
        }
        if (fabs(residual[abortVariable]) <= abortResidual) {
            ierr = TSSetConvergedReason(ts, TS_CONVERGED_USER);
CHKERRQ(ierr);
        }

    }

    PetscFunctionReturn(0);
}

/**
 * Below all the methods for implicit time stepping
 */
PetscErrorCode PetscIJacobian(
                               TS ts,         /*!< Time stepping object
(see PETSc TS)*/
                               PetscReal t,   /*!< Current time */
                               Vec Y,         /*!< Solution vector */
                               Vec Ydot,      /*!< Time-derivative of
solution vector */
                               PetscReal a,   /*!< Shift */
                               Mat A,         /*!< Jacobian matrix */
                               Mat B,         /*!< Preconditioning matrix
*/
                               void *ctxt     /*!< Application context */
```

```
                                )
{
    PetscErrorCode ierr;
    User           user = (User) ctxt;

    PetscFunctionBegin;

    user->isImplicit = isImplicit;
    user->shift = a;
    user->waqt = t;
    user->ts   = ts;
    /* Construct preconditioning matrix */
    if (user->flag_use_precon) { ierr = PetscComputePreconMatImpl(B, Y,
user); CHKERRQ(ierr); }

    PetscFunctionReturn(0);
}

static void NavierStokes2DJFunction(
                                const int dof,    /*!< Number of
degrees of freedom at a grid point (dof = 4) */
                                double    *J,     /*!< Jacobian
matrix: 1D array of size dof^2 = 16 */
                                double    *u,     /*!< solution at a
grid point (array of size dof = 4) */
                                double    gamma, /*!< ratio of
specific heats */
                                int       dir,    /*!< dimension (0 ->
x, 1 -> y) */
                                int       upw     /*!< 0 -> send back
complete Jacobian,
                                                      1 -> send back
Jacobian of right(+)-moving flux,
                                                     -1 -> send back
Jacobian of left(-)-moving flux */
                                )
{
    double R[dof*dof], D[dof*dof], L[dof*dof], DL[dof*dof];

    /* get the eigenvalues and left,right eigenvectors */
    _NavierStokes2DEigenvalues_       (u,D,gamma,dir,dof);
    _NavierStokes2DLeftEigenvectors_  (u,L,gamma,dir,dof);
    _NavierStokes2DRightEigenvectors_(u,R,gamma,dir,dof);

    int aupw = absolute(upw), k;
    k = 0;  D[k] = absolute( (1-aupw)*D[k] + 0.5*aupw*(1+upw)*max(0,D[k])
+ 0.5*aupw*(1-upw)*min(0,D[k]) );
    k = 5;  D[k] = absolute( (1-aupw)*D[k] + 0.5*aupw*(1+upw)*max(0,D[k])
+ 0.5*aupw*(1-upw)*min(0,D[k]) );
    k = 10; D[k] = absolute( (1-aupw)*D[k] + 0.5*aupw*(1+upw)*max(0,D[k])
+ 0.5*aupw*(1-upw)*min(0,D[k]) );
    k = 15; D[k] = absolute( (1-aupw)*D[k] + 0.5*aupw*(1+upw)*max(0,D[k])
+ 0.5*aupw*(1-upw)*min(0,D[k]) );

    MatMult4(DL,D,L);
    MatMult4(J,R,DL);
}
```

```
PetscErrorCode PetscComputePreconMatImpl(
                                          Mat Precon, /*!< Preconditioning
matrix to construct [global] */
                                          Vec Y,      /*!< Solution vector
[global] */
                                          void *ctxt  /*!< Application
context */
                                         )
{
    PetscErrorCode ierr;
    User           user  = (User) ctxt;
    DM             dmCell = NULL, plexCell = NULL;
    PetscInt       dim, dof;
    Vec            locY;
    const PetscScalar *arrY;
    DM             dmFace = NULL;
    DMLabel        ghostLabel;
    Vec            faceGeometryFVM, cellGeometryFVM;
    PetscInt       fStart, fEnd, face;
    const PetscScalar *facegeom;

    PetscFunctionBegin;

    /* Get the PETSc objects */
    /* - The DM associated with the timestepper */
    ierr = TSGetDM(user->ts, &dmCell); CHKERRQ(ierr);
    ierr = DMConvert(dmCell, DMPLEX, &plexCell); CHKERRQ(ierr);
    ierr = DMGetDimension(dmCell, &dim); CHKERRQ(ierr);
    dof  = user->model->physics->dof;
    /* -- Access the ghost layer to have the faces and cells labeled as
"ghost" */
    ierr = DMGetLabel(plexCell, "ghost", &ghostLabel); CHKERRQ(ierr);

