Recursive, parameter-free, explicitly defined interpolation nodes for simplices

Tobin Isaac*

Abstract

A rule for constructing interpolation nodes for nth degree polynomials on the simplex is presented. These nodes are simple to define recursively from families of 1D node sets, such as the Lobatto-Gauss-Legendre (LGL) nodes. The resulting nodes have attractive properties: they are fully symmetric, they match the 1D family used in construction on the edges of the simplex, and the nodes constructed for the $(d-1)$-simplex are the boundary traces of the nodes constructed for the d-simplex. When compared using the Lebesgue constant to other explicit rules for defining interpolation nodes, the nodes recursively constructed from LGL nodes are nearly as good as the *warp & blend* nodes of Warburton [War06] in 2D (which, though defined differently, are very similar), and in 3D are better than other known explicit rules by increasing margins for $n > 6$. By that same measure, these recursively defined nodes are not as good as implicitly defined nodes found by optimizing the Lebesgue constant or related functions, but such optimal node sets have yet to be computed for the tetrahedron. A reference python implementation has been distributed as the `recursivenodes` package, but the simplicity of the recursive construction makes them easy to implement.

1 Definition of the recursive rule

The motivating example for this work is the use of Lagrange polynomials as shape functions for the finite element approximation space $\mathcal{P}_n(\Delta^d)$: polynomials of degree at most n on the d-simplex. A Lagrange polynomial basis $\Phi_X = \{\varphi_i\} \subset \mathcal{P}_n(\Delta^d)$ is defined by a set of *interpolation nodes* $X = \{\boldsymbol{x}_i\} \subset \Delta^d$ as $\Phi_X = \{\varphi_i \in \mathcal{P}_n(\Delta^d) : \varphi_i(\boldsymbol{x}_j) = \delta_{ij}\}$. While some of the properties of an implementation of the finite element method depend only on the approximation space, the basis used, whether Lagrange or other, can affect the convergence, numerical stability, and computational efficiency of the method. Convergence is affected by the way the basis is used to approximate data, numerical stability by presence of round-off errors, in the construction of both the basis and the resulting systems of equations, and computational efficiency by the complexity

*tisaac@cc.gatech.edu, School of Computational Science and Engineering, Georgia Tech.

of common tasks like applying a mass or derivative matrix. Discussion of each of these aspects follows the definition of the interpolation nodes that are the main contribution of this work.

The nodes are dimensionally recursive, building from points on the interval $[0,1]$. A *1D node set* is a set of points $X_n = \{x_{n,i}\}_{i=0}^{n} \subset [0,1]$ that is increasing and symmetric about $1/2$, $x_{n,i} = 1 - x_{n,n-i}$. A *1D node family* is a collection $\boldsymbol{X} = \{X_n\}_{n\in\mathbb{N}_0}$. Examples include equispaced nodes, symmetric Gauss-Jacobi quadrature nodes, and symmetric Lobatto-Gauss-Jacobi quadrature nodes.

The new nodes are naturally defined on the barycentric d-simplex,

$$\Delta^d_{\text{bary}} = \{\boldsymbol{b} = (b_0, \ldots, b_d) \in \mathbb{R}^{d+1}_{+} : \textstyle\sum_i b_i = 1\},$$

and are naturally indexed by the multi-indices

$$A^d_n = \{\boldsymbol{\alpha} = (\alpha_0, \ldots, \alpha_d) \in \mathbb{N}^{d+1}_0 : |\boldsymbol{\alpha}| = n\}.$$

This work uses the standard notation $|\boldsymbol{\alpha}| = \sum_i \alpha_i$, and further defines:

- $\#\boldsymbol{x}$ as the length of a tuple (multi-index or vector),
- $\boldsymbol{x}_{\setminus i}$ as the tuple formed by removing the ith element, and
- $\boldsymbol{x}_{+i}$ as the augmentation of a tuple by inserting a zero for the ith element.

Given a 1D node family $\boldsymbol{X}$, the recursive definition of the interpolation node $\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}) \in \Delta^{\#\boldsymbol{\alpha}-1}_{\text{bary}}$ is

$$\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}) = \begin{cases} (1), & \#\boldsymbol{\alpha} = 1, \\ \dfrac{\sum_i x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|} \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i})_{+i}}{\sum_i x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|}}, & \#\boldsymbol{\alpha} > 1. \end{cases} \tag{1}$$

The full d-simplex node set is

$$R^d_{\boldsymbol{X},n} = \{\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}) : \boldsymbol{\alpha} \in A^d_n\}, \tag{2}$$

and the full d-simplex node family is

$$\boldsymbol{R}^d_{\boldsymbol{X}} = \{R^d_{\boldsymbol{X},n}\}_{n\in\mathbb{N}}. \tag{3}$$

Unless otherwise specified, the 1D node family $\boldsymbol{X}$ is taken to be the Lobatto-Gauss-Legendre (LGL) family $\boldsymbol{X}_{\text{LGL}}$. Some examples are illustrated in Fig. 1.

```
# Plot $R^2_{X,7}$ and $R^3_{X,7}$ for illustration
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from recursivenodes import recursive_nodes
fig = plt.figure(figsize=(3.5,1.75))
# Plot 2D n = 7 nodes
ax = fig.add_subplot(121)
nodes7 = recursive_nodes(2, 7, domain='equilateral')
```

```
# Plot lines
ax.plot([1.,0.,-1.,1.],[-0.577,1.155,-0.577,-0.577],'k',zorder=1)
# Plot nodes
ax.scatter(nodes7[:,0], nodes7[:,1],zorder=2)
_ = ax.axis('off')
# Plot 3D n = 7 nodes
ax = fig.add_subplot(122,projection='3d')
nodes7 = recursive_nodes(3, 7, domain='equilateral')
# Plot lines
_ = ax.plot([1.,0.,0.,-1.,1.],
            [-0.577,1.155,0.,-0.577,-0.577],
            [-0.408,-0.408,1.225,-0.408,-0.408],
            'k', zorder=1)
_ = ax.plot([0.,-1.],[1.155,-0.577],[-0.408,-0.408],'k',zorder=1)
_ = ax.plot([1.,0.],[-0.577,0.],[-0.408,1.225],'k',zorder=1)
# Plot nodes
_ = ax.scatter(nodes7[:,0], nodes7[:,1], nodes7[:,2],zorder=2)
ax.set_proj_type('ortho')
_ = ax.axis('off')
ax.auto_scale_xyz([-0.65, 0.6], [-.578, 0.6], [-0.409, 0.65])
ax.dist = 9.5
plt.tight_layout()
plt.show()
```

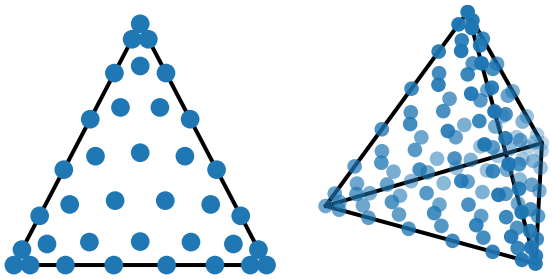

Figure 1: The nodes $R^2_{\boldsymbol{X},7}$ and $R^3_{\boldsymbol{X},7}$, mapped to equilateral simplices.

2 Intuition behind the recursive rule

Blyth and Pozrikidis [BP06] observed that Fekete points for the triangle, which have good interpolation properties, points in the interior nearly project onto LGL nodes on the edges of the triangle:

> Some intriguing observations can be made regarding the location of

> some of the Fekete points in a given set. [...] If an imaginary line is drawn through the nodes [...] the two Fekete nodes sit on this line, close to the two zeros of the second Lobatto polynomial, Lo2, scaled by the length of the imaginary line.

From this comes the idea that, if a good family of interpolation nodes on the $(d-1)$-simplex are already known, a heuristic for locating the node $\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha})$ in the d-simplex is to choose a point whose projection onto each facet from the opposite vertex is one of those good nodes,

$$\frac{\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha})_{\setminus i}}{1 - b_{\boldsymbol{X},i}(\boldsymbol{\alpha})} = \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i}), \quad \forall\, i \in \{0, \ldots, d\}. \tag{4}$$

Unfortunately, this is an overdetermined set of requirements.

Consider for example the placement of the interpolation node with multi-index $\boldsymbol{\alpha} = (1, 2, 3)$ in the barycentric triangle (this is one of the nodes for $n = |\boldsymbol{\alpha}| = 6$). The LGL nodes are good interpolation nodes, so the desire is for $\boldsymbol{b}_{\boldsymbol{X}}((1, 2, 3))$ to project onto the LGL nodes associated with the multi-indices $(2, 3)$ (one of the nodes for $n = 2 + 3 = 5$), $(1, 3)$ $(n = 4)$, and $(1, 2)$ $(n = 3)$, as illustrated in Fig. 2 (right). The projection lines nearly intersect at one point, but not quite. The system (4) has a solution if $\boldsymbol{X}$ is the family of equispaced nodes, and the solution is an equispaced node in the triangle, as seen in Fig. 2 (left).

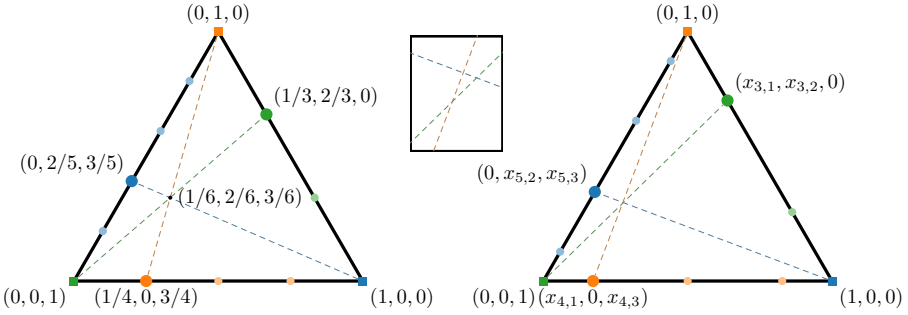

Figure 2: Desired projections for $\boldsymbol{b}_{\boldsymbol{X}}((1, 2, 3))$. If $\boldsymbol{X}$ is the equispaced 1D node family (left), the projections meet at an equispaced node in the triangle. If $\boldsymbol{X}$ is the LGL 1D node family (right), the lines do not actually meet at one point (detail, 32x magnified).

