Tetrahedral Lagrange Polynomial Interpolation

Aleksejs Fomins

1 Polynomial class

Here we discuss the analytical implementation and storage of polynomials for arbitrary dimension and parameter number.

1.1 Theory

Arbitrary polynomial of order up to n with d parameters can be represented in its expanded form as

$$p(\vec{u}) = \sum_i A_i \prod_{j=0}^{d} u_j^{\mathrm{pow}_{i,j}},$$

for example in 3D this can be written as

$$p(\vec{u}) = \sum_i A_i u^{pow_{u,i}} v^{pow_{v,i}} w^{pow_{w,i}},$$

Therefore, we define a PolySummand class, which stores a constant multiplier A and powers vector *pow*, and Polynomial class which stores a vector of PolySummands, and does required operations on them. Below we describe the implemented functionality

1.2 Methods

- Initialization: empty or with a single summand
- Operators: adding, subtracting or multiplying 2 polynomials or polynomial and a scalar.

  - to multiply by constant, need to loop and multiply each A_i by constant

  - to add or subtract 2 polynomials, merge the summand vectors and compactify

  - to multiply 2 polynomials, constructing a new polynomial from pairwise products of all summands, and compactify
- Evaluate: evaluates all summand and adds them up.

- Take derivative: lowers the corresponding power for each summand, multiplies prefactor by that power, and removes the summands that differentiate to 0
- Compactify: adds up all summands with the same power. Sorts the summands by $(x_1, y_1, z_1) < (x_2, y_2, z_2)$, where x has the highest priority and z has the lowest priority. Then all of the repeating powers will be consecutive. Simply loop over sorted polynomial, and to a new polynomial add the sums of all consecutive repeating polynomials.
- Integrate: Integration of a polynomial over a reference simplex. Naturally, it is the sum of integrals of summands. It can be shown that

  - 1D integral over the reference edge is $A\frac{1}{pow_u+1}$

  - 2D integral over the reference triangle is $A\frac{pow_u!pow_v!}{(pow_u+pow_v+2)!}$

  - 3D integral over the reference tetrahedron is $A\frac{pow_u!pow_v!pow_w!}{(pow_u+pow_v+pow_w+3)!}$

1.3 Tests

Currently the tests are only for 1, 2 and 3 dimensions. Most of the tests use intrinsic functionality like polynomial operators and derivatives to construct polynomials and print them to the screen, and request the user to to verify manually if they match the expected polynomials which are also printed. For each dimension there is one test which integrates a non-linear polynomial over simplex and prints out the result which is also compared manually.

TODO: These tests can and should be automatized in the future using integer string comparison. The test program should throw an error if a test fails

2 Curvilinear Element Interpolation class

Here we describe how to construct a curvilinear 3D object (edge, triangle or tetrahedron) which is interpolated over a set of points.

2.1 Interpolatory Points

We start by assuming a parameterization of the reference entity, namely

- edge: $\vec{r} = (u)$, such that $u \in [0, 1]$
- triangle: $\vec{r} = (u, v)$, such that $u \in [0, 1]$ and $v \in [0, 1 - u]$

- tetrahedron: $\vec{r} = (u, v, w)$, such that $u \in [0, 1]$, $v \in [0, 1 - u]$ and $w \in [0, 1 - u - v]$

Next, we provide a set of real geometry points $\vec{p}_i(\vec{r}_i)$ over which the 3D object is interpolated. By convention, we choose a rectangular grid in the parametric space, namely providing a real geometry point for each $\vec{r}_{i,j,k} = \Delta(i, j, k)$, where i, j, k are integers, $\Delta = 1/(Ord - 1)$ is the parametric step size, and Ord is the interpolation order of the surface. The parametric coordinates and their order determine the Lagrange Interpolatory polynomials and must be kept track of. Although the above convention is used in this writeup, the formalism itself can be applied to any set of interpolatory points, as long as their amount is preserved. In particular, for higher orders it should be beneficial to switch to Chebyshev sampling in order to minimize the Runge phenomenon.

