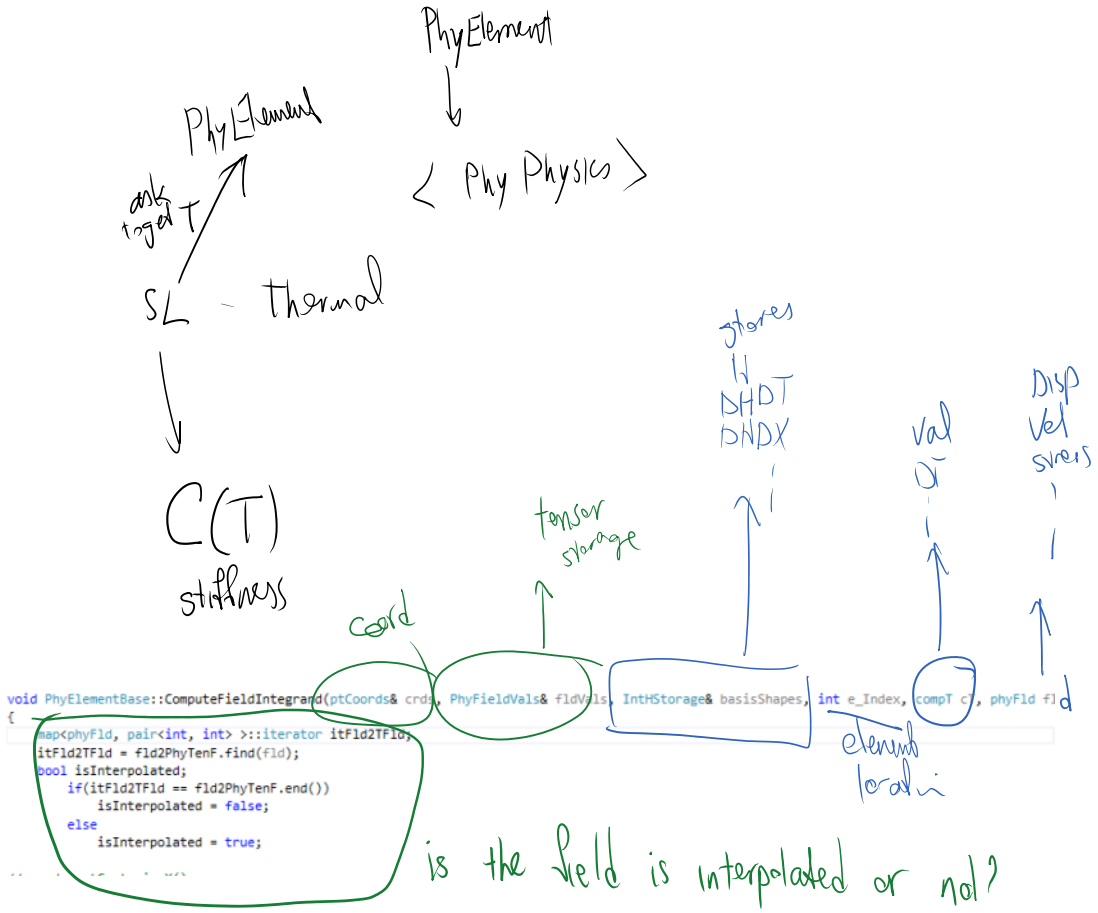
2018/04/25

Wednesday, April 25, 2018 11:36 AM

Computation of any type of tensor field needed in WR, error calculation etc.:



Side note:

Idea of function pointer:

```
extern PhyTensorFieldcompPtr PhyTFComputeIntegrand[NUM_COMPT];
```

Indexed by computation type

cVal
cDT
cDX

this name is

a function of class

a function of class

```
typedef void (PhyTensorField::*PhyTensorFieldcompPtr)
(ptCoords& , PhyFieldVals& , IntHStorage& , int, vsT, rotT);
```

Example:
A list of values of this array of function pointer must be specified:
For example for spatial derivative it's this function:

```
PhyTFComputeIntegrand[ctDX] = &PhyTensorField::ComputeDHDXIntegrand;
```

Other computation types:

```
void PhyElementBase::ComputeFieldIntegrand(ptCoords& crds
{
    map<phyFld, pair<int, int> >::iterator itFld2TFld;
    itFld2TFld = fld2PhyTenF.find(fld);
    bool isInterpolated;
        if(itFld2TFld == fld2PhyTenF.end())
            isInterpolated = false;
        else
            isInterpolated = true;

//  crds.setCartesianX();

    if ((int)cT < (int)ctSource)
    {
        if (isInterpolated)     // the field is interpola
        {
            int phyIndex = itFld2TFld->second.first;
            int tenFldIndex = itFld2TFld->second.second;
```

Interpolated things

.....

Non interpolated things

field name (U, V, S, ...)

