220922 Plan for assembly and scalar wrapping objects

Thursday, September 22, 2022   6:36 PM

In SHORT:
- The ComputeFullJacobianThread calls ONLY the computeOffDiagJacobian method, so you need to handle the computeJacobian when on the diagonal
- The ComputeJacobianThread calls ONLY the computeJacobian method, but some methods will ALSO provide coupling parts between its multiple primary variables

Kernel

For Kernel, there is one primary variable, and for solid mechanics the other variables happen as off diagonal
Assembly method for Jacobian (ComputeJacobianThread, computeJacobian) and for Full Jacobian (ComputeFullJacobianThread)

Field-Field	Field-Scalar	Field-Other_field	Field-Other_scalar
Scalar-Field	Scalar-Scalar	Scalar-Other_field	Scalar-Other_scalar

The green methods were added in the KernelScalarBase; others already are in existence
For AD Kernel, the scalar method is blank but could be added back in derived classes; otherwise handles all the same parts as regular

Integrated BC

For Integrated BC, the lower variable is treated as off diagonal, and so assembly happens in 4 groups for the full Jacobian (ComputeFullJacobianThread) or the two diagonal blocks for Jacobian only (ComputeJacobianThread, computeFaceJacobian)

Primary-Primary	Primary-Lower	Primary-Scalar	Primary-Other_field	Primary-Other_scalar
Lower-Primary	Lower-Lower	Lower-Scalar	Lower-Other_field	Lower-Other_scalar
Scalar-Primary	Scalar-Lower	Scalar-Scalar	Scalar-Other_field	Scalar-Other_scalar

The orange are not yet given and would be needed for a scalar IBC and a scalar I Lower BC
The BC assembly does have a scalar connection already
For AD Kernel, the scalar method is blank but could be added back in derived classes; otherwise handles all the same parts as regular

Interface Kernel

For Interface Kernel, the element and neighbor are treated as both primary variables of that object, so there is a (ComputeJacobianThread, computeInternalInterFaceJacobian) call and a ComputeFullJacobianThread call to

```
  const auto & ce = _fe_problem.couplingEntries(_tid);
  for (const auto & it : ce)
  {
    MooseVariableFieldBase & ivariable = *(it.first);
    MooseVariableFieldBase & jvariable = *(it.second);

    if (ivariable.isFV())
      continue;

    unsigned int ivar = ivariable.number(); (row-variable number)
    unsigned int jvar = jvariable.number(); (column-variable number)

      if (interface_kernel->variable().number() == ivar)
        interface_kernel->computeElementOffDiagJacobian(jvar);

      if (interface_kernel->neighborVariable().number() == ivar)
        interface_kernel->computeNeighborOffDiagJacobian(jvar);
```

Then internally, it checks if you are an interface variable or not:

```
InterfaceKernelTempl<T>
  bool is_jvar_not_interface_var = true;
  if (jvar == _var.number())
  {
    computeElemNeighJacobian(Moose::ElementElement);
    is_jvar_not_interface_var = false;
  }
  if (jvar == _neighbor_var.number())
  {
    computeElemNeighJacobian(Moose::ElementNeighbor);
    is_jvar_not_interface_var = false;
  }

  if (is_jvar_not_interface_var)
  {
    computeOffDiagElemNeighJacobian(Moose::ElementElement, jvar);
    computeOffDiagElemNeighJacobian(Moose::ElementNeighbor, jvar);
  }
```

For Jacobian, just call its computeJacobian and the scalar one
So: THIS one can get set up like the Kernel is for off diagonal Jac *-to-field, where you are either:
- _var, on the interface (Element or Neighbor, handled inside by calling computeElementOffDiagJacobian(jvar) or computeNeighborOffDiagJacobian(jvar) respectively, i.e. have 2 method calls, since ivar is LOCAL) along with a field-to-scalar for the Element and Neighbor); field-field and scalar-field
- _kappa_var, do nothing since not a field variable
- Jvar, a different variable (Element or Neighbor, handled inside by calling computeElementOffDiagJacobian(jvar) or computeNeighborOffDiagJacobian(jvar) respectively, i.e. have

