Scott Wurtz and Kyle Kent
Professor Roberts
cs335
1/26/15

Phase I Testing

Test ideas for each method:

- BF
  - BF_bf(VarFactory varFactory)
    - Create a BF and assert it's not null
  - void addTerm(LinearTermPtr trialTerm, LinearTermPtr testTerm)
    - Create two linear terms, add them to the BF. Create a solution for this BF, and then use BF.testFunctional() to ensure that the BF's solution is was is expected.
  - void addTerm (VarPtr trialVar, LinearTermPtr testTerm)
    - Create a linear term and a var, add them to the BF. Create a solution for this BF, and then use BF.testFunctional() to ensure that the BF's solution is was is expected.
  - void addTerm(VarPtr trialVar, VarPtr testVar)
    - Create two vars, add them to the BF. Create a solution for this BF, and then use BF.testFunctional() to ensure that the BF's solution is was is expected.
  - void addTerm(LinearTermPtr trialTerm, VarPtr testVar)
    - Create a linear terms and a var, add them to the BF. Create a solution for this BF, and then use BF.testFunctional() to ensure that the BF's solution is was is expected.
  - const string & testName(int testID)
    - Create a var with a given name, create a BF with that varFactory, and assert that the BF's return of the var's name is the same as the var's return of the name.
  - const string & trialName(int trialID)
    - Create a var with a given name, create a BF with that varFactory, and assert that the BF's return of the var's name is the same as the var's return of the name.
  - Camellia::EFunctionSpace functionSpaceForTest(int testID)
    - Create a var, and then create a bf with the same var factory. Test that the space returned by BF's method is the same as the space returned by the var created. (This one is not working in the final product)
  - Camellia::EFunctionSpace functionSpaceForTrial(int trialID)
    - Create a var, and then create a bf with the same var factory. Test that the space returned by BF's method is the same as the space returned by the var created. (This one is not working in the final product)
  - IPPtr graphNorm(double weightForL2TestTerms = 1.0)
    - Just test to make sure this method runs successfully. Don't verify output.

- IPPtr naiveNorm(int spaceDim)
  - Test to make sure this method runs successfully. Don't verify output.
- bool isFluxOrTrace(int trialID)
  - Create various vars, flux & trace or not, and make sure that this method returns true when the var in question is flux or trace, false otherwise.
- string displayString()
  - Was unsure about how to get a copy of this string any other way, so did not test this method.
- LinearTermPtr testFunctional(SolutionPtr trialSolution)
  - Same as tests for addTerm(). This is a rather circular test, but as you said in class sometimes this can't be avoided, and I didn't see another option in this case. Create two vars, add them to the BF. Create a solution for this BF, and then use BF.testFunctional() to ensure that the BF's solution is was is expected.
- VarFactory varFactory()
  - Create a var in a varFactory, and get it's ID. Create a BF with the same varFactory, and get the var's ID through BF.varFactory(). Assert the two ID's are equal.

- Solution
  - static SolutionPtr **solution**(MeshPtr mesh, BCPtr bc = Teuchos::null, RHSPtr rhs = teuchos::null, IPPtr ip = Teuchos::null);
    - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use that mesh to create a new Solution, then assert that the active elements of the Solution's Mesh is equal to 12, which is the number of active elements in the original Mesh.
  - void **addSolution**(Teuchos::RCP<Solution> soln, double weight, bool allowEmptyCells = false, bool replaceBoundaryTerms=false)
    - I could not get this method to take the parameters it asked for no matter what I tried so there is no working test for it yet.
  - void **addSolution**(Teuchos::RCP<Solution> soln, double weight, set<int> varsToAdd, bool allowEmptyCells = false)
    - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use Function to make three functions, x, one, zero, then use the PoissonFormulation to create two VarPtrs, phi and psi. Then use the Mesh to make two Solutions, projecting the first first one onto the mesh and then adding it with phi and a weight of 1.0. Then assert that the L2Norm of the projected and added Solution is different from the soln that was not touched.
  - void **clear**()
    - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use that mesh to create a new Solution, then clear it. Assert to make sure that it's not none, and then assert that the L2Norm of the Solution is equal to 0.
  - int **cubatureEnrichmentDegree**()
    - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use that mesh to create a new

Solution, then set its cubature enrichment degree to 8. Assert that the Solution's cubature enrichment degree is 8.

- void **setCubatureEnrichmentDegree**(int value)
  - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use that mesh to create a new Solution, then set its cubature enrichment degree to 9. Assert that the Solution's cubature enrichment degree is 9.
- double **L2NormOfSolution**(int trialID)
  - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use that mesh to create a new Solution. Create a new VarFactory, then use that to create a new basic fieldVar. Assert that the L2Norm of the field variable in the Solution is equal to 0.
- void **projectOntoMesh**(const std::map<int, FunctionPtr > &functionMap)
  - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use Function to make three functions, x, one, zero, then use the PoissonFormulation to create two VarPtrs, phi and psi. Then use the Mesh to make a Solution. Assert that the L2Norm of the phi variable in the Solution is equal to zero, then project the first one onto the mesh. Then assert that the L2Norm of the the phi variable of the Solution is not equal to 0.0.
- MeshPtr **mesh**()
  - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use that mesh to create a new Solution, then assert that the number of active elements in the mesh is equal to the number of active elements in Solution.mesh().
- BCPtr **bc**()
  - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use that mesh to create a new Solution. Create a new VarFactory, then use that to create a new basic fieldVar. Make a new BC named testBC, then set the Solution's BC to testBC. Assert that the value of bcsImposed of testBC with respect to field variable is equal to the value of soln.bc().bcsImposed with respect to the field variable.
- IPPtr **ip**()
  - This method was tricky to test and I couldn't figure out how to determine the identities of the two IPPtrs in question, so I just tested for the existence of the IP.
  - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Use that mesh to create a new Solution, then make a new IP names testIP. Set the Solution's IP to testIP, then assert that soln.ip() is not none.

- HDF5Exporter
    - **HDF5Exporter**(MeshPtr mesh, string outputDirName="output", string outputDirSuperPath = ".")
        - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Create a new HDF5Exporter using the mesh. Assert that the HDF5Exporter is not none.
    - void **exportFunction**(FunctionPtr function, string functionName="function", double timeVal=0)
        - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Create a new HDF5Exporter using the mesh. Create a new function, then export the function using the HDF5Exporter.exportFunction method.
    - void **exportFunction**(vector<FunctionPtr> functions, vector<string> functionNames, double timeVal=0)
        - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Create a new HDF5Exporter using the mesh. Create a vector of two new function, a vector of two function names, then export them using the HDF5Exporter.exportFunction method.
    - void **exportSolution**(SolutionPtr solution, VarFactory varFactory, double timeVal=0)
        - Create a PoissonFormulation, use it to create a BF, and then use that BF to create a RectilinearMesh. Create a new HDF5Exporter using the mesh. Create a new solution also using the mesh, then export the solution using the HDF5Exporter.exportFunction method.