(physics[phyIndex]->*PhyCompute[cT][fld])(crds, fldVals, basisShapes, e_Index, cVH, rT);

Computation type

```
extern PhycompPtr    PhyCompute[NUM_COMPT][NUM_PHYFLD];
```

```
typedef void (PhyPhysics::*PhycompPtr)(ptCoords& , PhyFieldVals& ,
    IntHStorage& , int, vsT, rotT);
```

For stress Value

```
PhyCompute[ctVal][pfStrsL] = &PhyPhysics::computeField_ctVal_pfStrsL;
```

The parent class (PhyPhysics has the declaration and empty implementation) and a derived class must provide the actual implementation
In PhyPhysics.h:

virtual void computeField_ctVal_pfStrsL(ptCoords& crds, PhyFieldVals& fldVals, IntHStorage& basisShapes, int e_Index, vsT cVH, rotT rT) {};

SLPhysics.cpp actual implementation
void SLPhysics::computeField_ctVal_pfStrsL(ptCoords& crds, PhyFieldVals& fldVals, IntHStorage& basisShapes, int e_Index, vsT cVH, rotT rT)
{
....

Now we can overview an integration routine:
Before that, there is an important enumeration:

Initial guess — Assembly

typedef enum { intSInitialGuess, intSPreAssembly, intSAssembly, intSPostSol, intSPostProcess, intSFreePostProcess, intFreeFCFPrint } intStage;

different post processing

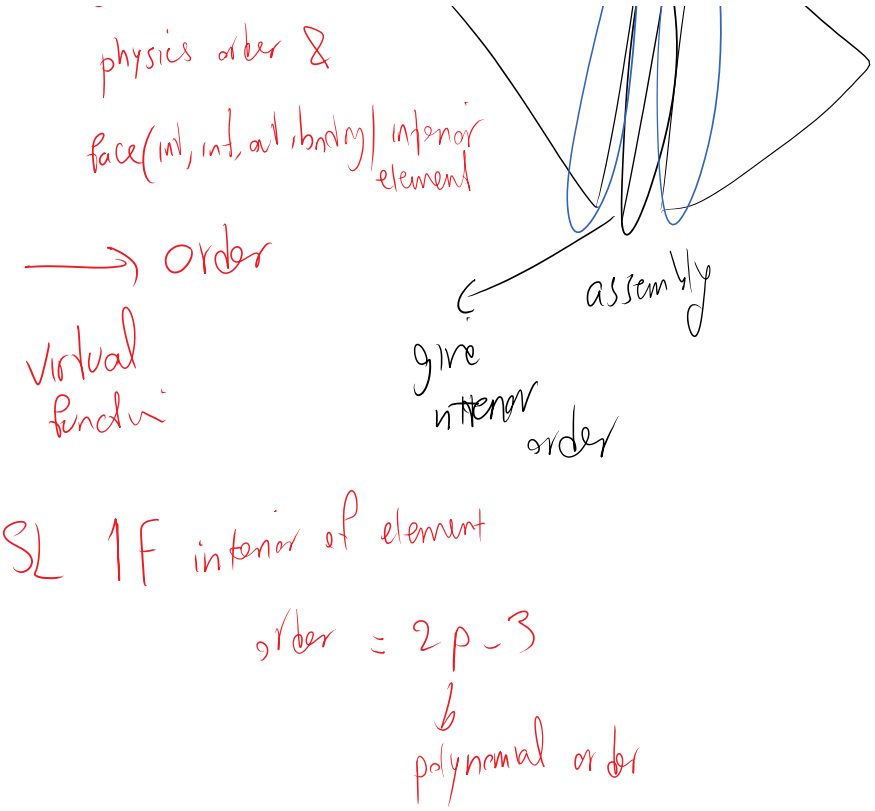
error analysis for RUN TIME adaptivity

Integration function:

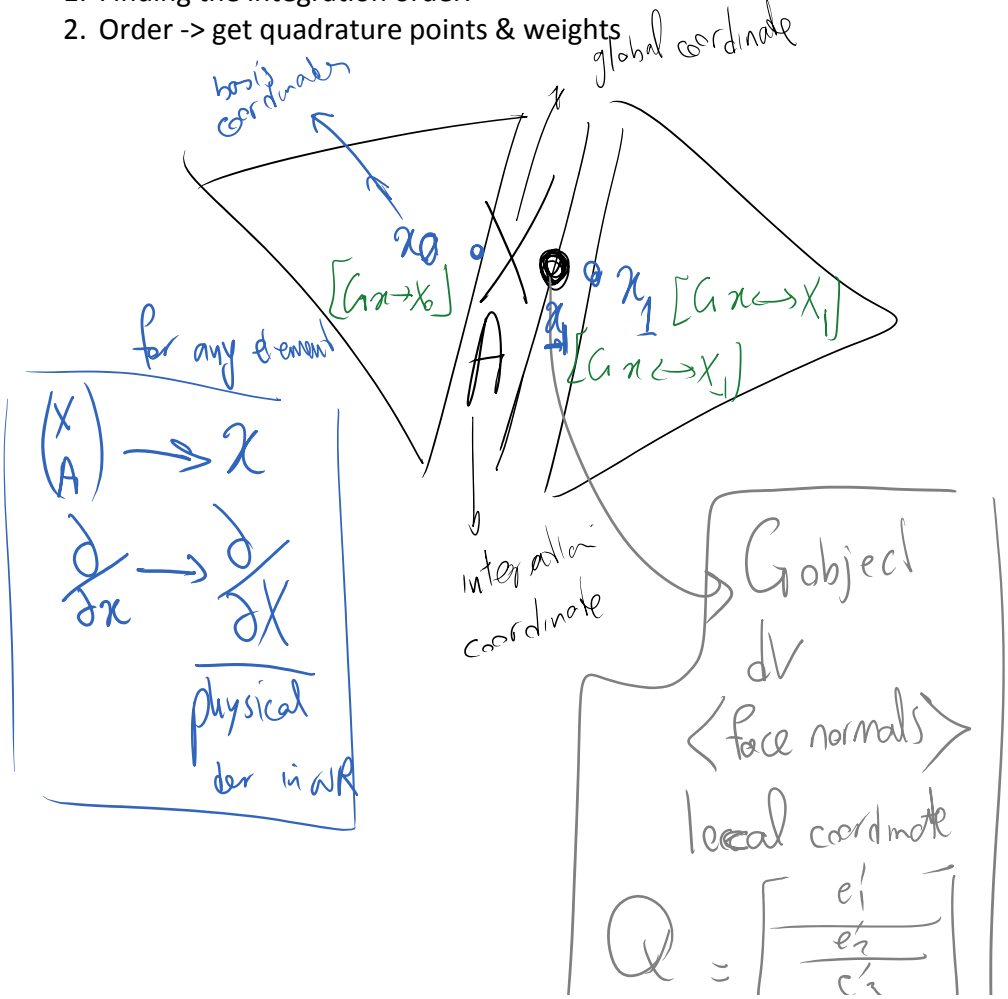
Integration of anything on an integration cell:

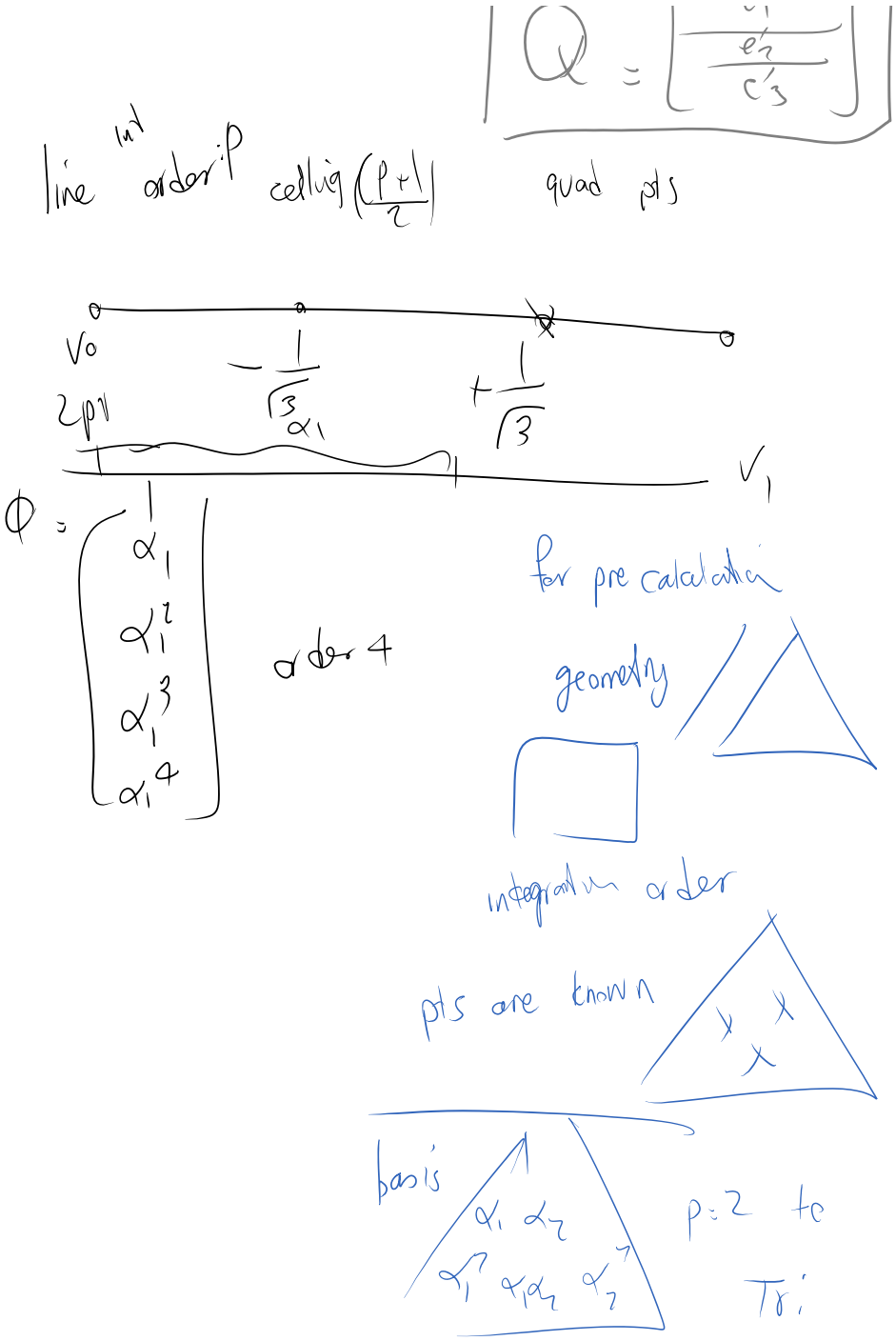
1.

