GEOCAM
Geometric Evolutions on Computational Abstract Manifolds

Daniel Champion, David Glickenstein, Yuliya Gorlina,
Alex Henniges, Kurtis Norwood, Jeff Taft,
Joseph Thomas, Andrea Young

Summer 2009

Contents

Preface

This is the preface. It is an unnumbered chapter. The `markboth` TeX field at the beginning of this paragraph sets the correct page heading for the Preface portion of the document. The preface does not appear in the table of contents.

Part I

Introductory Information

Chapter 1

Project Introduction

Chapter 2

The C++ Language

Working with a "memoized-pipeline" data structure (WORK IN PROGRESS)

Key Words

geometry, memoized-pipeline, extending, modifying, data structure, geoquant, quantities, singleton, observer, observable

Authors

- Alex Henniges
- Joseph Thomas

Introduction

The memoized-pipeline is a data structure we developed for investigating geometries defined on triangulations. It is particularly suited to the situation in which we need to specify the values of some geometric quantities (independent variables) and then need to rapidly calculate the values of some other quantities (the dependent variables). Basically, we achieve this speedup by trading space for time. Usually, the definitions of the dependent variables have many intermediate values in common. By saving these values the first time we compute them, and then reusing them later, we can avoid a lot of useless recalculation. This strategy of saving calculated values, which can be found in most algorithms textbooks, is called "memoization."

In implementing various geometries, we have already developed code and techniques for making memoization an automatic part of encoding a geometry. In this tutorial, we describe how to take advantage of this existing code.

Implementation Details

The underlying implementation of the pipeline is designed to solve two problems in a fairly user-friendly way:

1. We would like to be able to identify geometric quantities with positions on the triangulation. For example, we can speak of the dihedral angle associated with a particular edge on a tetrahedron. We would like to be able to write code in the same way.
2. We would like memoization to be nearly automatic. In other words, when writing a particular quantity, the programmer shouldn't have to think much about what happens to memoize that quantity's value.

Taking the programmer's perspective, we can view quantities as being specified by 3 pieces of information:

1. A position on the triangulation.
2. A definition of the other quantities (if any) needed to calculate the value of the current quantity, and where those quantities can be found on the triangulation.
3. A formula for calculating a quantity's value, given the values of the other quantities it depends on.

Usually, specifying just these 3 pieces of information is enough to create a new type of quantity. To help speed the development of quantities, we have developed a Ruby script, `makeQuantity.rb`, that generates much of the source code. This can be invoked at the command line as follows:

```
{\small > ruby makeQuantity.rb [quantity]
}
```

This produces two files, [`quantity`]`.h` and [`quantity`]`.cpp`.

2.0.1 The "anatomy" of `quantity.h`

In `C++`, header files serve several purposes. Among other uses, a header file can:

- Specify dependencies on other parts of the project.
- Define an interface for other parts of your project to use. This includes:
  - Definitions for new data-types (like classes).
  - Definitions for procedure calls (what arguments a procedure takes, and what it returns).

By default, `makeQuantity.rb` gives you the following header file to use (here, we chose `quantity/QUANTITY` as the quantity name, in practice, this is filled out by the script).

```
{\small #ifndef QUANTITY_H_
}
{\small #define QUANTITY_H_
}

{\small #include "geoquant.h"
}

{\small /******************REGION 1******************
}
{\small  * This is where you load the headers of the *
}
{\small  * quantities you require.                   *
}
{\small  ******************************************/
}

{\small class quantity : public virtual GeoQuant {
}
{\small protected:
}
{\small   quantity( SIMPLICES );
}
{\small   void recalculate();
}
{\small   /****************REGION 2********************
}
{\small    * The quantity references you need go here. *
}
{\small    ******************************************/
}

{\small public:
}
{\small   ~quantity();
}
{\small   static quantity* At( SIMPLICES );
}
{\small   static void CleanUp();
}
{\small };
}
{\small #endif /* QUANTITY_H_ */
}
```

The two important areas of the header are labeled `REGION 1` and `REGION 2`. In region 1, you specify the header files for the quantities and utilities you use in the rest of your quantity. These `#include` statements can be thought of as providing definitions for the data and procedures you want to use in building your quantity. In region 2, you specify the data associated with a given instance of the quantity; typically this amounts to several references to other quantities, or a data structure that manages references to other quantities. Lastly, you will need to modify the region tagged `SIMPLICES` so that it reflects a collection of simplices that describe your quantity's position on the triangulation.

2.0.2 The "anatomy" of `quantity.cpp`

In general, a `.cpp` file provides the internal implementation to support the operations described in the corresponding header file. Editing this file will be a little more complicated. By default, `makeQuantity.rb` will produce the following `.cpp` file:

```
{\small #include "quantity.h"
}

{\small #include <map>
}
{\small #include <new>
}
{\small using namespace std;
}

{\small
#define map<TriPosition, quantity*, TriPositionCompare> quantityIndex
}
{\small static quantityIndex* Index = NULL;
}

{\small quantity::quantity( SIMPLICES ){
}
{\small   /* REGION 1 */
}
{\small }
}

{\small quantity::~quantity(){
}
{\small   /* REGION 2 */
}
{\small }
}
```

```
{\small void quantity::recalculate(){
}
{\small   /* REGION 3 */
}
{\small }
}

{\small quantity* quantity::At( SIMPLICES ){
}
{\small   TriPosition T( NUMSIMPLICES, SIMPLICES );
}
{\small   if( Index == NULL ) Index = new quantityIndex();
}
{\small   quantityIndex::iterator iter = Index->find( T );
}

{\small   if( iter == Index->end() ){
}
{\small     quantity* val = new quantity( SIMPLICES );
}
{\small     Index->insert( make_pair( T, val ) );
}
{\small     return val;
}
{\small   } else {
}
{\small     return iter->second;
}
{\small   }
}
{\small }
}

{\small void quantity::CleanUp(){
}
{\small   if( Index == NULL ) return;
}
{\small   quantityIndex::iterator iter;
}
{\small   for(iter = Index->begin(); iter != Index->end(); iter++)
}
{\small     delete iter->second;
}
{\small   delete Index;
}
```

```
{\small }
}
```

There are a few smaller areas to fill out, but in general defining the quantity requires the following three definitions:

- Region 1 specifies how to obtain references on the quantities your quantity depends on. Typically, this will involve using the input simplex information and some utilities for inspecting the triangulation to look up the quantities needed for later calculations.
- Region 2 specifies how to release any data structures built up using dynamic memory. In many cases, this field will be left blank.
- Region 3 specifies how to calculate the value of an instance of the quantity. Typically, this will occur in two steps:
  1. Using the quantity references obtained in region 1, we acquire the current values of the quantities used in the calculation.
  2. Using a formula and the values found in step 1, we calculate the value of the current quantity.

An Extended Example

Perhaps the easiest way to understand the system is by examining a few working quantities. In this example, we consider the code written to represent the "Dual Area Segment¨ quantity discussed in ¨Discrete conformal variations and scalar curvature on piecewise flat two and three dimensional manifolds"[1] (in the paper, this quantity is also called $A_{ij,kl}$).

PICTURES AND A PROSE DESCRIPTION GO HERE

```
{\small #ifndef DUALAREASEGMENT_H_
}
{\small #define DUALAREASEGMENT_H_
}

{\small #include "geoquant.h"
}
{\small #include "triposition.h"
}

{\small #include "edge_height.h"
}
{\small #include "face_height.h"
}
```

[1] See URL http://arxiv.org/abs/0906.1560

```
{\small class DualAreaSegment : public virtual GeoQuant {
}
{\small private:
}
{\small   EdgeHeight* hij_k;
}
{\small   EdgeHeight* hij_l;
}
{\small   FaceHeight* hijk_l;
}
{\small   FaceHeight* hijl_k;
}

{\small protected:
}
{\small   DualAreaSegment( Edge& e, Tetra& t );
}
{\small   void recalculate();
}

{\small public:
}
{\small   ~DualAreaSegment();
}
{\small   static DualAreaSegment* At( Edge& e, Tetra& t );
}
{\small   static void CleanUp();
}
{\small   static void Record( char* filename );
}
{\small };
}

{\small #endif /* DUALAREASEGMENT_H_ */
}

{\small #include "dualareasegment.h"
}
{\small #include "miscmath.h"
}

{\small #include <stdio.h>
}

{\small
typedef map<TriPosition, DualAreaSegment*, TriPositionCompare> DualAreaSegmentIndex;
```

```
}
{\small static DualAreaSegmentIndex* Index = NULL;
}

{\small DualAreaSegment::DualAreaSegment( Edge& e, Tetra& t ){
}
{\small
  StdTetra st = labelTetra( t, e );  // We use the topological tools in miscmath to
}
{\small
                                     // label the tetrahedron with respect to edge e.
}

{\small   Face& fa123 = Triangulation::faceTable[ st.f123 ];
}
{\small   Face& fa124 = Triangulation::faceTable[ st.f124 ];
}

{\small
  hij_k = EdgeHeight::At( e, fa123 );  // Here we use the calculated topological values
}
{\small
  hij_l = EdgeHeight::At( e, fa124 );  // to look up the edge and face heights required
}
{\small
  hijk_l = FaceHeight::At( fa123, t ); // to calculate the dual area.
}
{\small   hijl_k = FaceHeight::At( fa124, t );
}

{\small
  hij_k->addDependent(this);  // Here we notify the quantities we reference
}
{\small
  hij_l->addDependent(this);  // that we wish to observe them (this is the
}
{\small
  hijk_l->addDependent(this); // where an important part of our observer-
}
{\small   hijl_k->addDependent(this); // observable design is implemented).
}
{\small }
}

{\small
DualAreaSegment::~DualAreaSegment(){} // We didn't allocate any memory to store
```

```
}
{\small
                                        // this quantity's data, so the destructor
}
{\small                                         // can be left blank.
}

{\small void DualAreaSegment::recalculate(){
}
{\small
  double Hij_k = hij_k->getValue();   // Step 1: We use the quantity references
}
{\small
  double Hijk_l = hijk_l->getValue(); // to obtain correct values for the referenced
}
{\small   double Hij_l = hij_l->getValue();   // quantities.
}
{\small   double Hijl_k = hijl_k->getValue();
}

{\small
  // Step 2: We use a formula to calculate the value of the dual-area segment.
}
{\small   value = 0.5*(Hij_k * Hijk_l + Hij_l * Hijl_k);
}
{\small }
}

{\small DualAreaSegment* DualAreaSegment::At( Edge& e, Tetra& t ){
}
{\small   TriPosition T( 2, e.getSerialNumber(), t.getSerialNumber() );
}
{\small   if( Index == NULL ) Index = new DualAreaSegmentIndex();
}
{\small   DualAreaSegmentIndex::iterator iter = Index->find( T );
}

{\small   if( iter == Index->end() ){
}
{\small     DualAreaSegment* val = new DualAreaSegment( e, t );
}
{\small     Index->insert( make_pair( T, val ) );
}
{\small     return val;
}
{\small   } else {
```

```
}
{\small     return iter->second;
}
{\small   }
}
{\small }
}

{\small void DualAreaSegment::CleanUp(){
}
{\small   if( Index == NULL ) return;
}
{\small   DualAreaSegmentIndex::iterator iter;
}
{\small   for(iter = Index->begin(); iter != Index->end(); iter++)
}
{\small     delete iter->second;
}
{\small   delete Index;
}
{\small }
}
```

Common Mistakes

A panoply of debugging hints should go here.

Fancier Tricks

Techniques for less common quantities go here.

Limitations, Areas to Improve

Discussion of the current topological assumptions our code makes.

