Article 8

A Framework for Computing Integral Geometries in VISH using Template Meta Programming

Marcel Ritter[1,3], Bidur Bohara[1,2], Werner Benger[1]
[1]Center for Computation & Technology at Louisiana State University (CCT/LSU), USA
[2]Department of Computer Science at Louisiana State University (LSU), USA
[3]Unit for Hydraulic Engineering, Department for Infrastructure, Faculty of Civil Engineering at the University of Innsbruck, Austria
`marcel.ritter@student.uibk.ac.at`

We present a framework using C++ template programming for the computation of integral geometries such as streamlines, pathlines, geodesics or time surfaces. Hereby, common needs are identified and certain features are shared between different integration geometries minimizing the overall implementation effort. The implementation is based on a fiber bundle data model opening the possibility to handle all kinds of space-time geometries in an unified interface.

8.1 Introduction

Visualization of vector fields is still an topic of active research in scientific visualization. The most widely used approach, visualization via arrow icons, is intuitive but does not scale to huge datasets, where methods such as fastLIC [Stalling, 1998] or Doppler speckles [Benger et al., 2009b] are superior. Aside direct visualization of the vectors at each data point a common approach is studying features of the vector field via integral curves and surfaces. The mathematical similarity of these feature indicators is not necessarily reflected by the actual implementation of the computation algorithms: the different integration scheme such as space-like vs. time-like or 1D vs. 2D may easily lead to independent implementations, whereas it were desirable to use one approach for all kinds of integral computations.

8.1.1 Mathematical Background and Motivation

Let $q(s)$ be a parameterized curve in a manifold M, $q : \mathbb{R} \to M : s \mapsto q(s)$. The variation of the curve parameter s defines the tangential vector $\dot{q} = dq/ds$ along the curve. In computational fluid dynamics (CFD) a vector field typically describes the motion of fluid particles. Let v be a vector field with the vector $v \in T_P(M)$ being an element of the tangential space at a point P of a manifold M. An integral curve is defined in a space-time manifold M by

$$\dot{q}(s) = v\big(q(s)\big) \quad \text{with} \quad q(0) = \sigma \in M \tag{8.1}$$

and σ the initial seeding point. Let us assume an evolving vector field in 3D coordinates. A **streamline** is an integral curve that is tangential to a vector field frozen at an instant of time, Fig. 8.1a:

$$\dot{q}(s) = v\big(q(0)^t, q(s)^x, q(s)^y, q(s)^z\big) \quad \text{with} \quad q(s)^t = q(0)^t \tag{8.2}$$

A **pathline** is evolving over time. It represents the motion of a fluid particle at a certain point in time, Fig. 8.1b:

$$\dot{q}(s) = v\big(q(s)^t, q(s)^x, q(s)^y, q(s)^z\big). \tag{8.3}$$

A bundle of integral curves yields a high dimensional object. An initial space-like seeding line, $\sigma : \mathbb{R} \to M : \lambda \mapsto \sigma(\lambda)$, defines an integral surface $\Sigma : \mathbb{R}^2 \to M : (s, \lambda) \mapsto \Sigma(s, \lambda)$:

$$d\Sigma/ds = v\big(\Sigma(s, \lambda)\big) \quad \text{with} \quad \Sigma(0, \lambda) = \sigma(\lambda) \tag{8.4}$$

A line at $\Sigma^t(s, \lambda) = const.$ is called a **material-line**, see Fig. 8.1c for an illustration. An initial seeding surface $S_0(\lambda, \mu) : \mathbb{R}^2 \to M : (\lambda, \mu) \mapsto S(\lambda, \mu)$ defines an integral-hyper-surface $H : \mathbb{R}^3 \to M : (s, \lambda, \mu) \mapsto H(s, \lambda, \mu)$:

$$dH/ds = v\big(H(s, \lambda, \mu)\big) \quad \text{with} \quad H(0, \lambda, \mu) = H_0(\lambda, \mu) \tag{8.5}$$

A surface at $H^t(s, \lambda, \mu) = const.$ is called a **time surface**, illustrated in Fig. 8.1d.