It is easy to verify that the above discretization generates the following number of interpolatory points as a function of interpolatory order (starting with 1, mentioning first 5 orders)

- edge: $N_{Ord} = \{2, 3, 4, 5, 6\}$
- face: $N_{Ord} = \{3, 6, 10, 15, 21\}$
- tetrahedron: $N_{Ord} = \{4, 10, 20, 35, 56\}$

2.2 Interpolatory Polynomials

Also, it is easy to verify, that the numbers of interpolatory points above exactly match the numbers of independent polynomials of all orders up to and including Ord. Let us define the functions $z^{(1,i)}(u)$, $z^{(2,i)}(u, v)$ and $z^{(3,i)}(u, v, w)$ as the set of all such polynomials, where the first number is dimension of the object, and the 2nd is the order:

- edge:
  $z^{(1,1)}(u) = \{1, u\}$,
  $z^{(1,2)}(u) = \{1, u, u^2\}$,
  $z^{(1,3)}(u) = \{1, u, u^2, u^3\}$,
  $z^{(1,4)}(u) = \{1, u, u^2, u^3, u^4\}$,
  $z^{(1,5)}(u) = \{1, u, u^2, u^3, u^4, u^5\}$,
  etc
- face:
  $z^{(2,1)}(u, v) = \{1, u, v\}$,
  $z^{(2,2)}(u, v) = \{1, u, v, u^2, uv, v^2\}$,
  etc
- tetrahedron:
  $z^{(2,1)}(u, v, w) = \{1, u, v, w\}$,

$z^{(2,2)}(u,v,w) = \{1, u, v, w, u^2, uv, v^2, wu, wv, w^2\}$,
etc

This means that we are able to use complete polynomials of given order for our interpolation. If one interpolates over other entities, for example, rectangles, one has to either use an incomplete basis of certain order in order to keep the rectangular sampling grid convention, or change the sampling convention to have exact number of interpolatory points for a certain order. [$Volakis2010$] choose the first approach, thus interpolating a 9 node 2nd order rectangle with 4th order incomplete polynomial which has a convenient separable dimension product form.

Now, **the idea of Lagrange Interpolation** is to generate a polynomial vector function $\vec{p}(\vec{r})$, which would correspond to a real point $\vec{p_i} = \vec{p}(\vec{r_i})$ for each parametric coordinate $\vec{r_i}$ corresponding to a point in the reference element. It can be easily seen that such function must take the form

$$\vec{p}(\vec{r}) = \sum_j L_j(\vec{r})\vec{p_j} \tag{1}$$

where the Lagrange Polynomials L_j are defined by their interpolatory property

$$L_j(\vec{r_i}) = \delta_{ij} \tag{2}$$

for all interpolatory points $\vec{r_i}$. It is easy to see that in general case there must be as many Lagrange Polynomials as there are interpolatory points N_{Ord}, and, the amount of free parameters within each polynomial must also be equal to that number, such that it can uniquely interpolate a set of N_{Ord} points ($N_{Ord} - 1$ zeroes and 1 one). As we have discussed above, this is indeed the case.
Next, one slightly tricky part, and then we are done. We would like to prove that the following equation holds:

$$z_i(\vec{r}) = \sum_j L_j(\vec{r}) z_i(\vec{r_j}) \tag{3}$$

where z_i is a vector of standard polynomials defined above. This equation should hold for all $z^{(\dim,Ord)}$, where $\dim = \{1,2,3\}$. For $\dim < 3$ where z is defined for less than 3 parameters, simply ignore the extra parameters in $\vec{r}$.

The proof is quite simple. Both LHS and RHS are polynomials of order at most Ord, which means that they have at most N_{Ord} free parameters, and therefore, if we can show that the equation holds for N_{Ord} different parameter sets, then it holds for all others as well. And indeed, (3) holds for all $\vec{r} = \vec{r_k}$ because of (2).

Finally, we can write (3) as a vector equation

$$\vec{z}(\vec{r}) = V\vec{L}(\vec{r}) \tag{4}$$

where $V_{ij} = z_i(\vec{r}_j)$, and find the Lagrange polynomials by inverting V, namely

$$\vec{L}(\vec{r}) = V^{-1}\vec{z}(\vec{r}) \tag{5}$$

Thus each Lagrange Polynomial is a linear combination of standard polynomials. They are polynomials of order (at most) Ord. Depending on whether we are dealing with edges, faces or tetrahedrons, L_j will be a function of 1, 2 or 3 parameters.