going from physics order &



1. Finding the integration order.
2. Order -> get quadrature points & weights





---

3. (we have all the quad points) ->

   Looping over the quad points
   a. Calculate all the shapes needed H, DHDx, DrDT, ...

   IntHStorage

b. Tensor field storage

intrStorage

b. Tensor field storage
   PhyFieldVals fldVals

4. Now that all the tensor fields are calculated in fldVals we can go ahead and evaluate / part the integration stage computations.

Overview of the code on what exactly is being done here

The part that we calculate order, quad points, and basis storage:

```
bool PhyIntCellBase::setAfterElementPolyOrderSetBeginingEachStage(intStage intS)
{
    bool b_time = ((intS == intSPreAssembly) || (intS == intSAssembly));
    if (b_time == true)
        intCellTimePtr->setStartgTimeIncremental();
    // don't want to recompute quad points if integration order does not change
    intNumQuadsTotal = intNumQuads.getBaseScalarValue();
    if (intNumQuadsTotal <= 0)
        intOrder.clear();
    GQuadIntOrder   intOrderPrevStep = intOrder;
    setIntegrationOrder(intS);

    // if (order NOT set previously OR the orders are not changed, no need to recompute quad points and shapes ....
    bool reComputeQuadPtVals = ((intOrderPrevStep.getSize() == 0) || (intOrderPrevStep != intOrder));
    if (reComputeQuadPtVals)
    {
        // setting all quad points needed for this int cell
        bool issueErrorIfPtsNotPrecomputed = true; // this can be reversed to false so no matter what the order of the element,
        bool useAlphaOrBeta0Order4MissingBeta = true; // if beta orders not provided, alpha orders will be used for them
        getQuadPointsWeightFunctions(intOrder, intCrds, intWeights, intT1, intNumQuads, issueErrorIfPtsNotPrecomputed, useAlphaO
```

```
    if ((intS != intSPreAssembly) &&
        (reComputeQuadPtVals || (intCellHStore.size() != intNumQuadsTotal)))
        PreComputeDistinctBaseShapes();
```

This place precalculates and sets ptCoords and Shapes

Now that ptCoords, and Shape storages are precomputed we can go to integration routine:

Actual integration routine:

Actual integration routine:

integration stage

```
bool PhyIntCellBase::IntegrateIntegrationCell(intStage intS,
    gTime *clockPreComputeFields, gTime *clockEvaluate)
{
    int facetT = (int)cellT.ct, intST = (int)intS;
```

Interior, boundary, inflow, outflow

```
    for (int quadPN = 0; quadPN < intNumQuadsTotal; ++quadPN)
    {
        //DBOUT
//      cout<<"quadPN "<<quadPN <<" / "<< intNumQuadsTotal <<endl;
        ptCoords *crdPtr = &intCrds(quadPN);
```

← get the precomputed quad pt

....

Storage for all the tensors

```
        // interior integration
        PhyFieldVals fldVals(this, crdPtr); // storage for shapes and values
        // Computing values before going into integrands
        bool succesfulPreCompute = (fldVals.*ComputeIntegrand[intST][cellT.ct])
            (*crdPtr, intCellHStore[quadPN]);
```

precomputed shape storage

Evaluation stage:

…

Again we use function pointers

```
if (bInteriorInt == true)
{
    pe = interiorIntBase.eh.getPhyE();
    if ((pe->*InteriorIntegrands[intST])(factor, eInteriorI, *crdPtr, this, quadPN, fldVals) == false)
        exit(0);//parallel edit
        //return false;
}
for (int facet = 0; facet < numFacetInActivePatch; ++facet)
{
    pe = facetIntBase[facet].eh.getPhyE();
    if ((pe->*FacetIntegrands[intST][facetT])(factor, facet, *crdPtr, this, quadPN, fldVals) == false)
        exit(0);//parallel edit
        //return false;
}
```