Finite differentiation schemes [Deuflhard & Bornemann, 2002] can be applied to solve these equations. In our visualization environment we take this common

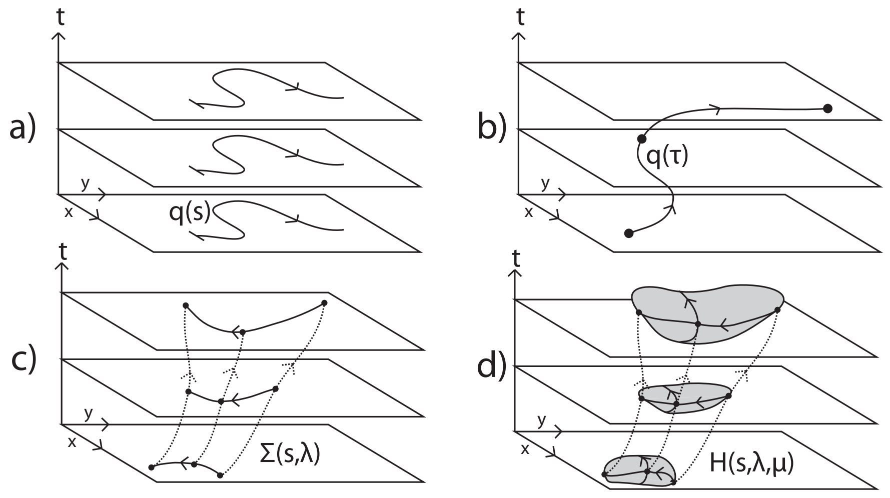


Figure 8.1: Different types of integral lines: a) streamlines, b) pathlines, c) material-line, d) time surface. The illustration shows time evolution in t-axis direction. The xy-plane is manifold hosting the integral curves.

foundation into account and we develop a template based integration framework suitable to all kinds of integration geometries in a unified way, sharing the algorithms used for solving differential equations, interpolating data fields and handling the underlying manifolds. The framework also extends to integrating geodesics on tensor fields stemming from numerical relativity or Magnetic Resonance Imaging (MRI). Here, the differential equation eq. 8.4 is replaced by the second order geodesic equation [Ritter & Benger, 2010] using $\dot{\Sigma} = d/ds\Sigma(s,\lambda)$:

$$\nabla_{\dot{\Sigma}}\dot{\Sigma}(s,\lambda) = 0 \ \ \forall\lambda \ \text{ with } \ \Sigma(0,\lambda) = \sigma(\lambda) \ \text{ and } \ \dot{\Sigma}(0,\lambda) = v\big(\sigma(\lambda)\big) \tag{8.6}$$

8.1.2 Previous Work

New development in vector field visualization is related to the computation of streaklines and streaksurfaces [Weinkauf & Theisel, 2010], the invention of new integration geometry types, such as time surfaces [Krishnan et al., 2009] or by [McLoughlin et al., 2009] on the improvement of algorithms to compute stream and path surfaces .

8.2 Framework Design and Implementation

Our earlier work includes the development of a streamline module for the computation and rendering of streamlines. Later, a pathlines module was developed mostly independently, only sharing some vector field interpolation methods. For the streamline module, the computation part was separated from the rendering of lines and it was further generalized to ease the implementation of geodesics in higher dimensional space-times [Ritter, 2010], and the computation part of the pathline module was inherited to compute the time surfaces [Bohara et al., 2010]. Besides a fast and simple Euler integration we had implemented the DOP853 integration for high accuracy [Hairer et al., 2000]. This is a Runge-Kutta (RK) integration of 8^{th} order using RK schemes of order 5 and 3 for error estimation and adaptive step size control and provides dense output. “The performance of this code, compared to methods of lower order, is impressive.” [Hairer et al., 2000]

8.2.1 Visualization Environment and Data Model

We use the *VISH* [Benger et al., 2007] visualization shell as our implementation platform. *VISH* supports the concept of a fiber bundles data model [Benger, 2004], which is a hierarchical data model structured in six levels. These levels are called Bundle, Slice (time), Grid, Topology(Skeleton), Representation and Field. Fields store the actual data arrays while the other levels are used similar to a directory structure to organize the data. Datasets such as position, velocity or connectivity information among points are stored in a Field. The collection of Skeletons which hold data Fields in their coordinate Representation is a Grid object. The collection of Grid objects over all time Slices is the Bundle of the dataset.

VISH uses a network structure to separate tasks in atomic entities, called *VISH*-objects, which are connected by input and output connections. A pull model is used updating using a separated control and data flow. Several levels of caching are provided throughout the network updating process [Benger et al., 2009a].

8.2.2 The Integration Module

To compute and visualize the integration geometry the task is split in three parts: definition of the seeding (emitter) geometry, integration based on a data field and rendering. Each task is taken care of by a different module in the *VISH* network.