2.3 Discussion

Important question is the interpretation of the interpolated manifold

- edge: a bounded curve which connects a certain set of points
- triangle: a bounded surface which connects a certain set of points
- tetrahedron: a bounded volume which connects a certain set of points

It is important to understand, especially for the tetrahedral case, that the parameterized bounded manifold is not exhaustively defined by the shape of its boundary. In tetrahedral case, interpolation over points associated with the element itself does not only mean that the interpolatory point is located inside of the tetrahedron. What it means that the geometry within the tetrahedron is also curved, such that all points within the parametric reference tetrahedron are uniquely mapped to the points in the real tetrahedron.

2.4 Implementation for Simplices

In this section we discuss how to efficiently enumerate the simplex interpolatory points, and to construct the reference simplex grid.

Let us place a set of points $\vec{\eta} \in Z^{\text{dim}}$ over simplex $\Delta^{\text{dim}}_{\text{len}}$. This can be done trivially by using 3 for loops and pushing vectors into a vector

- $\Delta^1_n = \{(i)\}$, for $i = [1 \text{ to } n]$
- $\Delta^2_n = \{(j,i)\}$, for $i = [1 \text{ to } n]$, $j = [1 \text{ to } n-i]$
- $\Delta^3_n = \{(k,j,i)\}$, for $i = [1 \text{ to } n]$, $j = [1 \text{ to } n-i]$, $k = [1 \text{ to } n-i-j]$

Then, each point $(\Delta^d_n)_i$ corresponds exactly to the power of u, v, w in the expansion of $(1+u+v+w)^n$, namely

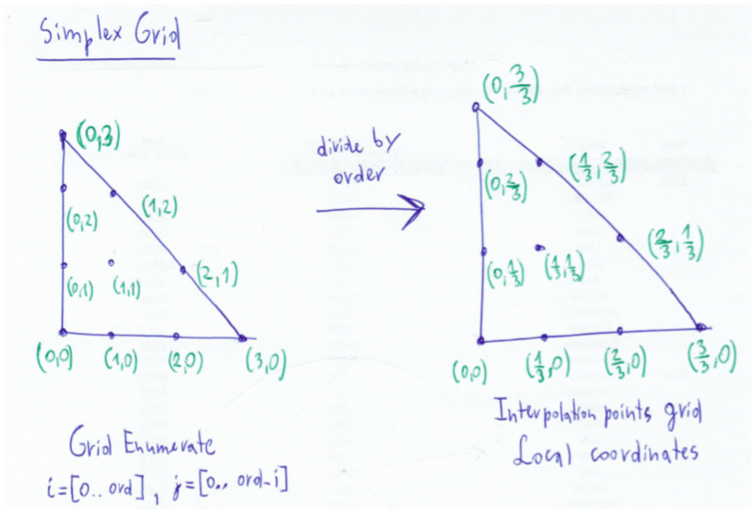
$$(1+u)^n = \sum_{i=0}^{n} C_n^{(\Delta^1_n)_i} u^{(\Delta^1_n)_{i,1}}$$

$$(1+u+v)^n = \sum_{i=0}^{n} C_n^{(\Delta_n^1)_i} u^{(\Delta_n^1)_{i,1}} v^{(\Delta_n^1)_{i,2}}$$

$$(1+u+v+w)^n = \sum_{i=0}^{n} C_n^{(\Delta_n^1)_i} u^{(\Delta_n^1)_{i,1}} v^{(\Delta_n^1)_{i,2}} w^{(\Delta_n^1)_{i,3}}$$

where $C_n^i, C_n^{i,j}$ and $C_n^{i,j,k}$ are the binomial, trinomial and quatranomial coefficients. The powers of the parameters given in this way are exactly the complete monomial basis for a polynomial of order up to and including d.