Any point in the interior of the triangle is in the convex hull of its projections onto the edges, so if a node location does satisfy (4), then it can be expressed as a barycentric combination of its projections. The equispaced nodes of the triangle not only have projections that are equispaced nodes on the edges, but their barycentric weights have a remarkable property.

Proposition 2.1. *Let the barycentric coordinates of the equispaced node associated with* $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \alpha_2)$, $|\boldsymbol{\alpha}| = n$, *be* $\boldsymbol{b} = (\alpha_0/n, \alpha_1/n, \alpha_2/n)$. *Let its projections onto the edges be*

$$\begin{aligned}\boldsymbol{b}_0 &= (0, \alpha_1/(n-\alpha_0), \alpha_2/(n-\alpha_0)),\\ \boldsymbol{b}_1 &= (\alpha_0/(n-\alpha_1), 0, \alpha_2/(n-\alpha_1)),\\ \boldsymbol{b}_2 &= (\alpha_0/(n-\alpha_2), \alpha_1/(n-\alpha_2), 0).\end{aligned}$$

Then $\boldsymbol{b}$ *is a convex combination of* $\boldsymbol{b}_0$, $\boldsymbol{b}_1$, *and* $\boldsymbol{b}_2$ *with (unnormalized) barycentric weights* $(1-\alpha_0/n) : (1-\alpha_1/n) : (1-\alpha_2/n)$.

In Proposition 2.1, the barycentric weights describing equispaced nodes in the triangle are themselves 1D equispaced nodes on the edge (see Fig. 3, left). By analogy, a heuristic for approximating a solution to the overdetermined system (4) is to use the same 1D node family $\boldsymbol{X}$ that was used for the projection points as barycentric weights for combining them (see Fig. 3, right), which restates (1).

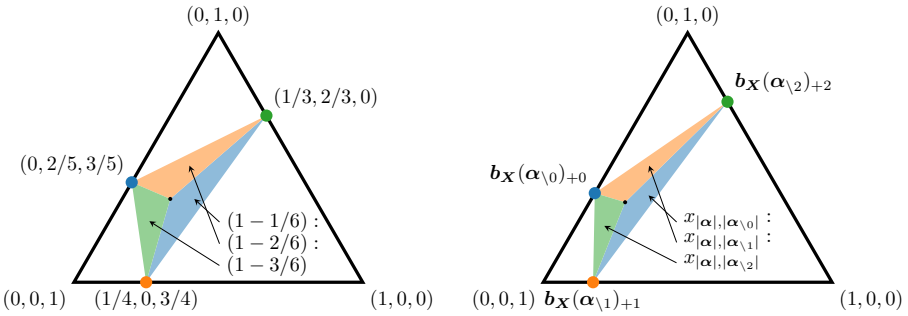

Figure 3: Defining $\boldsymbol{b}_{\boldsymbol{X}}((1,2,3))$ by barycentric coordinates relative to the projection points. If $\boldsymbol{X}$ is the equispaced 1D node family (left), it is the same as the point as the intersection of projection lines (see Fig. 2). If $\boldsymbol{X}$ is an arbitrary 1D node family (right), it is the recursive rule (1).

3 Comparison to other node families

The recursive rule (1) generates node families $\boldsymbol{R}^d_{\boldsymbol{X}}$ for the d-simplex in each dimension. This section compares them to other node families with respect to several metrics that are relevant to finite element computations.

3.1 Boundary and symmetry properties

The nodes $R^d_{\boldsymbol{X},n}$ have three non-numerical properties that make them convenient to use when implementing the finite element method.

I. **Symmetry:** The symmetry group of the d-simplex is the group S_{d+1}: for Δ^d_{bary}, each symmetry corresponds to a permutation of the coordinates.

It is clear that the recursive rule (1) respects these symmetries, that $\boldsymbol{b}_{\boldsymbol{X}}(\sigma(\boldsymbol{\alpha})) = \sigma(\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}))$. This is useful when a d-simplex is viewed from multiple orientations, such as when it is the interface between cells.

II. **Equivalence to $\boldsymbol{X}$ when $d = 1$:** The node sets in $\boldsymbol{X}$ must be symmetric about $1/2$, so if $\boldsymbol{\alpha} = (\alpha_0, \alpha_1)$ then $\sum_i x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|} = x_{|\boldsymbol{\alpha}|,\alpha_1} + x_{|\boldsymbol{\alpha}|,\alpha_0} = 1$. The recursive rule (1) then becomes

$$\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}) = x_{|\boldsymbol{\alpha}|,\alpha_1}(0,1) + x_{|\boldsymbol{\alpha}|,\alpha_0}(1,0) = (x_{|\boldsymbol{\alpha}|,\alpha_0}, x_{|\boldsymbol{\alpha}|,\alpha_1}).$$

In other words, $R^1_{\boldsymbol{X},n}$ is the 1D node set X_n mapped to the barycentric line Δ^1_{bary}.

III. **Recursive boundary traces:** Problems solved by the finite element method can have forms computed over surfaces: data for Neumann boundary conditions or jump terms in discontinuous Galerkin methods, for example. A good node set should induce good shape functions for $\mathcal{P}_n(\Delta^d)$, but also for the trace spaces on the boundary facets, which are embeddings of $\mathcal{P}_n(\Delta^{d-1})$.

The following proposition show that if the 1D node family $\boldsymbol{X}$ has nodes at the endpoints, then $R^d_{\boldsymbol{X},n}$ has nodes on each boundary facet of Δ^d_{bary} that are the $R^{d-1}_{\boldsymbol{X},n}$ nodes mapped onto that facet, and so they are appropriate for defining Lagrange polynomials on the trace space.

Proposition 3.1. *Let $\boldsymbol{X}$ be a 1D node family such that $x_{n,0} = 0$ and $x_{n,n} = 1$ for all $n \geq 1$. Let $\boldsymbol{\alpha}$ be a multi-index such that $|\boldsymbol{\alpha}| \geq 1$, $\#\boldsymbol{\alpha} > 1$, and $\alpha_j = 0$. Then $\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}) = \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j})_{+j}$.*

Proof. If $\alpha_j = 0$, then $|\boldsymbol{\alpha}_{\setminus j}| = |\boldsymbol{\alpha}|$, so $x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus j}|} = 1$. Therefore,

$$\begin{aligned}
\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}) &= \frac{\sum_i x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|} \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i})_{+i}}{\sum_i x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|}} \\
&= \frac{x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus j}|} \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j})_{+j} + \sum_{i \neq j} x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|} \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i})_{+i}}{x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus j}|} + \sum_{i \neq j} x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|}} \\
&= \frac{\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j})_{+j} + \sum_{i \neq j} x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|} \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i})_{+i}}{1 + \sum_{i \neq j} x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|}}. \qquad (5)
\end{aligned}$$

If $\#\boldsymbol{\alpha} = 2$, then $x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|} = x_{\alpha_i,0} = 0$ for $i \neq j$. Then (5) simplifies to $\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j})_{+j}$. This proves the base case.

Now assume the property holds if $\#\boldsymbol{\alpha} \leq d$, and let $\#\boldsymbol{\alpha} = d + 1$. If $i \neq j$, then $\boldsymbol{\alpha}_{\setminus i}$ has a zero at an index $\hat{\jmath} \in \{j, j-1\}$, so

$$\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i})_{+i} = \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i\setminus \hat{\jmath}})_{+\hat{\jmath}+i}.$$

The order can be switched: there is $\hat{\imath} \in \{i, i-1\}$ such that

$$\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i \setminus \hat{\jmath}})_{+\hat{\jmath}+i} = \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j \setminus \hat{\imath}})_{+\hat{\imath}+j}.$$

So by relabeling and using (1) this implies

$$\begin{aligned}\sum_{i \neq j} x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|} \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i})_{+i} &= \sum_{\hat{\imath}} x_{|\boldsymbol{\alpha}_{\setminus j}|,|\boldsymbol{\alpha}_{\setminus j \setminus \hat{\imath}}|} \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j \setminus \hat{\imath}})_{+\hat{\imath}+j} \\ &= \left(\sum_{\hat{\imath}} x_{|\boldsymbol{\alpha}_{\setminus j}|,|\boldsymbol{\alpha}_{\setminus j \setminus i}|}\right) \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j})_{+j}.\end{aligned}$$

From this equality (5) simplifies:

$$\begin{aligned}\frac{\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j})_{+j} + \sum_{i \neq j} x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|} \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus i})_{+i}}{1 + \sum_{i \neq j} x_{|\boldsymbol{\alpha}|,|\boldsymbol{\alpha}_{\setminus i}|}} &= \frac{(1 + \sum_{\hat{\imath}} x_{|\boldsymbol{\alpha}_{\setminus j}|,|\boldsymbol{\alpha}_{\setminus j \setminus \hat{\imath}}|}) \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j})_{+j}}{1 + \sum_{\hat{\imath}} x_{|\boldsymbol{\alpha}_{\setminus j}|,|\boldsymbol{\alpha}_{\setminus j \setminus \hat{\imath}}|}} \\ &= \boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha}_{\setminus j})_{+j}.\end{aligned}$$

□

Properties (II) and (III) together mean that the nodes of $R^d_{\boldsymbol{X},n}$ on an edge are always mappings of the 1D node set X_n. This is useful when simplices appear in hybrid meshes with tensor-product cells, which often use tensor products of 1D node sets, because common edges between the two cell types will have the same nodes.

