```
/*---------------------------------------------------------------------------*\
Class
    Foam::mcParticle

Description
    Incomplete example illustrating how to implement a custom particle class.

Author
    Michael Wild

SourceFiles
    mcParticleIO.C
    mcParticle.C

\*---------------------------------------------------------------------------*/

#ifndef mcParticle_H
#define mcParticle_H

#include "particle.H"
#include "contiguous.H"

class mcParticleCloud;

class mcParticle : public particle
{
  // Private Data

    // ORDER IN DESCENDING DATA SIZE!
    // IF YOU UPDATE THIS LIST, ALSO UPDATE mcParticleIO.C!
    scalar m_;   // 8 byte
    vector U_;   // contiguous 8 byte
    vector Ut_;
    scalar rho_;
    /*
    label  n_;   // 4 byte
    bool   b_;   // 1 byte
    */

public:

  TypeName("mcParticle")
  friend class Cloud<mcParticle>;

  //- Persistent storage and helper functions useful during tracking,
  //  e.g. interpolators.
  class trackingData :
    public particle::TrackingData<mcParticleCloud>
  {
     // ...
  };

  // Constructors

    //- Construct from components
    mcParticle(const mcParticleCloud&, const vector& pos, label celli,
               scalar m, const vector& U, scalar rho);
    //- Construct from Istream
    mcParticle(const Cloud<mcParticle>&, Istream&, bool readFields=true);
    //- Clone the particle
    autoPtr<particle> clone() const;
    //- Factory class for parallel transfer
    class iNew
    {
      /* Copy-paste from Foam::particle, s/particle/mcParticle/g */
    };

  // Access

      //- Return the particle mass (statistical weight)
      scalar m() const { return m_; }
      //- Return the particle mass (statistical weight)
```



```
      inline scalar& m() { return m_; }
      //- Return the particle velocity
      inline const vector& U() const { return U_; }
      //- Return the particle velocity
      inline vector& U() { return U_; }
      // ...

  // Public Member Functions

    //- Evolve particle for trackTime
    bool move(trackingData&, const scalar trackTime);

  // Patch interactions

    // SKIPPED hit*Patch FUNCTIONS
    hitPatch(/* ...*/); // called before other hit*Patch(/*...*/)

    //- Rotate particle properties (e.g. reflection)
    void transformProperties(const tensor&);
    //- Shift particle properties (e.g. cyclic patch)
    void transformProperties(const vector&);

  // I/O

    //- Read properties from files <time>/lagrangian/<cloudName>/<prop>
    static void readFields(Cloud<mcParticle>&);
    //- Write properties to files <time>/lagrangian/<cloudName>/<prop>
    static void writeFields(const Cloud<mcParticle>&);
    //- Write properties to stream (e.g. for parallel transfer)
    friend Ostream& operator<<(Ostream&, const mcParticle&);
};

// IF CONTAINS NON-CONTIGUOUS DATA
/*
template<>
inline bool contiguous<mcParticle>()
{
  return false;
}
*/

#endif
```

```
#include "mcParticle.H"
#include "mcParticleCloud.H"

namespace Foam
{
    defineTypeNameAndDebug(mcParticle, 0);
}


Foam::mcParticle::mcParticle
(const mcParticleCloud& c, const vector& pos,
label celli, scalar m, const vector& U, scalar rho);
:
    particle(c.pMesh(), pos, celli),
    m_(m),
    U_(U),
    Ut_(),
    rho_(rho),
    n_(0),
    b_(false)
{}


bool Foam::mcParticle::move
(
    mcParticle::trackingData& td,
    const scalar trackTime
)
{
    td.switchProcessor = false;
    td.keepParticle = true;
    const mcParticleCloud& mcpc = refCast<mcParticleCloud>(td.cloud());
    const polyMesh& mesh = mcpc.pMesh();
    const polyBoundaryMesh& pbMesh = mesh.boundaryMesh();

    scalar tEnd = (1.0 - stepFraction())*trackTime;
    scalar dtMax = tEnd;

    // At beginning of time step, update velocity
    if (stepFraction() < SMALL)
    {
        /* SKIPPED U_ +=  .... ; */
    }

    // Compute tracking velocity (i.e. handle 2D and wedge cases)
    Ut_ = U_;
    meshTools::constrainDirection(mesh, mesh.solutionD(), Ut_);
    point destPos = position() + tEnd * Ut_;
    if (mcpc.isAxiSymmetric())
    {
        vector rotatedCentreNormal = mcpc.axis()^destPos;
        rotatedCentreNormal /= mag(rotatedCentreNormal);
