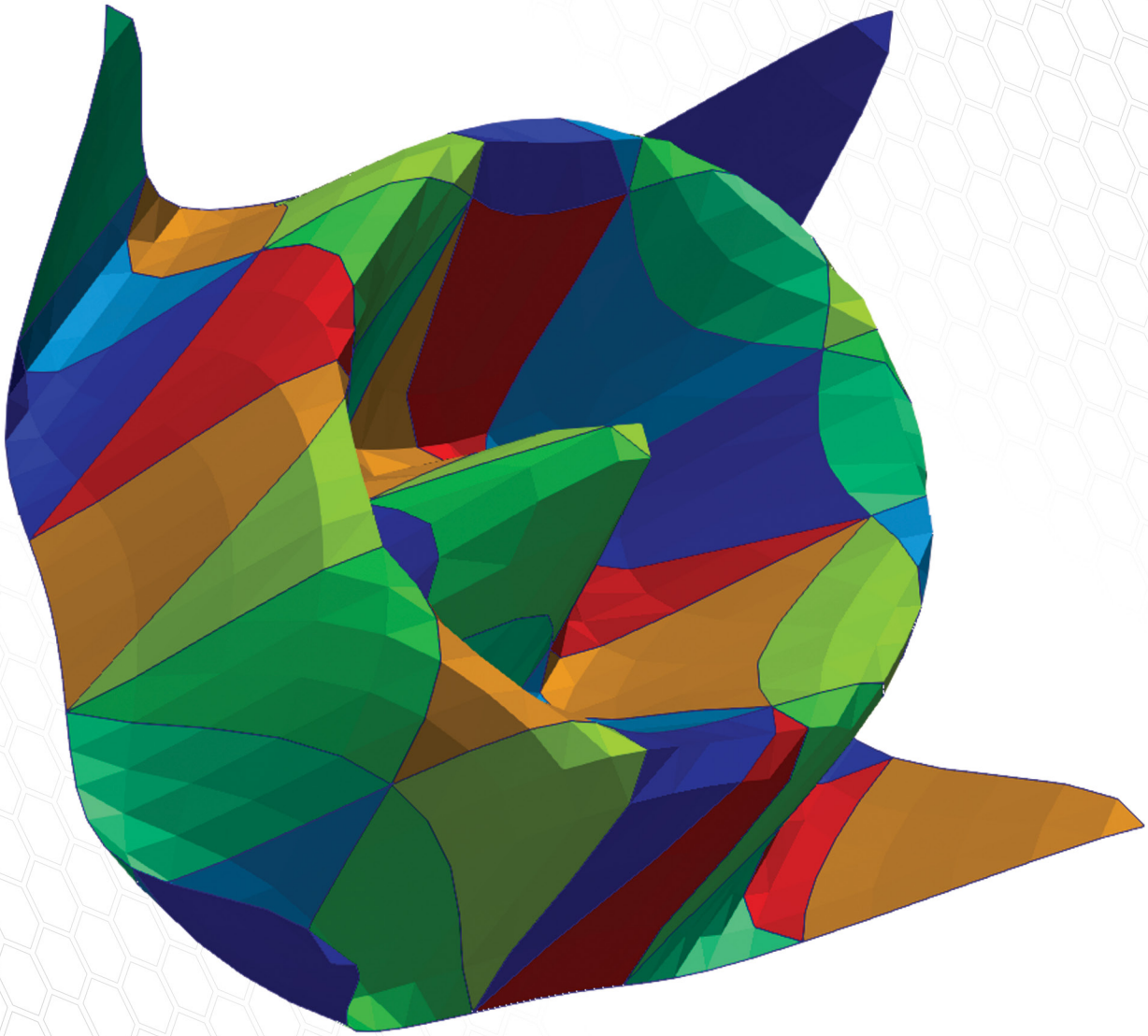
Kitware

SOURCE

A PUBLICATION FOR SOFTWARE DEVELOPERS

Issue 43



p.6

Arbitrary-order Lagrange cells in the Visualization Toolkit

CONTENTS

Kitware Source contains information on open source software. Since 2006, its articles have shared first-hand experiences from Kitware team members and those outside the company's offices who use and/or develop platforms such as CMake, the Visualization Toolkit, ParaView, the Insight Segmentation and Registration Toolkit, Resonant and the Kitware Image and Video Exploitation and Retrieval toolkit. Readers who wish to share their own experiences or subscribe to the publication can connect with the Kitware Source editor at comm@kitware.com.

Kitware Source comes in multiple forms. Kitware mails hard copies to addresses in North America, and it publishes each issue as a series of posts on https://blog.kitware.com.

GRAPHIC DESIGNER
Steve Jordan

EDITOR
Sandy McKenzie

COVER CONTENT
The new Lagrange cells in the Visualization Toolkit can capture complex behavior within a single cell. The image on the cover shows 50 fifth-order Lagrange triangles colored by cell. See "Modeling Arbitrary-order Lagrange Finite Elements in the Visualization Toolkit," which begins on page six, for more renderings of Lagrange cells.