3.2 Interpolation properties

A problem discretized by the finite element method may require the approximation of an arbitrary function f in $\mathcal{P}_n(\Delta^d)$. Certain problems have optimal projection operators for this purpose, such as L^2 projection or H^1 projection, but these operators can only be approximated with numerical integration rules, and may be implicit or expensive. When Lagrange polynomials are used as a basis, interpolation at the nodes is an appealing projection onto $\mathcal{P}_n(\Delta^d)$, because it requires the minimum number of function evaluations. Let $I_X : \mathcal{B}(\Delta^d) \to \mathcal{P}_n(\Delta^d)$ be the interpolation operator defined by nodes X acting on bounded, measurable functions on the d-simplex. The interpolation error can be bounded by

$$\|I_X f - f\|_\infty \leq (1 + \Lambda_n^{\max}(X)) \inf_{p \in \mathcal{P}_n(\Delta^d)} \|p - f\|_\infty,$$

where $\Lambda_n^{\max}(X)$ is the Lebesgue constant, defined by the shape functions Φ_X associated with X,

$$\Lambda_n^{\max}(X) = \max_{\boldsymbol{x} \in \Delta^d} \sum_{\varphi \in \Phi_X} |\varphi(\boldsymbol{x})|.$$

```
# Plot 2D Lebesgue constants, comparing the recursive nodes
# to other node families
import matplotlib.pyplot as plt
```

```
from matplotlib.ticker import MaxNLocator
from recursivenodes.lebesgue import lebesguemax_reference

# One plot for implicit, one for explicit
fig,(ax,ax2) = plt.subplots(ncols=2, figsize=(7,3.5), sharey=True)

# recursive and 2D implicit node families, their short names,
# and the degrees to plot
groups = [
    ('recursive', '$\mathbf{R}^2_{\mathbf{X}}$', (4,16,1)),
    ('Lebesgue', 'Roth-leb', (4,16,1)),
    ('Rapetti-Sommariva-Vianello-leb', 'RSV-leb', (4,19,1)),
    ('Rapetti-Sommariva-Vianello-lebgls', 'RSV-lebgls', (4,19,1)),
    ]
markers = ['+','x','1','2',]
# Plot Lebesgue constants
for (marker,group) in zip(markers,groups):
    key = group[0]
    label = group[1]
    first, last, step = group[2]
    ns = list(range(first, last, step))
    ls = [lebesguemax_reference(2, n, key) for n in ns]
    _ = ax2.scatter(ns, ls, label=label, marker=marker)
_ = ax2.legend()
_ = ax2.set_xlabel('polynomial degree $n$')
_ = ax2.xaxis.set_major_locator(MaxNLocator(integer=True))

# 2D explicit node families, their short names,
# and the degrees to plot
groups = [
    ('recursive', '$\mathbf{R}^2_{\mathbf{X}}$', (4,16,1)),
    ('equispaced', 'equispaced', (4,7,1)),
    ('blyth_luo_pozrikidis', 'BLP', (4,13,1)),
    ('warburton', 'warp & blend', (4,16,1)),
    ]
markers = ['+','x','1','2']
# Plot Lebesgue constants
for (marker,group) in zip(markers,groups):
    key = group[0]
    label = group[1]
    first, last, step = group[2]
    ns = list(range(first, last, step))
    ls = [lebesguemax_reference(2, n, key) for n in ns]
    _ = ax.scatter(ns, ls, label=label, marker=marker)
_ = ax.legend()
```

```
_ = ax.set_xlabel('polynomial degree $n$')
_ = ax.set_ylabel('$\Lambda_n^{\max}$')

ax2.get_shared_y_axes().join(ax, ax2)
plt.tight_layout()
fig.subplots_adjust(wspace=0)
_ = plt.show()
```

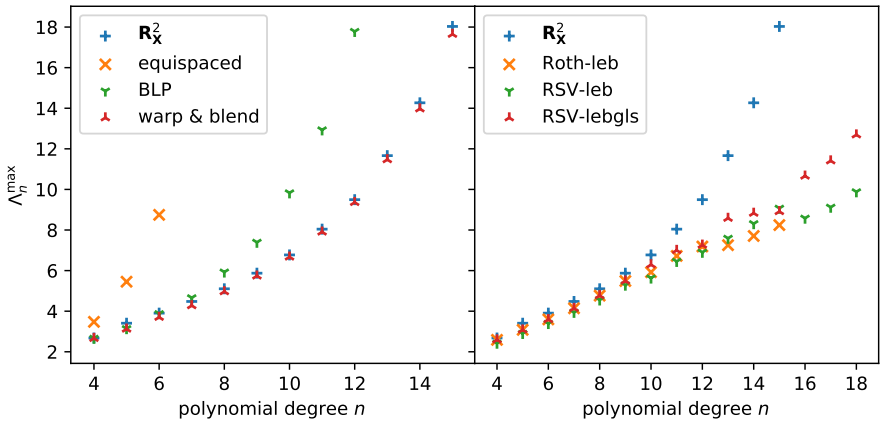

Figure 4: Lebesgue constants on the triangle, comparing $\boldsymbol{R}^2_{\boldsymbol{X}}$ against node families defined explicitly (left) and implicitly (right).

```
# Plot 3D Lebesgue constants, comparing the recursive nodes
# to other node families
import matplotlib.pyplot as plt
from recursivenodes.lebesgue import lebesguemax_reference

# One plot for implicit, one for explicit
fig,(ax,ax2) = plt.subplots(ncols=2, figsize=(7,3.5), sharey=True)

# recursive and 3D implicit node families, their short names,
# and the degrees to plot
groups = [
    ('recursive', '$\mathbf{R}^3_{\mathbf{X}}$', (4,16,1)),
    ('Chen-Babuska', 'CB', (4,10,1)),
    ('Hesthaven-Teng', 'HT', (4,10,1)),
    ]
markers = ['+','x','1']
# Plot Lebesgue constants
for (marker,group) in zip(markers,groups):
    key = group[0]
```

```
    label = group[1]
    first, last, step = group[2]
    ns = list(range(first, last, step))
    ls = [lebesguemax_reference(3, n, key) for n in ns]
    _ = ax2.scatter(ns, ls, label=label, marker=marker)
_ = ax2.legend()
_ = ax2.set_xlabel('polynomial degree $n$')

# 3D explicit node families, their short names,
# and the degrees to plot
groups = [
    ('recursive', '$\mathbf{R}^3_{\mathbf{X}}$', (4,16,1)),
    ('equispaced', 'equispaced', (4,11,1)),
    ('blyth_luo_pozrikidis', 'BLP', (4,14,1)),
    ('warburton', 'warp & blend', (4,16,1)),
    ]
markers = ['+','x','1','2']
# Plot Lebesgue constants
for (marker,group) in zip(markers,groups):
    key = group[0]
    label = group[1]
    first, last, step = group[2]
    ns = list(range(first, last, step))
    ls = [lebesguemax_reference(3, n, key) for n in ns]
    _ = ax.scatter(ns, ls, label=label, marker=marker)
_ = ax.legend()
_ = ax.set_xlabel('polynomial degree $n$')
_ = ax.set_ylabel('$\Lambda_n^{\max}$')

ax2.get_shared_y_axes().join(ax, ax2)
plt.tight_layout()
fig.subplots_adjust(wspace=0)
_ = plt.show()
```

Lebesgue constants for $\boldsymbol{R}^d_{\boldsymbol{X}}$ are compared against some other node families on the triangle in Fig. 4 and on the tetrahedron in Fig. 5. These include:

- **equispaced:** Equispaced nodes, defined by $b_{\text{eq},i}(\boldsymbol{\alpha}) = \alpha_i/|\boldsymbol{\alpha}|$.
- **BLP:** The nodes of Blyth, Luo, & Pozrikidis [BP06],[LP06], which like the recursively defined nodes are based on the LGL nodes $\boldsymbol{X}_{\text{LGL}}$. If $\boldsymbol{\alpha} > 0$, which indicates that the node will be in the interior of the simplex, they are defined by

$$b_{\text{BLP},i}(\boldsymbol{\alpha}) = \frac{1}{|\boldsymbol{\alpha}|}(1 + |\boldsymbol{\alpha}|x_{|\boldsymbol{\alpha}|,\alpha_i} - \textstyle\sum_j x_{|\boldsymbol{\alpha}|,\alpha_j}).$$

  Points on the boundary are mapped from the same rule applied to the

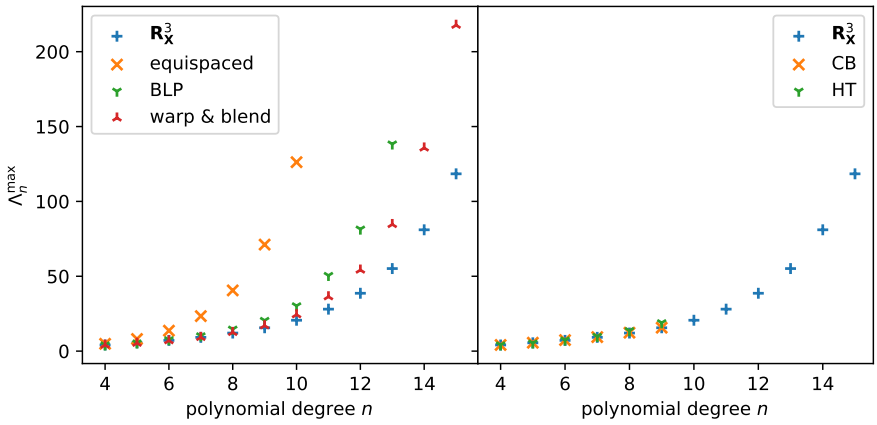

Figure 5: Lebesgue constants on the tetrahedron, comparing $\boldsymbol{R}^3_{\boldsymbol{X}}$ against node families defined explicitly (left) and implicitly (right).

$(d-1)$-simplex.