        tensor T = rotationTensor(rotatedCentreNormal, mcpc.centrePlaneNormal());
        transformProperties(T);
        destPos = transform(T, destPos);
        // constrain to kill numerical artifacts
        meshTools::constrainDirection(mesh, mesh.geometricD(), destPos);
        Ut_ = (destPos - position())/tEnd;
    }

    while (td.keepParticle && !td.switchProcessor && tEnd > SMALL)
    {
        scalar dt = min(dtMax, tEnd);
        destPos = position() + dt*Ut_;
        // do actual tracking
        scalar tf = trackToFace(destPos, td);
        ++nSteps_;
        dt *= tf;
        tEnd -= dt;
        stepFraction() = 1.0 - tEnd/trackTime;

        if (onBoundary() && td.keepParticle)
```



```
        {
            if (isA<processorPolyPatch>(pbMesh[patch(face())]))
            {
                td.switchProcessor = true;
            }
        }
    }
    return td.keepParticle;
}


void Foam::mcParticle::transformProperties (const tensor& T)
{
    particle::transformProperties(T);
    U_ = transform(T, U_);
    Ut_ = transform(T, Ut_);
}


void Foam::mcParticle::transformProperties(const vector& separation)
{
    particle::transformProperties(separation);
}

// SKIPPED hit*Patch(...) FUNCTIONS
```

```
#include "mcParticle.H"
#include "IOstreams.H"
#include "mcParticleCloud.H"

Foam::mcParticle::mcParticle
(
    const Cloud<mcParticle>& cloud,
    Istream& is,
    bool readFields
)
:
    particle(cloud.pMesh(), is, readFields)
{
    if (readFields)
    {
        if (is.format() == IOstream::ASCII)
        {
            m_ = readScalar(is);
            is  >> U_
                >> Ut_
                >> rho_
                ;
        }
        else
        {
            is.read
            (
                reinterpret_cast<char*>(&m_),
                sizeof(m_) + sizeof(U_) + sizeof(Ut_) + sizeof(rho_)
            );
        }
    }
    // Check state of Istream
    is.check("mcParticle::mcParticle(Istream&)");
}


void Foam::mcParticle::readFields(Cloud<mcParticle>& c)
{
    if (!c.size())
    {
        return;
    }
    particle::readFields(c);

    mcParticleCloud& mcpc = refCast<mcParticleCloud>(c);

    IOField<scalar> m(c.fieldIOobject("m", IOobject::MUST_READ));
    c.checkFieldIOobject(c, m);

    IOField<vector> U(c.fieldIOobject("U", IOobject::MUST_READ));
    c.checkFieldIOobject(c, U);

    IOField<scalar> rho(c.fieldIOobject("rho", IOobject::MUST_READ));
    c.checkFieldIOobject(c, rho);

    label i = 0;
    forAllIter(Cloud<mcParticle>, c, iter)
    {
        mcParticle& p = iter();

        p.m_ = m[i];
        p.U_ = U[i];
        p.rho_ = rho[i];
        ++i;
    }
}


void Foam::mcParticle::writeFields(const Cloud<mcParticle>& c)
{
    particle::writeFields(c);
```



```
    const mcParticleCloud& mcpc = refCast<const mcParticleCloud>(c);

    label np = c.size();

    IOField<scalar> m(c.fieldIOobject("m", IOobject::NO_READ), np);
    IOField<vector> U(c.fieldIOobject("U", IOobject::NO_READ), np);
    IOField<scalar> rho(c.fieldIOobject("rho", IOobject::NO_READ), np);

    label i = 0;
    forAllConstIter(Cloud<mcParticle>, c, iter)
    {
        const mcParticle& p = iter();

        m[i] = p.m_;
        U[i] = p.U_;
        rho[i] = p.rho_;
        i++;
    }

    m.write();
    U.write();
    rho.write();
}


Foam::Ostream& Foam::operator<<(Ostream& os, const mcParticle& p)
{
    if (os.format() == IOstream::ASCII)
    {
        os  << static_cast<const particle&>(p)
            << token::SPACE << p.m_
            << token::SPACE << p.U_
            << token::SPACE << p.Ut_
            << token::SPACE << p.rho_;
    }
    else
    {
        os  << static_cast<const particle&>(p);
        os.write
        (
            reinterpret_cast<const char*>(&p.m_),
            sizeof(p.m_) + sizeof(p.U_) + sizeof(p.Ut_) + sizeof(p.rho_)
        );
    }
    // Check state of Ostream
    os.check("Ostream& operator<<(Ostream&, const mcParticle&)");
    return os;
}
```

```
/*---------------------------------------------------------------------------*\
Class
    Foam::mcParticleCloud

Description
    Incomplete example illustrating how to implement a custom particle cloud.