Here, we present the framework that provides a computational module based on template specializations. Template programming allows to write flexible and reusable code, without performance losses due to late binding or loose coupling. The compiler directly inserts source code of template specializations dependent on their template parameters, which can be highly optimized during compilation. The

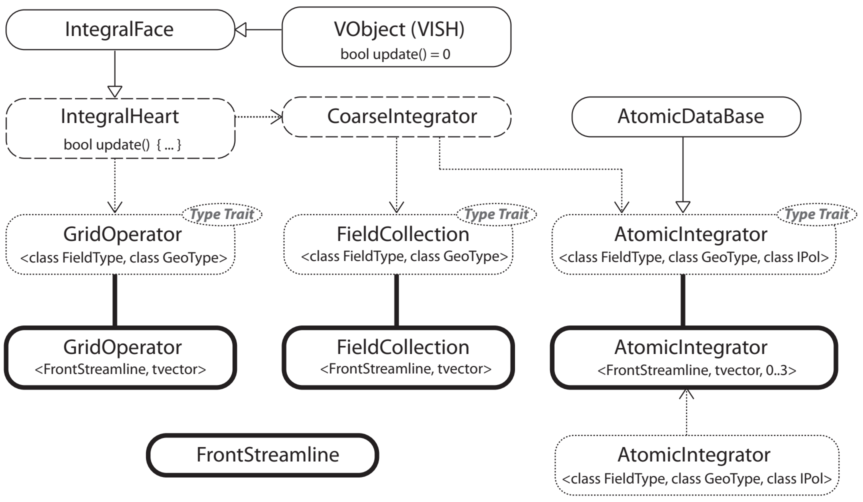


Figure 8.2: Class organization for the integration framework. Classes illustrated in dashed lines are template classes which define an interface but also provide a default implementation. Classes illustrated in dotted lines are template traits which only define an interface by empty functions. They have to be specialized and implemented. Four classes have to be implemented, indicated by the bold outline. The implementation of streamline integration is shown as an example.

overhead of function calls is not relevant in that context since it is removed during compilation. See section (8.3.2) for a comparison of timings runtime measurements between `optimize` versus `debug` compilation. Template programming provides a programming language on its own "executed" at compile time [Veldhuizen, 1995] [Furnish, 1998] [Vandevoorde & Josuttis, 2003].

Fig. 8.2 illustrates the main class relationships for the example of a streamline integration. The four classes outlined in bold have to be customized for that purpose: `GridOperator`, `FieldCollection` and `AtomicIntegrator` have to be specialized and an empty class defining a type has to be introduced. Here, this class is called `Streamline` in Fig. 8.2.

We derive a class called `IntegralFace` from `VObject` and define all the input and output connections necessary for a computation module. Derivation from `VObject` makes a *VISH* network module. `IntegralFace` implements no additional functionality. It provides the following input connections: input initial grid, input base field, interpolation type of the base field (linear, cubic, analytic), integration type (Euler, Dop853), length of integration geometry (or their trajectories), step size and maximum steps. The integration geometry grid is output by the module. From `IntegralFace` we derive the central computation class called

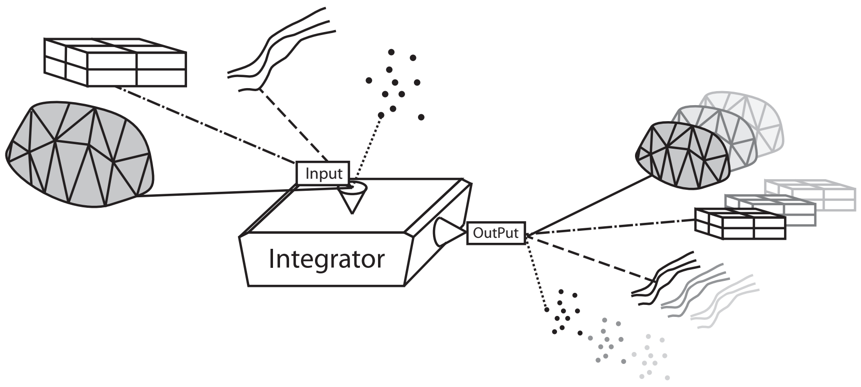


Figure 8.3: Any Grid object can define the seeding geometry and input to and output from the integrator. Examples are a triangular surface, a uniform grid, lines and points. A Grid may also hold data, for example scalar or vector field data. The output Grid depends on the input Grid and is of the same type but with an additional dimension.