- **warp & blend:** The nodes of Warburton [War06], which define the node location $\boldsymbol{b}_{\text{wb}}(\boldsymbol{\alpha})$ as the image of the equispaced node $\boldsymbol{b}_{\text{eq}}(\boldsymbol{\alpha})$ under a smooth bijection of the d-simplex. The bijection sends equispaced nodes to LGL nodes on the edges. The smooth map is nearly isoparametric, but a blending parameter is introduced that controls the distortion in the interior of the element, and optimal values of this blending parameter have been computed for n up to 15 in $d = 2$ and 3.
- **Roth-leb:** Nodes for the triangle computed by Roth [Rot05] by numerical minimization of $\Lambda_n^{\max}$.
- **RSV-leb:** Nodes for the triangle computed by Rapetti, Sommariva, and Vianello [RSV12] by numerical minimization of $\Lambda_n^{\max}$.
- **RSV-lebgls:** Nodes for the triangle computed by Rapetti, Sommariva, and Vianello [RSV12] by numerical minimization of $\Lambda_n^{\max}$, subject to the constraints that the nodes remain symmetric and that the nodes on the edges be LGL nodes.
- **CB:** Nodes for the tetrahedron computed by Chen and Babuška [CB96] by numerical minimization of the related interpolation metric

$$\Lambda_n^2(X) = \int_{\Delta^d} \sum_{\varphi \in \Phi_X} \varphi(\boldsymbol{x})^2 \, dx.$$

- **HT:** Nodes for the tetrahedron computed by Hesthaven and Teng [HT00] as the equilibrium distribution of charged particles.

All of these node families except *Roth-leb* and *RSV-leb* are symmetric for all n, and all except *equispaced*, *Roth-leb*, and *RSV-leb* have edge traces that are LGL nodes. The *equispaced*, *BLP*, and *warp & blend* nodes can be explicitly defined

in any dimension and have the recursive boundary property (III) from Section 3.1.[1]

Both Figs. 4 and 5 split the comparison of the $\boldsymbol{R}^d_{\boldsymbol{X}}$ node family into comparisons against families that are explicitly defined and nodes that are implicitly defined as the solution of an optimization problem.

2D: In 2D, the $\boldsymbol{R}^2_{\boldsymbol{X}}$ node family has Lebesgue constants that are not much worse than those for node families implicitly defined to minimize the Lebesgue constant for $n \leq 9$ (Fig. 4, left). For $n \geq 10$, the Lebesgue constant grows much faster for $\boldsymbol{R}^2_{\boldsymbol{X}}$ than for the best implicitly defined nodes.

Not coincidentally, at $n = 10$ the layout of implicitly defined nodes that minimize the Lebesgue constant changes significantly. Until then, the *RSV-lebgls* nodes look "lattice-like," as though they have been smoothly, symmetrically, and monotonically mapped from the equispaced nodes, the same as the explicit node families. At $n = 10$, however, this pattern changes (Fig. 6). This suggests that no node family that retains the lattice-like structure, including $\boldsymbol{R}^d_{\boldsymbol{X}}$, can attain a slow growth of $\Lambda_n^{\max}$ like the implicitly defined families.

```
# Demonstrate irregular structure of RSV-lebgls nodes for n >= 10.
import matplotlib.pyplot as plt
import numpy as np
from recursivenodes import recursive_nodes
from recursivenodes.nodes import rapetti_sommariva_vianello
fig = plt.figure(figsize=(5,2.5))
ax = fig.add_subplot(121)
nodes9 = rapetti_sommariva_vianello(9, domain='equilateral')
# Plot lines
_ = ax.plot([1.,0.,-1.,1.],[-0.577,1.155,-0.577,-0.577],'k',
            zorder=1)
# Plot nodes, n = 9
_ = ax.scatter(nodes9[:,0], nodes9[:,1], marker='x', zorder=2)
_ = ax.axis('off')
ax = fig.add_subplot(122)
nodes10 = rapetti_sommariva_vianello(10, domain='equilateral')
# Plot lines
_ = ax.plot([1.,0.,-1.,1.],[-0.577,1.155,-0.577,-0.577],'k',
            zorder=1)
# Plot nodes, n = 10
_ = ax.scatter(nodes10[:,0], nodes10[:,1], marker='x', zorder=2)
_ = ax.axis('off')
plt.tight_layout()
plt.show()
```

In comparison to the other explicitly defined nodes (Fig. 4, right), the $\boldsymbol{R}^2_{\boldsymbol{X}}$

[1] The *warp & blend* nodes have this property if the same value of the blending parameter is used for each dimension.

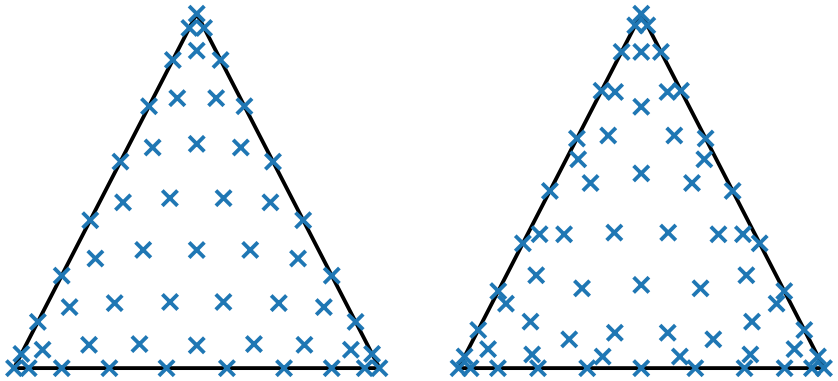

Figure 6: The LEBGLS nodes of Rapetti, Sommariva, and Vianello [RSV12], showing the abrupt change in layout between $n = 9$ (left) and $n = 10$ (right).

family is nearly as good as the *warp & blend* family, which has the best Lebesgue constants: $\Lambda_n^{\max}$ is never more than 10% different between them for $n \le 15$. In fact, despite the differences in their definitions—*warp & blend* by continuous bijections, $\boldsymbol{R}_{\boldsymbol{X}}^2$ by recursion—the node families are remarkably similar for $n \le 15$: $\|\boldsymbol{b}_{\boldsymbol{X}}(\alpha) - \boldsymbol{b}_{\mathrm{wb}}(\alpha)\| \le 0.01$ for every node in these node sets.

```
# Check some assertions from the text
from recursivenodes.lebesgue import lebesguemax_reference as lr
from recursivenodes import recursive_nodes
from recursivenodes.nodes import warburton
import numpy as np
diffmax = 0.
# Check assertion that the Lebesgue constants differ by
# less than 10%
for n in range(4,16):
    r = lr(2, n, 'recursive')
    w = lr(2, n, 'warburton')
    diffmax = max(diffmax, r/w)
assert(diffmax < 1.1)
# Check assertion that recursive nodes are close to
# warp & blend nodes in 2D
diffmax = 0.
for n in range(4,16):
    r = recursive_nodes(2, n, domain='barycentric')
    w = warburton(2, n, domain='barycentric')
    diff = np.max(np.linalg.norm(np.abs(r-w),axis=1))
    diffmax = max(diffmax,diff)
```

```
assert(diffmax < 0.01)
```

3D: In 3D there are no published examples of $\Lambda_n^{\max}$-optimal node sets that have been numerically computed in the same way as in 2D. $\Lambda_n^{\max}(X)$ is a nonconvex function of the node coordinates in X, and the number of coordinates grows cubically with n, so this is a challenging optimization problem. Instead, the implicitly defined node families *CB* and *HT* optimize simpler objectives: the Λ_n^2 interpolation metric and the electrostatic potential, respectively, and these have only been computed to $n \leq 9$. There is little difference in $\Lambda_n^{\max}$ between $\boldsymbol{R}_{\boldsymbol{X}}^3$ and these two families (Fig. 5, left), though it is slightly smaller than both for $n \geq 6$.

```
# Check assertion that the Lebesgue constant of recursive nodes
# is less than Chen-Babuska or Hesthaven-Teng for n >= 6
from recursivenodes.lebesgue import lebesguemax_reference as lr
for n in range(6,10):
    r = lr(3, n, 'recursive')
    c = lr(3, n, 'Chen-Babuska')
    h = lr(3, n, 'Hesthaven-Teng')
    assert(r < c)
    assert(r < h)
```

In comparison to the explicitly defined node families *BLP* and *warp & blend* (Fig. 5, right), there is little difference for $n \leq 6$ (all are within 7% of each other), but $\boldsymbol{R}_{\boldsymbol{X}}^3$ is increasingly superior for $n \geq 7$. For $n = 15$, the largest for which the *warp & blend* nodes' blending parameter has been optimized, the Lebesgue constant of $\boldsymbol{R}_{\boldsymbol{X}}^3$ is 40% smaller.

```
from recursivenodes.lebesgue import lebesguemax_reference as lr
# Check assertion that there is little difference in
# Lebesgue constants for n <= 6
for n in range(4,7):
    r = lr(3, n, 'recursive')
    w = lr(3, n, 'warburton')
    b = lr(3, n, 'blyth_luo_pozrikidis')
    minl = min(r,w,b)
    maxl = max(r,w,b)
    assert(minl > 0.93*maxl)
# Check assertion that there is the recursive nodes'
# Lebesgue constants are smaller than warp & blend for n >= 7,
# and 40% smaller for n = 15
for n in range(7,16):
    r = lr(3, n, 'recursive')
    w = lr(3, n, 'warburton')
    assert(r < w)
    if n == 15:
      assert(r < 0.6*w)
```

3.3 Asymptotic interpolation properties

A node family is good for approximation by interpolation if the interpolants are known to converge for a large class of functions. In particular, if f is analytic in the neighborhood of Δ^d, then there is a sequence of polynomials $p_n \rightrightarrows f$ (converging uniformly on Δ^d), so it is possible that given the right node family $\boldsymbol{X} = \{X_n\}$ that $I_{X_n} f \rightrightarrows f$ for all analytic f as well.