Author
    Michael Wild

SourceFiles
    mcParticleCloud.C

\*---------------------------------------------------------------------------*/

#ifndef mcParticleCloud_H
#define mcParticleCloud_H

#include "Cloud.H"
#include "mcParticle.H"
#include "dictionary.H"

class mcParticleCloud:
    public Cloud<mcParticle>
{
  // Private Data

    //- FV velocity
    const volVectorField& Ufv_;
    //- FV pressure
    const volScalarField& pfv_;
    //- Averaged 0th statistical moment
    DimensionedField<scalar, volMesh> mMom_;
    //- Averaged 1st statistical moment of inverse density (volume)
    DimensionedField<vector, volMesh> VMom_;
    //- Averaged 1st statistical moment of velocity
    DimensionedField<vector, volMesh> UMom_;
    //- Averaged 2nd statistical moment of fluctuating velocity
    DimensionedField<symmTensor, volMesh> uuMom_;
    //- Extracted, time-averaged density
    volScalarField rho_;
    //- Extracted, time-averaged particle-mass density
    volScalarField pmd_;
    //- Extracted, time-averaged velocity
    volVectorField U_;
    //- Extracted, time-averaged turbulent stress tensor
    volSymmTensorField Tau_;
    //- Extracted, time-averaged TKE
    volScalarField k_;

  // Private Member Functions

    //- Update moments and the quantities remembered by particles
    void updateCloudPDF(scalar existWt);

    //- Initialize statistical moments
    void initMoments();

    //- Disallow default bitwise copy construct
    mcParticleCloud(const mcParticleCloud&);
    //- Disallow default bitwise assignment
    void operator=(const mcParticleCloud&);

public:

  // Constructors

    //- Construct from components
    // If U, p or rho are NULL, they are looked up by the names defined by the
    // UName, pName rhoName entries (defaulting to U, p and rho, respectively)
    mcParticleCloud
    (
```



```
      const fvMesh& mesh,
      const dictionary& dict,
      const word& cloudName = "defaultCloud",
      const volVectorField* U = 0,
      const volScalarField* p = 0,
      const volScalarField* rho = 0
    );

  // Access

    //- Returns the FV TKE field (SKIPPED HERE)
    const volScalarField& kfv() const;

    //- Returns the FV turbulence frequency field (SKIPPED HERE)
    const volScalarField& omega() const;

    //- Returns true if the case uses a wedge geometry
    bool isAxiSymmetric() const;

    //- Axis of the wedge (only defined if isAxiSymmetric())
    const vector& axis() const;

    //- Centre plane of the wedge (only defined if isAxiSymmetric())
    const vector& centrePlaneNormal() const;

  // Public Member Functions

    //- Initialize the particle cloud
    void initReleaseParticles();

    //- Randomly generate N particles in celli, with provided cell-based
    // values and the scale of velocity fluctuation
    void particleGenInCell
    (
      label celli,
      label N,
      scalar m,
      const vector& U,
      const vector& urms
    );

    //- Evolve the particles
    void evolve();
};

#endif
```

```
#include "mcParticleCloud.H"
#include "fvMesh.H"
#include "interpolationCellPointFace.H"
#include "boundBox.H"
#include "fvc.H"

namespace Foam
{

    defineTemplateTypeNameAndDebug(Cloud<mcParticle>, 0);
}


Foam::mcParticleCloud::mcParticleCloud
(
    const fvMesh& mesh,
    const dictionary& dict,
    const word& cloudName,
    const volVectorField* U,
    const volScalarField* p,
    volScalarField* rho