Also, it is convenient to note that $(\Delta_n^d)_i/n$ is exactly the parametric coordinates of the interpolation points on a regular grid over simplex.



After the monomials and the parametric interpolation points have been constructed, it remains to construct the interpolation matrix by evaluating the monomials at the interpolation points, then to invert the matrix, and multiply the monomial vector by it obtaining lagrange polynomials. This has been implemented both explicitly, by calculating all the lagrange interpolatory polynomials for simplices and writing them as functions, and implicitly, by introducing a polynomial class, which has all the above functionality, and thus generates a set of interpolatory polynomials which can be evaluated and integrated analytically by the code.

2.5 Methods

- Initialization: Requires reference element/GeometryType, interpolatory vertex vector and interpolation order

- dofPerOrder: Number of interpolatory vertices required for a given element type, dimension and interpolation order. For a simplex this quantity is

$$DoF^{ord}_{dim} = \{\{2,3,4,5,6\},\{3,6,10,15,21\},\{4,10,20,35,56\}\}$$

- cornerID: The id number of a corner in the interpolatory vertex vector. For the simplices is calculated as follows:

  -for edge $\{0,\ ord\}$

  -for triangle $\{0,\ ord,\ DoF^{ord}_{dim} - 1\}$

  -for tetrahedron $\{0,\ ord,\ ord(ord+3)/2,\ DoF^{ord}_{dim} - 1\}$

- simplexGrid: These 3 methods implement the functionality discussed in section 2.4. They return a vector of indices which the grid points take in the d-dimensional matrix, and that vector divided by the interpolation order, which is exactly the local coordinates of the reference simplex grid.

- lagrangePolynomial: Evaluates the i-th Lagrange Polynomial $L_i(\vec{r})$ for a given i and a given local coordinate. Lagrange polynomials in this case are given explicitly for all orders to accellerate computation.

- realCoordinate: Evaluates the global coordinate given a local coordinate. Computes the scalar product $\vec{p}(\vec{r}) = \sum_i \vec{p}_i L_i(\vec{r})$.

- interpolatoryVectorAnalytical: Produces an analytical map from local to global coordinates in terms of polynomial vector.

  0) Constructs local grid points $\vec{r}_i$ as given in 2.4.

  1) Constructs monomial basis $\vec{z}^{ord}(\vec{r})$ as given in the beginning of section 2.2

  2) Evaluates all monomials $\vec{z}^{ord}(\vec{r})$ at all local grid points $\vec{r}_i$, assembling the DynamicMatrix V

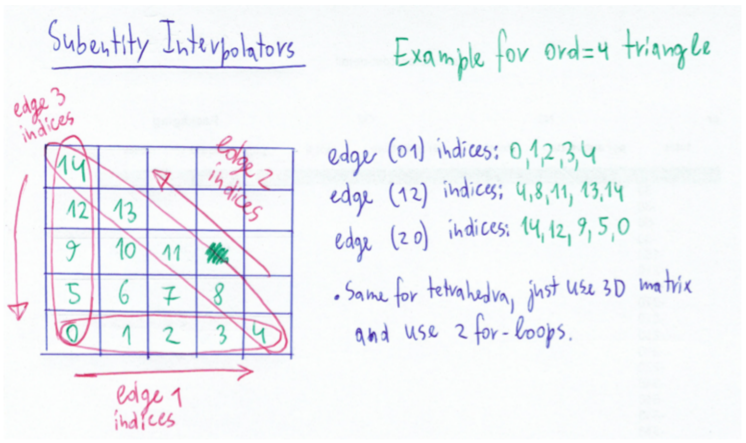
  3) Computes all lagrange polynomials using $\vec{L}(\vec{r}) = V^{-1}\vec{z}^{ord}(\vec{r})$

  4) Computes the analytical map $\vec{p}(\vec{r}) = \sum_i \vec{p}_i L_i(\vec{r})$.

- SubentityInterpolators Constructs Interpolator classes for each dim -1 subentity of the given element. Only allowed for elements of dimensions 2 and 3.