Part II

System Documentation

Chapter 3

System

Chapter 4

Subsystems

Chapter 5

Functions

Aij_kl

Function Prototype

`double Aij_kl( Vertex vi, Vertex vj, Vertex vk, Vertex vl)`

Key Words

Dual area, curvature, partial derivative, Einstein-Hilbert-Regge.

Authors

Daniel Champion

Introduction

`Aij_kl` calculates the dual area to an edge of a tetrahedron.

Subsidiaries

Functions:

```
hij_k
    dij
    Geometry::angle
hijk_l
    Geometry::dihedralAngle
```

Global Variables: radii, etas

Local Variables: none.

Description

`Aij_kl` is calculated with the formula:

$$\texttt{Aij\_kl (vi, vj, vk, vl)} = \frac{1}{2}\left(\begin{array}{l} \texttt{hij\_k(vi,vj,vk)} \cdot \texttt{hijk\_l(vi,vj,vk,vl)} \\ +\texttt{hij\_k(vi,vj,vl)} \cdot \texttt{hijk\_l(vi,vj,vl,vk)} \end{array} \right).$$

`hij_k` and `hijk_l` are calculated with the following formulae:

$$\texttt{hij\_k (vi, vj, vk)} = \frac{(\texttt{dij(vi,vk)} - \texttt{dij(vi,vj)}\cos(\alpha_{i,jk}))}{\sin(\alpha_{i,jk})}$$

$$\texttt{hijk\_l (vi, vj, vk, vl)} = \frac{\left(\texttt{hij\_k(vi,vj,vl)} - \texttt{hij\_k(vi,vj,vk)}\cos(\beta_{ij,kl})\right)}{\sin\left(\beta_{ij,kl}\right)}$$

where $\alpha_{i,jk}$ is the angle at vertex vi of triangle $\{vi, vj, vk\}$, and $\beta_{ij,kl}$ is the dihedral angle along edge $\{vi, vj\}$ of tetrahedron $\{vi, vj, vk, vl\}$ (implemented with the functions `Geometry::angle` and `Geometry::dihedralAngle` respectively).

`Aij_kl` was created for the calculation performed in the function `Lij_star`, which is used in the computation of the partial derivatives of curvature. These partial derivatives of curvature are used in the calculation of the second order partial derivatives of the Einstein-Hilbert-Regge functional for use in the optimization of said functional using Newton's method.

Practicum

As an example of the usage of this function, we will calculate the dual area to the edge $eij = \{vi, vj\}$ (see entry: `Lij_star`). To do this, we will sum the dual areas to each tetrahedron containing the edge eij.

```
vector<int> sum_over = *(eij.getLocalTetras());
double sum = 0.0;
vector<int> T_vertices, e_vertices;
Tetra T;
Vertex vi,vj,vk,vl;
for(i=0; i<sum_over.size(); ++i) {
    T = Triangulation::tetraTable[sum_over[i]];
    T_vertices = *(T.getLocalVertices());
    e_vertices = *(eij.getLocalVertices());
    vi = Triangulation::vertexTable[e_vertices[0]];
    vj = Triangulation::vertexTable[e_vertices[1]];
    vk = Triangulation::vertexTable[listDifference(&T_vertices, &e_vertices)[0]];
    vl = Triangulation::vertexTable[listDifference(&T_vertices, &e_vertices)[1]];
    sum += Aij_kl(vi, vj, vk, vl);
    }
return sum;
```

Limitations

`Aij_kl` if fully operational and has no known limitations. The function will output appropriate values provided it receives as input four distinct vertices that define a tetrahedron.

Revisions

subversion 757, 7/8/09, `Aij_kl` created.

Testing

Trials were run and the calculations returned were verified by hand.

Future Work

No future work planned.

Approximator::run

```
{\small int run(int numsteps, double stepsize)
}
{\small int run(double precision, double stepsize)
}
{\small int run(double precision, int maxNumSteps, double stepsize)
}
{\small
}
```

Keywords

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 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. The `run` function returns 0 on success or any other number if an error is encountered.

Subsidiaries

Functions:

- `Approximator::step`
- `Approximator::isPrecise`
- `Approximator::isAccurate`
- `Approximator::getLatest`

1. `Approximator::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, 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 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. 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).

A flow can also be run with a precision and a max number of steps that will stop once one of the conditions is reached. The last parameter of any run indicates the step size of the flow. A lower step size will lead to more accurate steps, but a longer time to convergence.

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:

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

{\small // Run a Yamabe flow for 300 steps with a stepsize of 0.01.
}
{\small app->run(300, 0.01);
}
{\small // Run with a precision bound of 0.000001 and a stepsize of 0.01
}
{\small app->run(0.000001, 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 is used, expects the values to converge. If a precision 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.
- subversion 787, 6/18/09: Added new `run` options to approximator. Removed accuracy. Checks for bad numbers.

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 [#Description Description] above). We also checked that the end values were within the precision and accuracy bounds when they were in effect.

Future Work

- 6/17 - Add more run options (ex. precision and maxNumSteps). **Complete (6/18)**
- 6/17 - Have a run stop the moment an undefined number appears. **Complete (6/18)**

No future work is planned at this time.

Curvature_Partial

Function Prototype

```
double Curvature_Partial ( int i, int l )
```

Key Words

Curvature, Einstein-Hilbert-Regge, functional, partial derivative

Authors

Daniel Champion

Introduction

Curvature_Partial calculates the partial derivative of the curvature at a vertex with respect to the log radius of another (possibly the same) vertex.

Subsidiaries

Functions:

```
isAdjVertex
Lij_star
    Aij_kl
        hijk_l
            hij_k
                dij
listDifference
listIntersection
```

Global Variables: curvature, dihedralAngle, eta, length, radius
Local Variables: none.

Description

`Curvature_Partial` receives as inputs two integers i and l. The first corresponds to the vertex of interest, the second corresponds to the vertex of differentiation. That is,

$$\texttt{Curvature\_Partial (i, l)} = \frac{\partial}{\partial \log r_l} K_i,$$

where r_l is the radius at vertex l, and K_i is the curvature at vertex i.

The function begins implementation by determining the relationship between i and l via the trichotomy $v_i = v_l$, v_i is adjacent to v_l, or v_i and v_l are not endpoints of any edge. Each of the three cases are calculated differently. The general formula for the variation of curvature w.r.t. log radii was calculated by Prof. David Glickenstein and is available at arXiv:0906.1560v1:

$$\delta K_i = -\sum_{edges\ \{i,j\}} \left(2\frac{l^*_{ij}}{l_{ij}} - \frac{r_i^2 r_j^2 \left(1-\eta_{ij}^2\right)}{l_{ij}^2} K_{ij} \right) (\delta f_j - \delta f_i) + K_i \delta f_i,$$

where $f_i = \log r_i$, l_{ij} is the length of the edge $\{i,j\}$, and l^*_{ij} is the dual area calculated with the function `Lij_star`.

When $i = l$, the formula for the partial derivative $\frac{\partial}{\partial \log r_l} K_i$ becomes:

$$\frac{\partial}{\partial \log r_i} K_i = \sum_{edges\ \{i,j\}} \left(2\frac{l^*_{ij}}{l_{ij}} - \frac{r_i^2 r_j^2 \left(1-\eta_{ij}^2\right)}{l_{ij}^2} K_{ij} \right) + K_i.$$

When v_i is adjacent to v_l only one term in the sum survives:

$$\frac{\partial}{\partial \log r_l} K_i = -\left(2\frac{l^*_{il}}{l_{il}} - \frac{r_i^2 r_l^2 \left(1-\eta_{il}^2\right)}{l_{il}^2} K_{il} \right).$$

When v_i and v_j are not endpoints of any edge the partial derivative is zero.

This function was created to assist in the computation of the second derivatives of the normalized Einstein-Hilbert-Regge functional:

$$EHR = \frac{\sum K_i}{\sqrt[3]{\sum_{tetra\ t} Vol(t)}}.$$

Surprisingly the first derivatives of the normalized EHR functional do not require the formula for the partial derivative of curvature since

$$\frac{\partial EHR}{\partial \log r_i} = K_i.$$

However, the second order partial derivatives of EHR certainly require the formulas for $\frac{\partial}{\partial \log r_l} K_i$ given above. These second order partial derivatives are used to construct a Hessian matrix which is then used the optimization of the EHR functional using Newton's method (implemented by `Newtons_Method`).

Practicum

When called, `Curvature_Partial (i, l)` returns the partial derivative $\frac{\partial}{\partial \log r_l} K_i$. An example of its usage is in the calculation of the second order partial derivatives of the normalized EHR functional. For this example let

$$\begin{aligned}
\texttt{VolSumPartial_i} &= \sum_{tetra\ t} \frac{\partial V_t}{\partial \log r_i}, \\
\texttt{VolSumPartial_j} &= \sum_{tetra\ t} \frac{\partial V_t}{\partial \log r_j}, \\
\texttt{VolSumSecondPartial} &= \sum_{tetra\ t} \frac{\partial^2 V_t}{\partial \log r_i \partial \log r_j} \\
K &= \sum_i K_i, \\
V &= \sum_{tetra\ t} V_t.
\end{aligned}$$

Then the second order partial derivative $\frac{\partial^2 EHR}{\partial \log r_i \partial \log r_j}$ is calculated by:

```
result = pow(V, (-4.0/3.0))*(1.0/3.0)*(3*V*Curvature_Partial(i,j)
        -Geometry::curvature(Triangulation::vertexTable[i])*VolSumPartial_j
        -Geometry::curvature(Triangulation::vertexTable[j])*VolSumPartial_i
        +(4.0/3.0)*K*pow(V, -1.0)*VolSumPartial_i*VolSumPartial_j
        -K*VolSumSecondPartial);
```

Limitations

The function `Curvature_Partial` is operational for all pairs of input integers i and l that are in the vertex table. If one of the arguments is not in the vertex table, the function will output zero.

Revisions

subversion 757, 7/6/09, `Curvature_Partial` created.

Testing

This function has not been tested.

Future Work

Using the calculation of the partial derivative of curvature in Mathematica, it should be compared to the output from `Curvature_Partial`.

dij

Function Prototype

```
double dij( Vertex vi, Vertex vj)
```

Key Words

Partial length

Authors

Daniel Champion, ???

Introduction

The function `dij` calculates the distance from a vertex to the center of an edge (as determined by the center of a decorated triangle).

Subsidiaries

Functions:

`Geometry::length`
`listIntersection`

Global Variables: radii, etas.
Local Variables: Vertex vi, vj.

Description

The function `dij` is calculated with the simple formula:

$$\texttt{dij(vi,vj)} = \frac{L_{ij}^2 + r_i^2 - r_j^2}{2L_{ij}},$$

where r_i, r_j are the radii at vertices `vi`, and `vj` respectively, and L_{ij} is the length of the edge $\{vi, vj\}$. Notice that this formula is not symmetric in i and j.

This function plays an important role in several areas of the project including curvature, Dirichlet energy, and the optimization of the Einstein-Hilbert-Regge functional. `dij` is used in the calculation of several quantities used in the implementation of the `Curvature_Partial` function, which is used in the optimization of the normalized Einstein-Hilbert-Regge functional.

Practicum

An example of the use of this function is in the calculation of the edge height function `hij_k`:

```
       double hij_k( Vertex vi, Vertex vj, Vertex vk) {
           Face fijk;
           vector<int> temp_ij =
               listIntersection(vi.getLocalFaces(), vj.getLocalFaces());
           vector<int> temp =
               listIntersection( &temp_ij, vk.getLocalFaces());
           fijk = Triangulation::faceTable[temp[0]];
           double result = (dij(vi, vk)-dij(vi,vj)
               *cos(Geometry::angle(vi, fijk)))/sin(Geometry::angle(vi,
fijk));
           return result;
           }
```

Limitations

`dij` must receive as input two vertices that define an edge in the triangulation. `dij` returns distinct values for each permutation of the input vertices.