COMPUTING GRADIENTS IN PARAVIEW FOR DATASETS WITH DIFFERENT CELL DIMENSIONS

Andrew Bauer

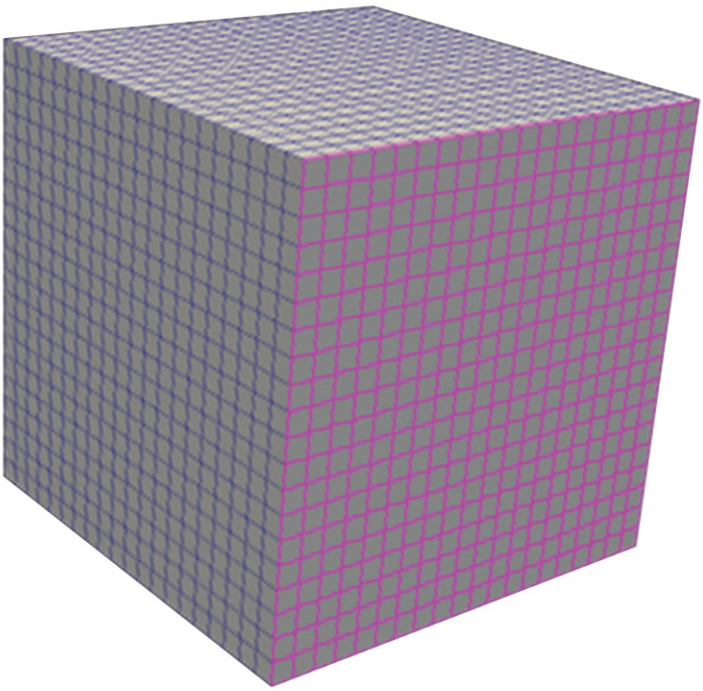
In scientific visualization, gradient computations vary based on the use case. For use cases with shallow water simulations or free surface simulations, cells of different dimensions should contribute to the computations in some manner. For other use cases, cells of different dimensions should not contribute to the computations.

ParaView, a very general tool for analyzing and visualizing data, can compute gradients. This article examines two use cases of ParaView. Both use cases rely on the `Calculator` filter to generate a Point Data field from input. The use cases then apply the `GradientOfUnstructuredDataSet` filter to compute results.

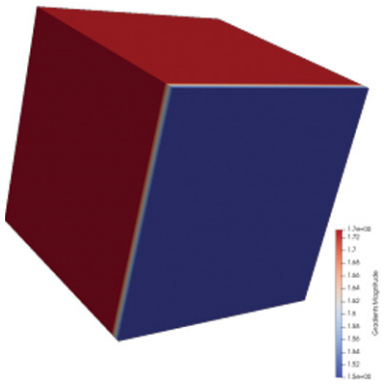
Use Case 1

The first use case employs a dataset that has both two-dimensional (2D) and three-dimensional (3D) cells, where the 2D cells serve as the faces of a subset of the 3D cells.

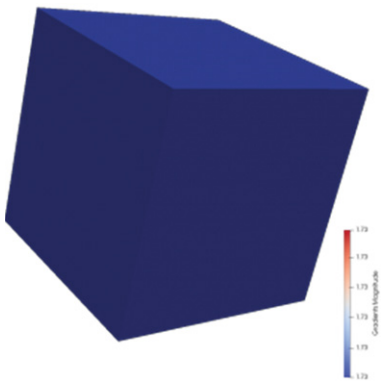
ParaView generates a Point Data field from the dataset, using the `Calculator` filter and the following input: `coordsX+coordsY+coordsZ`. When the `GradientOfUnstructuredDataSet` filter is applied to the Point Data field, the expected result is a tuple of [1, 1, 1] throughout the dataset. Since 2D cells incorrectly contribute to the computation, however, the gradient is wrong where 2D cells are present.



Two-dimensional cells are outlined in purple.



GradientOfUnstructuredDataSet produces an unexpected result.



DataSetMax produces the expected result.

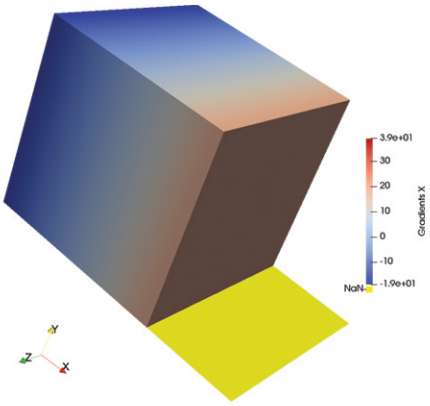
A new feature of `GradientOfUnstructuredDataSet` produces the expected result. This feature, called `ContributingCellOption`, helps to control what cells contribute to the computation. Note that this is an advanced feature; by default, it will not be visible in the `GradientOfUnstructuredDataSet` options in the graphical user interface of ParaView.