  - At the moment a rather crude algorithm is employed. To find the vertices corresponding to a given boundary, the order numbers of simplexGrid are written in a $(ord+1)\times(ord+1)\times(ord+1)$ matrix, which is then used to easier locate the indices of the boundary interpolatory points in the vertex vector.

  - Certain orientation convention is chosen for the provided sub-entitiy interpolators, to simplify the future calculation of the outwards normals. The convention for edge orientations for a triangle (012) are (01), (12) and (20). The convention for triangle orientations for a tetrahedron (0123) are (012), (023), (213) and (031).



edge (01) indices: 0,1,2,3,4
edge (12) indices; 4,8,11,13,14
edge (20) indices: 14,12,9,5,0
• Same for tetrahedra, just use 3D matrix and use 2 for-loops.

2.6 Tests

For each dimension several linear and polynomial Functors are defined which act as predefined local-to-global maps. Then, for a simplex of each dimension there is a testing routine which is run for every applicable Functor. The tests are as follows:

- For each order generate a local grid and sample the given functor to obtain global coordinates for interpolation points and thus construct the interpolator.
- First test requests a global coordinate for each local grid point, both using explicit function *realCoordinate* and by evaluating the analytical polynomial provided by *interpolatoryVectorAnalytical*. Then the 3 results are compared. For this test, all 3 results must match independent of Functor and interpolation order.
- Second test requests a global coordinate for a random set of local coordinates, also comparing the correct result with explicit and analytical functionality. Explicit and analytical results should be equal to each other for any test since they do the same thing. However, they will match to the true result only if the polynomial order of the Functor is lower or equal to the one being tested, and most likely should fail for lower orders.

TODO:

- The tests must be automatized such that the program throws an error if a test fails.
- Would be useful to test the method *SubentityInterpolators* which is not tested at the moment, but is indirectly tested later in the LagrangeGeometry tests.

3 Numerical Element Integrator class

Evaluates integral over element, approximating the integrand by an interpolatory polymomials of two hierarchical orders (2 and 4 at the moment). The running integration error is approximated by the difference between the analytical integrals calculated from these two interpolatory polynomials. The higher order element is split into into sub-elements of lower order, and the integration proceeds recursively. Every time an element is split, its previous running error is subtracted from the total error, and the running errors of the sub-elements are added to the total error. Thus, the integration is terminated when total approximated error is below selected tolerance. Heap structure ordered by the approximate error of the element is used to avoid recursion. At every iteration the element with the worst error is selected and then refined. When splitting, the previously calculated points are not re-calculated but hierarchically re-used by sub-elements. The sub-element only needs to be refined to a higher hierarchical order, by adding more points.

Possible improvement - Performance. As the the refined element does not check if the neighboring elements are also being refined, so they both sample on the boundary twice. Does there exist a method to store/find intersection refinements faster than just compute 2nd time. Using order 4 for every new refined triangle we sample 9 new points, out of which 2 are being wasted, thus 22% inefficient.

3.1 Methods

Numerical Integration is only available for Simplices at the moment.

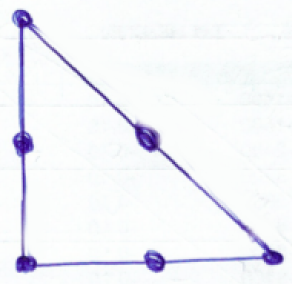
- Initialization: Requires GeometryType to know the type of the element being integrated
- Integrate: Integrates a function given by a functor object, until the expected integration error is below given tolerance.

3.2 Tests

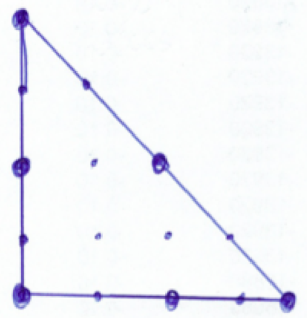
Numerical Integration is only implemented and hence tested for edges and triangles. A multitude of integrands are given in terms of Functors, starting with simple polynomial integrands and ending with integrands involving square roots of polynomials, as this is these are functions the method is expected to integrate well. The integrals are computed and compared to expected ones, which are sometimes given explicitly in numerical form, except of a lucky integral of $\iint \sqrt{xy}\, dxdy$ which happens to be $\pi/24$.