Revisions

subversion 757, 7/13/09, `dij` created.

Testing

`dij` was tested by working out several examples by hand.

Future Work

This function has been added to the Geometry class geoquants, and thus this entry needs to be updated.

EHR

Function Prototype

```
double Example_Function (Vertex, Edge1, Edge2, double, int,...)
```

Key Words

Enter all key words to this function here.

Authors

Type the primary authors here. Example:
Daniel "Cliff Jumper" Champion

Introduction

In this space provide a brief introduction to familiarize the reader with the function listed above.

Subsidiaries

List all functions used by this function; list all variables (and types) used by this function. Indent to show hierarchy.

Functions:
- Function1
- Function2
  - Function2.1
  - Function2.2
    - Function2.3

Global Variables:
Local Variables:

Description

Begin this section with a detailed description of the function. For simple functions provide sufficient theory to define the function, otherwise outline the theory and cite "calculations were performed in Mathematica..." etc. if applicable.

Conclude this section with an explanation of why this function exists. This would include initial motivation for the creation of the function, as well as all

primary programs (functions) that utilize the function. A brief history of the function can also be given if it serves to explain why the function exists.

Practicum

Place any and all practical information about the function in this section. Provide a short example of the use of this function if appropriate. This should be written in code or pseudo-code written in the format below:

```
begin repeat;
    result = n+5;
    end if result > 5;
    n=n+1;
end repeat;
```

Limitations

Provide a description of the limitations of the function. It should be clear what works and doesn't work about the function from reading this section.

Revisions

List the major revisions to the function with dates and a one sentence comment. Example:

subversion 617, 6/8/09, `Example_Function` created with severe limitations.

subversion 618, 6/9/09, `Example_Function` was fully commented and initial testing complete.

subversion 619, 6/10/09, `Example_Function` was augmented to utilize the Geometry class.

Testing

Describe how the function was tested. Include dates and names of test results if possible.

Future Work

In this section, describe what changes or increased functionality are desired for this function. It may be helpful to address some of the items listed in the "Limitations" section.

EHR_Partial

Function Prototype

```
double EHR_Partial (int i)
```

Key Words

Einstein-Hilbert-Regge, functional, Newton's method, partial derivative

Authors

Daniel Champion

Introduction

EHR_Partial calculates the partial derivative of the normalized Einstein-Hilbert-Regge functional with respect to log radii.

Subsidiaries

Functions:

- Total_Volume
- Total_Curvature
- Volume_Partial
  - listDifference
  - listIntersection

Global Variables: radii, etas, curvature, volume

Local Variables: none

Description

The normalized Einstein-Hilbert-Regge functional is given by the expression:

$$EHR = \frac{\sum\limits_{j} K_j}{\sqrt[3]{\sum\limits_{tetra\ t} V_t}},$$

where K_i is the curvature at vertex j, and V_t is the volume of tetrahedron t. It can be shown (see arXiv:0906.1560v1) that

$$\frac{\partial}{\partial \log r_i}\left(\sum_j K_j\right) = K_i,$$

hence the partial derivative of the normalized EHR functional becomes:

$$\frac{\partial}{\partial \log r_i} EHR = \frac{K_i \sqrt[3]{\sum_{tetra\ t} V_t} - \frac{1}{3}\left(\sum_{tetra\ t} V_t\right)^{-\frac{2}{3}} \sum_{tetra\ t} \frac{\partial V_t}{\partial \log r_i} \sum_j K_j}{\left(\sum_{tetra\ t} V_t\right)^{\frac{2}{3}}}$$
$$= V^{-\frac{4}{3}} \left(K_i V - \frac{1}{3} K \sum_{tetra\ t} \frac{\partial V_t}{\partial \log r_i} \right),$$

where V is the total volume of all tetrahedra in the triangulation and K is the sum of the curvatures over all vertices in the triangulation. `EHR_Partial (i)` calculates $\frac{\partial}{\partial \log r_i} EHR$.

The primary use of this function is in the calculation of the negative gradient of the EHR functional for use in optimization of the functional using Newton's method. The formula for $\frac{\partial}{\partial \log r_i} EHR$ given above was also used in the calculation of the second order partial derivatives of the EHR functional, implemented in `EHR_Second_Partial`.

Practicum

As an example of the use of this function, the calculation of the gradient of the EHR functional will be calculated. The negative gradient will be outputted as the array `EHRneg_gradient`.

```
double EHRneg_gradient[Triangulation::vertexTable.size()];
for(int i=0; i < Triangulation::vertexTable.size(); ++i) {
    EHRneg_gradient[i] = -1.0*EHR_Partial(i+1);
    }
```

Limitations

The function `EHR_Partial` is fully functional with no known limitations. It will return appropriate values so long as it is called with an integer in the vertex table.

Revisions

List the major revisions to the function with dates and a one sentence comment. Example:

subversion 757, 7/7/09, `EHR_Partial` created.

Testing

This function has not been tested.

Future Work

A testing regime should be instituted for this function.

EHR_Second_Partial

Function Prototype

```
double EHR_Second_Partial (int i, int j)
```

Key Words

Einstein-Hilbert-Regge, functional, partial derivative, Hessian.

Authors

Daniel Champion

Introduction

`EHR_Second_Partial` calculates the second order partial derivatives of the normalized Einstein-Hilbert-Regge functional with respect to log radii.

Subsidiaries

Functions:

```
Curvature_Partial
    isAdjVertex
    Lij_star
        Aij_kl
            hijk_l
                hij_k
                    dij
    listDifference
    listIntersection
Total_Curvature
    Geometry::curvature
Total_Volume
    Geometry::volume
Volume_Partial
    listDifference
    listIntersection.
Volume_Second_Partial
```

Global Variables: radii, etas.
Local Variables: none.

Description

The normalized Einstein-Hilbert-Regge functional is given by the expression:

$$EHR = \frac{\sum\limits_{j} K_j}{\sqrt[3]{\sum\limits_{tetra\ t} V_t}},$$

where K_i is the curvature at vertex j, and V_t is the volume of tetrahedron t. It can be shown (see arXiv:0906.1560v1) that

$$\frac{\partial}{\partial \log r_i}\left(\sum_j K_j\right) = K_i,$$

hence the partial derivative of the normalized EHR functional simplifies to become:

$$\frac{\partial}{\partial \log r_i} EHR = V^{-\frac{4}{3}}\left(K_i V - \frac{1}{3} K \sum_{tetra\ t} \frac{\partial V_t}{\partial \log r_i}\right),$$

where V is the total volume of all tetrahedra in the triangulation and K is the sum of the curvatures over all vertices in the triangulation. Differentiating this result with respect to $\log r_j$ yields:

$$\frac{\partial^2}{\partial \log r_i \partial \log r_j} EHR = V^{-\frac{4}{3}}\left(\begin{array}{c} V\frac{\partial K_i}{\partial \log r_j} - \frac{1}{3}K_i \sum\limits_t \frac{\partial V_t}{\partial \log r_j} - \frac{1}{3}K_j \sum\limits_t \frac{\partial V_t}{\partial \log r_i} \\ +\frac{4}{9}KV^{-1}\sum\limits_t \frac{\partial V_t}{\partial \log r_j}\sum\limits_t \frac{\partial V_t}{\partial \log r_i} - \frac{1}{3}K\sum\limits_t \frac{\partial^2 V_t}{\partial \log r_i \partial \log r_j} \end{array}\right).$$

When called, `EHR_Second_Partial` calculates the formula above, that is:

$$\texttt{EHR\_Second\_Partial (i,j)} = \frac{\partial^2}{\partial \log r_i \partial \log r_j} EHR.$$

The use of this function is in the population of the Hessian matrix for the normalized EHR functional. This Hessian matrix is used in the optimization of the EHR functional using Newton's method.

Practicum

As an example of the usage of `EHR_Second_Partial`, the Hessian matrix of the normalized EHR functional will be populated. In this example, the Hessian matrix is the array EHRhessian. The example reduced computation time by only calling `EHR_Second_Partial` for the upper triangular portion of the EHRhessian array, and symmetrically copies the entries above the diagonal to the corresponding location below the diagonal. Furthermore, in C++ arrays begin indexing at zero, however the vertices of the triangulations begin indexing at 1, requiring a shift of one in the population step. Note that triangulations that do not label the vertices consecutively will not be compatible with the following code.

```
      double EHRhessian[Triangulation::vertexTable.size()][Triangulation::vertexTable.size()];
      for(int i = 0; i < Triangulation::vertexTable.size(); ++i) {
          for(int j = 0; j < Triangulation::vertexTable.size(); ++j)
{
              if (i <= j) {
                  EHRhessian[i][j]=EHR_Second_Partial( i+1 , j+1 );
                  EHRhessian[j][i]=EHRhessian[i][j];
                  }
              }
          }
```

Limitations

`EHR_Second_Partial` is fully operational with no known limitations. The function will output appropriate values provided it receives as inputs a pair of integers in the vertex table.

Revisions

subversion 757, 7/7/09, `EHR_Second_Partial` created.

Testing

This function has not been tested.

Future Work

A testing regime should be instituted for this function.

flip

```
{\small Edge flip(Edge e)
}
```

Keywords

flip, delaunay

Authors

Kurt Norwood

Introduction

The `flip` function takes a single Edge as a parameter and performs a flip on it. This involves determining the new length of the edge after the flip and changing the topological information in the edge being flipped as well as all of the edge's adjacent simplices. This can be thought of as taking two triangles which share an edge (the parameter to flip) and making two new triangles which share an edge between the two vertices which were previously non-adjacent.

Subsidiaries

Functions:

```
{\small     void flipPP(struct simps b)
}

{\small     void flipPN(struct simps b)
}

{\small     void flipNN(struct simps b)
}

{\small     void topo_flip(Edge, struct simps)
}

{\small     bool prep_for_flip(Edge, struct simps*)
}
```

Global Variables:Local Variables:

```
{\small     Edge e
}
```

Description

flip begins by calling the prep_for_flip function, that will setup the struct given to it to contain all the important information necessary for the flip to occur, such as indices for the different simplices and the lengths of the triangles' edges, and the two angles which are not incident on the edge being flipped. The struct looks like:

```
{\small struct simps {
}
{\small        int v0, v1, v2, v3, e0, e1, e2, e3, e4, f0, f1;
}
{\small        double e0_len, e1_len, e2_len, e3_len, e4_len;
```

```
}
{\small        double a0, a2;
}
{\small };
}
```

With all this information known, the next step is to determine the type of flip that is to occur. The possibilities are broken up three ways: positive positive (PP), positive negative (PN), negative negative (NN); based on the initial condition of the two triangles. This will determine which of flipPP, flipPN, flipNN is called. Within these function is logic which should compute the new edge length and assign it to the edge e, and determine the positive/negative configuration of the two triangles and assign the appropriate boolean value to each face.

With the new edge length computed and assigned, the topo_flip function is called which performs the rearrangement of all the adjacencies of the different simplices which are adjacent to edge e.

The edge is returned.

Practicum

```
{\small   Edge e;
}
{\small   e = Triangulation::edgeTable[indexOfE];
}
{\small   e = flip(e);
}
```

one thing to note is that in future implementations the edge being given as the parameter may be different than the one returned

Limitations

The biggest limitation of the flip function is that it currently only works for bistellar flips. If higher dimensional flips are required this function will need to be modified heavily.