Interior

here we have all tensor fields available & process the integration stage

face type for face

face

Example

```
//  intSAssembly
bool PhyElementBase::IntegrandInterior_intSAssembly(double factor, int e_Index, ptCoords& crds, PhyIntCellBase*
{
    for (int phy = 0; phy < num_physics; ++phy)
    {
        PD4 ( db << "INTERIOR_ASSMBLY_phy" << phy << " / " << num_physics << '\n';)
        if (physics[phy]->IntegrandInterior_intSAssembly(factor, e_Index, crds, pic, quadPN, fldVals) == false)
            return false;
    }
```

```
//  intSAssembly
bool PhyElementBase::IntegrandInterior_intSAssembly(double factor, int e_Index, ptCoords& crds, PhyIntCellBase*
{
    for (int phy = 0; phy < num_physics; ++phy)
    {
        PD4 ( db << "INTERIOR_ASSMBLY_phy" << phy << " / " << num_physics << '\n';)
        if (physics[phy]->IntegrandInterior_intSAssembly(factor, e_Index, crds, pic, quadPN, fldVals) == false)
            return false;
    }
    return true;
}
```

Examples of function pointers for "processing of the interior OR faces attached to PIC:

extern PhyInteriorIntPtr InteriorIntegrands[NumIntStage];
extern PhyFacetIntPtr FacetIntegrands[NumIntStage][NumPhyCell_t];

```
typedef bool (PhyElementBase::*PhyInteriorIntPtr)(double, int, ptCoords& , PhyIntCellBase*, int, PhyFieldVals&);
typedef bool (PhyElementBase::*PhyFacetIntPtr)(double, int, ptCoords& , PhyIntCellBase*, int, PhyFieldVals&);
```

…
The definition of these function pointers
….

```
    //intSAssembly
    InteriorIntegrands[intSAssembly] = &PhyElementBase::IntegrandInterior_intSAssembly;

    FacetIntegrands[intSAssembly][noneCellT] = NULL;
    FacetIntegrands[intSAssembly][inflowT] = &PhyElementBase::IntegrandFacet_intSAssembly_inflowT;
    FacetIntegrands[intSAssembly][outflowT] = &PhyElementBase::IntegrandFacet_intSAssembly_outflowT;
    FacetIntegrands[intSAssembly][interiorT] = &PhyElementBase::IntegrandFacet_intSAssembly_interiorT;
    FacetIntegrands[intSAssembly][boundaryT] = &PhyElementBase::IntegrandFacet_intSAssembly_boundaryT;
```

….

```
bool PhyElementBase::IntegrandFacet_intSAssembly_interiorT(double factor, int e_Index, ptCoords& crds, PhyIntCellBase* pic, int quadPN, PhyFieldVals& fldVa
{
    for (int phy = 0; phy < num_physics; ++phy)
    {
        PD4 ( db << "INTERIOR_FACET_ASSMBLY_phy" << phy << " / " << num_physics << '\n';)
        if (physics[phy]->IntegrandFacet_intSAssembly_interiorT(factor, e_Index, crds, pic, quadPN, fldVals) == false)
            return false;
    }
    return true;
}
```

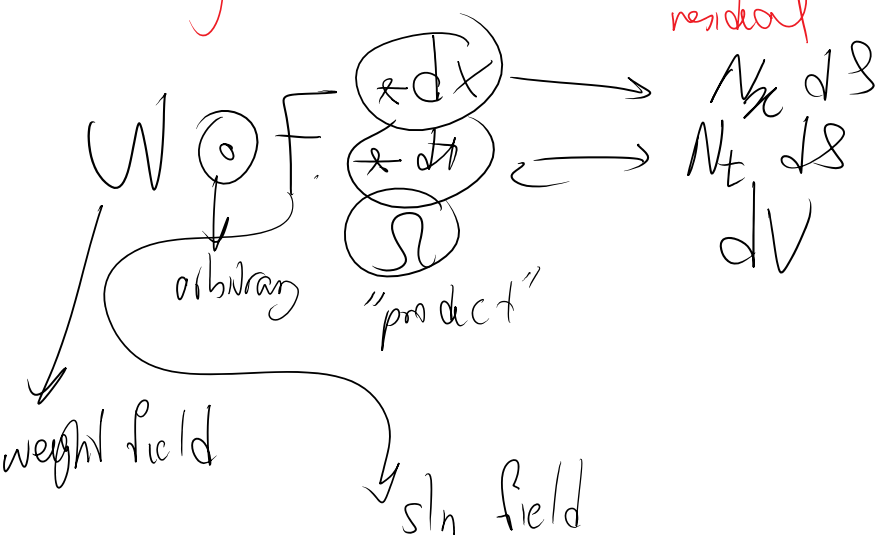
….

```
bool PhyPhysics::IntegrandFacet_intSAssembly_interiorT(double factor, int e_Index, ptCoords& crds, PhyIntCellBase* pic, int quadPN, PhyFieldVals& fldVals)
{
    double factorTerm;  // The factor in front of the integration term
    vsT cVH = patch->patchFlags.vs_intSAssemblyFldKRUpdate;
    FEM_Terms* femTerms = getFEM_AssemblyTerms();
    AssemblyIntegrandTerm* siTerm;

#if PRINT_DEBUG4
    VECTOR XCoord;
    crds.setCartesianX();
    crds.XC.getXV(XCoord);
    db << "ASMBLY_INT_FACE_quadPN\t" << quadPN << "\tcrd\t" << XCoord << "\tdesc\t" << pic->descFacet[e_Index] << "\tdecInterior\t" << pic->descInterior <<
    db << "Number_interior_sdx terms\t" << femTerms->num_weightField_Assembly_Facet_sdx_interiorT<< '\n';
#endif

    //*dx part
    for (int term = 0; term < femTerms->num_weightField_Assembly_Facet_sdx_interiorT; ++term)
```

