```
/********************************************************************
    NairnMPM.cpp
    nairn-mpm-fea

    Created by jnairn on Mon Nov 19 2001.
    Copyright (c) 2001 John A. Nairn, All rights reserved.
********************************************************************/

#include "stdafx.h"
#include "System/ArchiveData.hpp"
#include "System/UnitsController.hpp"
#include "NairnMPM_Class/NairnMPM.hpp"
#include "NairnMPM_Class/MeshInfo.hpp"
#include "NairnMPM_Class/MPMTask.hpp"
#include "NairnMPM_Class/InitializationTask.hpp"
#include "NairnMPM_Class/InitVelocityFieldsTask.hpp"
#include "NairnMPM_Class/ProjectRigidBCsTask.hpp"
#include "NairnMPM_Class/ExtrapolateRigidBCsTask.hpp"
#include "NairnMPM_Class/PostExtrapolationTask.hpp"
#include "NairnMPM_Class/SetRigidContactVelTask.hpp"
#include "NairnMPM_Class/MassAndMomentumTask.hpp"
#include "NairnMPM_Class/UpdateStrainsFirstTask.hpp"
#include "NairnMPM_Class/GridForcesTask.hpp"
#include "NairnMPM_Class/PostForcesTask.hpp"
#include "NairnMPM_Class/UpdateParticlesTask.hpp"
#include "NairnMPM_Class/UpdateStrainsLastTask.hpp"
#include "NairnMPM_Class/UpdateStrainsLastContactTask.hpp"
#include "NairnMPM_Class/UpdateMomentaTask.hpp"
#include "NairnMPM_Class/RunCustomTasksTask.hpp"
#include "NairnMPM_Class/MoveCracksTask.hpp"
#include "NairnMPM_Class/ResetElementsTask.hpp"
#include "NairnMPM_Class/XPICExtrapolationTask.hpp"
#include "Materials/MaterialBase.hpp"
#include "Custom_Tasks/CustomTask.hpp"
#include "Custom_Tasks/CalcJKTask.hpp"
#include "Custom_Tasks/PropagateTask.hpp"
#include "Custom_Tasks/DiffusionTask.hpp"
#include "Custom_Tasks/ConductionTask.hpp"
#include "Nodes/CrackVelocityFieldMulti.hpp"
#include "Cracks/CrackHeader.hpp"
#include "Cracks/CrackSurfaceContact.hpp"
#include "Elements/ElementBase.hpp"
#include "Patches/GridPatch.hpp"
#include "MPM_Classes/MPMBase.hpp"
#include "Global_Quantities/ThermalRamp.hpp"
#include "Global_Quantities/BodyForce.hpp"
#include "Boundary_Conditions/MatPtTractionBC.hpp"
#include "Boundary_Conditions/MatPtFluxBC.hpp"
#include "Boundary_Conditions/MatPtHeatFluxBC.hpp"
#include "Boundary_Conditions/InitialCondition.hpp"
#include "Exceptions/CommonException.hpp"
#ifdef RESTART_OPTION
#include "Custom_Tasks/AdjustTimeStepTask.hpp"
#endif
```

```
#include "NairnMPM_Class/XPICExtrapolationTaskTO.hpp"
#include "Exceptions/MPMWarnings.hpp"
#include <time.h>

// Activate this to print steps as they run. If too many steps happen before
failure
// better to use LOG_PROGRESS in MPMPrefix.hpp
//#define COUT_PROGRESS

enum { NO_RIGID_MM=0,RIGID_CONTACT_MM,RIGID_BLOCK_MM };

// global analysis object
NairnMPM *fmobj=NULL;
MPMTask *firstMPMTask;

// global variables
double timestep=1.;                     // time per MPM step (sec)
double strainTimestepFirst;             // time step for USF when doing USAVG
double strainTimestepLast;              // time step for USL when doing USAVG
double fractionUSF = 0.5;               // fraction time step for USF when doing
USAVG
double mtime=0.;                        // time for current step (sec)
double propTime=1.e30;          // time interval between propagation
calculations (sec)
int maxCrackFields=1;           // Maximum crack velocity fields at a node in a
material (it is MAX_FIELDS_FOR_CRACKS if there are cracks)
int maxMaterialFields;          // Maximum velocity fields or number of
independent materials in multimaterial mode (=1 in single mat mode)
int numActiveMaterials;         // Number of non-rigid materials used by at
least one material point
int maxShapeNodes=10;           // Maximum number of nodes for a particle (plus
1)
int nextPeriodicXPIC=-1;

#pragma mark CONSTRUCTORS

// Constructor
// throws std::bad_alloc
NairnMPM::NairnMPM()
{
        version=15;                                             // main version
        subversion=0;                                   // subversion (must be <
10)
        buildnumber=0;                                  // build number

        mpmApproach=USAVG_METHOD;               // mpm method
        ptsPerElement=4;                                // number of points per
element (2D default, 3D changes it to 8)
        ptsPerSide=2;
        propagate[0]=propagate[1]=NO_PROPAGATION;
        // default crack propagation type
        propagateDirection[0]=propagateDirection[1]=DEFAULT_DIRECTION;  //
default crack propagation direction
        propagateMat[0]=propagateMat[1]=0;
```

```
                // default is new crack with no traction law
        hasTractionCracks=FALSE;                // if any crack segment has a
traction law material
        maxtime=1.;                                             // maximum time
(sec)
        FractCellTime=.5;                               // fraction cell crossed
in 1 step at wave speed (CFL convergence condition)
    PropFractCellTime=-1.;          // fracture cell crossed in 1 step for
propagation time step (currently not user settable)
        TransFractCellTime=0.5;                         // scaling factor for
finding transport time step
        mstep=0;                                                // step number
        warnParticleLeftGrid=0;             // abort when this many leave the
grid (>0 push back, <0 delete, 0 quit when one leaves) (LeaveLimit)
    deleteLeavingParticles=false;
    warnParticleDeleted=-1;         // abort when this many are deleted
(DeleteLimit, <=1 means abort on nan particle)
        multiMaterialMode = false;              // multi-material mode
        skipPostExtrapolation = false;  // if changed to true, will extrapolate
for post update strain updates
        exactTractions=false;                   // exact traction BCs
        hasNoncrackingParticles=false;  // particles that ignore cracks in
multimaterial mode (rigid only in NairnMPM)
#ifdef RESTART_OPTION
    restartScaling = 0.;
    restartCFL = 0.5;
    warnRestartTimeStep = -1;
#endif

        // initialize objects
        archiver=new ArchiveData();             // archiving object

        // mechanics time step
        timeStepMinMechanics = 1.e30;
}

#pragma mark CORE MPM

// start analysis
// throws std::bad_alloc
void NairnMPM::CMPreparations(void)