Revisions

--r816 | kortox | 2009-06-29 12:41:33 -0700 (Mon, 29 Jun 2009) | 1 line

have all the new_flip stuff up to date and working with the new geometry classes

--r795 | kortox | 2009-06-18 17:58:30 -0700 (Thu, 18 Jun 2009) | 5 lines

anyway, this is a project for devopment of the flip algorithm, so far it contains a new flip function which is intended to replace the flip function that was previously in Triangulation/triangulationmorphs.cpp

main currently contains some test functions that can be called one at a time manually and should produce output that can indicate how the flip function is performing, this testing really needs to be improved

Testing

Initially testing was done inefficiently by manually analyzing what was written by the writeTriangulationFile function. Now that we have a way to display the triangulation, we can select an edge and flip it in the display and see that the flip occurred correctly. Granted this should at sometime in the future be automated, but for now if there is an issue we can try to debug it with the display.

Future Work

- Adding the ability to flip in higher dimensions. This would involve altering the function to take a Simplex object instead of an edge so that it is more general.
- We'll most likely want to have the function add an edge to the triangulation instead of just reposition the edge given, since this will lend itself better to the possible addition of 3-1 flips. Related to this would also be changing the return type to be a vector of Simplex objects for generality's sake.
- Moving the whole thing to a different file with an appropriate name other than new_flip

Geoquant::At

```
{\small Geoquant*  Geoquant::At(Simplex  s1,  ...)
}
{\small
}
```

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: mapLocal Variables: possible list of simplices

Description

The `At` function is a little diï¬erent 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 diï¬erent 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 ï¬rst 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 construction is removed. In addition, this is an eï¬ciency 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:

```
{\small
//  Get  the  Radius  quantity  from  the  first  vertex  in  the  triangulation.
}
{\small Radius  *r  =  Radius::At(Triangulation::vertexTable[0]);
}
{\small //  Get  the  angle  of  vertex  v  incident  on  face  f
}
{\small Vertex  v;
}
{\small Face  f;
}
{\small ...
}
{\small EuclideanAngle  *ang  =  EuclideanAngle::At(v,  f);
}
{\small
}
```

Limitations

The `At` function is limited in that a speciï¬c set of simplices will always return the exact same object. While this is in fact the design goal, this can limit oneâs ability to modify an object as a change in one place will aï¬ect 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 ï¬ow, 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.

hij_k

Function Prototype

```
double hij_k( Vertex vi, Vertex vj, Vertex vk)
```

Key Words

Edge height, partial edge.

Authors

Daniel Champion

Introduction

The function `hij_k` calculates the edge height (to the center of a triangular face) of an edge in the triangulation.

Subsidiaries

Functions:

`dij`
`Geometry::angle`
`listIntersection`

Global Variables: radii, etas.
Local Variables: Vertex vi, vj, vk.

Description

The calculation of `hij_k` involves the simple formula:

$$\texttt{hij\_k (vi, vj, vk)} = \frac{(\texttt{dij(vi,vk)} - \texttt{dij(vi,vj)}\cos(\alpha_{i,jk}))}{\sin(\alpha_{i,jk})},$$

where $\alpha_{i,jk}$ is the angle at vertex vi of triangle $\{vi, vj, vk\}$. A geometric interpretation of this quantity is a follows. Given a decorated triangle (triangle with radii and eta values assigned to the vertices and edges respectively), the center of this triangle can be calculated as the common power point of its embedding into two-dimensional Euclidean space. The perpendicular distance from this center point to the edge $\{vi, vj\}$ is exactly `hij_k (vi, vj, vk)`. Take note that the first two vertices in the function call correspond to the preferred edge, and the third vertex in the function call identifies the triangle.

A primary use of this function is in the calculation of several quantities needed for the `Curvature_Partial` function used in the optimization of the normalized Einstein-Hilbert-Regge functional.

Practicum

An example of the use of this function is in the calculation of the dual areas, `Aij_kl`, to an edge of a three dimensional triangulation.

```
double Aij_kl( Vertex vi, Vertex vj, Vertex vk, Vertex vl)
    {
    double result = 0.5*(hij_k(vi,vj,vk)*hijk_l(vi,vj,vk,vl)
        +hij_k(vi,vj,vl)*hijk_l(vi,vj,vl,vk));
    return result;
    }
```

hLimitations

`hij_k` must receive as input three vertices of a face of the triangulation. Moreover, the first two vertices in the function call identify an edge, and can be in any order, however the third vertex in the function call identifies the face and can not be permuted with the other two vertices.

Revisions

subversion 757, 6/8/09, `hij_k` created.

Testing

This function was not tested.

Future Work

This function has been incorporated into the Geometry class geoquants, and thus this entry needs to be updated.

hijk_l

Function Prototype

```
double hijk_l( Vertex vi, Vertex vj, Vertex vk, Vertex vl)
```

Key Words

Face height, edge height.

Authors

Daniel Champion

Introduction

The function `hijk_l` calculates the face height to the center of a tetrahedron.

Subsidiaries

Functions:

`Geometry::dihedralAngle`
`hij_k`
`listIntersection`

Global Variables: radii, etas.
Local Variables: Vertex vi, vj, vk, vl.

Description

The calculation of `hijk_l` involves the simple formula:

$$\texttt{hijk\_l (vi, vj, vk, vl)} = \frac{\left(\texttt{hij\_k(vi,vj,vl)} - \texttt{hij\_k(vi,vj,vk)}\cos(\beta_{ij,kl})\right)}{\sin\left(\beta_{ij,kl}\right)},$$

where $\beta_{ij,kl}$ is the dihedral angle along edge $\{vi, vj\}$ of tetrahedron $\{vi, vj, vk, vl\}$. A geometric interpretation of this quantity is a follows. Given a decorated tetrahedron (tetrahedron with radii and eta values assigned to the vertices and edges respectively), the center of this tetrahedron can be calculated as the common power point of its embedding into three-dimensional Euclidean space. The perpendicular distance from this center point to the face $\{vi, vj, vk\}$ is exactly `hijk_l (vi, vj, vk, vl)`. Take note that the first three vertices in the function call correspond to the preferred face, and the fourth vertex in the function call identifies the tetrahedron.

A primary use of this function is in the calculation of several quantities needed for the `Curvature_Partial` function used in the optimization of the normalized Einstein-Hilbert-Regge functional.

Practicum

An example of the use of this function is in the calculation of the dual areas, `Aij_kl`, to an edge of a three dimensional triangulation.

```
double Aij_kl( Vertex vi, Vertex vj, Vertex vk, Vertex vl)
    {
    double result = 0.5*(hij_k(vi,vj,vk)*hijk_l(vi,vj,vk,vl)
        +hij_k(vi,vj,vl)*hijk_l(vi,vj,vl,vk));
    return result;
    }
```

Limitations

`hijk_l` must receive as input four vertices of a tetrahedron of the triangulation. Moreover, the first three vertices in the function call identify a face and can be in any order, however the fourth vertex in the function call identifies the tetrahedron and can not be permuted with the other three vertices.

Revisions

subversion 757, 6/8/09, `hijk_l` created.

Testing

This function was not tested.

Future Work

This function has been incorporated into the Geometry class geoquants, and thus this entry needs to be updated.

Lij_star

Function Prototype

```
double Lij_star (Edge)
```

Key Words

Dual area, curvature, partial derivative, edge.

Authors

Daniel Champion

Introduction

`Lij_star` calculates the dual area of an edge.

Subsidiaries

Functions:

```
Aij_kl
       hijk_l
           hij_k
               dij
```

Global Variables:

Only radii and eta values are needed.

Local Variables:

none

Description

`Lij_star` is defined as:

$$\texttt{Lij\_star(edge (i,j))} = l^*_{ij} = \sum_{\substack{\text{all tetrahedra } (i,j,k,l) \\ \text{containing edge } (i,j).}} A_{ij,kl},$$

where $A_{ij,kl}$ is computed with the function `Aij_kl` applied to the vertices of the tetrahedron being summed over. Note that `Aij_kl` utilizes the functions `hijk_l`, `hij_l`, and `dij`, however only the radii and eta values are needed to calculate all of these quantities.

This function was created for use in the `Curvature_Partial` function which serves an essential role in calculating the second derivatives of the Einstein-Hilbert-Regge functional (`EHR_Second_Partial`). The second order partial derivatives of the EHR functional are used in the optimization of the EHR functional using Newton's method. `Lij_star` will eventually be used in the study of laplacians.

Practicum

Currently `Lij_star` is only used to calculate the partial derivative of curvature with respect to log radius. The following example calculates the partial derivative of the curvature at vertex V with respect to the log radius r_l corresponding to vertex Vprime (adjacent to V).

```
double sum = 0.0;
double dihedral_sum = 0.0;
Vprime = Triangulation::vertexTable[l];
E = Triangulation::edgeTable[listIntersection(V.getLocalEdges(),
    Vprime.getLocalEdges())[0]];
// This assumes that there is a unique edge between two vertices.
local_tetra = E.getLocalTetras();
for (int m=0; m < (*(local_tetra)).size(); ++m) {
    T = Triangulation::tetraTable[local_tetra->at(m)];
    dihedral_sum += Geometry::dihedralAngle(E,T);
}
result = Lij_star(E)/(Geometry::length(E))-(2*PI-dihedral_sum)
    *(pow(Geometry::radius(V), 2)*pow(Geometry::radius(Vprime),2)
    *(1-pow(Geometry::eta(E),2)))/pow(Geometry::length(E),3);
```

Limitations

`Lij_star` can operate on any and all edges of a 3D triangulation however it is only appropriate for triangulations where tetrahedra have distinct edges.

Revisions

Subversion 676, 5/15/09, `Lij_star` created within `Newtons_Method`.

Testing

`Lij_star` has not been tested.

Future Work

`Lij_star` should be moved to a more appropriate section of the code. A more general volume function should be created that would take any simplex object (including a boolean for dual simplices) and return the appropriate volume. This general volume function would be an excellent location for `Lij_star`. It should be tested some time as well.

makeTriangulationFile

```
{\small void makeTriangulationfile(char* fileIN, char* fileOUT)
}
```

Keywords

triangulation, Lutz, simplices

Authors

- Alex Henniges
- Mitch Wilson

Introduction

The `makeTriangulationFile` function converts a text file, given by `fileIn`, in the `Lutz` format to the standard format, printed to `fileOUT`. The file in standard format can then be read into the system to build the triangulation.

Subsidiaries

Functions:

- Pair::positionOf
- Pair::contains
- Pair::isInTuple

Global Variables:
Local Variables: `fileIN`, `fileOUT`

Description

This function is used to convert one format to another format that we consider to be the standard for reading in a triangulation. We have dubbed the format we are converting from `Lutz`. This is based on the source we retrieve this format from, http://www.math.tu-berlin.de/diskregeom/stellar/[1] .

The `Lutz` format provides a simpler interface than our standard format, and can therefore allow for a user to create a quick triangulation. The idea is to provide only the index of every vertex on each face of the triangulation. No information about edges or adjacencies need to be given. The file should begin with a "=¨ followed by ¨[¨ and ¨]"'s to contain the triangulation and each face. An example `Lutz` format for a tetrahedron is given [#Practicum below].

Note that the `makeTriangulationFile` is used only for two-dimensional triangulations, and that for three-dimensions, one should use make3DTriangulationFile.