The weakest known sufficient condition that guarantees this for $d > 1$ is sub-exponential growth of the Lebesgue constant: if $\Lambda_n^{\max}(X_n)^{1/n} \to 1$, then $I_{X_n} f \rightrightarrows f$ for f analytic in a neighborhood of Δ^d [Blo+92]. The values of $\Lambda_n^{\max}(R^d_{\boldsymbol{X},n})$ that appeared in Figs. 4 & 5 are tabulated in Table 1. In all tabulated values, $(\Lambda_n^{\max})^{1/n}$ continues to increase instead of converging to 1, so they show no evidence of sub-exponential growth.

```
# Tabulate Lebesgue constants for recursive nodes,
# and track their asymptotic growth rates in 2D and 3D
from tabulate import tabulate
from recursivenodes.lebesgue import lebesguemax_reference
L = {}
L['$n$'] = list(range(4,16))
L[r'$\Lambda_n^{\max}(R^2_{\XX,n})$'] = \
    [lebesguemax_reference(2, n) for n in range(4,16)]
L[r'$\Lambda_n^{\max}(R^2_{\XX,n})^{1/n}$'] = \
    [lebesguemax_reference(2, n)**(1./n) for n in range(4,16)]
L[r'$\Lambda_n^{\max}(R^3_{\XX,n})$'] = \
    [lebesguemax_reference(3, n) for n in range(4,16)]
L[r'$\Lambda_n^{\max}(R^3_{\XX,n})^{1/n}$'] = \
    [lebesguemax_reference(3, n)**(1./n) for n in range(4,16)]
print(tabulate(L, tablefmt="pipe", headers="keys"))
```

Table 1: Lebesgue constants computed for $\boldsymbol{R}^d_{\boldsymbol{X}}$

n	$\Lambda_n^{\max}(R^2_{\boldsymbol{X},n})$	$\Lambda_n^{\max}(R^2_{\boldsymbol{X},n})^{1/n}$	$\Lambda_n^{\max}(R^3_{\boldsymbol{X},n})$	$\Lambda_n^{\max}(R^3_{\boldsymbol{X},n})^{1/n}$
4	2.67857	1.27931	4.09308	1.42237
5	3.40745	1.27787	5.54727	1.40869
6	3.90448	1.25486	7.16891	1.38859
7	4.47897	1.23887	9.20205	1.37309
8	5.10406	1.226	12.0671	1.36521
9	5.87268	1.21738	15.5927	1.3569
10	6.77248	1.21081	20.6234	1.35343
11	8.04267	1.20867	28.034	1.35397
12	9.49527	1.20631	38.6495	1.35601
13	11.6647	1.208	55.1425	1.36132
14	14.2678	1.20908	81.0374	1.36878
15	18.0306	1.21265	118.42	1.37476

In fact, Bloom et al. [Blo+92] considered it an open question whether explicitly computed node families with uniformly convergent interpolants exist for *any* nontrivial set in $d > 1$. In the intervening time, analogues of the Chebyshev polynomials have been found for domains related to root systems [RM10], but these domains are not simplices. Bloom et al. [Blo+12] considered the question still open for simplices twenty years later, and it appears to still be open now.

3.4 Finite element matrix conditioning

Matrices that show up repeatedly in applications of the finite element method include the mass matrix $M_{ij} = \int_{\Delta^d} \varphi_i \varphi_j \, dx$ and the stiffness matrix $K_{ij} = \int_{\Delta^d} \nabla\varphi_{n,i} \cdot \nabla\varphi_{n,j} \, dx$. Let $G_{ijk} = \partial_j \varphi_k(\boldsymbol{x}_i)$ (considered as a matrix in $\mathbb{R}^{d|\Phi|\times|\Phi|}$), and $L_{ij} = \nabla \cdot \nabla\varphi_j(\boldsymbol{x}_i)$. These are the nodal gradient and Laplacian matrices, that appear in strong-form nodal discontinuous Galerkin methods. The condition numbers of these matrices (using the definition $\kappa_2(A) = \|A\|_2\|A^\dagger\|_2$) for $\boldsymbol{R}^d_{\boldsymbol{X}}$ are compared against the condition numbers for the equispaced, *BLP*, *warp & blend*, and *RSV-lebgls* nodes in Fig. 7. The condition number of M is affine invariant and in a quasiuniform mesh bounds the condition number of a fully assembled mass matrix [Wat87], while the condition numbers of K, G, and L depend on the choice of reference simplex: in this work, they are computed with respect to the biunit simplex $\Delta^d_{\text{bi}} = \{\boldsymbol{x} \in \mathbb{R}^d : \boldsymbol{x} \geq -1, \ \sum_i x_i \leq 2 - d\}$. The rankings of the node families by these metrics are essentially the same as by the Lebesgue constant in Section 3.2.

```
# Plot condition numbers of finite element matrices for
# recursive nodes
from recursivenodes import recursive_nodes as rn
from recursivenodes.nodes import (
    equispaced as eq,
    blyth_luo_pozrikidis as blp,
    warburton as wb,
    rapetti_sommariva_vianello as rsv)
from recursivenodes.metrics import (
    mass_matrix_condition as mmc,
    weak_laplacian_condition as wlc,
    nodal_laplacian_condition as nlc,
    nodal_gradient_condition as ngc)
import matplotlib.pyplot as plt
import numpy as np

# List of metrics to test and their names
metrics = (
    (mmc, '$\kappa_2(M)$, mass'),
    (wlc, '$\kappa_2(K)$, stiffness'),
    (ngc, '$\kappa_2(G)$, nodal gradient'),
    (nlc, '$\kappa_2(L)$, nodal Laplacian'),
```

```
    )
# List of node families to test and their
# names and markers
labeled = (
    (rn, '$R^d_{X,n}$', '+'),
    (eq, 'equispaced', 'x'),
    (blp, 'BLP', '1'),
    (wb, 'warp & blend', '2'),
    (rsv, 'RSV-lebgls', '3'),
    )
fig,axes = plt.subplots(ncols=2, nrows=4, figsize=(8.,12.5))
# For each dimension
for (i,d) in enumerate(range(2,4)):
    # For each metric
    for (j,named_metric) in enumerate(metrics):
        ax = axes[j,i]
        metric, metric_name = named_metric
        ymin = np.inf
        ymax = 0.
        # For each family
        for (family, label, marker) in labeled:
            if d == 3 and family == rsv: continue
            kappas = []
            ns = range(4,16)
            # For each degree
            for n in ns:
                # Get the node family
                if family == rsv:
                    nodes = family(n, domain='biunit')
                else:
                    nodes = family(d, n, domain='biunit')
                # Compute the metic
                kappa = metric(d, n, nodes, domain='biunit')
                kappas.append(kappa)
                # Use all node families but equispaced to
                # determine the range of the plot
                if label != 'equispaced':
                  ymin = min(ymin, kappa)
                  ymax = max(ymax, kappa)
            # Plot the family
            _ = ax.scatter(ns, kappas, label=label, marker=marker)
        _ = ax.set_yscale('log')
        # Scale to the data
        _ = ax.set_ylim([0.5*ymin, 2.*ymax])
        _ = ax.set_xlabel('polynomial degree $n$')
```

```
        _ = ax.set_title(f'{metric_name} matrix, d = {d}')
handles, labels = axes[3,0].get_legend_handles_labels()
_ = fig.legend(handles, labels, loc=(0.115,0.), ncol=5)
_ = plt.tight_layout(rect=(0,0.01,1,1))
_ = plt.show()
```

```
# Tabulate 2D finite element condition numbers and
# their asymptotic growth rates for the recursive
# nodes
from tabulate import tabulate
from recursivenodes import recursive_nodes as rn
from recursivenodes.metrics import (
    mass_matrix_condition as mmc,
    weak_laplacian_condition as wlc,
    nodal_gradient_condition as ngc,
    nodal_laplacian_condition as nlc,
    )
T = {}
ns = [4, 8, 16, 24, 32]
T['$n$'] = ns
M = r'$\kappa_2(M)$'
K = r'$\kappa_2(K)$'
G = r'$\kappa_2(G)$'
L = r'$\kappa_2(L)$'
T[M] = [mmc(2, n, nodes=rn(2, n, domain='biunit')) for n in ns]
T[r'$\kappa_2(M)^{1/n}$'] = [k**(1/n) for k,n in zip(T[M], ns)]
T[K] = [wlc(2, n, nodes=rn(2, n, domain='biunit')) for n in ns]
T[r'$\kappa_2(K)^{1/n}$'] = [k**(1/n) for k,n in zip(T[K], ns)]
T[G] = [ngc(2, n, nodes=rn(2, n, domain='biunit')) for n in ns]
T[r'$\kappa_2(G)^{1/n}$'] = [k**(1/n) for k,n in zip(T[G], ns)]
T[L] = [nlc(2, n, nodes=rn(2, n, domain='biunit')) for n in ns]
T[r'$\kappa_2(L)^{1/n}$'] = [k**(1/n) for k,n in zip(T[L], ns)]
print(tabulate(T, tablefmt="pipe", headers="keys",
  floatfmt=
    ('d','.1e','.3f','.1e','.3f','.1e','.3f','.1e','.3f')))
```

Table 2: Finite element matrix condition numbers $\boldsymbol{R}^2_{\boldsymbol{X}}$

n	$\kappa_2(M)$	$\kappa_2(M)^{1/n}$	$\kappa_2(K)$	$\kappa_2(K)^{1/n}$	$\kappa_2(G)$	$\kappa_2(G)^{1/n}$	$\kappa_2(L)$	$\kappa_2(L)^{1/n}$
4	4.7e+01	2.618	1.0e+02	3.196	1.7e+01	2.022	8.2e+00	1.691
8	2.0e+02	1.933	9.5e+02	2.358	7.0e+01	1.700	1.3e+02	1.840
16	1.3e+04	1.808	1.7e+05	2.124	1.2e+03	1.561	1.9e+04	1.848
24	2.8e+06	1.856	6.3e+07	2.113	2.8e+04	1.532	7.4e+06	1.933
32	8.0e+08	1.898	2.5e+10	2.114	6.2e+05	1.517	3.2e+09	1.982

