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

Side note: 
Idea of function pointer:

```c
extern PhyTensorFieldcompPtr PhyTFComputeIntegrand[NUM_COMPT];
```
Example:
A list of values of this array of function pointer must be specified:
For example for spatial derivative it’s this function:

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

Other computation types:

```c
extern PhycompPtr PhyCompute[NUM_COMP][NUM_PHYFLD];
```
For stress Value

\[
\text{PhyCompute}[\text{ctVal}][\text{pfStrsL}] = \&\text{Physics}::\text{computeField}_\text{ctVal}_\text{pfStrsL};
\]

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

In PhyPhysics.h:

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

SLPhysics.cpp actual implementation

```cpp
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:

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

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 for pre-calculation
      b. Tensor field storage

\[
Q = \left[ \begin{array}{c}
\frac{e^2}{c^3} \\
\end{array} \right]
\]

IntHStorage

\[
\phi, \quad \alpha_1, \alpha_2, \alpha_3, \alpha_4
\]

\[
V_0 = \frac{1}{\sqrt{3}}, \quad V_1 = \frac{1}{\sqrt{3}}
\]
b. Tensor field storage

\[
(\mathbf{u}^{\text{val}}, \quad (\mathbf{v}^{\text{val}}, \mathbf{s}^{\text{val}}))
\]

4. Now that all the tensor fields are calculated in `fldVals` we can go ahead and evaluate/partial 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:

```cpp
bool PhyIntCellBase::setAfterElementPolyOrderSetBeginningEachStage(intStage intS)
{
    bool b_time = ((intS == intSPreAssembly) || (intS == intSAccumulation));
    if (b_time == true)
    
    if (intNumQuadsTotal == 0)
        intNumQuadsTotal = intNumQuadsTotal + 1;
    
    // precompute and set ptCoords and Shapes storages
    if (precomputedPtCoords && precomputedShapes)
    
    if (recomputeQuadPtVals)
    {
        // setting quad points needed for this int cell
        if (intS == intSPreAssembly)
            // this can be reversed to false so no matter what the order of the element,
            // if a beta orders not provided, alpha orders will be used for them:
            // getQuadPointWeightFunctions(intOrder, intCoeffs, intWeights, intT), intNumQuads, issueErrorIfPtCoordRecomputed, useAlpha

        if ((intS == intSPreAssembly) &&

            (recomputeQuadPtVals || (intCellHStorage.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:

```cpp
bool PhyIntCellBase::IntegrateIntegrationCell(int Stage intS,
                                             gTime *clockPreComputeFields, gTime *clockEvaluate)
{
    // facetT = (int)cellT.ct, intST = (int)intS;
    Interior, boundary, inflow, outflow

    for (int quadPN = 0; quadPN < intNumQuadsTotal; ++quadPN)
    {
        cout << "quadPN " << quadPN << "/" << intNumQuadsTotal << endl;
        ptCords *crdPtr = &intCrsd(intS);
        
        // interior integration
        phyFieldVal fIdVal(this, crdPtr); // storage for shapes and values
        // Computing values before going into integrands
        bool successfulPrecompute = (fIdVal * ComputeIntegrand[intST][cellT.ct]
                                 (*crdPtr, intCellHStore[quadPN]);

        // get the precomputed quad pt
        Storage for all the tensor!
        // precomputed shape storage

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

        // get the precomputed quad pt
        Storage for all the tensor!
        // precomputed shape storage
    }
    
    // interior integration
    phyFieldVal fIdVal(this, crdPtr); // storage for shapes and values
    // Computing values before going into integrands
    bool successfulPrecompute = (fIdVal * ComputeIntegrand[intST][cellT.ct]
                                 (*crdPtr, intCellHStore[quadPN]);

    // get the precomputed quad pt
    Storage for all the tensor!
    // precomputed shape storage
}
```

Evaluation stage:

...

Again we use function pointers

```
if (InteriorInt == true)
{
    pe = InteriorIntBase.eh.get薏();
    if (pe->IntegrateIntegrationCell[intS](factor, eInterior, *crdPtr, this, quadPN, fIdVal) == false)
        exit(0); // Parallel edit
    //return false;
}
```

Example

```
// int SAssembly
bool PhyElementBase::IntegrateInnerIntSAssembly(double factor, int eIndex, ptCords& crds,
                                                        PhyIntCellBase*)
{
    for (int phy = 0; phy < num_physics; ++phy)
    {
        P04 ( phy << "INTERIOR.Assembly.phy" << phy << " \num.physics \" <> phy "");
        if (physics[phy]->IntegrateInnerIntSAssembly(factor, eIndex, crds, phyIntCellBase) == false)
            return false;
    }
```
Examples of function pointers for "processing of the interior OR faces attached to PIC:

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

The definition of these function pointers
For example, for Assembly processing involves handling contributors to $R \text{rendal}$ and $K$ stiffness for time terms.

\[ W \partial F = \int \left( \frac{\partial W}{\partial u} \partial F + \frac{\partial W}{\partial F} \partial u \right) \partial V \]

\[ \text{solid inside} \quad a \cdot dV \]

\[ \text{weight field} \quad v \text{ln field} \]

\[ \text{e.g.} \quad W = \int \left( \frac{\partial W}{\partial u} \partial F + \frac{\partial W}{\partial F} \partial u \right) \partial V \]

\[ F = p \text{u} = p \]
Please see the coordinate classes:
physics\PhyCoord.h

```cpp
class ptCoords
{
    eCoord eCrdsInterior; // the coordinate for element on interiorIntegration cell
    vector<eCoord> eCrdsFacet; // the vector of coordinates for elements on facetIntegration cells

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

    XCoord XC; // global cartesian coordinate

    geometry object storing
    geometry part

    \sum (8 - 6) N_i dS
    \sum (8 - 6) N_i dS
    \sum (8 - 6) N_i dS
    \sum (8 - 6) N_i dS
```

\$X_i\$ for elements with faces

\$X_i\$ for elements with faces

\$X_i\$
class XCoord {
    // global coordinate at once
    vector<double> X;
    // break-down of the global coordinate determined by CoordManager
    vector<vector<double>> xi;
}

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

    these objects (integration order & number of quad pt) can be used in employing precalculated shape functions

    break down of X (eg, X[i] space X i [0] time X)

    A quadrature coordinate geom object capable of transfers between & A (quad) (global)
class eCoord // coordinate class for element
{
GeomPropBase *gpb;
PhyInt2FBasePtr *i2ePtr;
BasisCoord basisC;
QCoord *quadC;
XCoord *XC;     // global ca
PhyElementBaseInterior* pei;
}

this is actually the basis coordinate "vector"

geometry class capable of
(basis)  global coordinate transformations

\[ \frac{\partial x}{\partial X} \]

basis derivatives of shapes to global derivatives