2 method calls, since ivar is LOCAL) along with a field-to-scalar for jvar, which if jvar is on both sides (likely for CZM), then you also want to pass Element and Neighbor); field-other_field and scalar-other_field

And you implement off diagonal scalar (*-to-scalar) with similar ideas, treating Element and Neighbor as a unit

- _var, do nothing since handled above
- _kappa_var, then add an Element, Neighbor, and Scalar contribution; field-scalar and scalar-scalar
- Jvar, a different scalar (these don't have a concept of element or neighbor side); add an Element, Neighbor, and Scalar contribution; field-other_scalar and scalar-other_scalar

To provide the overall matrix:

Element-Element	Element-Neighbor	Element-Scalar	Element-Other_Element	Element-Other_Neighbor	Element-Other_scalar
Neighbor-Element	Neighbor-Neighbor	Neighbor-Scalar	Neighbor-Other_Element	Neighbor-Other_Neighbor	Neighbor-Other_scalar
Scalar-Element	Scalar-Neighbor	Scalar-Scalar	Scalar-Other_Element	Scalar-Other_Neighbor	Scalar-Other_scalar

The orange are not yet given and would be needed for a scalar Interface Kernel
The yellow are missing methods that are NOT in the code at all yet
AD version integrates the same features as IK, handles same parts

DG Kernel

For DG, I don't know what scalar would need to be connected to it yet… but nothing is in place yet
It is like IK, it treats all variables as primary, so a Jacobian call is to all 4:

```
void
DGKernelBase::computeJacobian()
{
  if (!excludeBoundary())
  {
    precalculateJacobian();

    // Compute element-element Jacobian
    computeElemNeighJacobian(Moose::ElementElement);

    // Compute element-neighbor Jacobian
    computeElemNeighJacobian(Moose::ElementNeighbor);

    // Compute neighbor-element Jacobian
    computeElemNeighJacobian(Moose::NeighborElement);

    // Compute neighbor-neighbor Jacobian
    computeElemNeighJacobian(Moose::NeighborNeighbor);
  }
}
```

So does the off diagonal variable, the jvar is assumed to be on Element and Neighbor sides

Element-Element	Element-Neighbor	Element-Scalar	Element-Other_Element	Element-Other_Neighbor	Element-Other_scalar
Neighbor-Element	Neighbor-Neighbor	Neighbor-Scalar	Neighbor-Other_Element	Neighbor-Other_Neighbor	Neighbor-Other_scalar
Scalar-Element	Scalar-Neighbor	Scalar-Scalar	Scalar-Other_Element	Scalar-Other_Neighbor	Scalar-Other_scalar

The orange are not yet given and would be needed for a scalar DG Kernel
The yellow are missing methods that are NOT in the code at all yet
AD version is the same as the regular DG, handles same parts
So basically, the same as IK…
Well, it should look like this then:
For Jacobian, just call its computeJacobian (which gets all 4) and the scalar one
So: THIS one can get set up like the Kernel is for off diagonal Jac *-to-field, where you are either:

- _var, on the interface (Element and Neighbor, handled inside by calling computeJacobian, i.e. have 1 method call which gets all 4) along with a field-to-scalar for the Element and Neighbor); field-field and scalar-field
- _kappa_var, do nothing since not a field variable
- Jvar, a different variable (Element or Neighbor, handled inside by calling computeOffDiagJacobian, i.e. have 1 method call which gets all 4) along with a field-to-scalar for jvar, which if jvar is on both sides (likely for CZM), then you also want to pass Element and Neighbor); field-other_field and scalar-other_field

And you implement off diagonal scalar (*-to-scalar) with similar ideas, treating Element and Neighbor as a unit

- _var, do nothing since handled above
- _kappa_var, then add an Element, Neighbor, and Scalar contribution; field-scalar and scalar-scalar
- Jvar, a different scalar (these don't have a concept of element or neighbor side); add an Element, Neighbor, and Scalar contribution; field-other_scalar and scalar-other_scalar
    - If there IS supposed to be a scalar on both sides, THAT is just a constant DG function because EVERY element would have its o wn scalar…
    - So probably, this is like the temperature of the whole model on average, which evolves in time somehow