`IntegralHeart`:

```
template <typename FieldType, typename IntGeoType>
class IntegralHeart : public IntegralFace
```

It implements the `update()` function used in the network updating process and hosts the main integration loop. Tasks such as retrieving the initial geometry, the data field and the user control parameters and outputting the integration geometry are brought together here. The module handles all different kinds of input grids and data fields by utilizing the fiber bundle library. A input grid can, for example, be a point cloud, a surfaces or a connected curvilinear grid, as illustrated in Fig. 8.3. It is also possible to have additional data such as directional fields stored in the emitter grid, if additional initial conditions besides positions are required. Similarly, the field used in the integration can be of arbitrary type such as scalar, vector or a tensor field specified in any topology or representation.

The two template parameters `FieldType` and `IntGeoType` describe the types of the integration field and the type of the integration geometry. In the streamline example the field type would be a vector field and the integration geometry the empty class `Streamline`. The main computation loop utilized template type trait classes for doing the integration, Fig. 8.4, illustrated by the dotted outlined classes in Fig. 8.2. The traits define empty functions being called from `IntegralHeart`. One additional template parameter is introduced: `int InterpolationType`. It controls the choice of the interpolation scheme used in the integration field.

The main loop uses two template trait classes, see Fig. 8.2: `GridOperator` and `CoarseIntegrator`. The `GridOperator`

```
template <class FieldType, typename IntGeoType>
class GridOperator {...}
```

is responsible for the geometry that is created during the integration process and has to be implemented for a specific type. Several functions may be provided in the template specialization to control the geometry: `prepare()`, `advance()`, `refine()`, `store()` and `finalize()`. The functions `prepare()` and `finalize()` allow initial and final operation on the grid. The other functions are called in each call of the main loop and are responsible for advancing, storing and refining the grid. Using only `Grid` objects as function parameters allows maximum flexibility and power in the functions to modify or create geometry. All topological information is available in a `Grid`, which is necessary in the `refine()` function in case of doing adaptive geometry refinement.

The `CoarseIntegrator` trait provides two functions to the main loop:

```
template <class FieldType, typename IntGeoType, int InterpolType>
class  CoarseIntegrator {...}
```

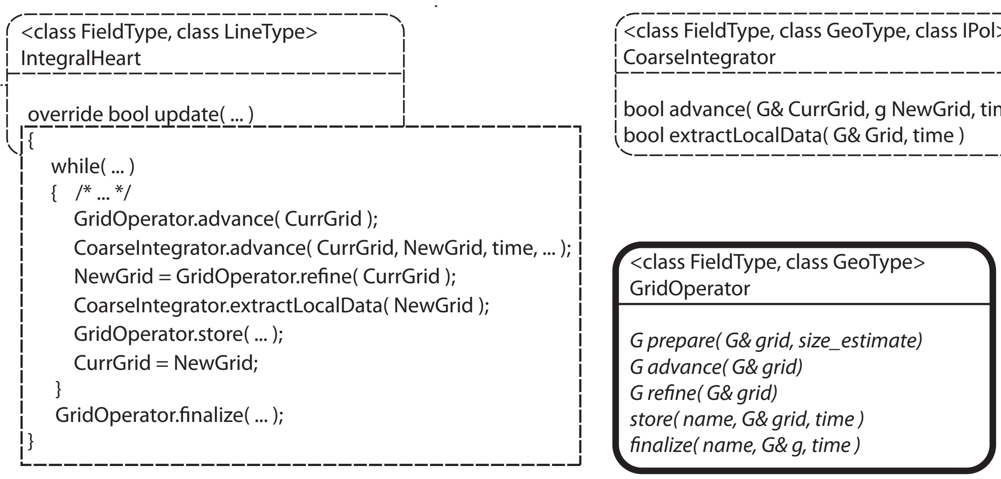
`advance()` and `extractLocalData()`. The `advance()` function is responsible for the integration in a coarse sense. It itself uses the trait `AtomicIntegrator`

```
template<typename FieldType, typename IntGeoType, int InterpolType>
class AtomicIntegrator {...}
```

which implements the integration on a low level per point basis. The `Coarse-Integrator` advances a collection of points. The default template implementation does a breadth-wise integration by advancing fronts. A depth-wise integration or line-wise integration can be added by providing a different template specialization. The `extractLocalData()` function collects all data besides the vertex data by again calling the according function from the `AtomicIntegrator` over a the collection of vertices which utilizes the `FieldInterpolator` template, a fully implemented template class that returns a linear or cubic interpolation value of a given point in the field. The interpolator also can return an analytic value if a formula for the data field available explicitly.