{
        // Transport tasks
        CreateTransportTasks();

        // check grid settings
        GridAndElementCalcs();

        // Preliminary particle Calculations
        PreliminaryParticleCalcs();

        // material mode settings
        SetupMaterialModeContactXPIC();
```

```
        // create patches or a single patch
        patches = mpmgrid.CreatePatches(np,numProcs);
        if(patches==NULL)
                throw CommonException("Out of memory creating the
patches","NairnMPM::PreliminaryParticleCalcs");

        // create buffers for copies of material properties

UpdateStrainsFirstTask::CreatePropertyBuffers(GetTotalNumberOfPatches());

        // if cracks
        PreliminaryCrackCalcs();

        // Output boundary conditions and mass matrix and grid info
        OutputBCMassAndGrid();

        // create warnings
        CreateWarnings();

        // Create step tasks
        CreateTasks();
}

// Do the MPM analysis
void NairnMPM::CMAnalysis(bool abort)
{
        try
        {       //-------------------------------------------------
                // Archiving
                if(!archiver->BeginArchives(IsThreeD(),maxMaterialFields))
                        throw "No archiving was specified or multiple archiving
blocks not monotonically increasing in start time";
                archiver->ArchiveResults(mtime);

                // optional validation of parameters
                ValidateOptions();

                // exit if do not want analysis
                if(abort) mtime=maxtime+1;

                // ------------------------------------------------------
                // Main MPM Loop
                while(mtime<=maxtime)
                {       // MPM Calculations
                        mstep++;                        // step number
            MPMStep();

                        // advance time and archive if desired
                        mtime+=timestep;
                        archiver->ArchiveResults(mtime);
        }
        }
        catch(CommonException& term)
        {       // calculation stopped, but still report results
```

```
                mtime+=timestep;
                archiver->ForceArchiving();
                archiver->ArchiveResults(mtime);
                cout << endl;
                PrintSection("ABNORMAL TERMINATION");
                term.Display(mstep,mtime);
        }
        catch(CommonException* term)
        {       // calculation stopped, but still report results
                mtime+=timestep;
                archiver->ForceArchiving();
                archiver->ArchiveResults(mtime);
                cout << endl;
                PrintSection("ABNORMAL TERMINATION");
                term->Display(mstep,mtime);
        }
        catch(const char *errMsg)
        {       // string error - exit to main
                throw errMsg;
        }
        catch(...)
        {       // unknown error - exit to main
                throw "Unknown exception in MPMStep() in MPM Loop in
NairnMPM.cpp";
        }
    cout << endl;

    //-------------------------------------------------
    // Custom task reports
    bool hasSectionTitle = false;
    CustomTask *nextTask=theTasks;
    while(nextTask!=NULL)
    {   if(!hasSectionTitle && nextTask->HasReport())
        {   PrintSection("CUSTOM TASK REPORTS");
            hasSectionTitle = true;
        }
        nextTask=nextTask->Report();
    }

    //-------------------------------------------------
    // Report warnings
        warnings.Report();

    //-------------------------------------------------
    // Report on time of exectution
    double execTime=ElapsedTime();
// elpased time in secs
        double cpuTime=CPUTime();
// cpu time in secs

    PrintSection("EXECUTION TIMES AND MEMORY");
    cout << "Calculation Steps: " << mstep << endl;

        char fline[100];
```

```
        sprintf(fline,"Elapsed Time: %.3lf secs\n",execTime);
    cout << fline;

    sprintf(fline,"CPU Time: %.3lf secs\n",cpuTime);
    cout << fline;

        if(mstep>0)
        {       double eTimePerStep = 1000.*execTime/((double)mstep);
                sprintf(fline,"Elapsed Time per Step: %.3lf ms\n",eTimePerStep);
                cout << fline;

                double timePerStep=1000.*cpuTime/((double)mstep);
                sprintf(fline,"CPU Time per Step: %.3lf ms\n",timePerStep);
                cout << fline;

        // profile task results
                MPMTask *nextMPMTask=firstMPMTask;
                while(nextMPMTask!=NULL)
                {
nextMPMTask->WriteProfileResults(mstep,timePerStep,eTimePerStep);
                        nextMPMTask=(MPMTask *)nextMPMTask->GetNextTask();
                }
        }

    //-----------------------------------------------------
    // Trailer
    cout << "\n***** " << CodeName() << " RUN COMPLETED\n";
}

// Main analysis loop for MPM analysis
// Made up of tasks created in MPMAnalysis()
void NairnMPM::MPMStep(void)
{
        // Step initialization
#ifdef LOG_PROGRESS
        char logLine[200];
        archiver->ClearLogFile();
        sprintf(logLine,"Step #%d: Initialize",mstep);
        archiver->WriteLogFile(logLine,NULL);
#endif
#ifdef RESTART_OPTION
    // make sure do get endless restarts
    int restarts=0;
#endif

        // loop through the tasks
        MPMTask *nextMPMTask=firstMPMTask;
        while(nextMPMTask!=NULL)
        {
#ifdef LOG_PROGRESS
                nextMPMTask->WriteLogFile();
#endif
#ifdef COUT_PROGRESS
                cout << "# " << mstep << ": " << nextMPMTask->GetTaskName() <<
```

```
endl;
#endif
                double beginTime=fmobj->CPUTime();
                double beginETime=fmobj->ElapsedTime();
#ifdef RESTART_OPTION
        if(!nextMPMTask->Execute(0))
        {   // restart this time step
            warnings.Issue(warnRestartTimeStep,-1);
            if(restarts==5)
                throw CommonException("Current time step still fails after 5
restarts","NairnMPM::MPMStep");
            restarts++;
            nextMPMTask = firstMPMTask;

AdjustTimeStepTask::ChangeTimestep(fabs(restartScaling)*timestep,fabs(restartSca
ling)*propTime,false);
            if(restartScaling<0.)