Pracicum

```
{\small
  // Convert the tetrahedron written in Lutz format to a file in standard format.
}
{\small
  makeTriangulationFile("./tetra_lutz.txt", "./tetra_standard.txt");
}
{\small
}
{\small   // Now read in the triangulation from standard format.
}
{\small   readTriangulationFile("./tetra_standard.txt");
```

[1] See URL http://www.math.tu-berlin.de/diskregeom/stellar/

```
}
{\small
}
```

The `Lutz` format may look like:

```
{\small =[[1,2,3],[1,2,4],[2,3,4],[1,3,4]]
}
{\small
}
```

The `makeTriangulationFile` would then create a file with:

```
{\small Vertex: 1
}
{\small 2 3 4
}
{\small 1 2 4
}
{\small 1 2 4
}
{\small Vertex: 2
}
{\small 1 3 4
}
{\small 1 3 5
}
{\small 1 2 3
}
{\small Vertex: 3
}
{\small 1 2 4
}
{\small 2 3 6
}
{\small 1 3 4
}
{\small Vertex: 4
}
{\small 1 2 3
}
{\small 4 5 6
}
{\small 2 3 4
}
{\small Edge: 1
}
```

```
{\small 1 2
}
{\small 2 3 4 5
}
{\small 1 2
}
{\small Edge: 2
}
{\small 1 3
}
{\small 1 3 4 6
}
{\small 1 4
}
{\small Edge: 3
}
{\small 2 3
}
{\small 1 2 5 6
}
{\small 1 3
}
{\small Edge: 4
}
{\small 1 4
}
{\small 1 2 5 6
}
{\small 2 4
}
{\small Edge: 5
}
{\small 2 4
}
{\small 1 3 4 6
}
{\small 2 3
}
{\small Edge: 6
}
{\small 3 4
}
{\small 2 3 4 5
}
{\small 3 4
}
```

```
{\small Face: 1
}
{\small 1 2 3
}
{\small 1 2 3
}
{\small 2 3 4
}
{\small Face: 2
}
{\small 1 2 4
}
{\small 1 4 5
}
{\small 1 3 4
}
{\small Face: 3
}
{\small 2 3 4
}
{\small 3 5 6
}
{\small 1 2 4
}
{\small Face: 4
}
{\small 1 3 4
}
{\small 2 4 6
}
{\small 1 2 3
}
{\small
}
```

Limitations

The limitation with the `Lutz` format that prevents it from being considered the standard format is that the user cannot create the most general of triangulations. To be more specific, it is impossible with the `Lutz` format to specify for there to be two edges with the same vertices.

A limitation of the `makeTriangulationFile` is that its requirements are unintuitive. There should be no "=" required, for example. Another limitation is that despite collecting enough information to build the triangulation, the function instead writes this to a file,

requiring the user to subsequently call the function readTriangulationFile.

Revisions

- subversion 545, 9/29/08: Added the `makeTriangulationFile` function.

Testing

This function has been tested through frequent use.

Future Work

- 7/1 - Improve the format system.
- 7/1 - Create the triangulation without performing a conversion to another file.

NewtonsMethod::optimize

```
{\small void optimize(double initial[], double soln[])
}
```

Keywords

Newtons Method, optimize, extremum, gradient, hessian

Authors

- Alex Henniges

Introduction

The `optimize` function of the NewtonsMethod class is designed to find either a maximum or minimum of a functional near a given point.

Subsidiaries

Functions:

- `NewtonsMethod::step`

Description

The `optimize` function is called once by the user and it will continue to loop until an extremum of the functional is found. The functional is given in the constructor for NewtonsMethod. The initial point is the first parameter of the `optimize` function and the solution point is placed in the second parameter. This means that if one cannot be found, the function will loop without end. This is unlike the `step` function used within `optimize` that can also be used by a client program to gain much greater flexibility in the optimization process, such as more leeway on when to stop and allowing for data collection in between. See the `step` function for a description of how the optimization is performed.

Practicum

Example:

```
{\small     double func(double vars[]) {
}
{\small        double val = 1 - pow(vars[0], 2) / 4 - pow(vars[1], 2) / 9;
}
{\small        return sqrt(val);
}
{\small     }
}
{\small
}
{\small     NewtonsMethod *nm = new NewtonsMethod(func, 2);
}
{\small     double initial[] = {1, 1};
}
{\small     double soln[2];
}

{\small     nm->optimize(initial, soln);
}
{\small
}
```

Limitations

The `optimize` function is limited in its termination condition. This must be a constant over any use of the `optimize` function. It is also limited in that it may not terminate at all and the user will be forced to quit the program. Instead of modifying the function,

these limitations are addressed by the `step` function which trades simplicity in terms of number of lines for greater flexibility.

Revisions

- subversion 876 7/16/09: Added a NewtonsMethod class for general maximizing.
- subversion 906 8/3/09: Changed the name of the function maximize to optimize in the NewtonsMethod class.

Testing

Newtons Method has been tested using several functions of 1 or 2 variables including the Gaussian function. It has been tested with both approximating the gradient and hessian and when both are given explicitly.

Future Work

- 8/4 - Add the ability to only move partially in the direction of the gradient.

pause

```
{\small    void pause();
}
{\small    void pause(char *fmt, ...);
}
{\small
}
```

Key Words

pause, print

Authors

Alex Henniges

Introduction

The `pause` freezes the current process until the user presses the **enter** key. This function also allows the user to print information at the pause line.

Subsidiaries

Functions:

- `vprintf`
- `scanf`
- `fflush`

Global Variables:
Local Variables:

Description

The `pause` function is designed to place break points in the code that will stop the process until the user presses the enter key. There are several uses to this. A standard one is debugging as it can allow a programmer to step through a procedure. While there are usually similar debugging options in code editors, this function can be added and removed easily from within the code. The second use is that the console for programs will close immediately after execution with some editors. Without a way to freeze the program, the console would close before the data could be read and interpreted.

There are two options for this `pause` function. If the default pause is used, the following message will be printed: "PAUSE..." Pressing the **enter** key will resume the process. The function can also print out a message provided to it. This uses the `vprintf` function so that the printed information can be formatted text. The user must still press **enter** to resume when this form is used. Pressing other keys will not affect the program.

Historically, the project has used

```
{\small   system("PAUSE");
}
{\small
}
```

to pause the program. However, this can only be used on a Windows machine, a limiting factor that we wish to remove from the project.

Practicum

Example:

```
{\small   pause("Done...press enter to exit."); // PAUSE
}
{\small
}
```

Limitations

One limitation of the `pause` function is that it only resumes after pressing the **enter** key. This is compared to the former pause function (see above) that would resume after pressing any key. This could also be considered an improvement.

Revisions

- subversion 909, 8/4/09: Added the fully functional `pause` function.

Testing

The `pause` function has been tested simply through using it extensively.

Future Work

No future work is planned at this time.

print3DResultsStep

```
{\small
   void print3DResultsStep(char* fileName, vector<double>* radii, vector<double>* curvs)
}
{\small
}
```

Key Words

radii, curvatures, file, flow, step, print, three-dimensional

Authors

Alex Henniges

Introduction

The `print3DResultsStep` function prints out the results of a curvature flow, with the results grouped by each step of the flow. These results will be written to the file given by `filename`.

Subsidiaries

Functions:
Global Variables:
Local Variables: `int vertSize`, `int numSteps`

Description

Prints the results of a curvature flow into the file given by `filename`. The results, that is, the radii and curvature values, are given by vectors of doubles. Most commonly, these vectors are taken from the Approximator class after the flow is run. The `print3DResultsStep` function determines the number of vertices of the current triangulation and the total number of steps are then derived from this and the size of the vectors.

There are several ways to display the results. The `print3DResultsStep` function groups by step. This means that for each step of the curvature flow, the radii and curvature values for each vertex is printed. In addition, since Yamabe flow converges with respect to curvature divided by radius, this value is printed as well. Therefore, this function should be used with three-dimensional curvature flows. An example is shown below. Other formats are given by printResultsStep , printResultsVertex, printResultsNum.

Practicum

Example:

```
{\small
  // Print the results of a curvature flow with Approximator app into file "ODEResult.t
}
{\small
  print3DResultsStep("./ODEResults.txt", app->radiiHistory, app->curvHistory);
}
{\small
}
```

The output of such an example may then be

```
{\small          :
}
{\small          :
}
{\small        Vertex   5     0.8324396       8.5301529       10.2471738
}
{\small        Total Curvature: 44.5286316
}

{\small        Step    74     Radius          Curvature        Curv:Radius
}
{\small        ---------------------------------------------------
}
{\small        Vertex   1     0.8883594       9.3071126       10.4767428
}
{\small        Vertex   2     0.8725496       9.0880458       10.4155064
}
{\small        Vertex   3     0.8579899       8.8858872       10.3566333
}
{\small        Vertex   4     0.8448655       8.7033021       10.3014058
}
{\small        Vertex   5     0.8333839       8.5432883       10.2513233
}
{\small        Total Curvature: 44.5276360
}

{\small        Step    75     Radius          Curvature        Curv:Radius
}
{\small        ---------------------------------------------------
}
{\small        Vertex   1     0.8873282       9.2928034       10.4727922
}
{\small          :
}
{\small          :
}
{\small
}
```

Limitations

Currently the `print3DResultsStep` function is limited in the information it prints. As our curvature flow has evolved to record additional information such as volumes, it may be time to explore

a more robust form for displaying results. As there is considerable dependence on the Approximator for the data vectors, it may be wise to place this and similar functions in the Approximator class.

Revisions

- subversion 545, 9/29/08: Moved the printing of results out of calcFlow and into a new function.
- subversion 783, 6/18/09: Small modifications in response to changes in the Approximator class.

Testing

The `print3DResultsStep` function was tested by running multiple curvature flows and printing the results. It was considered working when the format of the data was as desired.

Future Work

- 6/29 - Recreate the print functions to print more data and be more flexible.
- 6/29 - Move the print functions into the Approximator class.

printResultsNum

```
{\small
        void printResultsNum(char* fileName, vector<double>* radii, vector<double>* cu
}
```

Key Words

radii, curvatures, file, flow, vertex, print

Authors

Alex Henniges

Introduction

The `printResultsNum` function prints out the results of a curvature flow, with the results grouped by each vertex of the triangulation but without labels. This format is used for when a program, (GUI, Matlab, etc) needs to parse the data. These results will be written to the file given by `filename`.

Subsidiaries

Functions:
Global Variables:
Local Variables: `int vertSize`, `int numSteps`

Description

Prints the results of a curvature flow into the file given by `filename`. The results, that is, the radii and curvature values, are given by vectors of doubles. Most commonly, these vectors are taken from the Approximator class after the flow is run. The `printResultsNum` function determines the number of vertices of the current triangulation and the total number of steps are then derived from this and the size of the vectors.

There are several ways to display the results. The `printResultsNum` function groups by vertex but provides no labels. This means that for each vertex of the triangulation, the radii (first column) and curvature (second column) values for each step is given. This format is used for when a program, (GUI, Matlab, etc) needs to parse the data. Therefore, it would be difficult for a human to read, but allows the computer to do so much easier. An example is shown below. Other formats are given by printResultsStep, printResultsVertex, print3DResultsStep.

Practicum

Example:

```
{\small     // Print the results of a curvature flow with
}
{\small     // Approximator app into file "ODEResult.txt"
}
{\small
    printResultsNum("./ODEResults.txt", app->radiiHistory, app->curvHistory);
}
```

The output of such an example may then be

```
{\small          :
}
{\small          :
}
{\small       0.8272160717        3.1425515910
}
{\small       0.8272081392        3.1425294463
}
{\small       0.8272003900        3.1425078130
}

{\small       1.0000000000        3.1415926536
```

```
}
{\small       1.0000000000        3.5987926375
}
{\small       0.9954280002        3.5877016632
}
{\small       0.9909873062        3.5768691746
}
{\small       0.9866737711        3.5662905388
}
{\small          :
}
{\small          :
}
```

Limitations

Currently the `printResultsNum` function is limited in the information it prints. As our curvature flow has evolved to record additional information such as volumes, it may be time to explore a more robust form for displaying results. As there is considerable dependence on the Approximator for the data vectors, it may be wise to place this and similar functions in the [CurvatureFlow Approximator] class.