There are three enumerations for `ContributingCellOption`: `DataSetMax`, `All` and `Patch`. The default, `DataSetMax`, only uses cells with the highest dimension in a dataset to compute gradient quantities. Unlike `DataSetMax`, `All` uses each cell in the dataset to compute gradient quantities. This option provides the behavior for ParaView 5.4 and earlier. Alternatively, `Patch` only considers cells that contribute to the local gradient computation. From these cells, `Patch` selects those with the highest dimension.

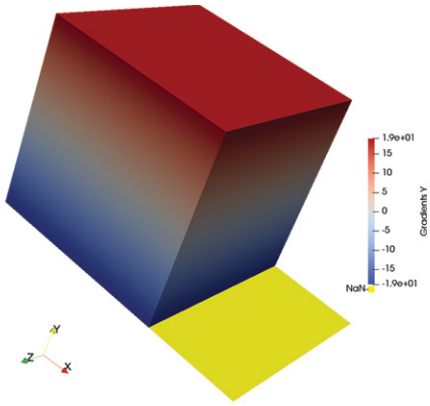
Use Case 2

Another use case employs a dataset in which 3D cells are present in one part of the domain, and 2D cells are present in another part. To be clear, these parts are connected.

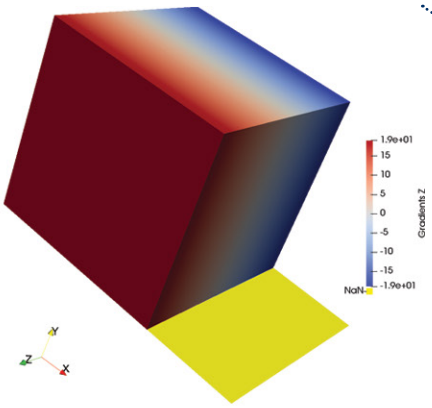
For this use case, `coordsX*coordsX+coordsY*coordsY+coordsZ*coordsZ` serves as the input to the `Calculator` filter. While the Points Data field that the filter generates is slightly more interesting than the one in the previous example, a numerical result from this use case can still be compared to an analytical result. Here are the results for the X-component, the Y-component and the Z-component of the gradient.