            {   cout << "# Restart #" << restarts << " of time step #" << mstep;
                cout << " with time step " <<
timestep*UnitsController::Scaling(1.e3) << " "
                << UnitsController::Label(ALTTIME_UNITS) << endl;
            }
            continue;
        }
#else
                nextMPMTask->Execute(0);
#endif
                nextMPMTask->TrackTimes(beginTime,beginETime);

        // on to next task
                nextMPMTask=(MPMTask *)nextMPMTask->GetNextTask();
#ifdef LOG_PROGRESS
                archiver->WriteLogFile("           Done",NULL);
#endif
#ifdef COUT_PROGRESS
                cout << "#            Done"  << endl;
#endif
        }
}

#pragma mark PREPARATION TASKS

// Create transport tasks if requested
void NairnMPM::CreateTransportTasks(void)
{
        // Active Transport Tasks
        if(fmobj->HasFluidTransport())
        {       diffusion = new DiffusionTask();
                transportTasks = diffusion;
        }
        if(ConductionTask::active)
        {
        conduction=new ConductionTask();
        if(transportTasks)
```

```
            transportTasks->nextTask=conduction;
        else
            transportTasks=conduction;
    }
    else
    {   // these can only be on when conduction is active
                ConductionTask::crackContactHeating = false;
                ConductionTask::matContactHeating = false;
                ConductionTask::crackTipHeating = false;
        }

        // are both the system and the particles isolated?
        if(!ConductionTask::active && ConductionTask::IsSystemIsolated())
        {   MaterialBase::isolatedSystemAndParticles = TRUE;
        }
}

// Check grid settings
// Unstructure grid (legacy code) checks element by element
// Initialize CPDI number of nodes
void NairnMPM::GridAndElementCalcs(void)
{
        // only allows grids created with the Grid command
        // (note: next two sections never occur uless support turned back on)
        if(mpmgrid.GetCartesian()==UNKNOWN_GRID)
                throw CommonException("Support for iunstructured grids is
currently not available","NairnMPM::GridAndElementCalcs");

        // Loop over elements, if needed, to determine type of grid
        if(mpmgrid.GetCartesian()==UNKNOWN_GRID)
        {       int userCartesian = NOT_CARTESIAN;
                double dx,dy,dz,gridx=0.,gridy=0.,gridz=0.;
                for(int i=0;i<nelems;i++)
                {       if(!theElements[i]->Orthogonal(&dx,&dy,&dz))
                        {   // exit if find one that is not orthogonal
                                userCartesian = NOT_CARTESIAN;
                                break;
                        }

                        if(userCartesian == NOT_CARTESIAN)
                        {       // first element, set grid size (dz=0 in 2D) and
set userCartesion flag
                                gridx=dx;
                                gridy=dy;
                                gridz=dz;
                                userCartesian = SQUARE_GRID;
                        }
                        else if(userCartesian==SQUARE_GRID)
                        {       // on sebsequent elements, if size does not
match current values, than elements differ
                                // in size. Set to variable grid but keep going
to check if rest are orthogonal
                                if(!DbleEqual(gridx,dx) || !DbleEqual(gridy,dy)
|| !DbleEqual(gridz,dz))
```

```
                                {       if(IsThreeD())
                                        {   gridz = 1.;
                                                userCartesian =
VARIABLE_ORTHOGONAL_GRID;
                                        }
                                        else
                                                userCartesian =
VARIABLE_RECTANGULAR_GRID;
                                }
                        }
                }

                // unstructured grid - set type, but elements not set so
meshInfo will not know many things
                // and horiz will be <=0
                mpmgrid.SetCartesian(userCartesian,gridx,gridy,gridz);
        }

        // only allows orthogonal grids
        if(mpmgrid.GetCartesian()<=0 ||
mpmgrid.GetCartesian()==NOT_CARTESIAN_3D)
                throw CommonException("Support for non-cartesian grids is
currently not available","NairnMPM::GridAndElementCalcs");

        // Ser number of CPDI nodes being used
        ElementBase::InitializeCPDI(IsThreeD());
}

// Preliminary particle Calclations prior to analysis in loop over particles
//  Verify material type, initialize history, damage, mass, and concentration
potential
//  Get time steps (mechanics and transport) with no CFL
//  Set velocity field
//  Allocate GIMP or CPDI info
//  Reorder rigid and nonrigid particles and rigid contact particles
// throws CommonException()
void NairnMPM::PreliminaryParticleCalcs(void)
{
    // future - make PropFractCellTime a user parameter, which not changed here
if user picked it
    if(PropFractCellTime<0.) PropFractCellTime=FractCellTime;
        double dcell = mpmgrid.GetMinCellDimension();                   // in mm
        maxMaterialFields = 0;
        numActiveMaterials = 0;
    nmpmsNR = 0;
        int hasRigidContactParticles = NO_RIGID_MM;
        int firstRigidPt = -1;
        //double area,volume;
    for(int p=0;p<nmpms;p++)
        {   // set number
        mpm[p]->SetNum(p);

        // verify material is defined
                int matid=mpm[p]->MatID();
```

```
                if(matid>=nmat)
                        throw CommonException("Material point with an undefined
material type","NairnMPM::PreliminaryParticleCalcs");

                // material point can't use traction law or contact law
                if(theMaterials[matid]->MaterialStyle()==TRACTION_MAT)
                        throw CommonException("Material point with traction-law
material","NairnMPM::PreliminaryParticleCalcs");
                if(theMaterials[matid]->MaterialStyle()==CONTACT_MAT)
                        throw CommonException("Material point with contact-law
material","NairnMPM::PreliminaryParticleCalcs");

        // initialize history-dependent material data on this particle
                // might need initialized history data so set it now, but
nonrigid only
                if(!theMaterials[matid]->IsRigid())

mpm[p]->SetHistoryPtr(theMaterials[matid]->InitHistoryData(NULL,mpm[p]));

                // Set material field (always field [0] and returns 1 in single
material mode)
                // Also verify allowsCracks - can only be true if multimaterial
mode; if rigid must be rigid contact material
                maxMaterialFields =
theMaterials[matid]->SetField(maxMaterialFields,multiMaterialMode,matid,numActiv
eMaterials);

                // see if any particle are ignoring cracks
                if(!theMaterials[matid]->AllowsCracks())
                        hasNoncrackingParticles = true;

                // element and mp properties
                double volume;
                Vector psize = mpm[p]->GetParticleSize();
                if(IsThreeD())
                {       volume = 8.*psize.x*psize.y*psize.z;
                }
                else
                {       // when axisymmetric, thickness is particle radial