)
:
    Cloud<mcParticle>(mesh, cloudName, false),
    mesh_(mesh),
    dict_(dict),
    runTime_(mesh.time()),
    Ufv_
    (
        U ? *U : mesh_.lookupObject<volVectorField>
                    (dict_.lookupOrDefault<word>("UName", "U"))
    ),
    pfv_
    (
        p ? *p : mesh_.lookupObject<volScalarField>
                    (dict_.lookupOrDefault<word>("pName", "p"))
    ),
    mMom_
    (
        IOobject
        (
            "mMoment",
            runTime_.timeName(),
            mesh_,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedScalar("mMoment", dimMass, 0)
    ),
    VMom_
    (
        IOobject
        (
            "VMoment",
            runTime_.timeName(),
            mesh_,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedScalar("VMoment", dimVolume, 0)
    ),
    UMom_
    (
        IOobject
        (
            "UMoment",
            runTime_.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
```



```
        ),
        mesh,
        dimensionedVector("UMoment", dimMass*dimVelocity, vector::zero)
    ),
    uuMom_
    (
        IOobject
        (
            "uuMoment",
            runTime_.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedSymmTensor("uuMoment", dimEnergy, symmTensor::zero)
    ),
    rho_
    (
        rho ? *rho : const_cast<volScalarField&>(
            mesh_.lookupObject<volScalarField>(
                dict_.lookupOrDefault<word>("rhoName", "rho")))
    ),
    pmd_
    (
        IOobject
        (
            "pmd",
            runTime_.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimDensity,
        mMom_/mesh_.V(),
        rhocPdf_.boundaryField()
    ),
    U_
    (
        IOobject
        (
            "UPdf",
            runTime_.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimVelocity,
        UMom_/max(mMom_, SMALL_MASS),
        Ufv_.boundaryField()
    ),
    Tau_
    (
        IOobject
        (
            "TauPdf",
            runTime_.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        /* SKIPPED INIT AND BC */
    ),

    k_
    (
        IOobject
        (
            "kPdf",
            runTime_.timeName(),
```

```
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh_,
        dimVelocity*dimVelocity,
        0.5*tr(TaucPdf_.dimensionedInternalField()),
        /* SKIPPED BC */
    )
{
    // If particle data found, read from files. Otherwise, initialize.
    if (size() > 0)
    {
        mcParticle::readFields(*this);
    }
    else
    {
        initReleaseParticles();
    }
    initMoments();
    updateParticlePDF();
}


Foam::scalar Foam::mcParticleCloud::evolve()
{
    // SKIPPED RELEASE PARTICLES AT INLET PATCHES

    // Time-averaging factor
    scalar existWt = 1.0/(1.0 + (runTime_.deltaT()/AvgTimeScale_).value());

    mcParticle::trackingData td(/* ... */);

    Cloud<mcParticle>::move(td, runTime_.deltaT().value());

    // Extract statistical averaging to obtain mesh-based quantities
    updateCloudPDF(existWt);
}


void Foam::mcParticleCloud::updateCloudPDF(scalar existWt)
{
    DimensionedField<scalar, volMesh> mMomInstant
    (
        IOobject
        (
            "mMomInstant",
            mesh_.time().timeName(),
            mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh_,
        dimensionedScalar("mMomInstant", dimMass, 0.0)
    );
    // ...
    // Similar for VMomInstant, UMomInstant and uuMomInstant
    // ...

