Nate Hills, Jeff Burge, Blake Hensel

**Testing Plan for Team Two Classes:**

**Function:**

Instance methods:

• virtual FunctionPtr x()

Returns the x component of this (if this is a vector-valued function).

// test returning the x component of a vector-valued function

// test returning the x component of a null vector-valued function

// test on a non-vector-valued function (scalar?)

// test on a null non-vector valued function

// test on various types of x components?

• virtual FunctionPtr y()

Returns the y component of this (if this is a vector-valued function).

// same as above

• virtual FunctionPtr dx()

Returns the x derivative of this.

// return the x derivative for various types of functions, not sure what range of derivatives it should be able to return

// test on null function

// test on functions with undefined derivatives?

• virtual FunctionPtr dy()

Returns the y derivative of this.

// same as above

• virtual FunctionPtr div()

If this is a vector-valued function, returns its divergence, defined as

f->x()->dx() + f->y()->dy().

// run similar tests as the two above but check consistency between dx and dy

// make sure to check null possibilities, undefined possibilities, and other outlying cases

• virtual FunctionPtr grad()

Returns the gradient of this, defined as Function::vectorize(f->dx(), f->dy()).

// run same tests as on dx and dy

// this basically just returns the two as a vector

• int rank()

Returns the rank of this; 0 for scalar-valued function, 1 for vector-valued, etc.

// test scalar, vector, etc for all

// test nulls on all

• double l2norm(MeshPtr mesh, int cubatureDegreeEnrichment = 0)

Returns a non-negative value measuring the this function on the specified mesh. For

non-polynomial functions, or functions of higher polynomial degree than the sum

of the test and trial spaces on the mesh, specifying a positive cubature enrichment

will allow more precise measurement.

// We are going to need help creating a test for this

• virtual string displayString()

Returns a string representing this function.

// test on different types of functions, make sure can convert to string

// test on null

Static (class) methods:

• static double evaluate(FunctionPtr f, double x, double y)

Returns this function evaluated at (x, y). Only valid for scalar-valued functions

(function with rank 0).

// test on general scalar-valued functions, testing with results equal, greater than, less than, negative, tending to infinity, undefined, etc.

// test on null scalar

// test on non scalar-valued functions

• static FunctionPtr composedFunction( FunctionPtr f, FunctionPtr arg g)

Returns f ◦ g = f (g(·, ·)) the composition of f with g; g must be vector-valued.

// test with g not vector-valued

// test with g null

// test with g vector-valued, f scalar

// test with g and f vector

// test with f null

// test test with g returning undefined on all values

• static FunctionPtr constant(double value)

Returns a constant function with the specified value.

// test with variety of values

For all of these methods below, they would be tested by feeding them parameters where the expected output is known, then compare the expected output to what was actually output and if they are the same the function works on those inputs. Need to test for null cases, and edge cases. For functions, maybe deal with undefined function cases.

• static FunctionPtr vectorize(FunctionPtr f1, FunctionPtr f2)

Returns a two-dimensional vector function with components f1 and f2.

• static FunctionPtr normal()

Returns a function representing the unit outward-facing normal on each element

boundary.

• static FunctionPtr solution(VarPtr var, SolutionPtr soln)

Returns a function representing the specified variable component of the specified

Solution.

• static FunctionPtr xn(int n=1)

Returns a function representing xn .

• static FunctionPtr yn(int n=1)

Returns a function representing y n .

• FunctionPtr operator\*(FunctionPtr f1, FunctionPtr f2)

Returns a function representing f1 f2 .

• FunctionPtr operator/(FunctionPtr f1, FunctionPtr scalarDivisor)

f1

Returns a function representing f2 , where f2 is the scalar-valued function scalarDivisor.

• FunctionPtr operator/(FunctionPtr f1, double divisor)

Returns a function representing fb1 , where b is the constant specified by divisor.

• FunctionPtr operator/(double value, FunctionPtr scalarDivisor)

a

Returns a function representing f2 , where a is the constant specified by value and

f2 is the scalar-valued function scalarDivisor.

• FunctionPtr operator\*(double weight, FunctionPtr f)

Returns af where a is the constant weight.

• FunctionPtr operator\*(FunctionPtr f, double weight)

Returns af where a is the constant weight.

• FunctionPtr operator\*(vector<double> weight, FunctionPtr f)

Returns the product of a and f where a is the vector-valued constant weight. If

f is a vector-valued function, this is the (scalar-valued) dot product (a · f ); if f is

scalar-valued, the result is a vector-valued function f a.

• FunctionPtr operator\*(FunctionPtr f, vector<double> weight)

Returns the product of a and f where a is the vector-valued constant weight. If

f is a vector-valued function, this is the (scalar-valued) dot product (a · f ); if f is

scalar-valued, the result is a vector-valued function f a.

• FunctionPtr operator+(FunctionPtr f1, FunctionPtr f2)

Returns f1 + f2 .

• FunctionPtr operator+(FunctionPtr f1, double value)

Returns a+f1 where a is the constant value value.

• FunctionPtr operator+(double value, FunctionPtr f1)

Returns a+f1 where a is the constant value value.

• FunctionPtr operator-(FunctionPtr f1, FunctionPtr f2)

Returns f1 − f2 .

• FunctionPtr operator-(FunctionPtr f1, double value)

Returns f1 − a where a is the constant value value.