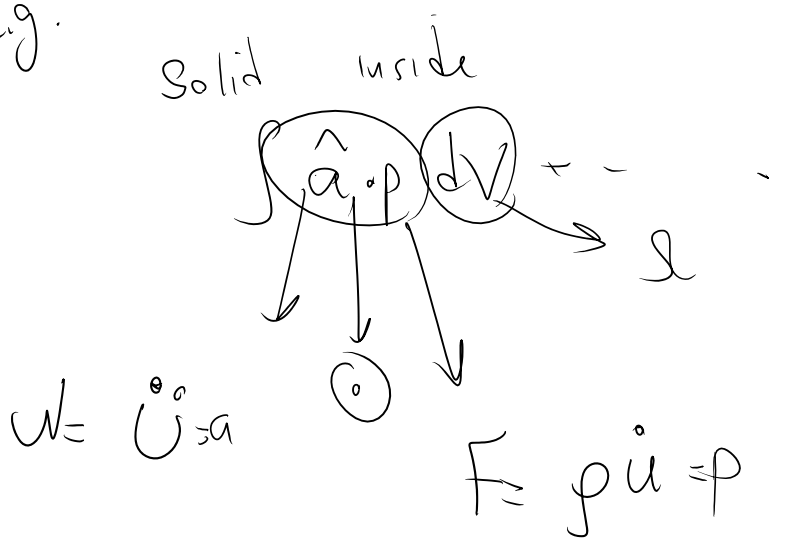
```
    siTerm = &femTerms->weightField_Assembly_Facet_sdx_interiorT[term];
    if (siTerm ->hasConstantFactor == true)
        factorTerm = siTerm->termConstantFactor;
    else
        factorTerm = siTerm->termConstantFactor * IntegrandFacet_intSAssembly_sdx_interiorT_ComponentFacor(term, e_Index, crds, pic, quadPN, fldVals);

    PD4 ( db << "term_INT_sdx" << term << '\n' << siTerm << "\tconstFact\t" << siTerm->termConstantFactor << "\tfactorTerm\t" << factorTerm << '\n';)
    #pragma omp critical(PhyPhysicsCE3)
    fldVals.updateAssemblyStiffnessReidual_Facet_sdx_interiorTboundaryT(pic, e_Index, *siTerm, factorTerm, factor, cVH);
}
fCompMode fcMode = fldVals.pfv_getFaceComputationMode();
//*dt part
PD4 (db << "Number_interior_sdt terms\t" << femTerms->num_weightField_Assembly_Facet_sdt_interiorT<< '\n';)
    PD4(db << "fcMode\t" << fcMode << '\n';)
    if (fcMode == dcm_only_sdxNZ)
    return true;                //*dt = 0 for vertical facet
for (int term = 0; term < femTerms->num_weightField_Assembly_Facet_sdt_interiorT; ++term)
{
```