The `advance()` function of the `CoarseIntegrator` extracts data fields of the current grid object into the so called

```
template <class FieldType, typename IntGeoType>
struct FieldCollection : public MemCore::ReferenceBase
                         <FieldCollection<FieldType,IntGeoType> >
{ ... } .
```



```
<class FieldType, class LineType>
IntegralHeart

override bool update( ... )
{
  while( ... )
  {  /* ... */
     GridOperator.advance( CurrGrid );
     CoarseIntegrator.advance( CurrGrid, NewGrid, time, ... );
     NewGrid = GridOperator.refine( CurrGrid );
     CoarseIntegrator.extractLocalData( NewGrid );
     GridOperator.store( ... );
     CurrGrid = NewGrid;
  }
  GridOperator.finalize( ... );
}
```

```
<class FieldType, class GeoType, class IPol>
CoarseIntegrator

bool advance( G& CurrGrid, g NewGrid, time )
bool extractLocalData( G& Grid, time )
```

```
<class FieldType, class GeoType>
GridOperator

G prepare( G& grid, size_estimate)
G advance( G& grid)
G refine( G& grid)
store( name, G& grid, time )
finalize( name, G& g, time )
```

Figure 8.4: More detailed illustration of class functions. All functionality is inserted into the `IntegralHeart`'s `update()` function by the compiler which uses the template trait functions of `GridOperator` and `CoarseIntegrator`. Reference pointers are used when pointers are required. `G` is short for `RefPtr<Grid>`.

The field collection provides data necessary for integration on an array basis used by the functions of the `AtomicIntegrator`. Thus, the atomic integration operation need not to extract this data from the grid in every integration step. `AtomicIntegrator` provides doEuler() and doDop853() functions to process every points on the integration coarse.

The following subsections describe the implementation of four different integration geometries using the presented framework.

8.2.3 Streamline Implementation

To implement the streamline integration several classes are gathered in a separate `cpp` file. An empty class `class FrontStreamline{};` is defined and the `GridOperator` specialized:

```
template<> class GridOperator<tvector, FrontStreamline> {...}
```

where `tvector` is the type of the vector field. Its `prepare()` member function retrieves the emitter grid and prepares a new integration geometry grid by copying the vertices. The geometry will later be stored as a set of lines in the bundle of the vector field. It also estimates the size of the data being computed and reserves memory. The `advance()` just passes the given grid through without any operation. The `refine()` function also passes the grid through. Refinement will by implemented in future. The `store()` function is an empty function. Nothing is stored per time-step. Finally, the `finalize()` function stores the filled data fields at the given time step into the bundle.

The `FieldSelection` specialization extracts `std::vectors` from the integration geometry grid and provides them to the `AtomicIntegrator` partial specialization:

```
template<int InterpolType>
class AtomicIntegrator<tvector, FrontStreamline, InterpolType>
: public AtomicDataBase {...};
```

Here, the `doEuler()` function is implemented which uses an index to access the correct vertex and direction. These are retrieved from the `Field- Collection` provided by the `CoarseIntegrator`. It computes and pushes the next vertex position and line connection into the `FieldCollection`. The `extractLocalData()` function extracts the interpolated vector field data for the vertex positions previously computed by the integration function by utilizing the `FieldInterpolator` template.

Finally, a *VISH* network module is provided by a template instantiation:

```
typedef IntegralHeart<tvector,FrontStreamline> FrontStreamlines;
```

Implementing streamline computation requires 350 lines of source code.

8.2.4 Pathlines Implementation

The pathline integration again requires a type `class FrontPathline{};` and a specialization of

```
template<> class GridOperator<tvector, FrontPathline> { ... };
```

The representation of a pathline differs from the representation of a streamline in the fiber data bundle. For a pathline a vertex is stored as grid in different time slices. The `prepare()` function also creates a new grid object and copies the vertex data from the emitter grid. However, the `advance()` function now creates a new grid object for each step of the integration front. The `refinement()` function is to be implemented in future and just passes the grid though. The `store()` function inserts the computed grid into the bundle for each integration front step. Here, the `finalize()` function is an empty implementation. The `FieldSelection` again extracts the array data from the grid object and provides array data.

The specialization of

```
template<int InterpolType>
class AtomicIntegrator<tvector, FrontPathline, InterpolType>
: public AtomicDataBase {...};
```

implements the `doEuler()` function, using a provided time step value. It uses direct array index access instead of using push backs (in the streamline case). The `extractLocalData()` function interpolates the vector field and stores directions at each vertex.

A *VISH* module is provided by the template instantiation:

```
typedef IntegralHeart<tvector,FrontPathline> FrontPathlines;
```

Pathline computation requires 250 lines of source code.