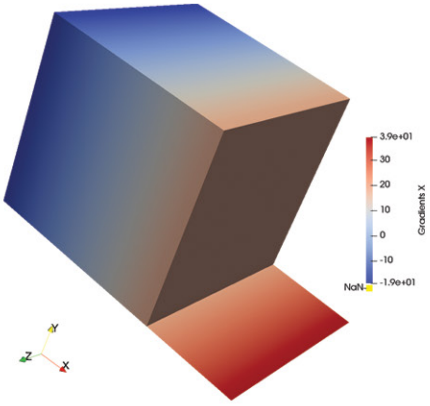
DataSetMax, X



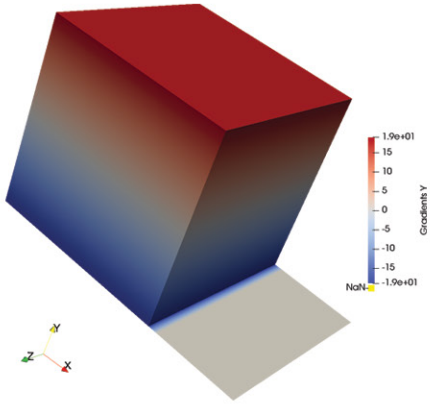
DataSetMax, Y



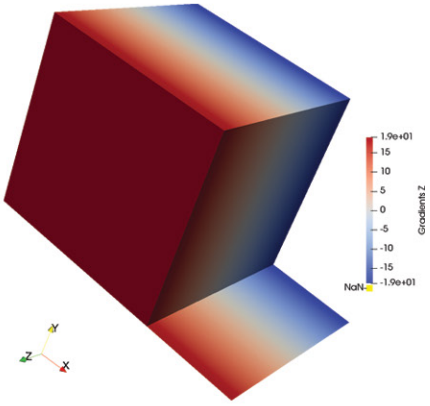
DataSetMax, Z



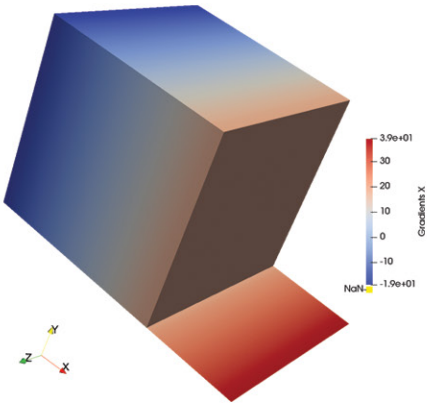
All, X



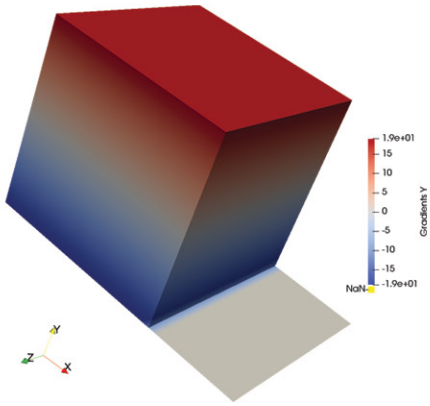
All, Y



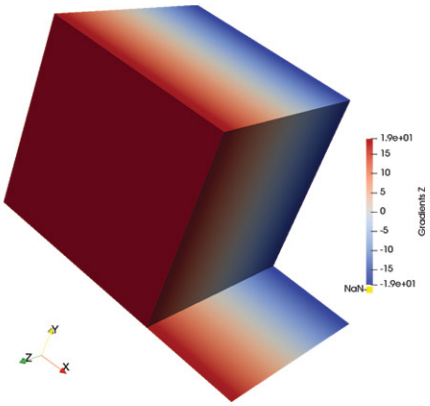
All, Z



Patch, X



Patch, Y



Patch, Z

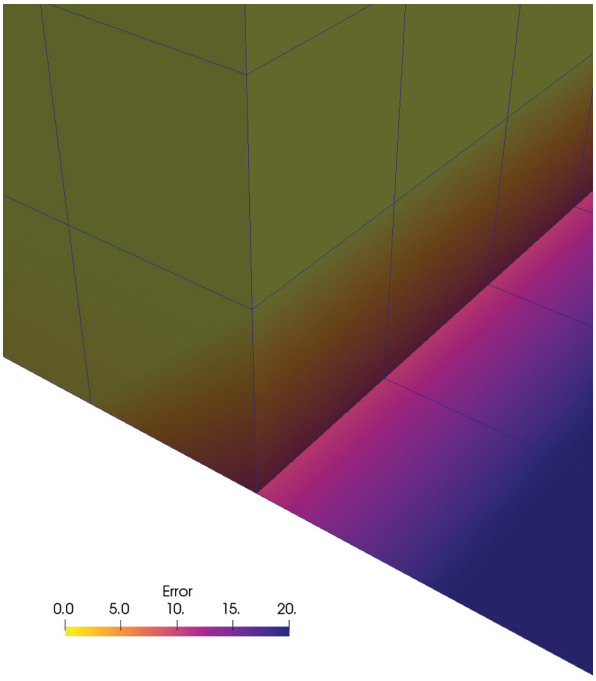
The results of `DataSetMax` are likely not desired in this case, as the filter produces Not-a-Number (NaN) values over the 2D cells. This is shown by the yellow pseudocoloring in the above images. While `All` and `Patch` give the expected results for the X-component and the Z-component, the results for the Y-component call for further analysis. This analysis can be performed by comparing the numerical result of the Y-component for each computation with the analytical result of `2*coordsY`. The `Calculator` filter can make this comparison using

the `abs(Gradients_Y-2*coordsY)` expression, if "Error" is designated in the field for Result Array Name.

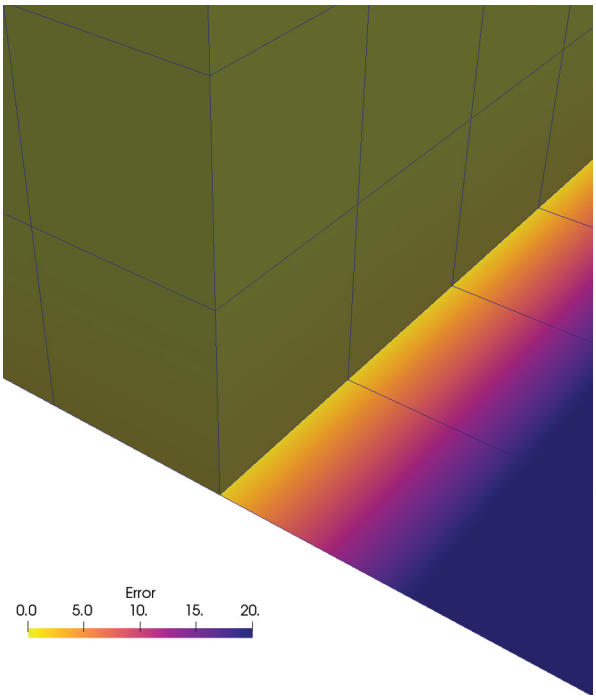
The images on the right show the results of `All` and `Patch` pseudocolored by "Error." In the top image, `All` produces an error in 3D cells that use the same points as 2D cells. `Patch` does not produce this error, as the bottom image demonstrates. This result is expected, as `Patch` ignores contributions from 2D cells for points that are connected to 3D cells.

In addition, both `All` and `Patch` produce an error in 2D cells. Since 2D cells are located in the XZ-plane, there is no depth in the Y-direction. Therefore, there is no variation in the Y-direction, and the computed Y-component of the gradient is zero.

In light of the fact that this use case does not have valid values for all gradient quantities, another new feature of `GradientOfUnstructuredDataSet` comes into play. This feature is called `ReplacementValueOption`. Like `ContributingCellOption`, `ReplacementValueOption` is an advanced feature. Its enumerations are `NaN`, which is the default; `Zero`; `DataTypeMin`; and `DataTypeMax`. For this use case, the results for `NaN` should trigger the realization that something is amiss with the `ContributingCellOption` results. A possible solution is to threshold out the cells with NaN values.