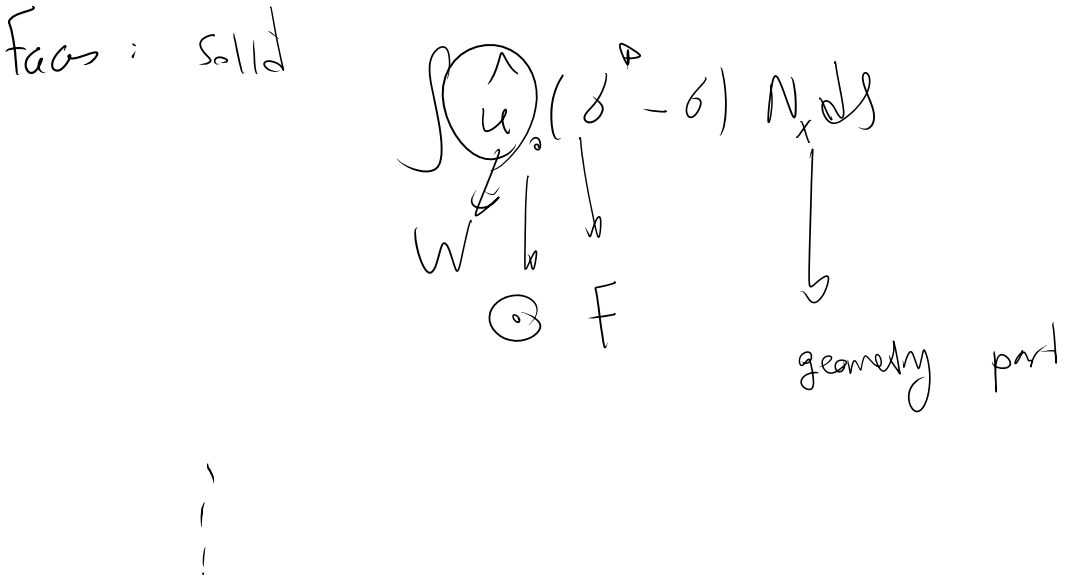
```
        siTerm = &femTerms->weightField_Assembly_Facet_sdt_interiorT[term];
        if (siTerm ->hasConstantFactor == true)
            factorTerm = siTerm->termConstantFactor;
        else
            factorTerm = siTerm->termConstantFactor * IntegrandFacet_intSAssembly_sdt_interiorT_ComponentFacor(term, e_Index, crds, pic, quadPN, fldVals);
        PD4 ( db << "term_INT_sdt" << term << '\n' << siTerm << "\tconstFact\t" << siTerm->termConstantFactor << "\tfactorTerm\t" << factorTerm << '\n';)
        #pragma omp critical(PhyPhysicsCE3)
        fldVals.updateAssemblyStiffnessReidual_Facet_sdt_interiorTboundaryT(pic, e_Index, *siTerm, factorTerm, factor, cVH);
    }
    return true;
}
```

for example for Assembly processing involves handling contributions to R residual & K stiffness for time terms



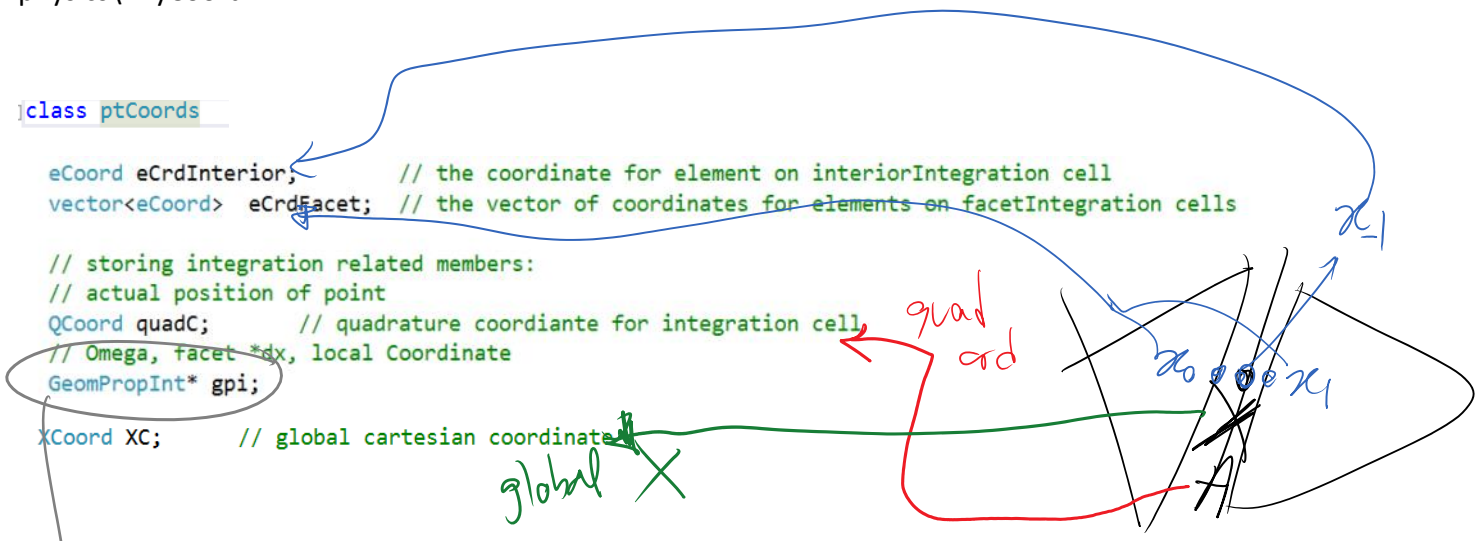
e.g.