8.2.5 Material-Line Implementation

The pathline module can be reused for the material-line implementation. The fiber bundle data stores topology information inside a Grid: A so called relative Representation on the vertices in a Skeleton. The pathline module is extended, copying this additional information in the `GridOperators` `prepare()` and `advance()` functions. This was implemented for an arbitrary number of additional Skeletons on the vertices using a Skeleton iterator. 50 additional lines are required in the framework and 4 lines for the pathline implementation.

8.2.6 Time Surfaces

The computation of the time surface did not need any additional development because the Skeleton iterator, implemented for material-lines, already copies the topological surface connectivity data.

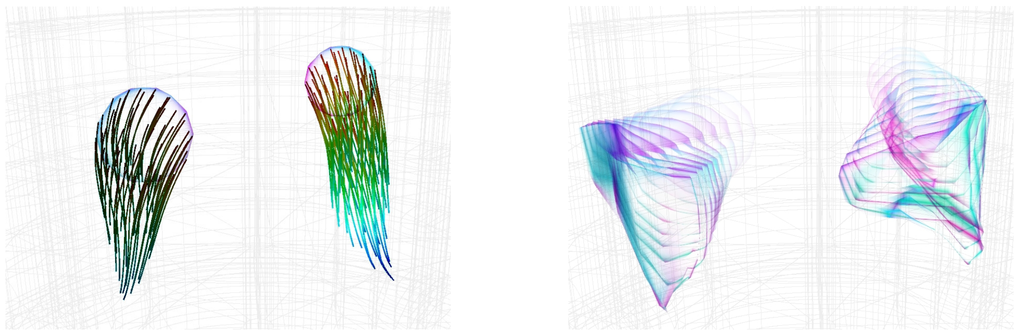


Figure 8.5: Images showing the streamlines [left] and the evolving time surfaces [right], with a two sphere geometry as a seeding grid.

8.3 Results

8.3.1 CFD Visualization of a Stirred Tank

Having the two computing modules implemented for streamline and pathline integration and having extended the modules providing the seeding geometries we are able to visualize streamlines, pathlines, material-lines and time surfaces in a uniform test grid and in a 2088 multi-block curvilinear dataset stemming from a CFD simulation of a stirred tank. The curvilinear grid is comprised of 3.1 million cells in total, with flow variables such as velocity and pressure measured at the cell vertices [Benger et al., 2009c]. The integration module currently implemented is explicit Euler. The figures show the results of the integration in the curvilinear dataset. The computation was done for 50 time steps using SVN revision 2557 of *VISH*. The material-lines, time surfaces and pathlines in Fig. 8.6 and 8.7 are rendered for every 5th time step. They are fading linearly in time, highlighting the current state.

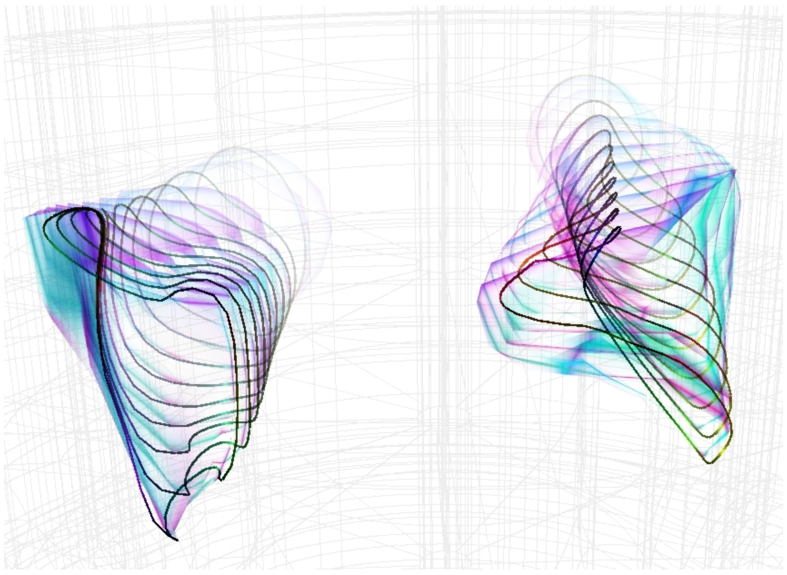


Figure 8.6: Image showing corresponding time surfaces overlapped on material lines [right], both computed for 50 time steps.