    // Loop through particles to accumulate moments (0, 1, 2 order)
    interpolationCellPointFace<vector> UInterp(Ufv_);
    forAllConstIter(mcParticleCloud, *this, pIter)
    {
        const mcParticle& p = pIter();
        label cellI = p.cell();
        vector U = UInterp.interpolate(p.position(), cellI, p.face());
        vector u = p.U() - U;

        const scalar m = p.m();
        mMomInstant[cellI]  += m;
        VMomInstant[cellI]  += m / p.rho();
        UMomInstant[cellI]  += m * p.U();
```



```
        uuMomInstant[cellI] += m * symm(u*u);
    }

    scalar newWt = 1.0 - existWt;
    // Do time-averaging of moments and compute mean fields
    mMom_  = existWt * mMom_  + newWt * mMomInstant;
    pmd_.internalField() = mMom_ / mesh_.V();

    VMom_  = existWt * VMom_  + newWt * VMomInstant;
    rho_.internalField()   = mMom_ / VMom_;
    rho_.correctBoundaryConditions();

    UMom_  = existWt * UMom_  + newWt * UMomInstant;
    U_.internalField()   = UMom_ / mMom_;
    U_.correctBoundaryConditions();

    uuMom_ = existWt * uuMom_ + newWt * uuMomInstant;
    Tau_.internalField() = uuMom_/mMom_;
    Tau_.correctBoundaryConditions();

    k_.internalField()   = 0.5*tr(Tau_.internalField());
    k_.correctBoundaryConditions();
}


void Foam::mcParticleCloud::initReleaseParticles()
{
    // Populate each cell with 30 particles in each cell
    forAll(Ufv_, celli)
    {
        scalar m = mesh_.V()[celli]*rho_[celli] / 30;
        vector U = U_[celli];
        scalar urms = sqrt(2./3.*kfv()()[celli]);
        vector uscales(urms, urms, urms);
        particleGenInCell(celli, 30, m, U, uscales);
    }
}


void Foam::mcParticleCloud::particleGenInCell
(
    label celli,
    label N,
    scalar m,
    const vector& Updf,
    const vector& uscales
)
{
    boundBox cellbb
    (
        pointField
        (
            mesh_.points(),
            mesh_.cellPoints()[celli]
        ),
        false
    );

    vector minb = cellbb.min();
    vector dimb = cellbb.max() - minb;

    label Npgen = 0;
    for (int i = 0; i < 100*N; ++i)
    {
        // Relative coordinate [0, 1] in this cell
        vector xi = random().vector01();
        // Random offset from min point
        scalar rx = min(max(10.0*SMALL, xi.x()), 1.0-10.0*SMALL);
        scalar ry = min(max(10.0*SMALL, xi.y()), 1.0-10.0*SMALL);
        scalar rz = min(max(10.0*SMALL, xi.z()), 1.0-10.0*SMALL);
        vector offsetRnd(rx*dimb.x(), ry*dimb.y(), rz*dimb.z());
```

```
        // Generate a particle position
        vector position = minb + offsetRnd;

        // If the case has reduced dimensionality, put the coordinate of the
        // reduced dimension onto the coordinate plane
        if (mesh_.nGeometricD() <= 2)
        {
            meshTools::constrainDirection(mesh_, mesh_.geometricD(), position);
        }

        // Initially put N particle per cell
        if (mesh_.pointInCell(position, celli))
        {
            // random() not shown here
            vector u
            (
              random().GaussNormal()*uscales.x(),
              random().GaussNormal()*uscales.y(),
              random().GaussNormal()*uscales.z()
            );
            vector Up = u + U;

            mcParticle* ptr = new mcParticle
            (
                *this,
                position,
                celli,
                m,
                Up,
                rho_[celli]
            );

            addParticle(ptr);
            ++Npgen;
        }

        // until enough particles are generated.
        if (Npgen >= N) break;
    }

    if (Npgen < N)
    {
        FatalErrorIn("mcParticleCloud::initReleaseParticles()")
            << "Only " << Npgen << " particles generated for cell "
            << celli << nl << "Something is wrong" << exit(FatalError);
    }
}


void Foam::mcParticleCloud::initMoments()
{
    bool readOk =
        mMom_.headerOk() &&
        VMom_.headerOk() &&
        UMom_.headerOk() &&
        uuMom_.headerOk();
    if (readOk)
    {
        Info<< "Moments read correctly." << endl;
    }
    else if (size() > 0)
    {
        Info<< "Moments are missing. Forced re-initialization." << endl;
        mMom_  = mesh_.V()*rho_;
        pmd_.internalField() = rho_.internalField();

        VMom_  = mMom_/rho_;

        UMom_  = mMom_*Ufv_;
        U_.internalField() = Ufv_.internalField();
        U_.correctBoundaryConditions();
```



```
        uuMom_ = mMom_*turbulenceModel().R()();

        k_.internalField() = kfv().internalField();
        k_.correctBoundaryConditions();
    }
    else
    {
        FatalErrorIn("mcParticleCloud::checkMoments()")
            << "Not all moment fields available and no particles present."
            << endl;
    }
}
```