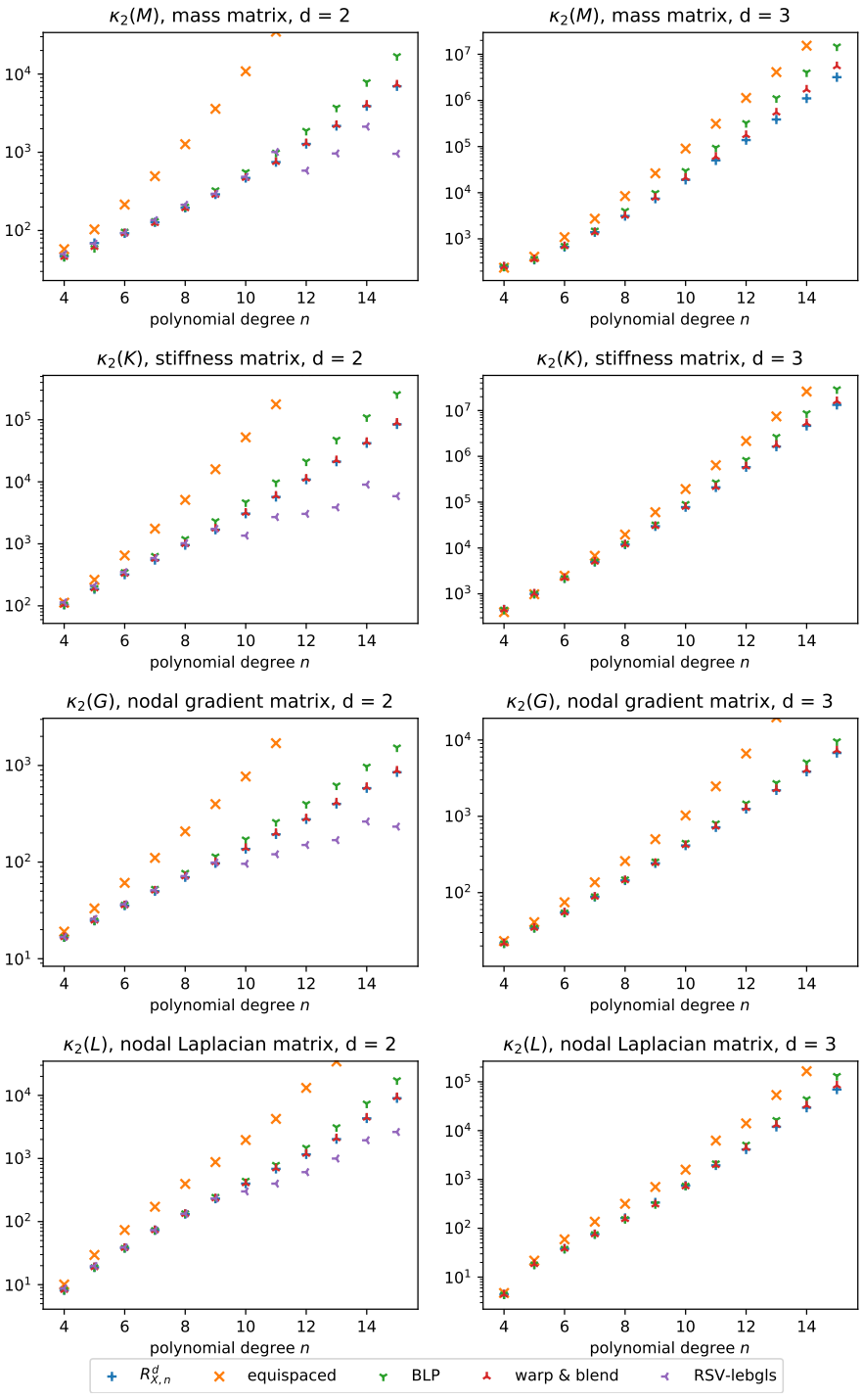

Figure 7: Condition numbers of finite element matrices.

```
# Tabulate 3D finite element condition numbers and
# their asymptotic growth rates for the recursive
# nodes
from tabulate import tabulate
from recursivenodes import recursive_nodes as rn
from recursivenodes.metrics import (
    mass_matrix_condition as mmc,
    weak_laplacian_condition as wlc,
    nodal_gradient_condition as ngc,
    nodal_laplacian_condition as nlc,
    )
T = {}
ns = [4, 8, 12, 16]
T['$n$'] = ns
M = r'$\kappa_2(M)$'
K = r'$\kappa_2(K)$'
G = r'$\kappa_2(G)$'
L = r'$\kappa_2(L)$'
T[M] = [mmc(3, n, nodes=rn(3, n, domain='biunit')) for n in ns]
T[r'$\kappa_2(M)^{1/n}$'] = [k**(1/n) for k,n in zip(T[M], ns)]
T[K] = [wlc(3, n, nodes=rn(3, n, domain='biunit')) for n in ns]
T[r'$\kappa_2(K)^{1/n}$'] = [k**(1/n) for k,n in zip(T[K], ns)]
T[G] = [ngc(3, n, nodes=rn(3, n, domain='biunit')) for n in ns]
T[r'$\kappa_2(G)^{1/n}$'] = [k**(1/n) for k,n in zip(T[G], ns)]
T[L] = [nlc(3, n, nodes=rn(3, n, domain='biunit')) for n in ns]
T[r'$\kappa_2(L)^{1/n}$'] = [k**(1/n) for k,n in zip(T[L], ns)]
print(tabulate(T, tablefmt="pipe", headers="keys",
  floatfmt=
    ('d','.1e','.3f','.1e','.3f','.1e','.3f','.1e','.3f')))
```

Table 3: Finite element matrix condition numbers $\boldsymbol{R}^3_{\boldsymbol{X}}$

n	$\kappa_2(M)$	$\kappa_2(M)^{1/n}$	$\kappa_2(K)$	$\kappa_2(K)^{1/n}$	$\kappa_2(G)$	$\kappa_2(G)^{1/n}$	$\kappa_2(L)$	$\kappa_2(L)^{1/n}$
4	2.5e+02	3.977	4.5e+02	4.615	2.2e+01	2.158	4.4e+00	1.449
8	3.1e+03	2.734	1.2e+04	3.231	1.4e+02	1.862	1.6e+02	1.889
12	1.4e+05	2.682	5.8e+05	3.022	1.3e+03	1.812	4.1e+03	2.001
16	9.3e+06	2.726	3.8e+07	2.979	1.2e+04	1.798	1.8e+05	2.132

In Tables 2 and 3 the growth rates of these condition numbers can be assessed. The values of $\kappa_2(M)^{1/n}$, $\kappa_2(K)^{1/n}$ and $\kappa_2(L)^{1/n}$ are not monotonically decreasing in both dimensions for values of n that have been calculated, which suggests super-exponential growth. $\kappa_2(G)^{1/n}$ appears to be monotonically decreasing towards some limit $\gamma_d > 1$ for $d = 2$, and $d = 3$, which suggests exponential growth, but there is no proof of this fact.

3.5 Finite element matrix efficiency

To evaluate the basis functions of $R^d_{\boldsymbol{X},n}$ at a set of nodes Q, one can compute the Vandermonde matrices $V_{R^d_{\boldsymbol{X},n}}$ and V_Q with respect to a stable basis for $\mathcal{R}_n(\Delta^d)$, such as the Proriol-Koornwinder-Dubiner basis [Pro57],[Koo75],[Dub91], and assemble $V_Q V^{-1}_{R^d_{\boldsymbol{X},n}}$. This approach goes back at least to [WPH00], and was improved with a singularity-free evaluation of the basis by Kirby [Kir10b]. Assuming $|Q| \in \Theta(n^d)$, the cost of constructing this matrix is $\Theta(n^{3d})$. There is no structure in $R^d_{\boldsymbol{X},n}$ that would allow for fast application to a vector of nodal coefficients, so the cost of a matrix vector product is $\Theta(n^{2d})$. The same costs hold for each directional derivative of the basis functions.

There appears to be no Lagrange polynomial basis for $\mathcal{R}_n(\Delta^d)$ that improves on this for $d > 1$, so all of the node families discussed above are equal with respect to this metric. It must be noted, however, that outside of Lagrange bases are bases that have fast algorithms, either through hierarchical construction, like Bernstein-Bézier polynomials, or through generalized tensor-product constructions related to the Duffy transformation, like the basis of Sherwin and Karniadakis [SK95]. The Bernstein-Bézier basis has been the subject of more recent work, and has fast algorithms that allow for optimal construction in $\Theta(n^{2d})$ and application in $\Theta(n^d)$ for these matrices, for constant coefficients matrices without quadrature [Kir10a], for evaluation at the Stroud quadrature points [AAD11], and for the inverse of the mass matrix [Kir16]. A drawback of the Bernstein-Bézier basis is mass-matrix condition numbers $\kappa_2(n,d) = \binom{2n+d}{n}$ that are worse even than equispaced nodes, though recent work by Allen and Kirby [AK20] shows that a condition number that uses a matrix norm based on the L_2 norm of the reconstructed polynomials grows like $\sqrt{\kappa_2(n,d)}$.

3.6 Ease of computation and implementation

Implicitly defined node families, including *Roth-leb*, *RSV-leb*, *RSV-lebgls*, *CB*, and *HT* from Section 3.2 and others not discussed, require the solution of an optimization problem over the choice of node coordinates, a problem size that, even with symmetries enforced, is $\Theta(n^d)$ in n. Objective functions like $\Lambda^{\max}_n(X)$ are quite nonconvex, so care must be taken to avoid local minima. It is fair to characterize these node sets as relatively expensive to compute from scratch.

The ease of implementing node sets from a node family is distinct from the computational complexity of computing the node sets from scratch. Most of the implicitly defined node sets discussed in this work have published node sets for moderate values of n for $d = 2$ [CB95][Hes98][RSV12], and a few for $d = 3$ [CB96][HT00].

Of the explicitly defined node families discussed in this work, the *equispaced* and *BLP* nodes are the cheapest to compute: the former requires $\Theta(d)$ operations and $\Theta(1)$ workspace, the latter requires $\Theta(d^2)$ operations and $\Theta(1)$ workspace per node. The *warp & blend* nodes additionally require $d + 1$ evaluations of 1D

Jacobi polynomials up to degree n at each node (one per facet of the simplex) and one $\Theta(n^3)$ inversion of a 1D Vandermonde matrix of size $n+1$ per node set.

The computational complexity of computing one node $\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha})$ in isolation by the rule (1) satisfies the recursion $T(d) = (d+1)T(d-1) + \Theta(d^2)$, which implies $T(d) \in \Theta((d+1)!)$. The workspace satisfies the recursion $S(d) = S(d-1) + \Theta(d)$, so $S(d) \in \Theta(d^2)$. Neither of these are a concern for $d = 2$ or 3.

