```
#include "Durbin.H"
#include "addToRunTimeSelectionTable.H"
#include "wallFvPatch.H"
#include "fixedInternalValueFvPatchField.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace incompressible
{
namespace RASModels
{

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

defineTypeNameAndDebug(Durbin, 0);
addToRunTimeSelectionTable(RASModel, Durbin, dictionary);

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Durbin::Durbin
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    transportModel& lamTransportModel
)
:
    RASModel(typeName, U, phi, lamTransportModel),
    GenElliptic(U, phi, lamTransportModel),

    solveK_(coeffDict_.lookupOrAddDefault<Switch>("solveK", true)),
    fBC_(coeffDict_.lookupOrAddDefault<word>("fBC", "automatic")),
    crossTurbDiffusion_(coeffDict_.lookupOrAddDefault<Switch>("crossTurbDiffusion", false)),
    wallsAlignedWithZ_(coeffDict_.lookupOrAddDefault<Switch>("wallsAlignedWithZ", true)),
{
    printCoeffs();
}


// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //


void Durbin::correct()
{
    GenElliptic::correct();

    if (!turbulence_)
    {
        return;
    }

    volSymmTensorField P = -twoSymm(R_ & fvc::grad(U_));
    volScalarField G("RASModel::G", 0.5*mag(tr(P)));

    volScalarField Ts("T", T());

    #include "../include/epsilonWallI2.H" // set patch internal eps values

    // split R_ into normal diffusion and cross diffusion terms
    volSymmTensorField Rdiag = R_;
    dimensionedScalar kzero = k0_ * 0.0;
    Rdiag.replace(symmTensor::XY, kzero);
    Rdiag.replace(symmTensor::YZ, kzero);
    Rdiag.replace(symmTensor::XZ, kzero);
    volSymmTensorField Rupper = R_ - Rdiag;
```



```
    symmTensor  minDiagR = gMin(Rdiag);

    surfaceScalarField Tsf = fvc::interpolate(Ts, "interpolate(T)");
    surfaceSymmTensorField Rdiagf  = fvc::interpolate(Rdiag, "interpolate(R)");
    surfaceSymmTensorField Rupperf = fvc::interpolate(Rupper, "interpolate(R)");

    // Dissipation equation
    tmp<fvScalarMatrix> epsEqn
    (
        fvm::ddt(epsilon_)
        + fvm::div(phi_, epsilon_)
        - fvm::Sp(fvc::div(phi_), epsilon_)
        - fvm::laplacian(Cmu_/sigmaEps_ * Tsf * Rdiagf, epsilon_, "laplacian(epsilon)")
        - fvm::laplacian(nu(), epsilon_, "laplacian(epsilon)")
      ==
        C1_ * G/Ts * ( 1.0 + 0.1*G/epsilon_)
        - fvm::Sp(C2_/Ts, epsilon_)
    );

    if(crossTurbDiffusion_)
    {
        epsEqn() -= fvc::laplacian(Cmu_/sigmaEps_ * Tsf * Rupperf, epsilon_, "laplacian(epsilon
)");
    }

    epsEqn().relax();
    epsEqn().boundaryManipulate(epsilon_.boundaryField());
    solve(epsEqn);
    bound(epsilon_, epsilon0_);

    // TKE equation
    if(solveK_)
    {
        tmp<fvScalarMatrix> kEqn
            (
                fvm::ddt(k_)
                + fvm::div(phi_, k_)
                - fvm::Sp(fvc::div(phi_), k_)
                - fvm::laplacian(Cmu_/sigmaK_ * Tsf * Rdiagf, k_, "laplacian(k)")
                - fvm::laplacian(nu(), k_, "laplacian(k)")
                ==
                G
                - fvm::Sp(epsilon_/k_, k_)
            );

        if(crossTurbDiffusion_)
        {
            kEqn() -= fvc::laplacian(Cmu_/sigmaK_ * Tsf * Rupperf, k_, "laplacian(k)");
        }

        kEqn().relax();
        solve(kEqn);
    }
    else
    {
        k_ = 0.5 * tr(R_);
    }
    bound(k_, k0_);

    // Reynolds stress equation
    #include "fWallI.H" // set patch internal f values

    tmp<fvSymmTensorMatrix> REqn
        (
            fvm::ddt(R_)
            + fvm::div(phi_, R_)
            - fvm::Sp(fvc::div(phi_), R_)
```

```
            - fvm::laplacian(Cmu_/sigmaK_ * Tsf * Rdiagf, R_, "laplacian(R)")
            - fvm::laplacian(nu(), R_, "laplacian(R)")
            + fvm::Sp(epsilon_/k_, R_)
            ==
            P                                          // production tensor
            + k_ * f_
        );

    if(crossTurbDiffusion_)
    {
        REqn() -= fvc::laplacian(Cmu_/sigmaK_*Ts*Rupper, R_, "laplacian(R)");
    }

    REqn().relax();
    solve(REqn);

    if(solveK_)
    {
        forAll(R_, celli)
        {
            symmTensor& rij = R_.internalField()[celli];
            rij.zz() = 2.0*k_.internalField()[celli] - rij.xx() - rij.yy();
        }
    }


    volScalarField Ls = L();
    Ts = T(); // re-compute time scale

    volSymmTensorField exSrc = -Clrr1_*dev(R_)/Ts - Clrr2_*dev(P);