position, which gives mp = rho*Ap*Rp
                        volume = 4.*psize.x*psize.y*mpm[p]->thickness();
                }
                double rho = theMaterials[matid]->GetRho(mpm[p]);
        // M/L^3, g/mm^3 in Legacy, Rigid 1/mm^3

        // assumes same number of points for all elements (but subclass can
override)
        // for axisymmetric xp = rho*Ap*volume/(# per element)
                mpm[p]->InitializeMass(rho,volume,false);
                // in g

                // IF rigid, do a few things then continue
                // mass will be in L^3 and will be particle volume
                if(theMaterials[matid]->IsRigid())
```

```
                {       // Trap rigid BC (used to be above element and mp
properties,
                        //     but moved here to get volume)
                        if(theMaterials[matid]->IsRigidBC())
                        {       if(firstRigidPt<0) firstRigidPt=p;
                                // CPDI or GIMP domain data only for rigid
contact particles (non-rigid done below)

if(!mpm[p]->AllocateCPDIorGIMPStructures(ElementBase::useGimp,IsThreeD()))
                                        throw CommonException("Out of memory
allocating CPDI domain structures","NairnMPM::PreliminaryParticleCalcs");
                                continue;
                        }

                        // rigid contact and block materials in multimaterial
mode
                        // Note: multimaterial mode implied here because
otherwise fatal error when SetField() above
                        hasRigidContactParticles |= RIGID_CONTACT_MM;
                        if(firstRigidPt<0) firstRigidPt=p;

                        // CPDI or GIMP domain data only for rigid contact
particles (non-rigid done below)

if(!mpm[p]->AllocateCPDIorGIMPStructures(ElementBase::useGimp,IsThreeD()))
                                throw CommonException("Out of memory allocating
CPDI domain structures","NairnMPM::PreliminaryParticleCalcs");

                        continue;
                }

        // now a nonrigid particle
        nmpmsNR = p+1;

        // zero external forces on this particle
                ZeroVector(mpm[p]->GetPFext());

                // concentration potential (negative interpreted as weight
fraction)
                if(mpm[p]->pConcentration<0. && fmobj->HasDiffusion())
                {       double
potential=-mpm[p]->pConcentration/mpm[p]->GetConcSaturation();
                        if(potential>1.000001)
                                throw CommonException("Material point with
concentration potential > 1","NairnMPM::PreliminaryParticleCalcs");
                        if(potential>1.) potential = 1.;
                        mpm[p]->pConcentration=potential;
                }

        // material dependent initialization
        theMaterials[matid]->SetInitialParticleState(mpm[p],np,0);

        // check mechanics time step
        double crot = theMaterials[matid]->WaveSpeed(IsThreeD(),mpm[p]);
```

```
                double tst = dcell/crot;
        if(tst<timeStepMinMechanics)
                {       timeStepMinMechanics = tst;
                }

        // propagation time
        tst = PropFractCellTime*dcell/crot;
        if(tst<propTime) propTime = tst;

                // Transport property time steps
                TransportTask *nextTransport = transportTasks;
                while(nextTransport!=NULL)
                {       // note that diffCon = k/(rho Cv) for conduction and D
for diffusion (max values)
                        double diffCon;
                        nextTransport->TransportTimeStepFactor(matid,&diffCon);
                        if(diffCon>0.)

nextTransport->CheckTimeStep((dcell*dcell)/(2.*diffCon));
                        nextTransport = nextTransport->GetNextTransportTask();
                }

        // CPDI orGIMP domain data for nonrigid particles

if(!mpm[p]->AllocateCPDIorGIMPStructures(ElementBase::useGimp,IsThreeD()))
            throw CommonException("Out of memory allocating CPDI domain
structures","NairnMPM::PreliminaryParticleCalcs");

        }

        // get unadjusted time steps to final time steps
        CFLTimeStep();

        // reorder the particles
        ReorderParticles(firstRigidPt,hasRigidContactParticles);

        // non-standard particle sizes
        archiver->ArchivePointDimensions();
}

// If needed, set up multimaterial more
void NairnMPM::SetupMaterialModeContactXPIC(void)
{
        // multimaterial checks and contact initialization
        if(maxMaterialFields==0 || numActiveMaterials==0)
                throw CommonException("No material points found with an actual
material","NairnMPM::SetupMaterialModeContactXPIC");

        else if(maxMaterialFields==1)
        {       // Turning off multimaterial mode because only one material is
used
                multiMaterialMode = false;
                mpmgrid.volumeGradientIndex = -1;
                mpmgrid.contactByDisplacements = true;
```

```
        }
        else
        {       mpmgrid.MaterialContactPairs(maxMaterialFields);
        }

        // ghost particles will need to know this setting when creating patches
        if(firstCrack!=NULL)
        {   if(firstCrack->GetNextObject()!=NULL)
                        maxCrackFields=MAX_FIELDS_FOR_CRACKS;
                else
                        maxCrackFields=MAX_FIELDS_FOR_ONE_CRACK;
        }

        if(firstCrack!=NULL || multiMaterialMode)
        {       // number of contact extrapolation vectors
                mpmgrid.numContactVectors = 0;

                // displacement extrapolations
                mpmgrid.displacementIndex = mpmgrid.numContactVectors;
                mpmgrid.numContactVectors++;

                // position extrapolations
                if(!mpmgrid.contactByDisplacements ||
!contact.crackContactByDisplacements)
                {       mpmgrid.positionIndex = mpmgrid.numContactVectors;
                        mpmgrid.numContactVectors++;
                }
                else
                        mpmgrid.positionIndex = -1;

                // gradient extrapolations
                if(mpmgrid.volumeGradientIndex>=0)
                {       mpmgrid.volumeGradientIndex = mpmgrid.numContactVectors;
                        mpmgrid.numContactVectors++;
                }

#if MM_XPIC == 1
                // need 3 more vectors to store delta p^alpha and process when
needed
                if(bodyFrc.XPICVectors()>=3)
                        bodyFrc.SetXPICVectors(6);
                // If using on first time step, set flag
                if(bodyFrc.UsingVstar()>0)
                        bodyFrc.SetUsingVstar(VSTAR_WITH_CONTACT);
#endif
        }

}

// crack calculations
void NairnMPM::PreliminaryCrackCalcs(void)
{
        // propagation time step and other settings when has cracks
        if(firstCrack!=NULL)
```

```
        {       CrackHeader *nextCrack=firstCrack;
                double dcell = mpmgrid.GetMinCellDimension();
                while(nextCrack!=NULL)
                {
                        nextCrack->PreliminaryCrackCalcs(dcell,false);
                        if(nextCrack->GetHasTractionLaws())
                                hasTractionCracks=TRUE;
