Linear Term - Paul

Spatial Filter - Connor

RHS - Connor

Function - Seth

**Function**

• virtual FunctionPtr x()

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

// Enter a function and check that the return is the same as the x component entered

• virtual FunctionPtr y()

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

// Enter a function and check that the return is the same as the x component entered

• virtual FunctionPtr dx()

Returns the x derivative of this.

// Enter a function with a known x derivative and check that the return = known value

• virtual FunctionPtr dy()

Returns the y derivative of this.

// Enter a function with a known x derivative and check that the return = known value

• virtual FunctionPtr div()

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

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

// Enter a function with known answer to dx()+dy() and make sure return = that answer

• virtual FunctionPtr grad()

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

// Enter a function with known gradient and assert that gradient = the return

• int rank()

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

//Enter a scalar-valued function and assert the return is 0

//Enter a vector-valued function and assert the return is 1

• 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.

// ||f-g|| == 0

• virtual string displayString()

Returns a string representing this function.

//Enter a string and make sure the correct string is displayed

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).

//Enter a function with known answer and make sure the answer = the return

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

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

//Enter two functions with known answer to composition of the two and check with the return

• static FunctionPtr constant(double value)

Returns a constant function with the specified value.

//Enter a value and check the return with constant function it should be

• static FunctionPtr vectorize(FunctionPtr f1, FunctionPtr f2)

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

//Enter two functions and check return with two-dimensional vector function it should be

• static FunctionPtr normal()

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

boundary.

//Enter a function with known normal and compare to the return

• static FunctionPtr solution(VarPtr var, SolutionPtr soln)

Returns a function representing the specified variable component of the specified

Solution.

//Enter a variable and a solution and compare the return with the function that should be returned

• static FunctionPtr xn(int n=1)

Returns a function representing x

//Enter an int and compare the return to the known answer

• static FunctionPtr yn(int n=1)

Returns a function representing y

//Enter an int and compare the return to the known answer

FunctionPtr operator\*(FunctionPtr f1, FunctionPtr f2)

Returns a function representing f1f2.

//Enter two function pointers and compare the return with the answer of f1\*f2

• FunctionPtr operator/(FunctionPtr f1, FunctionPtr scalarDivisor)

Returns a function representing f1

f2, where f2 is the scalar-valued function scalarDivisor.

//Enter two function pointers and compare the return with the answer of f1/f2

• FunctionPtr operator/(FunctionPtr f1, double divisor)

Returns a function representing f1

b, where b is the constant specified by divisor.

//Enter a function pointer and double and compare the return with the answer of f1/divisor

• FunctionPtr operator/(double value, FunctionPtr scalarDivisor)

Returns a function representing a f2, where a is the constant specified by value and f2 is the scalar-valued function scalarDivisor.

//Enter a function pointer and double and compare the return with the answer of value/scalarDivisor

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

Returns af where a is the constant weight.

//Enter a function pointer and double and compare the return with the answer of f\*weight

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

Returns af where a is the constant weight.

//Enter a function pointer and double and compare the return with the answer of f\*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.

//Enter a function pointer and double type vector and compare the return with the answer of f\*weight

• 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.

//Enter a function pointer and double type vector and compare the return with the answer of f\*weight

• FunctionPtr operator+(FunctionPtr f1, FunctionPtr f2)

Returns f1 + f2.

//Enter two function pointers and compare the return with the answer of f1+f2

• FunctionPtr operator+(FunctionPtr f1, double value) Returns a+f1 where

a is the constant value value.

//Enter a function pointer and a value and compare the return with the answer of f1+value

• FunctionPtr operator+(double value, FunctionPtr f1) Returns a+f1 where

a is the constant value value.

//Enter a function pointer and a value and compare the return with the answer of f1+value

• FunctionPtr operator-(FunctionPtr f1, FunctionPtr f2)

Returns f1 − f2.

//Enter two function pointers and compare the return with f1-f2

• FunctionPtr operator-(FunctionPtr f1, double value)

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

//Enter a function pointer and a value and compare the return with the answer of f1-value

• FunctionPtr operator-(double value, FunctionPtr f1)

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

//Enter a function pointer and a value and compare the return with the answer of value-f1

• FunctionPtr operator-(FunctionPtr f)

Returns −f.

//Enter a function pointer and compare the return with the answer of -1\*f

**LinearTerm**

• LinearTerm()

Constructor; creates an empty (zero) linear term.

// Check that some variable holding this is not null.

• const set<int> & varIDs()

Returns IDs for all the Vars in the LinearTerm.

// Create several vars and put their id’s into variables which are stored into an array

// Create the linearterm using these vars.

// Run this method and check that each of the vars corresponds to the right varID.

• VarType termType()

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

flux, trace, or mixed.

// Create a linear term using a predefined variable type and store this type.

// Check the variable type returned from this method with the one that was stored.

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

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

the LinearTerm.

//

• int rank()

Returns 0 for scalar, 1 for vector, etc.

// Test with arbitrary examples of scalar, vector, etc

• string displayString()

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

// Test with various LinearTerms and see if the printed string is an accurate representation of it

// Test with empty LinearTerm

*operator overloads:*

LinearTermPtr operator+(LinearTermPtr a1, LinearTermPtr a2)

Returns a1 + a2.

// test with various random LinearTerms to make sure that an arbitrary one won’t crash the program

• LinearTermPtr operator+(VarPtr v, LinearTermPtr a)

Returns v + a.

// test with various random Vars and LinearTerms to make sure that an arbitrary one won’t crash the program

• LinearTermPtr operator+(LinearTermPtr a, VarPtr v)

Returns a + v.

// test with various random Vars and LinearTerms to make sure that an arbitrary one won’t crash the program

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

Returns fv.