    tmp<fvSymmTensorMatrix> fEqn
        (
            fvm::laplacian(f_)
            ==
            fvm::Sp(1.0/sqr(Ls), f_)
            -
            (
                exSrc/k_ + dev(R_)/(k_*Ts)
            ) / sqr(Ls)
            );

    fEqn().relax();
    fEqn().boundaryManipulate(f_.boundaryField());
    solve(fEqn);

}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace RASModels
} // End namespace incompressible
} // End namespace Foam

// ************************************************************************* //
```



```
#include "muGenElliptic.H"
#include "wallFvPatch.H"
#include "wallDistData.H"
#include "wallPointYPlus.H"
#include "gaussLaplacianScheme.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{
namespace fv
{
  makeFvLaplacianTypeScheme(gaussLaplacianScheme, symmTensor, symmTensor)
}
namespace incompressible
{
namespace RASModels
{

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

muGenElliptic::muGenElliptic
(
    const volVectorField& U,
    const surfaceScalarField& phi,
    transportModel& lamTransportModel,
    const volSymmTensorField & RAvg,
    const volScalarField & epsilonAvg
)
:
    muRASModel(typeName, U, phi, lamTransportModel, RAvg, epsilonAvg),
    mesh_(U.mesh()),

{
    IOdictionary relaxParameters
        (
            IOobject
            (
                "relaxParameters",
                runTime_.constant(),
                "../../constant",
                mesh_,
                IOobject::MUST_READ,
                IOobject::NO_WRITE
            )
        );

    dictionary couplingDict(relaxParameters.subDictPtr("couplingOptions"));

    imposeTurbEvery_ =
        couplingDict.lookupOrDefault<label>("mapL2REvery", 1, true);

    Info << "(If enabled) directly imposing turb quantities every: " << imposeTurbEvery_ << end
l;
}


void muGenElliptic::updateKolmogorovFlag()
{
    // wall unit as defined by nu/sqrt(tauw/rho)
    volScalarField ystar
    (
        IOobject
        (
            "ystar",
            mesh_.time().constant(),
            mesh_
```

```
        ),
        mesh_,
        dimensionedScalar("ystar", dimLength, 1.0)
    );

    const fvPatchList& patches = mesh_.boundary();
    forAll(patches, patchi)
    {
        if (isA<wallFvPatch>(patches[patchi]))
        {
            const fvPatchVectorField& Uw = U_.boundaryField()[patchi];
            const scalarField& nuw = nu().boundaryField()[patchi];
            // Note: nuw is used instead of nueff
            // for wall-resolving mesh, nut should be zero at wall
            ystar.boundaryField()[patchi] =
                nuw/sqrt(nuw*mag(Uw.snGrad()) + VSMALL);
        }
    }

    wallPointYPlus::yPlusCutOff = 500;
    wallDistData<wallPointYPlus> y(mesh_, ystar);

    KolmogorovFlag_ = pos(yStarLim_ - y/ystar);
}


tmp<volScalarField> muGenElliptic::T() const
{
    return max
        (
            k_/(epsilon_ + epsilonSmall_),
            KolmogorovFlag_ * 6.0 * sqrt(nu()/(epsilon_ + epsilonSmall_))
        );
}

tmp<volScalarField> muGenElliptic::L() const
{
    return
        CL_*max
        (
            pow(k_,1.5)/(epsilon_ + epsilonSmall_),
            KolmogorovFlag_ * CEta_ * pow(pow(nu(),3.0)/(epsilon_ + epsilonSmall_),0.25)
        );
}


tmp<volSymmTensorField> muGenElliptic::devReff() const
{
    return tmp<volSymmTensorField>
    (
        new volSymmTensorField
        (
            IOobject
            (
                "devRhoReff",
                runTime_.timeName(),
                mesh_,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            R_ - nu()*dev(twoSymm(fvc::grad(U_)))
        )
    );
}


tmp<fvVectorMatrix> muGenElliptic::divDevReff(volVectorField& U) const
```



```
{
    if(implicitDiv_)
    {
        return
            (
                fvc::div(R_)
                + fvc::laplacian(nut(), U, "laplacian(nuEff,U)")
                - fvm::laplacian(nuEff(), U)
            );
    }
    else
    {
        return
            (
                fvc::div(R_)
                - fvm::laplacian(nu(), U)
            );
    }
}


void muGenElliptic::correct()
{
    updateKolmogorovFlag();
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace RASModels
} // End namespace incompressible
} // End namespace Foam

// ************************************************************************* //
```