Revisions

subversion 545, 9/29/08: Moved the printing of results out of calcFlow and into a new function. subversion 783, 6/18/09: Small modifications in response to changes in the Approximator class.

Testing

The `printResultsNum` function was tested by running multiple curvature flows and printing the results. It was considered working when the format of the data was as desired.

Future Work

6/29 - Recreate the print functions to print more data and be more flexible. 6/29 - Move the print functions into the Approximator class.

printResultsNumSteps

```
{\small
   void printResultsNumSteps(char* fileName, vector<double>* radii, vector<double>* curvs)
}
{\small
}
```

Key Words

radii, curvatures, file, flow, vertex, print

Authors

Alex Henniges

Introduction

The `printResultsNumSteps` function prints out the results of a curvature flow, with the results grouped by each step of the triangulation but without labels. This format is used for with the GUI to create a polygonal representation of curvatures. These results will be written to the file given by `filename`.

Subsidiaries

Functions:
Global Variables:
Local Variables: `int vertSize`, `int numSteps`

Description

Prints the results of a curvature flow into the file given by `filename`. The results are curvature divided by radii values, and are given by vectors of doubles. Most commonly, these vectors are taken from the Approximator class after the flow is run. The `printResultsNumSteps` function determines the number of vertices of the current triangulation and the total number of steps are then derived from this and the size of the vectors.

There are several ways to display the results. The `printResultsNumSteps` function groups by step but provides no labels and does not print out radii, but instead curvautre divided by radii. The purpose for this format is to create the "Polygon flows" in the GUI. Therefore, it would be difficult for a human to read, but allows the computer to do so much easier. An example is shown below.

Practicum

Example:

```
{\small
  // Print the results of a curvature flow with Approximator app into file "ODEResult.t
}
{\small
  printResultsNumSteps("./ODEResults.txt", app->radiiHistory, app->curvHistory);
}
{\small
}
```

The output of such an example may then be

```
{\small          :
}
{\small          :
}
{\small       3.1425515910
}
{\small       3.1425294463
}
{\small       3.1425078130
}

{\small       3.1415926536
}
{\small       3.5987926375
}
{\small       3.5877016632
}
{\small       3.5768691746
}
{\small       3.5662905388
}
{\small          :
}
{\small          :
}
{\small
}
```

Limitations

Unlike the other print functions, the purpose of `printResultsNumSteps` is to only display the curvature divided by radii, and so is not limited

in the information it prints. On the otherhand, an overhaul of the entire printing system would likely involve modifying this function.

Revisions

- subversion 545, 9/29/08: Moved the printing of results out of calcFlow and into a new function.
- subversion 783, 6/18/09: Small modifications in response to changes in the Approximator class.

Testing

The `printResultsNumSteps` function was tested by running multiple curvature flows and printing the results. It was considered working when the format of the data was as desired.

Future Work

- 6/29 - Recreate the print functions to print more data and be more flexible.
- 6/29 - Move the print functions into the Approximator class.

printResultsStep

```
{\small
   void printResultsStep(char* fileName, vector<double>* radii, vector<double>* curvs)
}
{\small
}
```

Key Words

radii, curvatures, file, flow, step, print

Authors

Alex Henniges

Introduction

The `printResultsStep` function prints out the results of a curvature flow, with the results grouped by each step of the flow. These results will be written to the file given by `filename`.

Subsidiaries

Functions:
Global Variables:
Local Variables: `int vertSize`, `int numSteps`

Description

Prints the results of a curvature flow into the file given by `filename`. The results, that is, the radii and curvature values, are given by vectors of doubles. Most commonly, these vectors are taken from the Approximator class after the flow is run. The `printResultsStep` function determines the number of vertices of the current triangulation and the total number of steps are then derived from this and the size of the vectors.

There are several ways to display the results. The `printResultsStep` function groups by step. This means that for each step of the curvature flow, the radii and curvature values for each vertex is printed. An example is shown below. Other formats are given by printResultsVertex, printResultsNum, print3DResultsStep.

Practicum

Example:

```
{\small
  // Print the results of a curvature flow with Approximator app into file "ODEResult.t
}
{\small
  printResultsStep("./ODEResults.txt", app->radiiHistory, app->curvHistory);
}
{\small
}
```

The output of such an example may then be

```
{\small          :
}
{\small          :
}
{\small        Vertex   4     0.5397923       2.1411007
```

```
}
{\small        Total Curvature: 12.5663706
}

{\small        Step     9     Radius          Curvature
}
{\small        ---------------------------------------------------
}
{\small        Vertex   1     1.1681789       3.9076805
}
{\small        Vertex   2     0.9668779       3.5170408
}
{\small        Vertex   3     0.7605802       2.9772908
}
{\small        Vertex   4     0.5451929       2.1643586
}
{\small        Total Curvature: 12.5663706
}

{\small        Step    10     Radius          Curvature
}
{\small        ---------------------------------------------------
}
{\small        Vertex   1     1.1592296       3.8916213
}
{\small          :
}
{\small          :
}
{\small
}
```

Limitations

Currently the `printResultsStep` function is limited in the information it prints. As our curvature flow has evolved to record additional information such as volumes, it may be time to explore a more robust form for displaying results. As there is considerable dependence on the Approximator for the data vectors, it may be wise to place this and similar functions in the Approximator class.

Revisions

- subversion 545, 9/29/08: Moved the printing of results out of calcFlow and into a new function.

- subversion 783, 6/18/09: Small modifications in response to changes in the Approximator class.

Testing

The `printResultsStep` function was tested by running multiple curvature flows and printing the results. It was considered working when the format of the data was as desired.

Future Work

- 6/29 - Recreate the print functions to print more data and be more flexible.
- 6/29 - Move the print functions into the Approximator class.

printResultsVertex

```
{\small
   void printResultsVertex(char* fileName, vector<double>* radii, vector<double>* curvs
}
{\small
}
```

Key Words

radii, curvatures, file, flow, vertex, print

Authors

Alex Henniges

Introduction

The `printResultsVertex` function prints out the results of a curvature flow, with the results grouped by each vertex of the triangulation. These results will be written to the file given by `filename`.

Subsidiaries

Functions:
Global Variables:
Local Variables: `int vertSize`, `int numSteps`

Description

Prints the results of a curvature flow into the file given by `filename`. The results, that is, the radii and curvature values, are given by vectors of doubles. Most commonly, these vectors are taken from the Approximator class after the flow is run. The `printResultsVertex` function determines the number of vertices of the current triangulation and the total number of steps are then derived from this and the size of the vectors.

There are several ways to display the results. The `printResultsVertex` function groups by vertex. This means that for each vertex of the triangulation, the radii and curvature values for each step is given. An example is shown below. Other formats are given by printResultsStep, printResultsNum, print3DResultsStep.

Practicum

Example:

```
{\small
  // Print the results of a curvature flow with Approximator app into file "ODEResult.txt"
}
{\small
  printResultsVertex("./ODEResults.txt", app->radiiHistory, app->curvHistory);
}
{\small
}
```

The output of such an example may then be

```
{\small          :
}
{\small          :
}
{\small        Step  298        0.8272161        3.1425516
}
{\small        Step  299        0.8272081        3.1425294
}
{\small        Step  300        0.8272004        3.1425078
}

{\small        Vertex:   2        Radius                Curv
}

{\small        ---------------------------------
}
{\small        Step    0        1.0000000        3.1415927
```

```
}
{\small        Step    1        1.0000000        3.5987926
}
{\small        Step    2        0.9954280        3.5877017
}
{\small        Step    3        0.9909873        3.5768692
}
{\small        Step    4        0.9866738        3.5662905
}
{\small          :
}
{\small          :
}
{\small
}
```

Limitations

Currently the `printResultsVertex` function is limited in the information it prints. As our curvature flow has evolved to record additional information such as volumes, it may be time to explore a more robust form for displaying results. As there is considerable dependence on the Approximator for the data vectors, it may be wise to place this and similar functions in the Approximator class.

Revisions

- subversion 545, 9/29/08: Moved the printing of results out of calcFlow and into a new function.
- subversion 783, 6/18/09: Small modifications in response to changes in the Approximator class.

Testing

The `printResultsVertex` function was tested by running multiple curvature flows and printing the results. It was considered working when the format of the data was as desired.

Future Work

- 6/29 - Recreate the print functions to print more data and be more flexible.
- 6/29 - Move the print functions into the Approximator class.

Total_Volume

Function Prototype

```
double Total_Volume ()
```

Key Words

Volume, tetrahedron, Cayley-Menger determinant.

Authors

Daniel Champion

Introduction

The function `Total_Volume` calculates the total volume of a three dimensional triangulated manifold.

Subsidiaries

Functions:

Geometry::volume

Global Variables: radii, etas.

Local Variables: none.

Description

`Total_Volume` is calculated by summing the volumes of each tetrahedron in a triangulation. The volume of a tetrahedron is calculated with the Cayley-Menger determinant:

$$288V^2 = \det \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & L_{12}^2 & L_{13}^2 & L_{14}^2 \\ 1 & L_{12}^2 & 0 & L_{23}^2 & L_{24}^2 \\ 1 & L_{13}^2 & L_{23}^2 & 0 & L_{34}^2 \\ 1 & L_{14}^2 & L_{24}^2 & L_{34}^2 & 0 \end{bmatrix}$$

where the lengths were determined from the radii and eta values using the formula

$$L_{ij}^2 = r_i^2 + r_j^2 + 2r_i r_j Eta_{ij}.$$

The formula was obtained using calculations within Mathematica and was output into the C programming language using the function CForm.

The total volume of a triangulation is used in multiple locations within the project. One example of its use is in the calculation of the normalized Einstein-Hilbert-Regge functional:

$$\widetilde{EHR} = \frac{\sum_i K_i}{(\texttt{Total\_Volume()})^{\frac{1}{3}}}$$

where K_i is the curvature at vertex i.

Practicum

An excellent example of the use of this function is in the calculation of the normalized Einstein-Hilbert-Regge functional.

```
double EHR () {
    double result;
    result = (Total_Curvature ())/pow(Total_Volume (), 1.0/3.0);
    return result;
    }
```

Limitations

Since `Total_Volume()` relies critically on the Geometry::volume function, it thus has the same limitations. Specifically, if the edge lengths of any tetrahedron do not satisfy the necessary conditions to produce a positive volume tetrahedron, `Total_Volume()` will output an undefined number. The Cayley-Menger determinant can be used to check this condition on the edge lengths.

Revisions

subversion 757, 7/11/09, `Total_Volume()` created.

Testing

Using known volumes of several tetrahedra, the total volume was calculated by hand and compared with `Total_Volume()`.

Future Work

No planned future work.

Volume_Partial

Function Prototype

```
double Volume_Partial (int i, Tetra t)
```

Key Words

Volume, tetrahedron, vertex, radius, Cayley-Menger determinant, standard form.

Authors

Daniel Champion

Introduction

`Volume_Partial` calculates the partial derivative of the volume of a tetrahedron with respect to the logarithm of the radius of a vertex.

Subsidiaries

Functions:
`listDifference`
`listIntersection`
`Simplex::isAdjVertex`
Global Variables: radii, etas.
Local Variables: int i, Tetra t.

Description

The volume of a tetrahedron only depends on the lengths of its edges as calculated from the Cayley-Menger determinant. Thus for a given tetrahedron t, it's partial derivatives with respect to log radii will vanish except for those radii corresponding to the vertices of t. The function `isAdjVertex` of the simplex class determines this condition. `Volume_Partial` then proceeds with an initialization procedure that labels the vertices and edges (radii and etas) in standard form. Specifically, `Volume_Partial` receives as inputs an integer `i` corresponding to a vertex (in the vertex table) which is labeled vertex v1. The remaining vertices are labeled $v2, v3, v4$, and the edges $e12, e13, e14, e23, e24, e34$ are labeled preserving the structure implied by the assignment of the vertices. The radii r_1, r_2,... and eta values Eta_{12}, Eta_{13},... are assigned to the corresponding vertices and edges.