    /* Get the geometry informations */
    /* - All cell and face geometry */
    ierr = DMPlexGetGeometryFVM(plexCell, &faceGeometryFVM,
&cellGeometryFVM, NULL); CHKERRQ(ierr);
    /* - Get the "face" layer and underlying geometrical organization
specifically */
    ierr = DMPlexGetHeightStratum(plexCell, 1, &fStart, &fEnd);
CHKERRQ(ierr);
    ierr = VecGetDM(faceGeometryFVM, &dmFace); CHKERRQ(ierr);
    ierr = VecGetArrayRead(faceGeometryFVM, &facegeom); CHKERRQ(ierr);

    /* - Get the local version of the solution vector */
    ierr = DMGetLocalVector(plexCell, &locY); CHKERRQ(ierr);
    ierr = VecZeroEntries(locY); CHKERRQ(ierr);
    ierr = DMGlobalToLocalBegin(plexCell, Y, INSERT_VALUES, locY);
CHKERRQ(ierr);
    ierr = DMGlobalToLocalEnd  (plexCell, Y, INSERT_VALUES, locY);
CHKERRQ(ierr);
    ierr = DMPlexInsertBoundaryValues(plexCell, PETSC_TRUE, locY, 0.0,
faceGeometryFVM, cellGeometryFVM, NULL); CHKERRQ(ierr);
    ierr = VecGetArrayRead(locY, &arrY); CHKERRQ(ierr);

    /* Loop on the faces */
    for (face = fStart; face < fEnd; ++face) {
        /* - Some inner loop declarations */
```

```
        PetscBool         boundary;
        PetscInt          ghost, numChildren;
        PetscInt          numCells;
        const PetscInt   *cells;
        PetscInt          c, pd, d;
        PetscFVFaceGeom  *fg;
        PetscScalar      *cy[2];
        PetscReal         J[2][dof*dof], deltaJ[dof*dof];
        PetscInt         irow[dof], icol[dof];
        /* - Loop immediately if it is a ghost face, on a boundary or a
face that has no children */
        ierr = DMLabelGetValue(ghostLabel, face, &ghost); CHKERRQ(ierr);
        ierr = DMIsBoundaryPoint(plexCell, face, &boundary);
CHKERRQ(ierr);
        ierr = DMPlexGetTreeChildren(plexCell, face, &numChildren, NULL);
CHKERRQ(ierr);
        // if (ghost >= 0 || boundary || numChildren) continue;
        // if (ghost >= 0) continue;
        if (boundary) continue;
        // if (numChildren) continue;
        /* - Get the support of the face, i.e. its two neighbouring cells
*/
        ierr = DMPlexGetSupportSize(plexCell, face, &numCells);
CHKERRQ(ierr);
        if (numCells != 2) SETERRQ2(PETSC_COMM_SELF, PETSC_ERR_PLIB,
"facet %d has %d support points: expected 2",face,numCells);
        ierr = DMPlexGetSupport(plexCell, face, &cells); CHKERRQ(ierr);
        /* - Loop on the support of the current face to access the
conservative variables */
        ierr = DMPlexPointLocalRead(dmFace, face, facegeom, &fg);
CHKERRQ(ierr);
        for (c = 0; c < numCells; ++c) {
            ierr = DMPlexPointLocalRead(plexCell, cells[c], arrY,
&cy[c]); CHKERRQ(ierr);
        }
        /* - Reconstruct the gradient of the jacobian */
        for (d = 0; d < dim; ++d) {
            /* -- For each cell of the support, get the JFunction (exact
jacobian) */
            for (c = 0; c < numCells; ++c) {
                NavierStokes2DJFunction((const int)dof, (double*)J[c],
(double*)cy[c], gam, d, 0);
            }
            /* -- For cell 0 (left cell), add the difference weighted
with the positive quadrature weights */
            for (pd = 0; pd < dof*dof; ++pd) {
                deltaJ[pd] = fg->grad[0][d] * (J[1][pd] - J[0][pd]);
            }
            for (pd = 0; pd < dof; ++pd) {irow[pd] = dof*cells[0] + pd;
icol[pd] = dof*cells[0] + pd;}
            ierr = MatSetValuesLocal(Precon, dof, irow, dof, icol,
deltaJ, ADD_VALUES); CHKERRQ(ierr);
            /* -- For cell 1 (right cell), add the difference weighted
with the negative quadrature weights */
            for (pd = 0; pd < dof*dof; ++pd) {
                deltaJ[pd] = -fg->grad[1][d] * (J[1][pd] - J[0][pd]);
            }
```