                        nextCrack=(CrackHeader *)nextCrack->GetNextObject();
                }
        }
}

// create warnings
void NairnMPM::CreateWarnings(void)
{
        if(firstCrack!=NULL)
        {       // crack warnings
                CrackHeader::warnNodeOnCrack=warnings.CreateWarning("Unexpected
crack side; possibly caused by node or particle on a crack",-1,5);
                CrackHeader::warnThreeCracks=warnings.CreateWarning("Node with
three or more cracks",-1,5);
        }

    // get place to delete particles leaving the grid or nan particles
    if(warnParticleLeftGrid<0 || warnParticleDeleted>1)
    {   if(!mpmgrid.IsStructuredGrid())
        {   // error for now, could add option to enter of location
            throw CommonException("Deleting lost or nan particles needs a
structured grid (or a store location)","NairnMPM::CreateWarnings");
        }
        int cornerElement = mpmgrid.GetCornerNonEdgeElementNumber();

theElements[cornerElement-1]->FindCentroid(&ResetElementsTask::storeDeleted);
    }

    //  particles leaving grid (0 reverts to 1% pushed back >0 is pushed back,
<0 is deleted)
    // See LeaveLimit
        if(warnParticleLeftGrid==0)
        {       // use default setting to quit if 1% of particle leave the grid
        // To abort on first leave set DeleteLimit to 1
                warnParticleLeftGrid = nmpms/100;
        }
    if(warnParticleLeftGrid>1)
        warnParticleLeftGrid = warnings.CreateWarning("Particle has left the
grid and was pushed back",warnParticleLeftGrid,0);
    else if(warnParticleLeftGrid==1)
        warnParticleLeftGrid = warnings.CreateWarning("Particle has left the
grid; simulations will stop",-1,0);
    else
    {   deleteLeavingParticles = true;
        warnParticleLeftGrid = warnings.CreateWarning("Particle has left the
grid and was deleted",-warnParticleLeftGrid,0);
    }
```

```
    // particle deleted because of nan warning
    // See DeleteLimit
    if(warnParticleDeleted>1)
        warnParticleDeleted = warnings.CreateWarning("Particle with nan was
deleted",warnParticleDeleted,0);
        else
                warnParticleDeleted = warnings.CreateWarning("Particle with nan;
simulation will stop",-1,0);

#ifdef RESTART_OPTION
    if(fabs(restartScaling)>1.e-6)
        warnRestartTimeStep = warnings.CreateWarning("MPM step restarted with
smaller time step",-1,0);
#endif

        // nodal point calculations
        // Note that no crack velocity or materical velocity fields created
before this call
        NodalPoint::PrepareNodeCrackFields();

        // finish warnings
        warnings.CreateWarningList();

}

// create all the tasks needed for current simulation
// Custom task (add CalcJKTask() if needed) and Initialize them all
// MPM time step tasks
// throws std::bad_alloc
void NairnMPM::CreateTasks(void)
{
    CustomTask *nextTask;

        // if there are cracks, create J/K task and optionally a propagation
task
        //              (it is essential for propagation task to be after the JK
task)
        //              (insert these tasks before other custom tasks)
        if(firstCrack!=NULL)
        {       if(propagate[0] || archiver->WillArchiveJK(false))
                {   theJKTask = new CalcJKTask();
                        if(propagate[0])
                        {       nextTask = new PropagateTask();
                                // task order will be theJKTasks -> propagate
task -> theTasks
                                theJKTask->nextTask = nextTask;
                                nextTask->nextTask = theTasks;
                        }
                        else
                        {       // task order will be theJKTasks -> theTasks
                                theJKTask->nextTask = theTasks;
                        }
                        // switch to theJKTask to be the new first one
```

```
                        theTasks=theJKTask;
                        ElementBase::AllocateNeighbors();
                }
                else
                {   // Turn off any J-K settings that can cause problems
                        JTerms = -1;
                }
        }
    else
    {   // Turn off any crack settings that can cause problems
        JTerms = -1;
    }

        // see if any custom or transport tasks need initializing
        if(theTasks!=NULL || transportTasks!=NULL)
        {       PrintSection("SCHEDULED CUSTOM TASKS");

                // transport tasks
                TransportTask *nextTransport = transportTasks;
                while(nextTransport!=NULL)
                {       nextTransport = nextTransport->Initialize();
                        cout << endl;
                }

                // custom tasks
                nextTask=theTasks;
                while(nextTask!=NULL)
                {       nextTask=nextTask->Initialize();
                        cout << endl;
                }
        }

        //-------------------------------------------------
        // Create all the step tasks

        // INITIALIZATION Tasks
        MPMTask *lastMPMTask,*nextMPMTask;
        lastMPMTask=firstMPMTask=(MPMTask *)new
InitializationTask("Initialize");

        if(firstCrack!=NULL || maxMaterialFields>1)
        {       nextMPMTask=(MPMTask *)new InitVelocityFieldsTask("Decipher
Crack and Material Fields");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                lastMPMTask=nextMPMTask;
        }

        // MASS AND MOMENTUM TASKS

        // if rigid BCs by extrapolation, extrapolate now
        if(nmpms>nmpmsRC && MaterialBase::extrapolateRigidBCs)
        {       nextMPMTask=(MPMTask *)new ExtrapolateRigidBCsTask("Rigid BCs by
Extrapolation");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
```

```
                lastMPMTask=nextMPMTask;
        }

        // if rigid contact, add task to change their velocity
        if(nmpmsRC > nmpmsRB)
        {       nextMPMTask=(MPMTask *)new SetRigidContactVelTask("Set Rigid
Contact Velocities");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                lastMPMTask=nextMPMTask;
        }

        nextMPMTask=(MPMTask *)new MassAndMomentumTask("Extrapolate Mass and
Momentum");
        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
        lastMPMTask=nextMPMTask;

        // if rigid BCs not using extrapolations, project to grid BCs
        if(nmpms>nmpmsRC && !MaterialBase::extrapolateRigidBCs)
        {       nextMPMTask=(MPMTask *)new ProjectRigidBCsTask("Rigid BCs by
Projection");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                lastMPMTask=nextMPMTask;
        }

        nextMPMTask=(MPMTask *)new PostExtrapolationTask("Post Extrapolation
Tasks");
        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
        lastMPMTask=nextMPMTask;

        // UPDATE STRAINS FIRST AND USAVG
    if(mpmApproach==USF_METHOD || mpmApproach==USAVG_METHOD)
        {       nextMPMTask=(MPMTask *)new UpdateStrainsFirstTask("Update
Strains First");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                lastMPMTask=nextMPMTask;
                USFTask = (UpdateStrainsFirstTask *)nextMPMTask;
        }

#ifdef RESTART_OPTION
    if(fabs(restartScaling)>1.e-6)
    {   // disallow greater than 1
        if(restartScaling>1.)