If the nodes must be computed for higher dimensions, the cost of computing a full node set can be reduced by caching the lower-dimensional nodes. Then the cost of computing the node sets $\{R^d_{\boldsymbol{X},i}\}_{i=0}^n$ with caching satisfies the recursion $T(d,n) = T(d-1,n) + \Theta(\binom{n+d+1}{d+1}d^2)$, which implies $T(d,n) \in O(\binom{n+d+2}{d+1}d^2)$, leading to an amortized cost per node that is $O((n+d)d^2/n)$. The workspace with caching satisfies the recursion $S(d,n) = S(d-1,n) + \Theta(\binom{n+d+1}{d+1}d)$, so $S(d,n) \in O(\binom{n+d+2}{d+1}d)$, which is an amortized space per node that is $O((n+d)d/n)$.

In terms of implementation, once the 1D node family $\boldsymbol{X}$ is available, the code to compute $\boldsymbol{b}_{\boldsymbol{X}}(\boldsymbol{\alpha})$ is very short. Here is an example implementation in python:

```
import numpy as np

def recursive(alpha, family):
    '''The barycentric d-simplex coordinates for a multi-index
    alpha with length d+1 and sum n, based on a 1D node family.'''
    d = len(alpha) - 1
    n = sum(alpha)
    xn = family[n]
    b = np.zeros((d+1,))
    if d == 1:
        b[:] = xn[[alpha[0], alpha[1]]]
        return b
    weight = 0.
    for i in range(d+1):
        alpha_noti = alpha[:i] + alpha[i+1:]
        w = xn[n - alpha[i]]
        br = recursive(alpha_noti, family)
        b[:i] += w * br[:i]
        b[i+1:] += w * br[i:]
        weight += w
    b /= weight
    return b
```

A reference python implementation, which includes all numerical methods used to evaluate and compare against other node families in this work, is available as the `recursivenodes` package [Isa20a]. The website for the package [Isa20b] hosts a version of this manuscript showing how it was used to generate all figures and tables.

4 Using 1D families other than LGL nodes

The analysis and comparison in Section 3 was conducted under the assumption that the 1D node family $\boldsymbol{X}$ was $\boldsymbol{X}_{\text{LGL}}$, the Lobatto-Gauss-Legendre nodes on the interval $[0, 1]$. The recursive rule (1) allows for an arbitrary 1D node family. While $\boldsymbol{X}_{\text{LGL}}$ appears to be the best choice according to the metrics in Section 3, for completeness a few alternate choices of $\boldsymbol{X}$ are presented here.

$\boldsymbol{X}_{\text{eq}}$ (equispaced): It is not surprising, given the discussion in Section 2 where equispaced nodes provided the intuition behind the recursive rule, that using $\boldsymbol{X} = \boldsymbol{X}_{\text{eq}}$ reproduces the equispaced nodes, $\boldsymbol{b}_{\boldsymbol{X}_{\text{eq}}}(\boldsymbol{\alpha}) = \boldsymbol{b}_{\text{eq}}(\boldsymbol{\alpha})$.

$\boldsymbol{X}_{\text{LGC}}$ (Lobatto-Gauss-Chebyshev): The 1D Lobatto-Gauss-Chebyshev (LGC) node family has good interpolation properties in 1D while being a nested family, with $X_{n,\text{LGC}} \subset X_{2n,\text{LGC}}$. The recursive nodes $\boldsymbol{R}^d_{\boldsymbol{X}_{\text{LGC}}}$ inherit the nested property, as demonstrated in Fig. 8, left.

$\boldsymbol{X}_{\text{GL}}$ (Gauss-Legendre): The 1D Gauss-Legendre (GC) node family does not include the endpoints 0 and 1, but can still be used to construct nodes. Property (III) in Section 3.1 does not hold, and in fact all nodes will be in the interior of the d-simplex as demonstrated in Fig. 8, right.

```
# Demonstrate the nested recursive nodes built from
# the Lobatto-Gauss-Chebyshev nodes, and the interior nodes
# built from the Gauss-Legendre nodes
from recursivenodes import recursive_nodes
import matplotlib.pyplot as plt
fig,(ax,ax2) = plt.subplots(ncols=2, figsize=(7,3.5))
n4 = recursive_nodes(2, 4, family='lgc', domain='equilateral')
n8 = recursive_nodes(2, 8, family='lgc', domain='equilateral')
# Plot lines
_ = ax.plot([1.,0.,-1.,1.],[-0.577,1.155,-0.577,-0.577],'k',
            zorder=1)
# Plot LGC nodes, n = 4
_ = ax.scatter(n4[:,0],n4[:,1],marker='x',label='$X_{LGC}, n=4$',
               zorder=3)
# Plot LGC nodes, n = 8
_ = ax.scatter(n8[:,0],n8[:,1],marker='+',label='$X_{LGC}, n=8$',
               zorder=2)
ax.set_aspect('equal')
_ = ax.axis('off')
n4 = recursive_nodes(2, 4, family='gl', domain='equilateral')
# Plot lines
_ = ax2.plot([1.,0.,-1.,1.],[-0.577,1.155,-0.577,-0.577],'k',
             zorder=1)
# Plot GL nodes, n = 4
_ = ax2.scatter(n4[:,0],n4[:,1],marker='+',label='$X_{GL}, n=4$',
                zorder=2)
```

```
ax2.set_aspect('equal')
_ = ax2.axis('off')
plt.tight_layout()
plt.show()
```

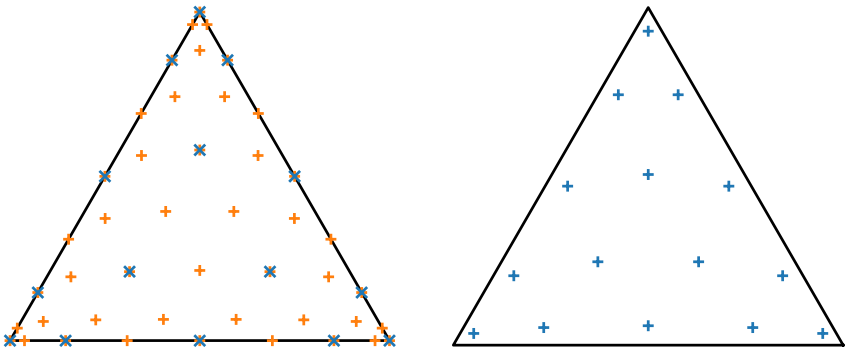

Figure 8: (left) The node sets $R^2_{\boldsymbol{X}_{\mathrm{LGC},4}}$ and $R^2_{\boldsymbol{X}_{\mathrm{LGC},8}}$, demonstrating nested node sets, a property inherited from the 1D LGC node family. (right) The node set $R^2_{\boldsymbol{X}_{\mathrm{GL},4}}$, demonstrating node sets contained in the interior.

The metrics from Section 3 are used to compare $\boldsymbol{R}^d_{\boldsymbol{X}}$ for $\boldsymbol{X}_{\mathrm{LGL}}$, $\boldsymbol{X}_{\mathrm{LGC}}$, and $\boldsymbol{X}_{\mathrm{GL}}$ in Fig. 9.[2] The results for $d = 2$ and $d = 3$ resemble the results of the 1D node families, with GL nodes having worse interpolation properties but better conditioned mass matrices than either set of Lobatto nodes, and with LGC nodes having interpolation properties and matrix condition numbers that similar but slightly worse to LGL nodes. The most interesting trend to be observed in Fig. 9 is that, while the growth rate of the Lebesgue constant for the GL nodes is worse than the Lobatto nodes in 2D, it is much closer in 3D.

```
# Plot Lebesgue constants and finite element matrix condition
# numbers for LGC and GL nodes in comparison to the default
# LGL nodes
from recursivenodes import recursive_nodes as rn
from recursivenodes.nodes import expand_to_boundary
from recursivenodes.metrics import (
    mass_matrix_condition as mmc,
    weak_laplacian_condition as wlc,
    nodal_laplacian_condition as nlc,
    nodal_gradient_condition as ngc)
```

[2]The only metric omitted is the condition number of the nodal Laplacian matrix, where the node families show little distinction.

```
from recursivenodes.lebesgue import lebesguemax as lm
import matplotlib.pyplot as plt
import numpy as np

# Metrics used and their names
metrics = (
    (lm, '$\Lambda_n^{max}$'),
    (mmc, '$\kappa_2(M)$, mass matrix'),
    (wlc, '$\kappa_2(K)$, stiffness matrix'),
    (ngc, '$\kappa_2(G)$, nodal gradient matrix'),
    )
# Nodes measured, their names and markers
labeled = (
    ('lgl', 'LGL', '+'),
    ('lgc', 'LGC', 'x'),
    ('gl', 'GL', '1'),
    )
fig,axes = plt.subplots(ncols=2, nrows=4, figsize=(8.,12.5))
# For each dimension
for (i,d) in enumerate(range(2,4)):
    # For each metric
    for (j,named_metric) in enumerate(metrics):
        ax = axes[j,i]
        metric, metric_name = named_metric
        ymin = np.inf
        ymax = 0.
        # For each node family
        for (family, label, marker) in labeled:
            kappas = []
            ns = range(4,16)
            # For each degree
            for n in ns:
                # Get the node set
                nodes = rn(d, n, family=family, domain='biunit')
                # Compute the metric
                if metric == lm:
                  if family == 'gl':
                    # The method for computing the Lebesgue
                    # constant when the nodes are all in the
                    # interior requires some extra nodes on the
                    # boundary to help place initial guesses for
                    # local maxima of the Lebesgue function
                    ex_nodes = rn(d, n+d+1, family='lgl',
                                  domain='biunit')
                    ex_nodes = expand_to_boundary(d, n, nodes,
```

```
                                                    ex_nodes)
                    kappa, _ = lm(d, n, nodes, ex_nodes=ex_nodes)
                  else:
                    kappa, _ = lm(d, n, nodes)
                else:
                  kappa = metric(d, n, nodes)
                kappas.append(kappa)
                ymin = min(ymin, kappa)
                ymax = max(ymax, kappa)
            # Plot the metric
            _ = ax.scatter(ns, kappas, label=label, marker=marker)
        _ = ax.set_yscale('log')
        _ = ax.set_ylim([0.5*ymin, 2.*ymax])
        _ = ax.set_xlabel('polynomial degree $n$')
        _ = ax.set_title(f'{metric_name}, d = {d}')
handles, labels = axes[2,0].get_legend_handles_labels()
_ = fig.legend(handles, labels, loc=(0.37,0.), ncol=5)
_ = plt.tight_layout(rect=(0,0.01,1,1))
_ = plt.show()
```

5 Interpolation tests

To show that the bounds implied by the Lebesgue constants in 3.2 are in line with interpolation errors in practice, this section compares those errors for the recursively constructed nodes against other node families for two benchmark functions that have appeared previously.