A close-up of the All option reveals an error.



A close-up of the Patch option reveals a properly computed gradient.

Acknowledgment

The datasets for both use cases are available at https://data.kitware.com/#folder/5a1c85008d777f7ddd99ab8b. While this article explores the datasets in ParaView, the new features of `GradientOfUnstructuredDataSet` come from `vtkGradientFilter` in the Visualization Toolkit (VTK). For backwards compatibility, the default behavior for `vtkGradientFilter` uses the `All` option in `ContributingCellOption`.

The work in VTK and ParaView was funded by Sandia National Laboratories. In particular, Ken Moreland from Sandia took part in discussions on how to improve gradient computation with VTK and ParaView for a variety of use cases.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Meet the Author

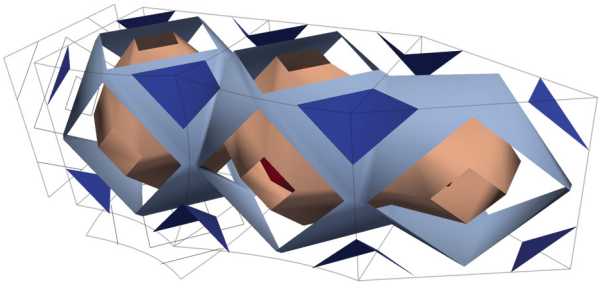
Andrew Bauer *is a staff R&D engineer on the high-performance computing (HPC) and visualization team at Kitware. He primarily works on enabling tools and technologies for HPC simulations.*

MODELING ARBITRARY-ORDER LAGRANGE FINITE ELEMENTS IN THE VISUALIZATION TOOLKIT

Thomas J. Corona, David Thompson

Two techniques that are used by simulations are geometric refinement (also known as hierarchical refinement, or h-refinement) and polynomial refinement (also known as p-refinement). Traditional simulations often apply geometric refinement, which splits cells with low accuracy to resolve finer spatial features. More recent finite element simulations may also apply polynomial refinement. This technique enhances existing cells, so they can fit a higher-order polynomial to the simulation solution. As a result, the cells can represent more complex functions.

Cells that experience polynomial refinement have an order greater than one. For many years, the Visualization Toolkit (VTK) has handled different orders with different cell types. More specifically, it has complete sets of linear (order one) and quadratic (order two) cells for all cell types. Until recently, VTK only supported cells of order greater than two on a case-by-case basis. Thanks to support for new cells, called Lagrange cells, VTK can now render approximations to curves, triangles, quadrilaterals, tetrahedra, hexahedra and wedges of any order up to 10. VTK can also manage cells of order greater than 10 with simple, compile-time changes.

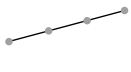


A wireframe outlines several Lagrange cells—two hexahedra, two quadrilaterals and one wedge—along with isocontours of ellipsoidal scalar functions defined by the cells.

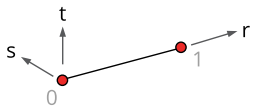
The new cells in VTK use Lagrange polynomials, which are common in mathematics. The polynomials allow VTK to recursively evaluate shape functions for arbitrary orders. Depending on the polynomial order, the number of points for edges, faces and volumes may vary in Lagrange cells. Thus, Lagrange cells differ from their pre-existing analogues, which expect a fixed number of points in a predetermined order.

Lagrange Cells

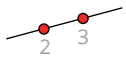
The number of points in a Lagrange cell determines the order over which they are iterated relative to the parametric coordinate system of the cell. The first points that are reported are vertices. They appear in the same order in which they would appear in linear cells. Mid-edge points are reported next. They are reported in sequence. For two- and three-dimensional (3D) cells, the following set of points to be reported are face points. Finally, 3D cells report points interior to their volume.



Lagrange curve

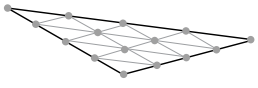


vertices

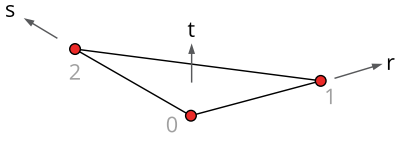


edges

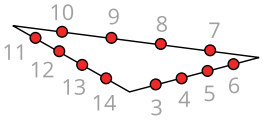
For simplicial shapes such as triangles, edge points are reported in a counterclockwise order. This order matches the order in which the points would appear in standard representations. Face points are reported next. They are reported as the points of a lower-degree Lagrange triangle. Vertices on this lower-degree triangle are reported first, followed by edge points and then face points, as described above. For higher orders, the process of reporting face points repeats until no points remain.