The formula for the partial derivative in terms of these standard form variables was calculated in Mathematica using the Cayley-Menger determinant, that is:

$$288V^2 = \det \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & L_{12}^2 & L_{13}^2 & L_{14}^2 \\ 1 & L_{12}^2 & 0 & L_{23}^2 & L_{24}^2 \\ 1 & L_{13}^2 & L_{23}^2 & 0 & L_{34}^2 \\ 1 & L_{14}^2 & L_{24}^2 & L_{34}^2 & 0 \end{bmatrix},$$

where the lengths were determined from the radii and eta values using the formula

$$L_{ij}^2 = r_i^2 + r_j^2 + 2r_i r_j Eta_{ij}.$$

The formula obtained from Mathematica was outputted into the C programming language using the function CForm.

This function was designed for use in the optimization of the Einstein-Hilbert-Regge functional using Newton's method. In this procedure the gradient of the EHR functional is needed which contains the partial derivatives of the volume.

Practicum

Usage:

`Volume_Partial (int i, Tetra t)`

The integer `i` corresponds to a vertex in the vertex table, that is

$$\texttt{Volume\_Partial (i, t)} = \frac{\partial}{\partial \log r_i} Volume(t).$$

Limitations

`Volume_Partial` was designed to output the correct partial derivative for any integer i in the vertex table and any tetrahedron t in the triangulation.

Revisions

subversion 757, 7/7/09, `Volume_Partial` created within Newtons_Method.cpp.

Testing

The partial derivative of volume of several known tetrahedra were calculated using `Volume_Partial` and verified using Mathematica.

Future Work

The procedure that initializes the tetrahedron into standard form should be removed from this program and placed elsewhere. There are several occurrences of this type of procedure that should be consolidated.

Volume_Second_Partial

Function Prototype

```
double Volume_Second_Partial ( int i, int j, Tetra t)
```

Key Words

Volume, Hessian Matrix, Newton's Method, partial derivative, Einstein-Hilbert-Regge functional.

Authors

Daniel Champion

Introduction

`Volume_Second_Partial` calculates the second order partial derivatives of the volume of a tetrahedron with respect to log radii for all pairs of indices (not necessarily distinct) in the vertex table.

Subsidiaries

Functions:

`listDifference`
`listIntersection`
`Simplex::isAdjVertex`
Global Variables: radii, etas.
Local Variables: int i, int j, Tetra t.

Description

The volume of a tetrahedron only depends on the lengths of its edges as calculated from the Cayley-Menger determinant. Thus for a given tetrahedron t, it's second order partial derivatives with respect to log radii will vanish except for pairs of radii (not necessarily distinct) corresponding to the vertices of t. The first step in the implementation of `Volume_Second_Partial` is the determination of the following trichotomy for a pair $\{i, j\}$ of indices in the vertex table:

A. $i = j$ and i is a vertex of tetrahedron t
B. $i \neq j$ and both i and j belong to t
C. at least one of i or j doesn't belong to t.

Each condition of the trichotomy requires a distinct calculation to determine the desired partial derivative. Nevertheless, the next step in the implementation is to place the tetrahedron in "standard form" relative to the indices i and j (for conditions A and B only). More specifically, for condition A the radius for vertex i is stored as r_1, and the remaining radii of the tetrahedron t are assigned r_2, r_3, and r_4 in no particular order. The eta values $Eta_{12}, Eta_{13}, ...$ are then assigned preserving the preceding assignments. In the case of condition B, the radii at vertices i and j are assigned to r_1 and r_2 respectively, and r_3, and r_4 the remaining radii of t. The eta values $Eta_{12}, Eta_{13}, ...$ are again assigned preserving the preceding assignments.

The formulas for the second order partial derivatives in terms of these standard form variables was calculated in Mathematica using the Cayley-Menger

determinant, that is:

$$288V^2 = \det \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & L_{12}^2 & L_{13}^2 & L_{14}^2 \\ 1 & L_{12}^2 & 0 & L_{23}^2 & L_{24}^2 \\ 1 & L_{13}^2 & L_{23}^2 & 0 & L_{34}^2 \\ 1 & L_{14}^2 & L_{24}^2 & L_{34}^2 & 0 \end{bmatrix},$$

where the lengths were determined from the radii and eta values using the formula

$$L_{ij}^2 = r_i^2 + r_j^2 + 2r_i r_j Eta_{ij}.$$

The formula obtained from Mathematica was outputted into the C programming language using the function CForm.

This function was designed for use in the optimization of the Einstein-Hilbert-Regge functional using Newton's method. In this procedure the Hessian matrix of the normalized EHR functional is needed, each entry of which uses the second order partial derivatives of volume. See the entry on `EHR_Second_Partial`.

Practicum

Usage:

`Volume_Second_Partial (int i, int j, Tetra t)`

The integers `i` and `j` correspond to vertices in the vertex table and `t` is a tetrahedron in the triangulation. Specifically the function returns:

$$\texttt{Volume\_Second\_Partial (i, j, t)} = \frac{\partial^2}{\partial \log r_i \partial \log r_j} Volume(t).$$

Limitations

`Volume_Second_Partial` is fully operational with no know limitations. The function will output appropriate values when given indices i, and j in the vertex table, and a tetrahedron t.

Revisions

subversion 757, 7/9/09, `Volume_Second_Partial` created.

Testing

Several trials were run outputting the values of `Volume_Second_Partial` for a variety of indices and tetrahedra. These values were compared with calculations performed on Mathematica.

Future Work

The procedures that place the tetrahedron's geometric data in standard form need to be redeveloped. Currently these procedures are very expensive and labyrinthine. A general procedure should be written to place any simplex in standard form relative to any sub-simplex.

Chapter 6

Classes

`Approximator`

Key Words

curvature flow, differential equations, Euler's method, Runga Kutta

Authors

- Joe Thomas
- Alex Henniges

Introduction

The Approximator class runs a curvature flow using one of several methods. The class itself is abstract and the method is chosen by the instantiating object.

Subsidiaries

Functions:

- [Functions#Approximator::run run]

Sub-classes:

- EulerApprox
- RungaApprox

Public Variables:

- radiiHistory
- curvHistory
- areaHistory
- volumeHistory

Description

The `Approximator` class is the shell for running a curvature flow. The class provides the functionality to determine what system of differential equations to use, how to perform a step, and even which values to record for later use. The system is provided by the user at run-time and is defined to be a function that takes in an empty array of `double`s and fills the array with the values calculated in the system of equations. The `Approximator` class is abstract with an abstract method `step`. A class that extends `Approximator` implements `step` with the method of approximation to use (i.e Euler's method). Lastly, the `Approximator` stores values after each `step` of a flow according to which values were requested at construction. Values include radii, curvatures, areas, and volumes. These histories can then be accessed directly from the `Approximator` object.

Constructor

The constructor takes in a function that defines a system of differential equations and a string of characters representing what histories to record. For the function to be a `sysdiffeq` it must not return a value and its only parameter is an array of doubles that will be filled in with values after the function completes. The history string must be nul-terminated and consisting of only valid characters. The valid characters currently are:

- r - Record radii
- 2 - Record two-dimensional curvatures
- 3 - Record three-dimensional curvatures
- a - Record areas
- v - Record volumes

One cannot list both two- and three-dimensional curvatures.

```
{\small   typedef void (*sysdiffeq)(double derivs[]);
}
{\small   Approximator(sysdiffeq funct, char* histories);
```

```
}
{\small
}
```

Practicum

This example will show how to run a Yamabe curvature flow on the pentachoron using precision and accuracy bounds (see [Functions#run run]) while recording radii, curvatures, and volumes. It will show how to initialize the system and also how to print out results at the end.

```
{\small int main(int argc, char** argv) {
}
{\small    map<int, Vertex>::iterator vit;
}
{\small    map<int, Edge>::iterator eit;
}
{\small    map<int, Face>::iterator fit;
}
{\small    map<int, Tetra>::iterator tit;
}
{\small
}
{\small    vector<int> edges;
}
{\small    vector<int> faces;
}
{\small    vector<int> tetras;
}
{\small
}
{\small
}
{\small    time_t start, end;
}
{\small
}
{\small    // File to read in triangulation from.
}
{\small
   char from[] = "./Triangulation Files/3D Manifolds/Lutz Format/pentachoron.txt";
}
{\small    // File to convert to proper format.
}
{\small    char to[] = "./Triangulation Files/manifold converted.txt";
```

```
}
{\small    // Convert, then read in triangulation.
}
{\small    make3DTriangulationFile(from, to);
}
{\small    read3DTriangulationFile(to);
}

{\small    int vertSize = Triangulation::vertexTable.size();
}
{\small    int edgeSize = Triangulation::edgeTable.size();
}
{\small
}
{\small    // Set the radii
}
{\small    for(int i = 1; i <= vertSize; i++) {
}
{\small
      Radius::At(Triangulation::vertexTable[i])->setValue(1 + (0.5 - i/5.0) );
}
{\small    }
}
{\small    // Set the etas
}
{\small    for(int i = 1; i <= edgeSize; i++) {
}
{\small        Eta::At(Triangulation::edgeTable[i])->setValue(1.0);
}
{\small    }
}

{\small
   // Construct an Approximator object that uses the Euler method and Yamabe flow while
}
{\small    // recording radii, 3D curvatures, and volumes.
}
{\small    Approximator *app = new EulerApprox((sysdiffeq) Yamabe, "r3v");
}

{\small
   // Run the Yamabe flow with precision and accuracy bounds of 0.0001 and stepsize of
}
{\small    app->run(0.0001, 0.0001, 0.01);
}
```

```
{\small    // Print out radii, curvatures and volumes
}
{\small
   printResultsStep("./Triangulation Files/ODE Result.txt", &(app->radiiHistory), &(app->curvHist
}
{\small
   printResultsVolumes("./Triangulation Files/Volumes.txt", &(app->volumeHistory));
}

{\small    return 0;
}
{\small }
}
```

NewtonsMethod

Key Words

gradient, hessian, extrema

Authors

- Alex Henniges
- Dan Champion

Introduction

The `NewtonsMethod` class is used to find an extremum of a given functional. In addition to the functional given, the user can provide the gradient or hessian function. If not, these are approximated during run-time.

Subsidiaries

Functions:

- NewtonsMethod::maximize
- NewtonsMethod::step
- NewtonsMethod::setPrintFunc
- NewtonsMethod::printInfo

Description

As a class, `NewtonsMethod` is used as a general way to perform Newton's method on a function in order to find its extrema. Newton's method will find an extrema much faster than Euler's method, but is also more complicated. In order for Newton's method to work, it requires the first and second-order partial derivatives in addition to the original function. If the user knows these explicitly, they can be passed in the [#Constructor constructor] and should lead to more accurate and possible quicker calculations. If the first and secord-partial derivatives are not provided, then they will be approximated using quotients.

For our purposes within the `NewtonsMethod` class, the original function is defined to take as a parameter an array of `double`s that represent the values of each variable. The function should return a double. The gradient function is defined to take an array of doubles that also represent the point at which the partial derivatives should be calculated. In addition, it takes another array of doubles for the partial derivatives to be placed in. Lastly, a hessian function is also defined to take an array of doubles for the point at which the calculation is being done. It also takes a two-dimensional array of double for the second-order partial derivatives to be placed in. Both the gradient and hessian function do not return a value.