8.3.2 Time Measurements

We did measurements of the computation time comparing our old implementations and the new ones in a simple uniform grid based dataset and the stirred tank data set. For testing we used a machine equipped with a six core Intel Xeon

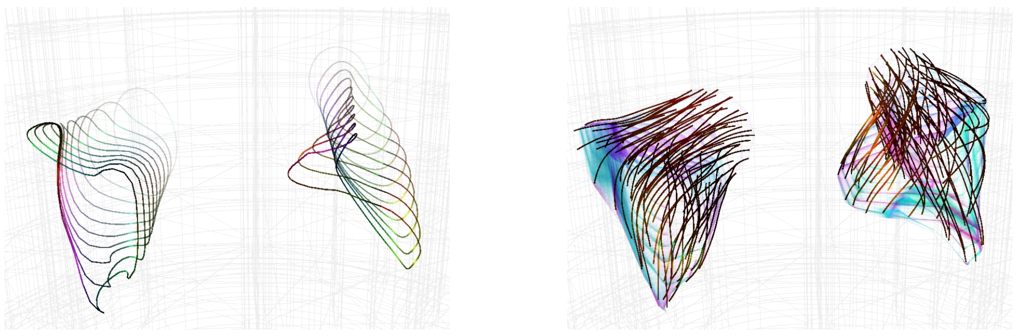


Figure 8.7: Images showing the material lines [left] and time surfaces with pathlines overlapped [right], all computed for 50 time steps.

W3680@3.33GHz, 12MB L3, 6.4GT/s with 6GB of 1333MHz DDR3 SDRAM and a NVIDIA Quadro FX 3800 1GB running gcc version 4.4.4 20100630 (Red Hat 4.4.4-10).

Imple-mentaion	**Data-set**	**Comp-ilation**	**Steps** #	**Step-Time** [msec]	**N.-Time** [%]	**Spd-Up** [-]	**Spd-Up** [-]
stream old	uni.	debug	2704	0.19	*100*		
stream old	curvi.	debug	6739	3.10	100		
stream old	uni.	opt.	2704	0.03	***16***	*6.3*	
stream old	curvi.	opt.	6739	1.40	45	2.2	
stream new	uni.	debug	2600	0.19	*100*		1.0
stream new	curvi.	debug	6600	5.40	174		**0.6**
stream new	uni.	opt.	2600	0.03	***16***	*6.3*	1.0
stream new	curvi.	opt.	6600	2.60	84	2.1	**0.5**
path old	uni.	debug	2600	0.20	*105*		
path old	curvi.	debug	6732	429.88	13867		
path old	uni.	opt.	2600	0.03	***16***	*6.7*	
path old	curvi.	opt.	6732.21	224.00	7226	1.9	
path new	uni.	debug	2600	0.21	*111*		1.0
path new	curvi.	debug	6600	5.61	181		**76.6**
path new	uni.	opt.	2600	0.03	***15***	*7.0*	1.0
path new	curvi.	opt.	6600	2.61	84	2.1	**85.8**

The table gathers: Old and new stream and pathline using debug and optimized compilation, number of integration steps, time per integration step, normalized-time with respect to the old uniform and curvilinear grid streamline computation, speedup by optimized compilation mode and speedup by the new implementation.

When comparing the timings of the old and the new integration in the uniform grid the measurements show no difference. The introduced overhead of the framework does not result in a longer computation time. In the curvilinear case of streamlines the old computation is faster by a factor of about two. This has to be investigated. However, the new curvilinear pathlines benefit a speedup of about factor of 80 by making faster interpolation and point search algorithms available.

8.4 Conclusion

While the earlier version of integration modules were implemented independent of each other with redundant computation code and time, we successfully designed and implemented a framework based on template specializations that provides a common computational module for different integral geometries. We introduced the visualization of material lines with minimal programming effort: 350 lines for streamlines, 250 lines for pathlines, 54 lines for material lines and none for time surfaces.

8.5 Future Work

We will adapt our existing three and four dimensional geodesic tensor field integration code to the framework, enable DOP853 integration, introduce grid refinement during integration, introduce a module to extract a path-surface from computed material lines and work on a thread-based parallelization on the CPU using OpenMP [OpenMP Architecture Review Board, 2010].

Acknowledgments

We thank the VISH development team, among them Georg Ritter, Science and Technology Research Institute at the University of Hertfordshire, and Hans-Peter Bischof, Rochester Institute of Technology, for their support. This research employed resources of the Center for Computation & Technology at Louisiana State University and was supported by the projects "Modeling and Visualizing the Effect of Severe Storms on Oil Spill Trajectories with the Cactus Framework" and "Visualization of Oil Spill Related Data" awarded by the LONI Network Deployment & Operation [The LONI Institute, 2010].

Bibliography

[Benger, 2004] Benger, W. (2004). *Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model*. PhD thesis, FU Berlin.