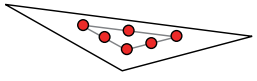
Lagrange triangle



vertices

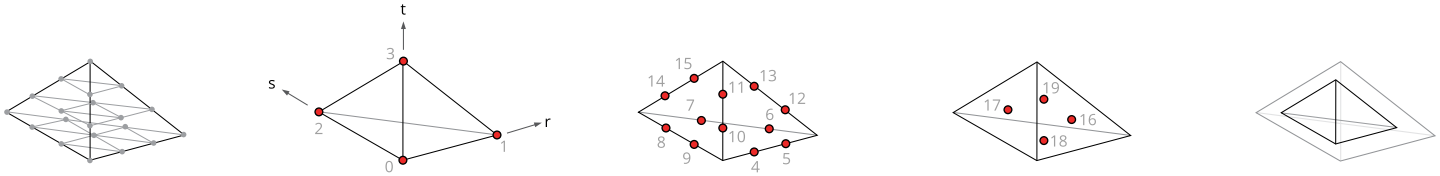


edges



faces

Like triangles, tetrahedra are simplicial shapes. Therefore, tetrahedral points are reported using the same method of recursion that is used to report triangular points.

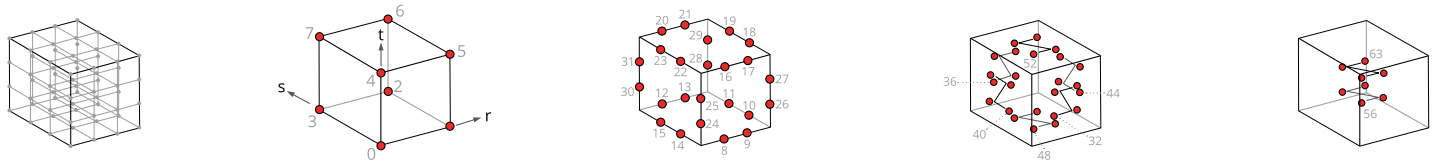


Lagrange tetrahedron | *vertices* | *edges* | *faces* | *volume*

For higher orders, the process of reporting face points repeats per triangular face. To report volume points, the outer shell of vertices, edge points and face points is removed. What remains is a lower-order Lagrange tetrahedron. Vertices on this inner tetrahedron are reported first, followed by edge points and face points. The process of removing shells repeats until no points remain. In this manner, reporting points is like peeling an onion.

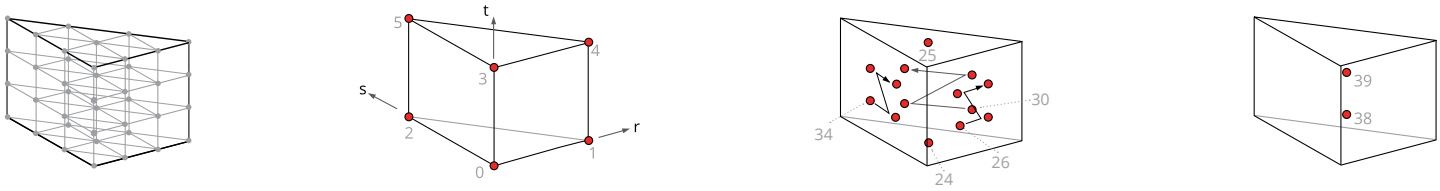
For prismatic shapes such as curves, quadrilaterals and hexahedra, edge points are reported from the lowest r, s or t parameter to the highest. Face points and volume points are also reported this way.

Lagrange quadrilateral | *vertices* | *edges* | *faces*



Lagrange hexahedron | *vertices* | *edges* | *faces* | *volume*

Wedges adopt the prismatic cell ordering with two exceptions: one for triangular faces and one for quadrilateral faces. On triangular faces, edge points are ordered counterclockwise. On quadrilateral faces, face points are reported in axis order. The axis that connects the first two points on each face can be traced from quadrilateral to quadrilateral on a path that proceeds counterclockwise around the triangular faces.



Lagrange wedge | *vertices* | *faces* | *volume*

Example Code

The pre-existing eXtensible Markup Language-based (XML-based) readers/writers in VTK have been extended to handle Lagrange cells. An example file is located on https://data.kitware.com in the Lagrange Cell Examples collection. While some elements in the example have a curved shape, every element has a curved (ellipsoidal) scalar function that is defined over each cell. This scalar function illustrates that higher-order cells can have interior minima, maxima and other critical points, unlike many linear cells.

