Approximator::run

Function Prototype

```
run ( int numsteps, double stepsize)
run (double precision, double accuracy, double stepsize)
```

Key Words

flow, curvature, stepsize, precision, accuracy, approximator

Authors

Joseph Thomas, Alex Henniges

Introduction

The `run` function of the Approximator class runs a system of differential equations representing a curvature flow for either a number of steps or until the values are within a desired accuracy and precision. The system to use and how steps are performed is given in the constructor of the approximator. The type of run is based on the parameters given.

Subsidiaries

Functions:
- step
- isPrecise
- isAccurate
- getLatest
  - recordState

Global Variables: radii, curvatures
Local Variables: **int** numsteps, **double** stepsize, **double** precision, **double** accuracy

Description

If the `run` function is given a number of steps, it will call its step function that number of times. In between steps, the `run` function will record the current state of any values that have been requested to be recorded (this is specified in the constructor).

If the `run` function is given a precision and accuracy, it will continue to call its step function until the desired quantities (curvature in two dimensions and curvature divide by radius in three dimensions) have converged within the precision and accuracy bounds. Precision is defined to be the difference between subsequent values of a quantity. Therefore, precision is a measure of how much a value is changing. Accuracy is defined to be the difference in value between quantities. Therefore, accuracy is a measure of how close values are to each other.

In between steps, the `run` function will record the current state of any values that have been requested to be recorded (this is specified in the constructor).

The `run` function and the overarching Approximator class exists as an improvement over the curvature flows of earlier versions of the Geocam project. The `run` function provides the skeleton that is similar for all types of curvature flows. Beyond the constructor, this should be the only thing a user calls from the Approximator class.

Practicum

Example:

```
// Create an approximator that uses the Euler method on a Yamabe flow.
Approximator *app = new EulerApprox(Yamabe);

// run a Yamabe flow for 300 steps with a stepsize of 0.01.
app->run(300, 0.01);
// run with a precision and accuracy bounds of 0.0001 and a stepsize of 0.01
app->run(0.0001, 0.0001, 0.01);
```

Limitations

The `run` function is limited in the systems of differential equations that it can `run`. It is designed to run with curvature flows and, when precision and accuracy are used, expects the values to converge. If a precision/accuracy run is performed on a flow that does not converge, the `run` function will not stop. If a new curvature flow is created whose convergence is not the usual (as in curvature divided by radius in Yamabe flow) then the `run` function will have to be modified to accommodate for this.

Revisions

subversion 659, 5/1/09, Initial `run` function uploaded to the code. subversion 679, 6/3/09, `run` function modified to work with new Geometry structure. subversion 761, 6/12/09, `run` function modified to work with new quantity structure.

Testing

The function was tested by performing two and three dimensional flows on familiar triangulations. The start and end values for radii and curvature was then compared with our expected values. The expected values were obtained from the earlier curvature flows we had (see). We also checked that the end values were within the precision and accuracy bounds when they were in effect.

Future Work

No future work is planned at this time, though possible changes would include providing for more general systems to be able to run. This would involve changing the way precision and accuracy are determined as well as what values are recorded.

`Geoquant::At`

Function Prototype

```
Geoquant* Geoquant::At(Simplex s1, ...)
```

Key Words

geoquant, recalculate, dependent, triposition, simplex

Authors

Joseph Thomas

Introduction

The `At` function is defined for every type of geoquant as a way to retrieve that quantity. Once the quantity is retrieved, a value can be set or asked of the quantity. A quantity is retrieved by providing a list of simplices that describe the position of the quantity in the triangulation.

Subsidiaries

Functions: getSerialNumber
Global Variables: map
Local Variables: possible list of simplices

Description

The `At` function is a little different for every type of geoquant, but in all cases it is a static function for that class that serves as an object retrieval in place of a constructor. The function takes as a parameter a list of simplices which may be different for each type of geoquant. The list is the natural description of where the quantity is in the triangulation. For example, a radius is described by a vertex, whereas an angle is described as a vertex on a certain face. The `At` function returns a pointer to the requested quantity.

When the `At` function is called, it searches a local map for a quantity with the given list of simplices. If it is found, a pointer to that quantity in the map is simply returned. If it is not found, the quantity is constructed and placed into the map. If the construction of

the object requires other types of quantities not yet created, then these will be constructed automatically at this time. Lastly, this quantity is returned.

The constructor is hidden from the user for several reasons. The first is that this avoids redundant construction and the need for an encapsulating object to hold a large set of geoquants (like the Geometry class in a previous version). In the same vein, the need for an initial build step and a required order of cunstruction is removed. In addition, this is an efficiency improvement as quantities that are never requested are never created, decreasing memory use and large dependency trees which can take a while for an invalidate to traverse.

Practicum

Example:

```
// Get the length value from the first edge in the triangulation.
Length *l = Length::At(Triangulation::edgeTable[0]);

// Get the angle of vertex v incident on face f
Vertex v;
Face f;
...
EuclideanAngle *ang = EuclideanAngle::At(v, f);
```

Limitations

The `At` function is limited in that a specific set of simplices will always return the exact same object. While this is in fact the design goal, this can limit one's ability to modify an object as a change in one place will affect its use elsewhere in the code. The function also will require the user to handle pointers, a powerful yet fragile and sometimes daunting aspect of the programming language.

Revisions

subversion 761, 6/12/09, A working copy of `At` and the Geoquant system.

Testing

The `At` function was tested in small modularized systems, then tested in a three dimensional flow, which required many varied uses of `At`. Some retrieved quantities had their values set while others had their values accessed and compared with what mathematica calculations predicted.

Future Work

No future work is planned at this time.