For lower DG, then the Jacobian case has E-N and N-E but only L-L part. The full Jacobian call will all 9.
So, basically like IBC lower: the LOWER is an OTHER variable type that is considered owned by this element

Element-Element	Element-Neighbor	Element-Lower	Element-Scalar	Element-Other_Element	Element-Other_Neighbor	Element-Other_Lower	Element-Other_scalar
Neighbor-Element	Neighbor-Neighbor	Neighbor-Lower	Neighbor-Scalar	Neighbor-Other_Element	Neighbor-Other_Neighbor	Neighbor-Other_Lower	Neighbor-Other_scalar
Lower-Element	Lower-Neighbor	Lower-Lower	Lower-Scalar	Lower-Other_Element	Lower-Other_Neighbor	Lower-Other_Lower	Neighbor-Other_scalar
Scalar-Element	Scalar-Neighbor	Scalar-Lower	Scalar-Scalar	Scalar-Other_Element	Scalar-Other_Neighbor	Scalar-Other_Lower	Scalar-Other_scalar

For Jacobian, just call its computeJacobian (which gets all 5) and the scalar one
The yellow are missing methods that are NOT in the code at all yet
No AD version of this exists yet
So: THIS one can get set up like the Kernel is for off diagonal Jac *-to-field, where you are either:

- _var or _lower, on the interface (Element and Neighbor, handled inside by calling computeOffDiagJacobian, i.e. have 1 method call which gets all 9) along with a field-to-scalar for the Element and Neighbor); field-field and scalar-field
- _lower

- _kappa_var, do nothing since not a field variable
- Jvar, a different variable (Element or Neighbor, handled inside by calling computeOffDiagJacobian, i.e. have 1 method call which gets all 4) along with a field-to-scalar for jvar, which if jvar is on both sides (likely for CZM), then you also want to pass Element and Neighbor); field-other_field and scalar-other_field

And you implement off diagonal scalar (*-to-scalar) with similar ideas, treating Element and Neighbor as a unit

- _var, do nothing since handled above
- _lower, do nothing since handled above
- _kappa_var, then add an Element, Neighbor, and Scalar contribution; field-scalar and scalar-scalar
- Jvar, a different scalar (these don't have a concept of element or neighbor side); add an Element, Neighbor, and Scalar contribution; field-other_scalar and scalar-other_scalar
    - If there IS supposed to be a scalar on both sides, THAT is just a constant DG function because EVERY element would have its o wn scalar…
    - So probably, this is like the temperature of the whole model on average, which evolves in time somehow

The mortar case is like DG and IK, but doesn't seem to have any OTHER variables yet; recall Element = Secondary and Neighbor= Primary

Element-Element	Element-Neighbor	Element-Lower	Element-Scalar	Element-Other_Element	Element-Other_Neighbor	Element-Other_Lower	Element-Other_scalar
Neighbor-Element	Neighbor-Neighbor	Neighbor-Lower	Neighbor-Scalar	Neighbor-Other_Element	Neighbor-Other_Neighbor	Neighbor-Other_Lower	Neighbor-Other_scalar
Lower-Element	Lower-Neighbor	Lower-Lower	Lower-Scalar	Lower-Other_Element	Lower-Other_Neighbor	Lower-Other_Lower	Neighbor-Other_scalar
Scalar-Element	Scalar-Neighbor	Scalar-Lower	Scalar-Scalar	Scalar-Other_Element	Scalar-Other_Neighbor	Scalar-Other_Lower	Scalar-Other_scalar

The AD mortar objects DO handle the other variable couplings, and this happens in the computeJacobian call; so the whole K contribution is ALWAYS provided
So, the tagging system should cover those variables naturally as well, and we could add that into the non-AD version in the computeJacobian routine
The yellow are missing methods that are NOT in the code at all yet
The orange are not yet given and would be needed for a scalar Mortar constraint
I would need to hear from MOOSE guys whether following the idea of LowerDG or the idea of IK (i.e. how primary is the lower variable) would be the approach for adding off diagonal field and off diagonal scalar. Since the AD objects assemble off diagonal variables for all problems (no Jacobian or Full Jacobian distinction), then likely the non-AD version should be taken for how to do the off diagonal field