```
                                RKweight[0] = 1.0/6.0; RKweight[1] = 1.0/6.0;
RKweight[2] = 2.0/3.0;
                                break;
                        }
                        break;
                    }
                    case 4: {
                        switch (timeOrder) {
                            case 4:
                                if (!rank) printf("| RK44 Activated\n");
                                RKmatrix[4] = 0.5; RKmatrix[9] = 0.5;
RKmatrix[14] = 1.0;
                                RKnode[1] = 0.5; RKnode[2] = 0.5; RKnode[3] =
1.0;
                                RKweight[0] = 1.0/6.0; RKweight[1] = 1.0/3.0;
RKweight[2] = 1.0/3.0; RKweight[3] = 1.0/6.0;
                                break;
                        }
                        break;
                    }
                    default:
                        if (!rank) printf("| ERROR: RK %d stages, order %d is
not implemented.", nRKstages, timeOrder);
                        exit(1);
                }
            }
        }
        /* - For PETSc usage */
#else
        /* -- Create the time stepping entity */
        PetscErrorCode ierr;
        ierr = TSCreate(PetscObjectComm((PetscObject)dm), &ts);
CHKERRV(ierr);
        /* -- Assign to it the right integrator based on order and number of
stages */
        if (isRKLowStorage)
            PetscPrintf(user->model->comm, "| WARNING: Parameter
isRKLowStorage ignored with PETSc.\n");
        if (timeOrder == 1) {
            if (isImplicit) { ierr = TSSetType(ts, TSBEULER); CHKERRV(ierr);
}
            else            { ierr = TSSetType(ts, TSEULER);  CHKERRV(ierr);
}
            PetscPrintf(user->model->comm, "| Time stepping scheme: Euler
%s\n", (isImplicit)?"implicit":"explicit");
        } else {
            if (isImplicit) {
                PetscPrintf(user->model->comm, "| ERROR: Implicit temporal
discretization of high order not available with PETSc yet.\n");
                PETSCABORT(user->model->comm, 7);
            } else {
                if (timeOrder <= 3) {
                    ierr = TSSetType(ts, TSSSP); CHKERRV(ierr);
                    switch (timeOrder) {
                        case 2:
                            ierr = TSSSPSetType(ts, TSSSPRKS2);
CHKERRV(ierr);
```

```
                        ierr = TSSSPSetNumStages(ts, nRKstages);
CHKERRV(ierr);
                        PetscPrintf(user->model->comm, "| Time stepping
scheme: Runge-Kutta SSP 2 with %d stages.\n", nRKstages);
                        break;
                    case 3:
                        ierr = TSSSPSetType(ts, TSSSPRKS3);
CHKERRV(ierr);
                        ierr = TSSSPSetNumStages(ts, nRKstages *
nRKstages); CHKERRV(ierr);
                        PetscPrintf(user->model->comm, "| Time stepping
scheme: Runge-Kutta SSP 3 with %d stages.\n", nRKstages*nRKstages);
                        break;
                }
            } else if (timeOrder == 4) {
                ierr = TSSetType(ts, TSSSP); CHKERRV(ierr);
                ierr = TSSSPSetType(ts, TSSSPRK104); CHKERRV(ierr);
                PetscPrintf(user->model->comm, "| Time stepping scheme:
Runge-Kutta SSP 4 with 10 stages.\n");
            } else {
                PetscPrintf(user->model->comm, "| ERROR: RK order %d is
not implemented with PETSc.", timeOrder);
                PETSCABORT(user->model->comm, 8);
            }
        }
    }
    /* -- Get infos from options */
    ierr = TSSetFromOptions(ts); CHKERRV(ierr);
    /* -- Set the exit criteria (max time, and max # of iterations) */
    ierr = TSSetMaxTime(ts, stopTime);
CHKERRV(ierr);
    ierr = TSSetMaxSteps(ts, maxIter);
CHKERRV(ierr);
    ierr = TSSetExactFinalTime(ts, TS_EXACTFINALTIME_MATCHSTEP);
CHKERRV(ierr);
    /* -- Associate the temporal integrator to the spatial discretization
*/
    ierr = TSSetDM(ts, dm); CHKERRV(ierr);
    if (user->vtkmon) {
        ierr = TSMonitorSet(ts, vtkMonitor, user, NULL); CHKERRV(ierr);
    }
    ierr = DMTSSetBoundaryLocal(dm, DMPlexTSComputeBoundary, user);