Constructor

There are three constructors for the `NewtonsMethod` class, allowing for a combination of potential functions that can be given explicitly. In addition, every constructor must be given an integer that indicates the number of variables given to the functional.

```
{\small     typedef double (*orig_function)(double vars[])
}
{\small     typedef void   (*gradient)(double vars[], double sol[])
}
{\small     typedef void   (*hessian)(double vars[], double *sol[])
}

{\small     NewtonsMethod(orig_function func, int numVars)
}
{\small     NewtonsMethod(orig_function func, gradient df, int numVars)
}
{\small
    NewtonsMethod(orig_function func, gradient df, hessian d2f, int numVars)
}
{\small
}
```

Practicum

Below is a full example of how to use the `NewtonsMethod` class to find the minimum of an ellipse. In this case, the gradient and hessian are not given. The maximum found is at (0,0).

```
{\small    // This function takes two variables.
}
{\small    // f(x, y) = (1 - x^/4 - y^2/9) ^(1/2)
}
{\small    double ellipse(double vars[]) {
}
{\small        double val = 1 - pow(vars[0], 2) / 4 - pow(vars[1], 2) / 9;
}
{\small        return sqrt(val);
}
{\small    }
}

{\small    int main(int arg, char** argv) {
}
{\small     // Create the NewtonsMethod object, 2 variables
}
{\small     NewtonsMethod *nm = new NewtonsMethod(ellipse, 2);
}
{\small     // Build the array that holds the initial values.
}
{\small     double initial[] = {0.1, 2.5};
}
{\small     // Build the array that will hold the final solution.
}
{\small     double soln[2];
}
{\small     // Run the maximize function
}
{\small     nm->maximize(initial, soln);
}

{\small     // Display the results
}
{\small     printf("\nSolution: %.10f, %.10f\n", soln[0], soln[1]);
}

{\small     return 0;
}
{\small    }
}
```

```
{\small
}
```

Using the same ellipse, one can use the `step` function instead of `maximize` to gain greater flexibility over the procedure. In this case, we also print out useful information after each step.

```
{\small      // This function takes two variables.
}
{\small    double ellipse(double vars[]) {
}
{\small        double val = 1 - pow(vars[0], 2) / 4 - pow(vars[1], 2) / 9;
}
{\small        return sqrt(val);
}
{\small    }
}

{\small    int main(int arg, char** argv) {
}
{\small     NewtonsMethod *nm = new NewtonsMethod(ellipse, 2);
}
{\small     double x_n[] = {1, 1};
}
{\small     int i = 1;
}
{\small     fprintf(stdout, "Initial\n-----------------\n");
}
{\small     for(int j = 0; j < 2; j++) {
}
{\small       fprintf(stdout, "x_n_%d[%d] = %f\n", i, j, x_n[j]);
}
{\small     }
}
{\small
    // Continue with the procedure until the length of the gradient is
}
{\small     // less than 0.000001.
}
{\small     while(nm->step(x_n) > 0.000001) {
}
{\small       fprintf(stdout, "\n***** Step %d *****\n", i++);
}
{\small       nm->printInfo(stdout);
}
{\small       for(int j = 0; j < 2; j++) {
```

```
}
{\small         fprintf(stdout, "x_n_%d[%d] = %f\n", i, j, x_n[j]);
}
{\small       }
}
{\small     }
}
{\small     printf("\nSolution: %.10f\n", x_n[0]);
}

{\small     return 0;
}
{\small    }
}
{\small
}
```

In this example, we use a one variable Gaussian function, but provide a gradient and hessian, as well. The maximum is found at `x = 0`.

```
{\small    // The function, e^(-x^2).
}
{\small    double gaussian(double vars[]) {
}
{\small        return exp(-pow(vars[0], 2));
}
{\small    }
}

{\small    // The gradient function, -2x * e^(-x^2).
}
{\small    // Note that the solution is placed in the array.
}
{\small    void gradFunc(double vars[], double sol[]) {
}
{\small      sol[0] = -2 * vars[0] * func(vars);
}
{\small    }
}

{\small    // The hessian function, e^(-x^2)(4x^2 - 2).
}
{\small    // Note that the solution is placed in a matrix.
}
{\small    void hessFunc(double vars[], double *sol[]) {
```

```
}
{\small      sol[0][0] = func(vars) * (4 * pow(vars[0], 2) - 2);
}
{\small    }
}

{\small    int main(int arg, char** argv) {
}
{\small     // Create the NewtonsMethod object
}
{\small
    NewtonsMethod *nm = new NewtonsMethod(gaussian, gradFunc, hessFunc, 1);
}
{\small     // Build the array that holds the initial value.
}
{\small     double initial[] = {0.1};
}
{\small     // Build the array that will hold the final solution.
}
{\small     double soln[1];
}
{\small     // Run the maximize function
}
{\small     nm->maximize(initial, soln);
}

{\small     // Display the results
}
{\small     printf("\nSolution: %.10f\n", soln[0]);
}

{\small     return 0;
}
{\small    }
}
{\small
}
```

Limitations

There are limitations with the `NewtonsMethod` class with regards to the approximation of the gradient and hessian. In both cases, a delta value for the quotients is hard-coded as 10^{-5}. This could lead to accuracy issues when the point where the derivative is being calculated is less than this value. It can also be too accurate at times and lead to unnecessarily slowing down the procedure. One solution

could be to provide a function where a delta value is set by the user.

Revisions

- subversion 876, 7/16/09: Added a NewtonsMethod class for general maximizing.

Future Work

- 7/16 - Provide greater flexibility to the user for approximating the gradient and hessian.

Part III

Theory

Chapter 7

Glossary

circle power Given a circle C and a point P, let L be the line through the point P that passes through the center of the circle. The A and B be the intersection points of the L with the circle. Define a signed distance $\left\|\overline{PX}\right\|$ for a line segment $\overline{PX}$ to be negative if the line segment lies entirely within the circle, and positive otherwise. The *circle power* of P relative to C, denoted by $pow_C(P)$, is given by:

$$pow_C(P) = \left\|\overline{PA}\right\| \left\|\overline{PB}\right\|.$$

Alternatively, if C is defined implicitly by $(x - x_c)^2 + (y - y_c)^2 = r_c^2$, then the circle power can be expressed as:

$$pow_C(P) = (x - x_c)^2 + (y - y_c)^2 - r_c^2.$$

common power point The point of the plane (or $\mathbb{R}^3$) containing a decorated triangle (or tetrahedron) that has the same circle power with respect to each of the weight circles.

decorated simplex (edge, triangle, tetrahedron,...) A simplex is called *decorated* when weights are assigned to the vertices of the simplex and then actualized by embedding the simplex into Euclidean space together with spheres centered at the vertices with radii determined by the weights. The orthocircle (if one exists) is sometimes considered part of the decorated simplex when appropriate.

edge curvature Given a three-dimensional piecewise flat manifold $(M, \mathcal{T}, d)$, the *edge curvature* along an edge $\{i, j\}$, measures how much that edge differs from Euclidean space. Specifically, the edge curvature K_{ij} is given by

$$K_{ij} = \left(2\pi - \sum_{\substack{k,l, \text{ such that} \\ \{i,j,k,l\} \in \mathcal{T}}} \beta_{ij,kl}\right) l_{ij},$$

where l_{ij} is the edge length, and $\beta_{ij,kl}$ is the dihedral angle of the edge $\{i,j\}$ of the tetrahedron $\{i,j,k,l\}$. In a triangulation of three-dimensional Euclidean space $K_{ij} = 0$ for all edges.

Einstein constant For a piecewise flat manifold $(M, \mathcal{T}, d)$, the *Einstein constant* λ is given by

$$\lambda = \frac{EHR\,(M, \mathcal{T}, d)}{3\mathcal{V}},$$

where $EHR\,(M, \mathcal{T}, d)$ is the Einstein-Hilbert-Regge functional and $\mathcal{V}$ is the total volume.

Einstein metric Given a piecewise flat manifold $(M, \mathcal{T}, d)$, we say that d is an *Einstein metric* provided that for all edges $\{i,j\}$ in the triangulation we have:

$$K_{ij} = \lambda l_{ij} \frac{\partial \mathcal{V}}{\partial l_{ij}},$$

where K_{ij} is the edge curvature, l_{ij} is the edge length, $\mathcal{V}$ is the total volume, and λ is the Einstein constant.

inversive distance Start with a decorated edge e_{ij}, that is, an edge of length l_{ij} with weight circles of radius r_i, r_j centered on its vertices. The inversive distance η_{ij} of the edge e_{ij} can be calculated with the formula:

$$\eta_{ij} = \frac{l_{ij}^2 - r_i^2 - r_j^2}{2r_i r_j}.$$

When the two weight circles intersect with angle θ_{ij}, we have the simple formula:

$$\eta_{ij} = \cos\left(\theta_{ij}\right).$$

The former formula was obtained by using the law of cosines and solving for the $\cos\left(\theta_{ij}\right)$ term. When the weight circle do not intersect and do not contain one or the other $\eta_{ij} > 1$. When the weight circles intersect at some angle then $-1 \leq \eta_{ij} \leq 1$. If one weight circle contains the other we have $\eta_{ij} < 1$.

manifold A second countable, Hausdorff topological space M is a *manifold* provided there is an integer $n > 0$ such that for each $x \in M$ there is an open set U_x containing x and a homeomorphism $h_x : U_x \to B\,(1,0) \subset \mathbb{R}^n$.

orthocircle Given a decorated triangle, provided the common power point is outside all of the weight circles, there exists a circle that is orthogonal to each of the weight circles. That is, the *orthocircle* is the circle that intersects each of the weight circles orthogonally. The orthocircle does not exist when the common power point is on or inside all three circles, however if the common power point is at infinity, there is a line that is orthogonal to the three weight circles which will also be identified as the orthocircle.

piecewise flat manifold A triple $(M, \mathcal{T}, d)$ where $(M, \mathcal{T})$ is a compact triangulated manifold with triangulation $\mathcal{T}$, d is a metric on M so that the restriction of d to each simplex of $\mathcal{T}$ is isometric to a Euclidean simplex of the same dimension.

triangulation A collection of n-dimensional simplices $\mathcal{T}$ together with pairwise identifications for the $(n-1)$-dimensional faces of the simplices. More restrictions are needed to ensure that the resultant space is a manifold. Alternatively, given a space M of dimension n, a *triangulation* of M is a subdivision of M into components $\{\sigma_i\}$ by (hyper)surfaces (of dimension $n-1$) so that each component is homeomorphic to an n-dimensional ball, and each component is combinatorially (as determined by the subdivisions) equivalent to an n-simplex.

pseudomanifold

dimension

metric

curvature

scalar curvature

constant scalar curvature

geometric flow

curvature flow

Yamabe flow

normalized total scalar curvature functional

Einstein-Hilbert functional

Einstein-Hilbert-Regge functional

conformal class

conformal deformation

min-max procedure

Yamabe constant

flip

Pachner move

flip algorithm

hinge

convex hinge

nonconvex hinge

bone

Delaunay triangulation (or hinge)

weighted triangulation (or hinge)

weighted Delaunay triangulation

Voronoi diagram (or cell)

weighted Voronoi diagram (or cell)

power diagram (or cell)

negative triangles

dual, Poincare dual

dual length

dual volume

Appendix A

Appendix

The appendix fragment is used only once. Subsequent appendices can be created using the Chapter Section/Body Tag.

Afterword

The back matter often includes one or more of an index, an afterword, acknowledgements, a bibliography, a colophon, or any other similar item. In the back matter, chapters do not produce a chapter number, but they are entered in the table of contents. If you are not using anything in the back matter, you can delete the back matter TeX field and everything that follows it.