The following Python code describes how to create individual cells and add them to an unstructured grid to make a Lagrange tetrahedron.

```
import math
import vtk

# Let's make a sixth-order tetrahedron
order = 6
# The number of points for a sixth-order tetrahedron is
nPoints = (order + 1) * (order + 2) * (order + 3) / 6;

# Create a tetrahedron and set its number of points. Internally, Lagrange cells
 # compute their order according to the number of points they hold.
tet = vtk.vtkLagrangeTetra()
tet.GetPointIds().SetNumberOfIds(nPoints)
tet.GetPoints().SetNumberOfPoints(nPoints)
tet.Initialize()

point = [0.,0.,0.]
barycentricIndex = [0, 0, 0, 0]

# For each point in the tetrahedron...
for i in range(nPoints):
  # ...we set its id to be equal to its index in the internal point array.
  tet.GetPointIds().SetId(i, i)

  # We compute the barycentric index of the point...
  tet.ToBarycentricIndex(i, barycentricIndex)

  # ...and scale it to unity.
  for j in range(3):
      point[j] = float(barycentricIndex[j]) / order

  # A tetrahedron comprised of the above-defined points has straight
  # edges.
  tet.GetPoints().SetPoint(i, point[0], point[1], point[2])

# Add the tetrahedron to a cell array
tets = vtk.vtkCellArray()
tets.InsertNextCell(tet)

# Add the points and tetrahedron to an unstructured grid
uGrid =vtk.vtkUnstructuredGrid()
uGrid.SetPoints(tet.GetPoints())
uGrid.InsertNextCell(tet.GetCellType(), tet.GetPointIds())

# Visualize
mapper = vtk.vtkDataSetMapper()
mapper.SetInputData(uGrid)

actor = vtk.vtkActor()
actor.SetMapper(mapper)

renderer = vtk.vtkRenderer()
renderWindow = vtk.vtkRenderWindow()
renderWindow.AddRenderer(renderer)
renderWindowInteractor = vtk.vtkRenderWindowInteractor()
renderWindowInteractor.SetRenderWindow(renderWindow)

renderer.AddActor(actor)
renderer.SetBackground(.2, .3, .4)

renderWindow.Render()
renderWindowInteractor.Start()
```

Presently, the unstructured grid stores the X-Y-Z coordinate data, point field values and offsets for each point in data arrays. The overhead for this storage increases quickly with order. For hexahedra of order N along each axis, the overhead is $(N + 1)^3$.

In the future, it may be possible to condense connectivity storage to a fixed size per cell shape. It may also be possible to specify order per axis rather than infer it from the number of points that define a cell. If order is explicitly specified, the following can be defined in memory: the offset to the first point and the order over which the points are iterated relative to the parametric coordinate system of the cell.

The VTK development team has already taken the first step toward these possibilities by keeping points for edges and faces together in Lagrange cells. After further steps are taken, the ability to relate points to a particular boundary will allow VTK to refer to the points with a single offset. This will allow VTK to significantly reduce its memory footprint.

Unit Tests

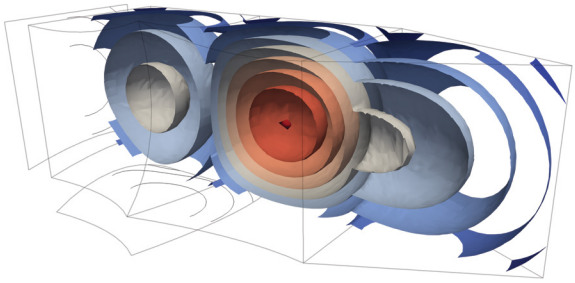
VTK offers unit tests for Lagrange cells. Some C++ unit tests are available in Common/DataModel/Testing/Cxx. LagrangeInterpolation.cxx, for example, evaluates the shape function that is common to tensor product shapes such as curves, quadrilaterals, hexahedra and wedges. Alternatively, TestLagrangeTetra.cxx, TestLagrangeTriangle.cxx and LagrangeHexahedron.cxx contain unit tests for inherited cell functions.

Python integration tests are located in Filters/Geometry/Testing/Python/LagrangeGeometricOperations.py. These tests demonstrate how to read in new cells from an XML file, intersect the cells with lines, glyph the resulting points and run filters on the unstructured grid that contains the cells. One such filter is `vtkUnstructuredGridGeometryFilter`. It has been extended to demonstrate high-fidelity polynomial refinement. The technique renders representations that reveal the smooth nature of elements.

Adaptive Tessellation

To further reveal the true shape of elements, VTK performs adaptive tessellation. This technique is executed by `vtkTessellatorFilter`. The filter adaptively subdivides edges according to tolerances that are specified on the chord error and field value differences. The resulting isocontours are smoother than those produced with the default option. The reason why `vtkTessellatorFilter` is not the default option is because it is computationally expensive.

In addition to VTK, `vtkTessellatorFilter` can be used in ParaView. ParaView is a VTK-based open source software platform. As a result of recent changes, it now also supports Lagrange cells.



The filter for adaptive tessellation in VTK produces curvy outlines and an improved approximation to the volumetric function.

Acknowledgment

For funding the work on the Lagrange cells, the authors thank the Data Analysis and Assessment Center (DAAC) of the Department of Defense. Information on DAAC is available on https://daac.hpc.mil. DAAC has generously licensed the work so that others can benefit from it through the license for VTK. The license is detailed on https://www.vtk.org/licensing.

Meet the Authors

Thomas J. Corona *is a senior R&D engineer at Kitware. Among his interests are computational electromagnetics, mesh generation and high-performance computing.*

David Thompson *is a staff R&D engineer at Kitware. His interests include computational simulation and visualization, conceptual design, solid modeling and mechatronics.*

KITWARE NEWS

Kitware Keeps Growing

Throughout the summer, the new Kitware headquarters continued to sprout in Clifton Park, New York. Its key differentiators from the current headquarters include a single-building layout; more natural light; an auditorium that connects to a kitchen; and showers for team members to use after boot camp workouts, yoga sessions, etc. After Kitware moves to the new headquarters, it will have over 60,000 square feet of office space in the U.S.