```
            for (pd = 0; pd < dof; ++pd) {irow[pd] = dof*cells[0] + pd;
icol[pd] = dof*cells[1] + pd;}
            ierr = MatSetValuesLocal(Precon, dof, irow, dof, icol,
deltaJ, ADD_VALUES); CHKERRQ(ierr);
        }
    }

    ierr = MatAssemblyBegin(Precon, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = MatAssemblyEnd  (Precon, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = MatShift        (Precon, user->shift);        CHKERRQ(ierr);

    MatView(Precon, NULL);

    /* Free local memory */
    ierr = VecRestoreArrayRead(faceGeometryFVM, &facegeom);
CHKERRQ(ierr);
    ierr = VecRestoreArrayRead(locY, &arrY); CHKERRQ(ierr);
    ierr = DMRestoreLocalVector(plexCell, &locY); CHKERRQ(ierr);
    ierr = DMDestroy(&plexCell); CHKERRQ(ierr);

    PetscFunctionReturn(0);
}

PetscErrorCode PetscJacobianFunction_JFNK(
                                          Mat Jacobian, /*!< Jacobian
matrix */
                                          Vec Y,      /*!< Input vector
[global] */
                                          Vec F       /*!< Output vector
(Jacobian times input vector) [global] */
                                         )
{
    PetscErrorCode ierr;
    User           user;
    PetscReal      normY, epsilon;
    DM             locdm;
    Vec            RHS_hyp, locY;

    PetscFunctionBegin;

    /* Get back the context to access everything */
    ierr = MatShellGetContext(Jacobian, &user); CHKERRQ(ierr);

    /* Get the DM from the matrix */
    ierr = TSGetDM(user->ts, &locdm); CHKERRQ(ierr);

    /* Get the norm of the increment vector Y */
    ierr = VecNorm(Y, NORM_2, &normY); CHKERRQ(ierr);

    /* - If the increment is infinitesimal */
    if (normY < 1e-16) {
        ierr = VecZeroEntries(F); CHKERRQ(ierr);
        ierr = VecAXPBY(F, user->shift, 0.0, Y); CHKERRQ(ierr);
    }
    /* - If the increment is non negligible */
    else {
        /* -- U0 + eps * Y */
        epsilon = user->jfnk_eps / normY;
```

```
        ierr = VecAYPX(Y, epsilon, user->U0_ref); CHKERRQ(ierr);
        /* -- F(U0 + eps * Y) */
        ierr = DMGetGlobalVector(locdm, &RHS_hyp); CHKERRQ(ierr);
        ierr = VecZeroEntries(RHS_hyp); CHKERRQ(ierr);
        ierr = DMGetLocalVector(locdm, &locY); CHKERRQ(ierr);
        ierr = DMGlobalToLocalBegin(locdm, Y, INSERT_VALUES, locY);
CHKERRQ(ierr);
        ierr = DMGlobalToLocalEnd(locdm, Y, INSERT_VALUES, locY);
CHKERRQ(ierr);
        ierr = DMPlexTSComputeRHSFunctionFVM(locdm, user->waqt, locY,
RHS_hyp, (void*) user); CHKERRQ(ierr);
        ierr = DMRestoreLocalVector(locdm, &locY); CHKERRQ(ierr);
        /* -- J = F(U0 + eps * Y) - F(U0) */
        ierr = VecZeroEntries(F); CHKERRQ(ierr);
        ierr = VecWAXPY(F, -1.0, user->RHS_ref, RHS_hyp); CHKERRQ(ierr);
        /* -- J = shift * Y - 1/eps * (F(U0+eps*Y)-F(U0)) */
        ierr = VecAXPBY(F, user->shift, (-1.0/epsilon), Y);
CHKERRQ(ierr);
        /* -- Restore accesses */
        ierr = DMRestoreGlobalVector(locdm, &RHS_hyp); CHKERRQ(ierr);
    }

    PetscFunctionReturn(0);
}

PetscErrorCode PetscJacobianFunction_Linear(Mat Jacobian, Vec Y, Vec F)
{
    PetscErrorCode ierr;

    PetscFunctionBegin;

    PetscFunctionReturn(0);
}


#endif
```