            restartScaling = 0.5;
        else if(restartScaling<-1.)
            restartScaling = -0.5;
        MPMTask *priorTask = firstMPMTask;
        while(priorTask!=NULL)
        {   if(priorTask->BlockRestartOption())
            {   // can't use restart with this tasks
                char errMsg[200];
                strcpy(errMsg,"Restart option not allowed by the '");
                strcat(errMsg,priorTask->GetTaskName());
                strcat(errMsg,"' task");
                throw CommonException(errMsg,"NairnMPM::CreateTasks()");
```

```
            }
            priorTask = (MPMTask *)(priorTask->GetNextTask());
        }

        // restart is allowed, show that it is active
        if(theTasks==NULL && transportTasks==NULL) PrintSection("SCHEDULED
CUSTOM TASKS");
        cout << "Restart time step if nodal travel >" << restartCFL << "*(cell
length)" << endl;
        cout << "     Rescale time step by factor " << fabs(restartScaling) << "
when restarted" << endl;
        if(restartScaling<0.)
            cout << "     Display notice whenever restarted" << endl;
        cout << endl;
    }
#endif

        // EXTRAPOLATE FORCES
        nextMPMTask=(MPMTask *)new GridForcesTask("Extrapolate Grid Forces");
        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
        lastMPMTask=nextMPMTask;

        nextMPMTask=(MPMTask *)new PostForcesTask("Post Force Extrapolation
Tasks");
        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
        lastMPMTask=nextMPMTask;

        // UPDATE MOMENTA
        nextMPMTask=(MPMTask *)new UpdateMomentaTask("Update Momenta");
        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
        lastMPMTask=nextMPMTask;

        // XPIC extrapolations is needed only if order 2 or higher
        if (bodyFrc.XPICVectors() > 1)
        {       XPICMechanicsTask = new XPICExtrapolationTask("XPIC
Extrapolations");
        }

        // XPIC extrapolations for transport is needed if activated
        if(TransportTask::hasXPICOption)
        {       nextMPMTask=(MPMTask *)new XPICExtrapolationTaskTO("XPIC
Transport Extrapolations");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                lastMPMTask=nextMPMTask;
                XPICTransportTask=(XPICExtrapolationTaskTO *)nextMPMTask;
        }

        // UPDATE PARTICLES
        nextMPMTask=(MPMTask *)new UpdateParticlesTask("Update Particles");
        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
        lastMPMTask=nextMPMTask;

#ifdef MOVECRACKS_EARLY
        // MOVE CRACKS
```

```
        if(firstCrack!=NULL)
        {       nextMPMTask=(MPMTask *)new MoveCracksTask("Move Cracks");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                lastMPMTask=nextMPMTask;
        }
#endif

        // UPDATE STRAINS LAST AND USAVG
        // Energy calcs suggest the contact method, which re-extrapolates,
should always be used
        if(mpmApproach==USL_METHOD || mpmApproach==USAVG_METHOD)
        {       if(fmobj->skipPostExtrapolation)
                {       nextMPMTask=(MPMTask *)new UpdateStrainsLastTask("Update
Strains Last");
                        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                        lastMPMTask=nextMPMTask;
                }
                else
                {       nextMPMTask=(MPMTask *)new
UpdateStrainsLastContactTask("Update Strains Last with Extrapolation");
                        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                        lastMPMTask=nextMPMTask;
                }
        }

        // CUSTOM TASKS (including J Integral and crack propagation)
        if(theTasks!=NULL)
    {   nextMPMTask=(MPMTask *)new RunCustomTasksTask("Run Custom Tasks");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                lastMPMTask=nextMPMTask;
        }

#ifndef MOVECRACKS_EARLY
        // MOVE CRACKS
        if(firstCrack!=NULL)
        {       nextMPMTask=(MPMTask *)new MoveCracksTask("Move Cracks");
                lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
                lastMPMTask=nextMPMTask;
        }
#endif

        // RESET ELEMEMTS
        nextMPMTask=(MPMTask *)new ResetElementsTask("Reset Elements");
        lastMPMTask->SetNextTask((CommonTask *)nextMPMTask);
        //lastMPMTask=nextMPMTask;

}

#pragma mark PREPARATION SUPPORT METHODS

// Reorder Particles in order
//      NonRigid, Rigid Blok, Rigid Contact, Rigid BCs
// Also Initialize RigidBlck particles
void NairnMPM::ReorderParticles(int firstRigidPt,int hasRigidContactParticles)
```

```
{
	// Step 1: reorder NonRigid followed by (Rigid Block, Rigid Contact and
Rigid BCs intermixed)
	int p;
	if(firstRigidPt<nmpmsNR && firstRigidPt>=0)
	{	p = firstRigidPt;

		// loop until reach end of non-rigid materials
		while(p<nmpmsNR)
		{	int matid = mpm[p]->MatID();
			if(theMaterials[matid]->IsRigid())
			{	// if any rigid particle, switch with particle
at nmpmsNR-1
		SwapMaterialPoints(p,nmpmsNR-1);
				nmpmsNR--;
// now 0-based pointer of previous NR particle

				// fix particle based boundary conditions
				ReorderPtBCs(firstLoadedPt,p,nmpmsNR);
				ReorderPtBCs(firstTractionPt,p,nmpmsNR);
				ReorderPtBCs(firstFluxPt,p,nmpmsNR);
				ReorderPtBCs(firstHeatFluxPt,p,nmpmsNR);
				ReorderPtBCs(firstDamagedPt,p,nmpmsNR);

				// back up to new last nonrigid
				while(nmpmsNR>=0)
				{	matid = mpm[nmpmsNR-1]->MatID();
					if(!theMaterials[matid]->IsRigid())
break;
					nmpmsNR--;
				}
			}

			// next particle
			p++;
		}
	}

	// Step 2: reorder rigid particles as (Rigid Block&Rigid Contact
intermixed, Rigid BCs)
	nmpmsRC = nmpmsNR;
	nmpmsRB = nmpmsNR;
	if(hasRigidContactParticles!=NO_RIGID_MM)
	{	// segment rigid particles if have any for contact or for block

		// back up to new last rigid contact or block particle
		nmpmsRC = nmpms;
		while(nmpmsRC > nmpmsNR)
		{	int matid = mpm[nmpmsRC-1]->MatID();
			if(!theMaterials[matid]->IsRigidBC()) break;
			nmpmsRC--;
		}

		// loop until reach end of rigid contact or block particles
```

```
                p = nmpmsNR;
                while(p < nmpmsRC)
                {       int matid=mpm[p]->MatID();
                        if(theMaterials[matid]->IsRigidBC())
                        {       // if rigid BC particle, switch with rigid
contact particle at nmpmsRC-1
                SwapMaterialPoints(p,nmpmsRC-1);
                                nmpmsRC--;

                                // fix particle based boundary conditions