• FunctionPtr operator-(double value, FunctionPtr f1)

Returns a − f1 where a is the constant value value.

• FunctionPtr operator-(FunctionPtr f)

Returns −f .

**IP:**

• IP()

Constructor; produces an empty (zero) IP object.

// make an IP and check to see if what we made is null or throws an error.

// Check to see if IP is empty

• void addTerm( LinearTermPtr a )

Adds term a to the IP.

// add a linear term, see if result is correct

// try adding a null LinearTermPtr, make sure it handles it correctly

• void addTerm( VarPtr v )

Adds term v to the IP.

// add a var, see if result is correct

// try addint a null VarPtr, make sure it handles it correctly

• LinearTermPtr evaluate(map< int, FunctionPtr> &varFunctions)

Evaluates the IP by substituting the Functions in varFunctions for variables with

the corresponding IDs in the IP. The IP is understood as a symmetrical bilinear

form, and here we substitute for one of the two linear term arguments, leaving us

with a linear term.

// most complicated to test

// test general case of substituting the valid functions in varFunctions

// try substituting a null function

**RHS:**

• RHS(bool legacySubclass)

Constructor. The Python interface should eliminate the legacySubclass argument,

and always pass false to the C++ constructor.

// make an RHS and check to see if what we made is null or throws an error.

• bool nonZeroRHS(int testVarID)

// create one where the values are known then check a zero and a non zero and if they both work then it ought to function correctly.

Returns true if the RHS is not identically zero for the specified test variable.

• void addTerm( LinearTermPtr rhsTerm )

Adds the term specified to the RHS.

//add then return and see if what was added is returned.

• void addTerm( VarPtr v )

//same as prev.

Adds the term specified to the RHS.

• LinearTermPtr linearTerm()

//if returned and altered, then returned again, we know it works properly.

Returns a mutable reference to the RHS as a LinearTerm (change the returned

linear term, and RHS will change!).

• LinearTermPtr linearTermCopy()

Returns a copy of RHS as a LinearTerm.

//if initialized with a specific linear term then you copy it, it should be the same

**Linear Term:**

• LinearTerm()

Constructor; creates an empty (zero) linear term.

// test if LinearTerm is null or if constructor throws an error

// test to see if LinearTerm is empty

• const set<int> & varIDs()

Returns IDs for all the Vars in the LinearTerm.

// test general case

// test empty LinearTerm

// test on null

• VarType termType()

Returns the variable type for the variables in the LinearTerm. Can be test, field,

flux, trace, or mixed.

// test on all combinations of variable types

// test on null

• FunctionPtr evaluate(map< int, FunctionPtr> &varFunctions)

Evaluates a LinearTerm as a Function by substituting Functions for the Vars in

the LinearTerm.

• int rank()

For all of these methods below, they would be tested by feeding them parameters where the expected output is known, then compare the expected output to what was actually output and if they are the same the function works on those inputs.

Returns 0 for scalar, 1 for vector, etc.

• string displayString()

Returns a text string, TeX by convention, representing the linear term.

// test on empty

// test on null

// test on LinearTerm of different types

• LinearTermPtr operator+(LinearTermPtr a1, LinearTermPtr a2)

Returns a1 + a2 .

// test on general case

// test on null LinearTerms

// test on adding LinearTerms of different types

• LinearTermPtr operator+(VarPtr v, LinearTermPtr a)

Returns v + a.

// test on general

/// test on null for each

// test on different types?

• LinearTermPtr operator+(LinearTermPtr a, VarPtr v)

Returns a + v. // same as above

• LinearTermPtr operator\*(FunctionPtr f, VarPtr v)

Returns f v. // same as above

• LinearTermPtr operator\*(VarPtr v, FunctionPtr f)

Returns f v. // same as above

• LinearTermPtr operator\*(double weight, VarPtr v)

Returns av, where a is the constant weight weight.

// same as above

• LinearTermPtr operator\*(VarPtr v, double weight)

Returns av, where a is the constant weight weight.

// same as above

• LinearTermPtr operator\*(vector<double> weight, VarPtr v)

Returns av, where a is the constant vector-valued weight weight.

// same as above

• LinearTermPtr operator\*(VarPtr v, vector<double> weight)

Returns av, where a is the constant vector-valued weight weight.

// same as above

• LinearTermPtr operator\*(FunctionPtr f, LinearTermPtr a)

Returns f a.

// same as above

• LinearTermPtr operator+(VarPtr v1, VarPtr v2)

Returns v1 + v2 .

// same as above

• LinearTermPtr operator/(VarPtr v, double weight)

v

// same as above

Returns a , where a is the constant weight weight.

• LinearTermPtr operator/(VarPtr v, FunctionPtr f)

v

// same as above

Returns f , where f is scalar-valued.

• LinearTermPtr operator-(VarPtr v1, VarPtr v2)

Returns v1 − v2 .

// same as above

• LinearTermPtr operator-(VarPtr v)

Returns −v1 .

// same as above

• LinearTermPtr operator-(LinearTermPtr a)

Returns −a.

// same as above

• LinearTermPtr operator-(LinearTermPtr a, VarPtr v)

Returns a − v.

// same as above

• LinearTermPtr operator-(VarPtr v, LinearTermPtr a)

Returns v − a.

// same as above

• LinearTermPtr operator-(LinearTermPtr a1, LinearTermPtr a2)

Returns a1 − a2

// same as above