CHKERRV(ierr);
#ifndef navierstokes
    if (!isImplicit) {
        /* --- For explicit TS, only RHS method needs to be given */
        ierr = DMTSSetRHSFunctionLocal(dm, DMPlexTSComputeRHSFunctionFVM,
user); CHKERRV(ierr);
    } else {
        /* --- For implicit TS, both RHS and LHS methods need to be given
*/
        /* ---- Prepare the vector that receives the 'reference' RHS,
i.e. F(U0) */
        ierr = DMCreateGlobalVector(dm, &user->RHS_ref); CHKERRV(ierr);
        ierr = VecZeroEntries(user->RHS_ref); CHKERRV(ierr);
        /* ---- RHS here */
        ierr = DMTSSetRHSFunctionLocal(dm, PetscRHSFunctionExpl, user);
CHKERRV(ierr);
```

```
        /* ---- Matrix-free representation of the jacobian for LHS now */
        SNES           snes;
        KSP            ksp;
        PC             pc;
        TSProblemType  ptype;
        SNESType       snestype;
        Mat            A, B;
        Vec            locX;
        PetscInt       total_size;
        /**/
        ierr = TSGetSNES(ts, &snes); CHKERRV(ierr);
        ierr = SNESGetType(snes, &snestype); CHKERRV(ierr);
        ierr = TSGetProblemType(ts, &ptype); CHKERRV(ierr);
        ierr = DMGetLocalVector(dm, &locX); CHKERRV(ierr);
        ierr = VecGetSize(locX, &total_size); CHKERRV(ierr);
        ierr = DMRestoreLocalVector(dm, &locX); CHKERRV(ierr);
        total_size = total_size * user->model->physics->dof;
        ierr = MatCreateShell(user->model->comm, total_size, total_size,
PETSC_DETERMINE, PETSC_DETERMINE, user, &A); CHKERRV(ierr);
        if ((!strcmp(snestype, SNESKSPONLY)) || (ptype == TS_LINEAR)) {
            /* linear problem */
            user->flag_is_linear = 1;
            ierr = MatShellSetOperation(A, MATOP_MULT, (void
(*)(void))PetscJacobianFunction_Linear); CHKERRV(ierr);
            ierr = SNESSetType(snes, SNESKSPONLY); CHKERRV(ierr);
        } else {
            /* nonlinear problem */
            user->flag_is_linear = 0;
            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 = 0;
        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 = MatCreateAIJ(user->model->comm, total_size,
total_size, PETSC_DETERMINE, PETSC_DETERMINE,
                                (dim*2+1)*user->model->physics->dof,
NULL,
                                2*dim*user->model->physics->dof, NULL,
&B); CHKERRV(ierr);
            ierr = MatSetBlockSize(B, user->model->physics->dof);
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 = KSPGetPC(ksp, &pc);     CHKERRV(ierr);
            ierr = PCSetType(pc, PCNONE);  CHKERRV(ierr);
        }
    }
#else
    ierr = DMTSSetRHSFunctionLocal(dm, PetscRHSFunctionExpl, user);
CHKERRV(ierr);
#endif
    /* -- 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 = TSSetUp(ts);                          CHKERRV(ierr);
    ierr = TSSetApplicationContext(ts, user);  CHKERRV(ierr);
#endif

    /* Handle the parameters for a stationary computation */
    if (isStationary) {
        doAbortOnClResidual = doAbortOnCdResidual = false;

        abortResidual = getDbl("abortResidual", "1e-6");

        abortVariable = getInt("abortVariable", "1") - 1;
        switch (abortVariable) {
        case RHO:
            strcpy(abortVariableName, "RHO");
            break;
        case MX:
            strcpy(abortVariableName, "MX");
            break;
        case MY:
            strcpy(abortVariableName, "MY");
            break;
        case E:
            strcpy(abortVariableName, "E");
            break;
        }

        clAbortResidual = getDbl("cl_abortResidual", "0.0");
        cdAbortResidual = getDbl("cd_abortResidual", "0.0");

        if (!rank) printf("| Stationary Problem\n");
        if ((clAbortResidual == 0.0) && (cdAbortResidual == 0.0)) {
            if (!rank) printf("| Abort Residual '%s': %g\n",
abortVariableName, abortResidual);
        } else if (clAbortResidual > 0.0) {
            if (!rank) printf("| Cl Abort Residual: %g\n",
clAbortResidual);
            doAbortOnClResidual = true;
```