                                ReorderPtBCs(firstLoadedPt,p,nmpmsRC);

                                // back up to new last nonrigid
                                while(nmpmsRC > nmpmsNR)
                                {       matid=mpm[nmpmsRC-1]->MatID();
                                        if(!theMaterials[matid]->IsRigidBC())
break;
                                        nmpmsRC--;
                                }
                        }

                        // next particle
                        p++;
                }

                // Step 3: finally reorder rigid particles as (Rigid Block,
Rigid Contact, Rigid BCs)
                nmpmsRB = nmpmsNR;
#ifdef MODEL_RIGID_BODIES
                if(hasRigidContactParticles&RIGID_BLOCK_MM &&
hasRigidContactParticles&RIGID_CONTACT_MM)
                {       // If has RB and RC, need to reorder that block now
                        // back up to new last rigid block particle
                        nmpmsRB = nmpmsRC;
                        while(nmpmsRB > nmpmsNR)
                        {       int matid = mpm[nmpmsRB-1]->MatID();
                                if(theMaterials[matid]->IsRigidBlock()) break;
                                nmpmsRB--;
                        }

                        // loop until reach end of rigid blocks
                        p = nmpmsNR;
                        while(p < nmpmsRB)
                        {       int matid=mpm[p]->MatID();
                                if(!theMaterials[matid]->IsRigidBlock())
                                {       // if not rigid block, switch with rigid
block particle at nmpmsRB-1
                    SwapMaterialPoints(p,nmpmsRB-1);
                                        nmpmsRB--;

                                        // fix particle based boundary
conditions
                                        ReorderPtBCs(firstLoadedPt,p,nmpmsRB);
```

```
                                        // back up to new last nonrigid
                                        while(nmpmsRB > nmpmsNR)
                                        {       matid=mpm[nmpmsRB-1]->MatID();

if(theMaterials[matid]->IsRigidBlock()) break;
                                                nmpmsRB--;
                                        }
                                }

                                // next particle
                                p++;
                        }
                }
                else if(hasRigidContactParticles&RIGID_BLOCK_MM)
                {       // if rigid block but no contact already done
                        nmpmsRB=nmpmsRC;
                }

                // for Rigid block materials: 1 Get initial values, 2. Find
block properties, 3. Apply to particles
                if(nmpmsRB>nmpmsNR)
                {       for(p=nmpmsNR;p<nmpmsRB;p++)
                        {       RigidBlock *rbmat = (RigidBlock
*)theMaterials[mpm[p]->MatID()];

rbmat->InitialBlockValues(mpm[p]->mp,&mpm[p]->vel,&mpm[p]->pos);
                        }

                        // finish material initiationization
                        RigidBlock::FinishInitialValues();

                        for(p=nmpmsNR;p<nmpmsRB;p++)
                        {       RigidBlock *rbmat = (RigidBlock
*)theMaterials[mpm[p]->MatID()];

rbmat->InitialParticleValues(&mpm[p]->vel,&mpm[p]->pos);
                        }
                }
#endif
        }
}

// Swap material points p1 and p2 and swap material num too
void NairnMPM::SwapMaterialPoints(int p1,int p2) const
{
    MPMBase *temp = mpm[p1];
    mpm[p1] = mpm[p2];
    mpm[p1]->SetNum(p1);
    mpm[p2] = temp;
    mpm[p2]->SetNum(p2);
}

// calculate time step
void NairnMPM::CFLTimeStep()
```

```
{
	// verify time step and make smaller if needed
	// FractCellTime and TransFractCellTime are CFL factors for mechanics
and transport
#ifndef TRANSPORT_ONLY
	double timeStepMin = FractCellTime*timeStepMinMechanics;
#else
	double timeStepMin = 1.e30;
#endif
	// Transport property time steps
	TransportTask *nextTransport=transportTasks;
	while(nextTransport!=NULL)
	{	double tst = TransFractCellTime*nextTransport->GetTimeStep();
		if(tst < timeStepMin) timeStepMin = tst;
		nextTransport = nextTransport->GetNextTransportTask();
	}

	// use timeStepMin, unless specified time step is smaller
	if(timeStepMin<timestep) timestep = timeStepMin;
	if(fmobj->mpmApproach==USAVG_METHOD)
	{	// fraction USF = 0.5, coded with variable in case want to
change
		strainTimestepFirst = fractionUSF*timestep;
		strainTimestepLast = timestep - strainTimestepFirst;
	}
	else
	{	strainTimestepFirst = timestep;
		strainTimestepLast = timestep;
	}

	// propagation time step (no less than timestep)
	if(propTime<timestep) propTime=timestep;
}

// When NR particle p2 moves to p1, reset any point-based BCs that use that
point
void NairnMPM::ReorderPtBCs(MatPtLoadBC *firstBC,int p1,int p2)
{
	while(firstBC!=NULL)
		firstBC = firstBC->ReorderPtNum(p1,p2);
}

// Called just before time steps start
// Can insert code here to black runs with invalid options
// throws CommonException()
void NairnMPM::ValidateOptions(void)
{
	// Disable non-structured or variable element grid sizes in NairnMPM -
need OSParticulas for those options
	if(!mpmgrid.IsStructuredEqualElementsGrid())
	{	throw CommonException("Non-structured grids or grids with
unequal element sizes are currently disabled in NairnMPM because they are not
verified for all features.",
```

```
"NairnMPM::ValidateOptions");
        }

        // GIMP and CPDI and SPLINE require regular
    //  and qCPDI not allowed in 3D
        if(ElementBase::useGimp != POINT_GIMP)
    {   // Not using classic MPM shape functions
                if(!mpmgrid.IsStructuredEqualElementsGrid())
                {       switch(ElementBase::useGimp)
                        {       case UNIFORM_GIMP:
                                {       throw CommonException("GIMP needs a
generated structured grid with equally sized elements",

"NairnMPM::ValidateOptions");
                                }
                                case BSPLINE_GIMP:
                                {       throw CommonException("B2GIMP needs a
generated structured grid with equally sized elements",

 "NairnMPM::ValidateOptions");
                                }
                                case BSPLINE:
                                {       throw CommonException("B2SPLINE needs a
generated structured grid with equally sized elements",

 "NairnMPM::ValidateOptions");
                                }