Chief Technical Officer Bill Hoffman, Technical Support Specialist Bryan Garrant and CEO Lisa Avila take a look around the new Kitware headquarters at 1712 Route 9 in Clifton Park, New York. (Jake Stookey/Kitware)

The company already increased its geographic footprint this year, as it leased an office at 411 North Fairfax Drive, Suite 302, Arlington, Virginia. Multiple team members have relocated to the suite, and Kitware has advertised job postings for the office on https://jobs.kitware.com.

As the "Who We Are" section of the website indicates, "Each person at Kitware brings something special to the table." Since Issue 42 of the Kitware Source publication, the following team members came to Kitware's table: Ian Sacklow (systems administrator), Kyle Edwards (junior technical staff member), Kristen Rinaldi (compliance manager), Alessandro Genova (R&D engineer), Meaghan Hickey (human resources generalist), Jared Tomek (UX/UI designer), William Hicks (senior R&D engineer), Emmett Hitz (R&D engineer), Lauren MacPherson (annotation specialist), Cameron Johnson (R&D engineer) and Maryann Olstad (annotation specialist). Former interns Ameya Shringi and Taylor Cook became R&D engineers, and Preston Law, Jordan Blue, Max Hoffman, James Hoctor and Reilley Blue returned for internships. Several interns came to Kitware for the first time, including Shreeraj Jadhav, Patrick Avery, David Russell, Adam Romlein, Joshua Beard, Deirdre Kelliher, Brian LaFleche, Wesley Turner, Tyler Burse, Matthew Purri, Zhixin Li and Bhavan Vasu.

In Carrboro, North Carolina, the office gained Matthew Phillips (staff R&D engineer), Riley Johnson (intern), Brian Clipp (staff R&D engineer), David Joy (R&D Engineer) and Pablo Hernández-Cerdán (intern). Across the Atlantic Ocean in Lyon, France, Kitware hired Michael Migliore, Matthieu Zins and Nick Laurenson as R&D engineers. The team also brought in Charly Griot, Gabriel Devillers, Youness Hamdi and Edern Haumont for internships.

As CEO Lisa Avila notes in the blog entry "Spring Promotions Bloom at Kitware," Kitware's growth "was made possible by the professional development of our team, which has allowed us to expand our technical expertise, scientific impact and customer base." To recognize their development, the company promoted Meredith Lapati to lead contracts administrator, Tim Thirion to technical leader, Deepak Chittajallu to principal engineer, Adrien Beaudet to operation support specialist, Kimberly O'leary to accountant, Chris Harris to principal engineer, Kellie Corona to data informatics analyst and David Stoup to principal engineer. From software development to process improvement, these team members have brought unique skills and abilities to Kitware. Summaries of their promotions are available on https://blog.kitware.com/spring-promotions-bloom-at-kitware.

Offices Cater Free Courses on Software Platforms

Kitware dished out a sequence of courses on the Visualization Toolkit (VTK), ParaView and CMake at its headquarters in Clifton Park, New York. The courses ran from March 13 to March 15, 2018.

With a mix of sessions that ranged from basic to advanced, the VTK and ParaView courses spoke to first-time and experienced users. The initial sessions toured the fundamental features and capabilities of the software, and the later sessions re-examined them with data for medical visualization, geoscience and computational fluid dynamics.

Staff R&D Engineer Dan Lipsa gets set to start a course on ParaView.

Kitware devoted the third course to Modern CMake. The course taught attendees how to install CMake, create files, build systems and test software. Looking ahead, modern features in CMake will be part of VTK 9.0. As a result, language wrapping in VTK will become more compartmentalized, and the bulk of dependencies for third-party libraries will get updated and imported in a unified manner. In addition, the number of global variables that represent a module's state will reduce. Kitware aims to release VTK 9.0 before 2019.

Down in Carrboro, North Carolina, VTK was a primary topic of a course on biomedical image analysis. Exercises taught attendees how to read a Digital Imaging and Communications in Medicine (DICOM) dataset; annotate a slice of a medical image; and create a 3D volume rendering through ray casting, shading, cropping, clipping, etc. Along with VTK, the course had exercises on 3D Slicer and the Insight Segmentation and Registration Toolkit. These exercises addressed subjects such as tractography, dynamic analysis and image filtering. Kitware posted material from the exercises on https://data.kitware.com under the "Courses" collection.

CMake Developer Opens the Floor for Q&A

Principal Engineer Robert Maynard made a special appearance on Reddit in June 2018 for an Ask Me Anything (AMA) on CMake. With support from Chief Technical Officer Bill Hoffman and Staff R&D Engineers Zack Galbreath and Chuck Atkins, Maynard fielded numerous questions on syntax, Modern CMake, reference documentation, precompiled header support, Lua, performance and other matters.

"The AMA was an excellent new way for us to engage with users and developers who have different levels of experience with CMake," Maynard said. "I can see us doing more AMAs down the road."

To read the questions and replies, please see http://redd.it/8sie4b.