// test with various random Vars and Functions to make sure that an arbitrary one won’t crash the program

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

Returns fv.

// test with various random Vars and Functions to make sure that an arbitrary one won’t crash the program

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

Returns av, where a is the constant weight weight.

// test with various random Vars and weights to make sure that an arbitrary one won’t crash the program

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

Returns av, where a is the constant weight weight.

// test with various random Vars and weights to make sure that an arbitrary one won’t crash the program

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

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

// test with various random Vars and weights to make sure that an arbitrary one won’t crash the program

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

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

// test with various random Vars and weights to make sure that an arbitrary one won’t crash the program

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

Returns f a.

// test with various random Functions and LinearTerms to make sure that an arbitrary one won’t crash the program

• LinearTermPtr operator+(VarPtr v1, VarPtr v2)

Returns v1 + v2.

// test with various random Vars to make sure that an arbitrary one won’t crash the program

• LinearTermPtr operator/(VarPtr v, double weight)

Returns va, where a is the constant weight weight.

// test with various random Vars and weights to make sure that an arbitrary one won’t crash the program

• LinearTermPtr operator/(VarPtr v, FunctionPtr f)

Returns vf, where f is scalar-valued.

// test with various random Vars and Functions to make sure that an arbitrary one won’t crash the program

• LinearTermPtr operator-(VarPtr v1, VarPtr v2)

Returns v1 − v2.

// test various random Vars to make sure that no general term could cause issues

// test w/ null arguments to make sure it handles them correctly

• LinearTermPtr operator-(VarPtr v)

Returns −v1.

// test various random Vars to make sure that no general term could cause issues

// test w/ null arguments to make sure it handles them correctly

• LinearTermPtr operator-(LinearTermPtr a)

Returns −a.

// test various random LinearTerms to make sure that no general term could cause issues

// test w/ null arguments to make sure it handles them correctly

• LinearTermPtr operator-(LinearTermPtr a, VarPtr v)

Returns a − v.

// test various random LinearTerms and Vars to make sure that no general term could cause issues

// test w/ null arguments to make sure it handles them correctly

• LinearTermPtr operator-(VarPtr v, LinearTermPtr a)

Returns v − a.

// test various random LinearTerms and Vars to make sure that no general term could cause issues

// test w/ null arguments to make sure it handles them correctly

• LinearTermPtr operator-(LinearTermPtr a1, LinearTermPtr a2)

Returns a1 − a2.

// test various random LinearTerms to make sure that no general term could cause issues

// test w/ null arguments to make sure it handles them correctly

**SpatialFilter**

• virtual bool matchesPoint(double x, double y)

Returns true if the filter matches the provided point, false otherwise.

// test points that are known to be on the filter

// test points that are known not to be on the filter

• virtual bool matchesPoint(vector<double> &point)

Returns true if the filter matches the provided point, false otherwise.

// test points that are known to be on the filter

// test points that are known not to be on the filter

• static SpatialFilterPtr allSpace()

Returns a filter that returns true for every point.

// compare this to the union of a filter and its negation (which would be true)

• static SpatialFilterPtr unionFilter(SpatialFilterPtr a, SpatialFilterPtr

b)

Returns a filter that returns true at a point if either of the provided filters return

true at that point.

// union two filters with a known X and known Y, assert that it matches the point at that x and y

• static SpatialFilterPtr intersectionFilter(SpatialFilterPtr a, SpatialFilterPtr

b)

Returns a filter that returns true at a point if both provided filters return true at

that point.

// intersect one filter with an allSpace filter and assert that the intersection must be equal to the first filter

• static SpatialFilterPtr negatedFilter(SpatialFilterPtr filterToNegate)

Returns a filter that returns true at points where the provided filter returns false,

and vice versa.

// intersect a filter with its negation and assert that it’s equal to the negation of allSpace (which is false at all points)

• static SpatialFilterPtr matchingX(double x)

Returns a filter that matches any point which has the provided x coordinate.

// check that the filter matches X

• static SpatialFilterPtr matchingY(double y)

Returns a filter that matches any point which has the provided y coordinate.

// check that the filter matches Y

• static SpatialFilterPtr lessThanX(double x)

Returns a filter that matches any point whose x coordinate is less than x.

// check that the filter is less than X

• static SpatialFilterPtr lessThanY(double y)

Returns a filter that matches points whose y coordinate is less than y.

// check that the filter is less than Y

• static SpatialFilterPtr greaterThanX(double x)

Returns a filter that matches any point whose x coordinate is greater than x.

// check that the filter is greater than X

• static SpatialFilterPtr greaterThanY(double y)

Returns a filter that matches any point whose y coordinate is less than y

// check that the filter is greater than Y

**RHS**

• RHS(bool legacySubclass)

Constructor. The Python interface should eliminate the legacySubclass argument,

and always pass false to the C++ constructor.

// test that both the legacy constructor and the new constructor properly initialize RHS

• bool nonZeroRHS(int testVarID)

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

// test that it doesn’t return true when it isn’t identically zero

// test that it does return true on a variable where it should

• void addTerm( LinearTermPtr rhsTerm )

Adds the term specified to the RHS.

// make sure the RHS has the added term

• void addTerm( VarPtr v )

Adds the term specified to the RHS.

// make sure the RHS has the added term

• LinearTermPtr linearTerm()

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

linear term, and RHS will change!).

// displayString on the LinearTerm and make sure that it is a copy of the RHS

// make sure that changing the LinearTerm does change the RHS

• LinearTermPtr linearTermCopy()

Returns a copy of RHS as a LinearTerm.

// call displayString on the LinearTerm and make sure that it is a copy of the RHS

// make sure that changing the LinearTerm doesn’t change the RHS (since this is a copy)