                                case BSPLINE_CPDI:
                                {       throw CommonException("B2CPDI needs a
generated structured grid with equally sized elements",

 "NairnMPM::ValidateOptions");
                                }
                                default:
                                        break;
                        }
                }

        if(ElementBase::useGimp == QUADRATIC_CPDI)
        {   if(IsThreeD())
                throw CommonException("3D does not allow qCPDI shape functions;
use lCPDI instead","NairnMPM::ValidateOptions");
        }

                else if(ElementBase::useGimp == BSPLINE ||
ElementBase::useGimp==BSPLINE_GIMP
                                        || ElementBase::useGimp==BSPLINE_CPDI)
                {
                        if(firstTractionPt!=NULL && exactTractions)
                        {       throw CommonException("Exact tractions not yet
supported for spline shape functions.","NairnMPM::ValidateOptions");
                        }
                }
        }
```

```
        // Only allowed if 2D and ExactTractions
        if(exactTractions && fmobj->IsThreeD())
                throw CommonException("Exact tractions not supported yet in 3D
simulations.","NairnMPM::ValidateOptions");

        // limits here are because element.filled is 32 bit int. Making it a
long would allow 64 particle per element
    // 3D requires orthogonal grid and 1,8, or 27 particles per element
    // 2D requires 1,4, 9, 16, or 25 particles per element
        if(IsThreeD())
        {       if(mpmgrid.GetCartesian()!=CUBIC_GRID &&
mpmgrid.GetCartesian()!=ORTHOGONAL_GRID &&
mpmgrid.GetCartesian()!=VARIABLE_ORTHOGONAL_GRID)
                        throw CommonException("3D calculations require an
orthogonal grid","NairnMPM::ValidateOptions");
                if(ptsPerElement>27)
                        throw CommonException("3D analysis requires 1 or 8 or 27
particles per cell","NairnMPM::ValidateOptions");
        }
        else
        {       if(ptsPerElement>25)
                        throw CommonException("2D analysis requires 1, 4, 9, 16,
or 25 particles per cell","NairnMPM::ValidateOptions");
        }

    // Axisymmetric requirements and adjustments
    if(IsAxisymmetric())
    {   if(ElementBase::useGimp == POINT_GIMP)
        {   // require cartesian grid
            if(mpmgrid.GetCartesian()<=0)
            {   throw CommonException("Axisymmetric with Classic MPM needs an
orthogonal grid","NairnMPM::ValidateOptions");
            }
        }
        else if(ElementBase::useGimp == UNIFORM_GIMP)
                {       ElementBase::useGimp = UNIFORM_GIMP_AS;
                        // verify r=0,cell,... on grid lines if grid minimum <
2*cell
                        if(!mpmgrid.ValidateASForGIMP())
                        {       throw CommonException("GIMP needs r=0 on grid
line (if mesh minimum<2*cell size)",

"NairnMPM::ValidateOptions");
                        }
                }
        else if(ElementBase::useGimp == LINEAR_CPDI)
                {       ElementBase::useGimp = LINEAR_CPDI_AS;
        }
                else if(ElementBase::useGimp == BSPLINE_GIMP)
                {   ElementBase::useGimp = BSPLINE_GIMP_AS;
                        // verify r=0,cell,... on grid lines if grid minimum <
2*cell
                        if(!mpmgrid.ValidateASForGIMP())
```

```
                        {       throw CommonException("B2GIMP needs r=0 on grid
line (if mesh minimum<2*cell size)",

"NairnMPM::ValidateOptions");
                        }
                }
                else if(ElementBase::useGimp == QUADRATIC_CPDI)
                {   throw CommonException("Axisymmetric does not allow qCPDI
shape functions","NairnMPM::ValidateOptions");
                }
    }

        // The input commands have options to enter arbitray elements and nodes
for an unstructured grid, but code
        // does nut current work with that style so it is isabled here. To use
unstructure grid need at least:
        // 1. FindElementFromPoint() would need to search all elements
        // 2. Parallel patching would need rewriting
        if(!mpmgrid.IsStructuredGrid())
                throw CommonException("This code currently requies use of a
generated structured grid","NairnMPM::ValidateOptions");

        // check each material type (but only if it is used it at least one
material point)
        int i;
        for(i=0;i<nmat;i++)
        {       if(theMaterials[i]->GetField()>=0)
                        theMaterials[i]->ValidateForUse(np);
        }
}

#pragma mark ACCESSORS

// verify analysis type
bool NairnMPM::ValidAnalysisType(void)
{
        // change defaults for 3D - this will be called in <Header> which will
be before any user changes
        if(np==THREED_MPM)
        {       ptsPerElement=8;                                // number of
points per element
                ptsPerSide=2;
                nfree=3;
                maxShapeNodes=28;                               // increase if
CPDI is used
        }

        return np>BEGIN_MPM_TYPES && np<END_MPM_TYPES;
}

// if crack develops tractionlaw, call here to turn it on
void NairnMPM::SetHasTractionCracks(bool setting) { hasTractionCracks=setting; }

// Get Courant-Friedrichs-Levy condition factor for convergence
```

```
double NairnMPM::GetCFLCondition(void) { return FractCellTime; }
double NairnMPM::GetTransCFLCondition(void) { return TransFractCellTime; }
double *NairnMPM::GetCFLPtr(void) { return &FractCellTime; }
double *NairnMPM::GetTransCFLPtr(void) { return &TransFractCellTime; }

// to check on diffusion or poroelasticity
bool NairnMPM::HasDiffusion(void) { return
DiffusionTask::active==MOISTURE_DIFFUSION; }
#ifdef POROELASTICITY
bool NairnMPM::HasPoroelasticity(void) { return
DiffusionTask::active==POROELASTICITY_DIFFUSION; }
#endif
bool NairnMPM::HasFluidTransport(void) { return
DiffusionTask::active!=NO_DIFFUSION; }

// Get Courant-Friedrichs-Levy condition factor for convergence for propagation
calculations
double NairnMPM::GetPropagationCFLCondition(void) { return PropFractCellTime; }

#pragma mark ARCHIVER PASS THROUGHS

// archiver pass throught while setting up analysis
void NairnMPM::ArchiveNodalPoints(int np) { archiver->ArchiveNodalPoints(np); }
void NairnMPM::ArchiveElements(int np) { archiver->ArchiveElements(np); }
void NairnMPM::SetInputDirPath(const char *xmlFIle,bool useWorkingDir)
{       archiver->SetInputDirPath(xmlFIle,useWorkingDir);
}
```