[Benger et al., 2007] Benger, W., Ritter, G., & Heinzl, R. (2007). The Concepts of VISH. In *4th High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007* (pp. 26–39).: Berlin, Lehmanns Media-LOB.de.

[Benger et al., 2009a] Benger, W., Ritter, G., Ritter, M., & Schoor, W. (2009a). Beyond the visualization pipeline. In *5th High-End Visualization Workshop, Baton Rouge, Louisiana, March 18th - 21st, 2009*: Berlin, Lehmanns Media-LOB.de.

[Benger et al., 2009b] Benger, W., Ritter, G., Su, S., Nikitopoulos, D. E., Walker, E., Acharya, S., Roy, S., Harhad, F., & Kapferer, W. (2009b). Doppler speckles - a multi-purpose vectorfield visualization technique for arbitrary meshes. In *CGVR'09 - The 2009 International Conference on Computer Graphics and Virtual Reality*.

[Benger et al., 2009c] Benger, W., Ritter, M., Acharya, S., Roy, S., & Jijao, F. (2009c). Fiberbundle-based visualization of a stir tank fluid. In *17th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (pp. 117–124). URL: `http://wscg.zcu.cz/WSCG2009/Papers_2009/!_WSCG2009_Short_final.zip`.

[Bohara et al., 2010] Bohara, B., Harhad, F., Benger, W., Brener, N., Iyengar, S., Ritter, M., Liu, K., Ullmer, B., Shetty, N., Natesan, V., Cruz-Neira, C., Acharya, S., Roy, S., & Karki, B. (2010). Evolving time surfaces in a virtual stirred tank. In V. Skala (Ed.), *18th International Conference on Computer Graphics, Visualization and Computer Vision'2010*.

[Deuflhard & Bornemann, 2002] Deuflhard, P. & Bornemann, F. (2002). *Scientific Computing with Ordinary Differential Equations*. Springer Verlag, New York.

[Furnish, 1998] Furnish, G. (1998). Container-Free Numerical Algorithms in C++. *Computers in Physics*, 12(3).

[Hairer et al., 2000] Hairer, E., Norsett, S. P., & Wanner, G. (2000). *Solving Differential Equations I*. Springer-Verlag Berlin Heidelberg.

[Krishnan et al., 2009] Krishnan, H., Garth, C., & Joy, K. I. (2009). Time and streak surfaces for flow visualization in large time-varying data sets. *Proc. IEEE Visualization '09*.

[McLoughlin et al., 2009] McLoughlin, T., Laramee, R. S., & Zhang, E. (2009). Easy integral surfaces: a fast, quad-based stream and path surface algorithm. In *Proceedings of the 2009 Computer Graphics International Conference*, CGI '09 (pp. 73–82). New York, NY, USA: ACM. URL: `http://doi.acm.org/10.1145/1629739.1629748`.

[OpenMP Architecture Review Board, 2010] OpenMP Architecture Review Board (2010). OpenMP. URL: `http://openmp.org/wp/`.

[Ritter, 2010] Ritter, M. (2010). Computing Geodesics in Numerical Space Times. Master's thesis, Institute of Computer Science, University of Innsbruck. `http://www.fiberbundle.net/papers/da_geodesics_print.pdf`.

[Ritter & Benger, 2010] Ritter, M. & Benger, W. (2010). Visualizing coordinate acceleration and christoffel symbols. *IADIS Computer Graphics, Visualization, Computer Vision and Image Processing 2010 (CGVCVIP 2010)*.

[Stalling, 1998] Stalling, D. (1998). *Fast Texture-Based Algorithms for Vector Field Visualization*. PhD thesis, Free University Berlin.

[The LONI Institute, 2010] The LONI Institute (2010). LONI. URL: `http://www.loni.org/`.

[Vandevoorde & Josuttis, 2003] Vandevoorde, D. & Josuttis, N. M. (2003). *C++ Templates - The Complete Guide*. Addison Wesley.

[Veldhuizen, 1995] Veldhuizen, T. L. (1995). Using C++ template metaprograms. *C++ Report*, 7(4), 36–43. Reprinted in C++ Gems, ed. Stanley Lippman. URL: `http://extreme.indiana.edu/~tveldhui/papers/`.

[Weinkauf & Theisel, 2010] Weinkauf, T. & Theisel, H. (2010). Streak lines as tangent curves of a derived vector field. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization 2010)*, 16(6), 1225–1234. Received the Vis 2010 Best Paper Award. URL: `http://tinoweinkauf.net/`.