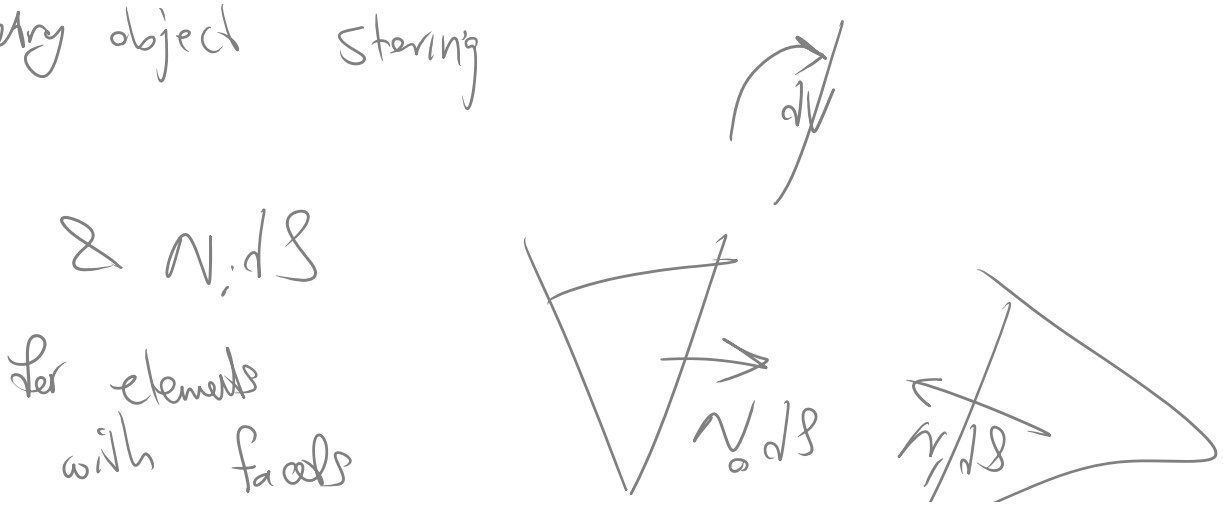


Please see the coordinate classes:
physics\PhyCoord.h

```
class ptCoords

  eCoord eCrdInterior;       // the coordinate for element on interiorIntegration cell
  vector<eCoord>  eCrdFacet;  // the vector of coordinates for elements on facetIntegration cells

  // storing integration related members:
  // actual position of point
  QCoord quadC;       // quadrature coordiante for integration cell
  // Omega, facet *dx, local Coordinate
  GeomPropInt* gpi;

XCoord XC;      // global cartesian coordinate
```



geometry object storing



with facets on pic

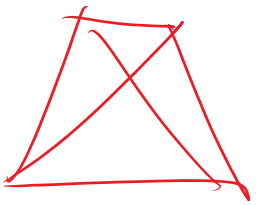
$\int N_i dS$ $\sum_i N_i dS$

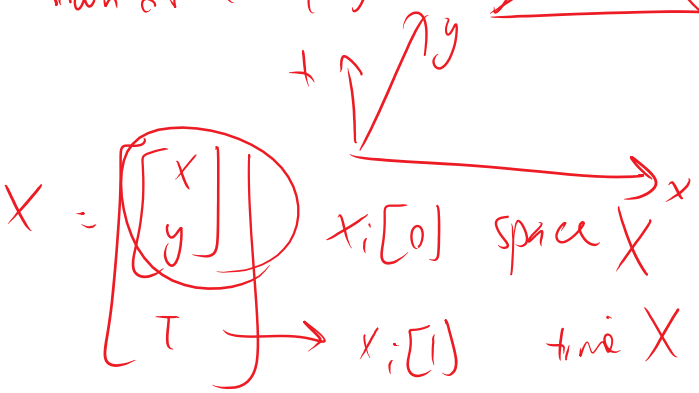
($N_i dS \approx \pm dx$ in differential forms notation)

```
class XCoord
{
    // global coordinate at once
    vector<double> X;
    // break-down of the global coordinatede determined by CoordManager
    vector< vector< double> > Xi;
```

global Cartesian X

break down of X (e.g.





```
class QCoord : public GQuadCoord
{
    GeomPropBase int gpb;
    GQuadIntOrder int_order;
    GQuadNumber int_number;
    XCoord *XC;
```

A quadrature coordinate

geom Object capable of transfers between & A (quad) X (global)

these objects (integration order & number of quad pt)

can be used in employing precalculated shape functions

```
class eCoord // coordinate class for element
{
    GeomPropBase e_gpb;
    PhyInt2EBasePtr* i2ePtr;
    BasisCoord basisC;
    QCoord *quadC;
    XCoord *XC;     // global ca
    PhyElementBaseInterior* pei;

}
```

x

basis coordinate

this is actually the basis coordinate "vector"

geometry class capable of

(basis) x $\longleftrightarrow$ global X coordinate transformations

&

$$\frac{\partial}{\partial x} \longrightarrow \frac{\partial}{\partial X}$$

$$\frac{\partial^2}{\partial x \partial x} \qquad \frac{\partial^2}{\partial X \partial X}$$

basis derivatives of shapes to global derivatives