TODO:

Numerical Integration — Adaptive interpolation method

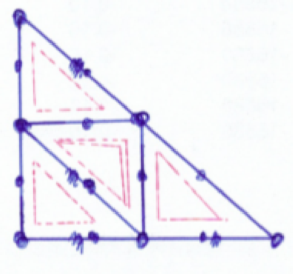


1) Sample function using 2nd order interpolatory points
2) Interpolate
3) Approx. Integral by analytically integrating interpolatory polynomial.



4) Refine to order 4
5) Repeat steps 1-3 for order 4
6) Approx. total int. by ord 4
   Approx. total error by $|I_4 - I_2|$

////// Loop ///////



7) for this element, subtract its integral and error from total (element from heap)
8) Split this ord=4 into 4 ord=2
9) Repeat steps 1-6 for each of 4 trs.
10) Add their Int. & Err. to total
11) Push back to heap.

Heap.pop() always returns the element with worst error estimate

///// until tot_err < tolerance

- These tests should be automatized, which is easy to do, as they are only comparing numerical values.
- Integrals of complicated functions may exceed 100000 samples for this method given relative precision 10^{-5}, which is too slow for the desired application

A Proof for polynomial summand integrals

Note that the series for beta-function can be written as

$$B(a+1,b+1) = \int_0^1 x^a(1-x)^b dx = \int_0^1 x^a \sum_{i=0}^{b} C_b^i (-1)^i x^i = \sum_{i=0}^{b} \frac{(-1)^i C_b^i}{a+1+i}$$

where $C_b^i = \frac{b!}{i!(b-i)!}$ is the combinations number.

A.1 1D

$$I_{1D} = \int_0^1 x^\alpha dx = \left.\frac{x^{a+1}}{a+1}\right|_0^1 = \frac{1}{a+1} \tag{6}$$

A.2 2D

$$\begin{aligned} I_{2D} = \int_0^1 \int_0^{1-x} x^a y^b dxdy &= \frac{1}{b+1}\int_0^1 x^a(1-x)^{b+1}dx = \frac{1}{b+1}\beta(a+1,b+2) \\ &= \frac{1}{b+1}\frac{a!(b+1)!}{(a+b+2)!} = \frac{a!b!}{(a+b+2)!} \end{aligned}$$

A.3 3D

$$\begin{aligned}
I_{3D} &= \int_0^1 \int_0^{1-x} \int_0^{1-x-y} x^a y^b z^c dxdydz \\
&= \frac{1}{c+1} \int_0^1 \int_0^{1-x} x^a y^b (1-x-y)^{c+1} dxdy \\
&= \frac{1}{c+1} \int_0^1 \int_0^{1-x} x^a y^b \sum_{i=0}^{c+1} C_{c+1}^i (-1)^i y^i (1-x)^{c+1-i} dxdy \\
&= \sum_{i=0}^{c+1} \frac{C_{c+1}^i (-1)^i}{c+1} \int_0^1 \ x^a (1-x)^{c+1-i} \int_0^{1-x} y^{b+i} dxdy \\
&= \sum_{i=0}^{c+1} \frac{C_{c+1}^i (-1)^i}{(c+1)(b+1+i)} \int_0^1 \ x^a (1-x)^{c+1-i} (1-x)^{b+1+i} dx \\
&= \sum_{i=0}^{c+1} \frac{C_{c+1}^i (-1)^i}{(c+1)(b+1+i)} \int_0^1 \ x^a (1-x)^{b+c+2} dx \\
&= \sum_{i=0}^{c+1} \frac{C_{c+1}^i (-1)^i}{(c+1)(b+1+i)} \beta(a+1, b+c+3) \\
&= \frac{1}{c+1} \beta(b+1, c+2) \beta(a+1, b+c+3) \\
&= \frac{1}{c+1} \frac{b!(c+1)!}{(b+c+2)!} \frac{a!(b+c+2)!}{(a+b+c+3)!} \\
&= \frac{a!b!c!}{(a+b+c+3)!}
\end{aligned}$$