The first function is

$$f_A(\boldsymbol{x}) = \prod_i (x_i + 1) \cosh\left(\sum_i x_i - 1\right), \tag{6}$$

which has appeared in [Hei05][War06][CW15] as an example of a smooth, non-polynomial function to which even the equispaced polynomial interpolants converge. In Table 5, the absolute interpolation errors $\|I_X(f_A) - f_A\|_\infty$ on the biunit simplex is tested for node families that appeared in Section 3.

The next is the "Witch of Agnesi" function,

$$f_B(\boldsymbol{x}) = \frac{1}{1 + \alpha|\boldsymbol{x}|^2}, \tag{7}$$

which for $\alpha = 25$ is the classic Runge function, for which the equispaced interpolants diverge in some domains. Table 5 reports $\|I_X(f_B) - f_B\|_\infty$ on an equilateral simplex centered at the origin with edge length 2. For $d = 2$, the

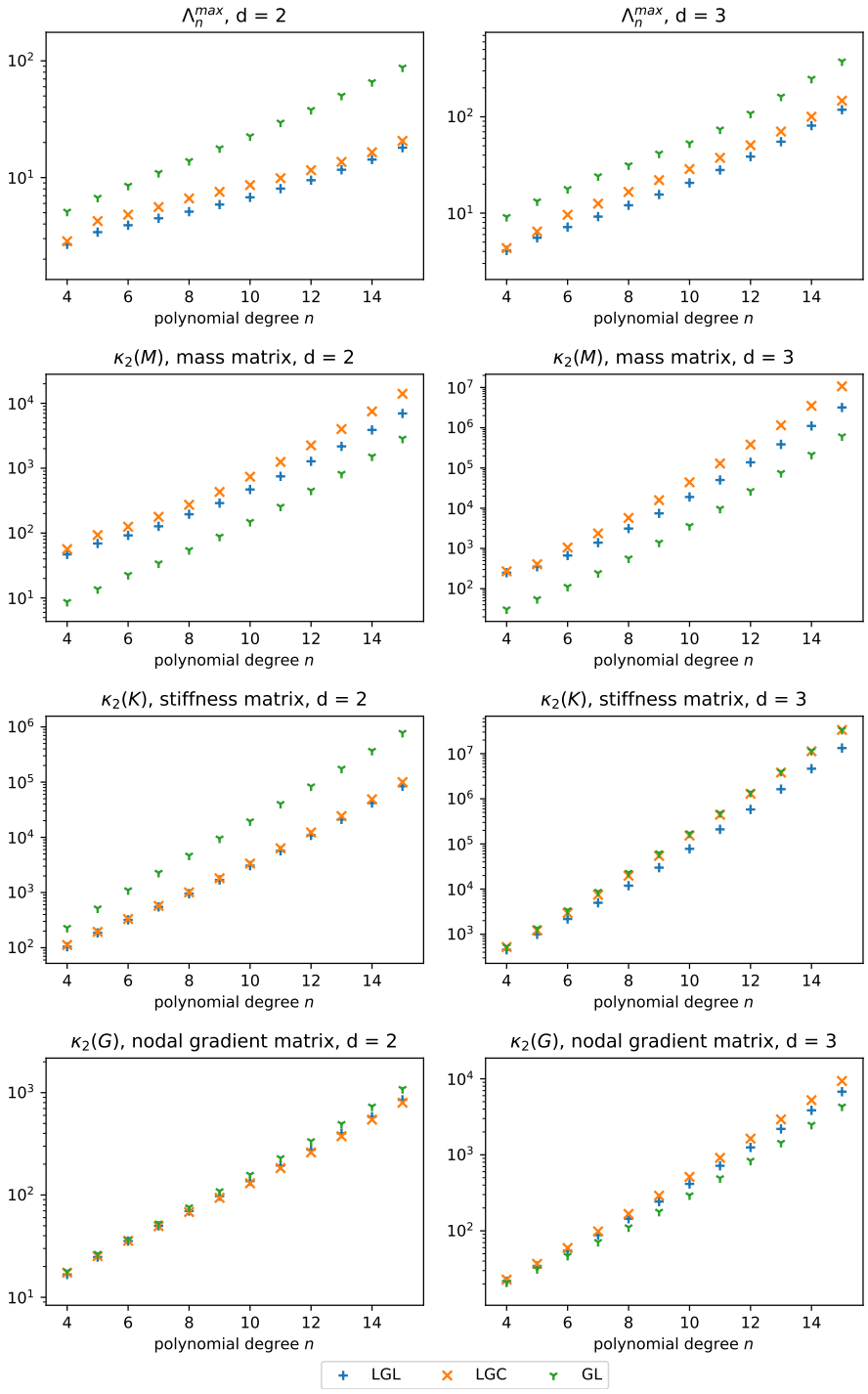

Figure 9: Comparing $\boldsymbol{R}^d_{\boldsymbol{X}_{\mathrm{LGL}}}$, $\boldsymbol{R}^d_{\boldsymbol{X}_{\mathrm{LGC}}}$, and $\boldsymbol{R}^d_{\boldsymbol{X}_{\mathrm{GL}}}$ according to metrics from Section 3.

standard $\alpha = 25$ is used, but for $d = 3$, $\alpha = 60$ is used instead to make errors of the equispaced interpolants roughly the same as for $d = 2$.

```
# Setup a function to compute the maximum interpolation error
# using the refining random search used by Warburton in "An explicit
# construction of interpolation nodes on the simplex"
# (doi:10.1007/s10665-006-9086-6)

import numpy as np
from recursivenodes.nodes import recursive as rec
from recursivenodes.nodes import equispaced as eqs
from recursivenodes.nodes import warburton as war
from recursivenodes.nodes import blyth_luo_pozrikidis as blp
from recursivenodes.nodes import rapetti_sommariva_vianello as rsv
from recursivenodes.polynomials import \
    proriolkoornwinderdubinervandermonde as pkdvdm
from recursivenodes.utils import coord_map
from tabulate import tabulate

class interpolation_error(object):
    '''Take a function and a polynomial interpolant and return a callable for
    the the absolute value of the difference'''
    def __init__(self, n, biunit_nodes, func, domain):
        nodes = coord_map(biunit_nodes, 'biunit', domain)
        funcval = func(nodes)
        self.func = func
        self.domain = domain
        self.n = n
        self.d = biunit_nodes.shape[1]
        V = pkdvdm(self.d, n, biunit_nodes)
        self.coeffs = np.linalg.solve(V,funcval)[:,np.newaxis]

    def __call__(self, x):
        funcval = self.func(x)
        x_biunit = coord_map(x, self.domain, 'biunit')
        polyval = pkdvdm(self.d, self.n, x_biunit, C=self.coeffs)
        return np.abs(funcval - polyval[:,0])

def max_interpolation_error(d, ie):
    npoints = 10000
    # construct random points in d + 1 unit cube
    points = np.random.rand(npoints, d + 1)
    # project them onto the barycentric triangle
    points = points / (np.sum(points, axis=1)[:,np.newaxis])
    niters = 20
    for iter in range(niters):
```

```
        # map them to the desired domain
        equipoints = coord_map(points, 'barycentric', ie.domain)
        # compute the errors
        err = ie(equipoints)
        if iter == niters - 1:
          return max(err)
        # get the top 10% error points
        sortidx = sorted(list(range(npoints)), key=lambda k: err[k])
        # get the barycentric points associated with high errors
        keep_points = points[sortidx[int(0.9*npoints):],:]
        newpoints = np.ndarray((npoints,d+1))
        for (j,point) in enumerate(keep_points):
            # generate new points in a box around them
            boxshift = point-0.5**(iter+2)
            boxshift[boxshift < 0.] = 0.
            boxmax = point+0.5**(iter+2)
            boxmax[boxmax > 1.] = 1.
            boxscale = boxmax - boxshift
            newpoints[10*j:10*(j+1),:] = \
                np.random.rand(10, d + 1) * boxscale + boxshift
            # keep the point that generated these random points
            newpoints[10*j,:] = point
        # new points (reprojected to the barycentric triangle)
        points = newpoints / (np.sum(newpoints, axis=1)[:,np.newaxis])

def err_table(d, nodes, func, domain, ns):
    ie = lambda n: interpolation_error(n, nodes(d, n, domain='biunit'), func, domain)
    return [max_interpolation_error(d, ie(n)) for n in ns]

def rsv2(d, n, domain='barycentric'): return rsv(n, domain=domain)

# Use the maximum interpolation error calculator on a smooth function

def war_func_II(x):
    return np.prod(x + 1., axis=1) * np.cosh(np.sum(x, axis=1) - 1.)

ns = [6, 9, 12, 15, 18]

E = {}
E['$d$'] = [2] * len(ns) + [3] * len(ns)
E['$n$'] = ns + ns
E['equispaced'] = err_table(2, eqs, war_func_II, 'biunit', ns) \
    + err_table(3, eqs, war_func_II, 'biunit', ns)
E['BLP'] = err_table(2, blp, war_func_II, 'biunit', ns) \
    + err_table(3, blp, war_func_II, 'biunit', ns)
```

```
E['warp \& blend'] = err_table(2, war, war_func_II, 'biunit', ns) \
    + err_table(3, war, war_func_II, 'biunit', ns[:-1]) + [None]
E['RSV-lebgls'] = err_table(2, rsv2, war_func_II, 'biunit', ns) \
    + [None] * len(ns)
E[r'${\RR}^d_{\XX}$'] = err_table(2, rec, war_func_II, 'biunit', ns) \
    + err_table(3, rec, war_func_II, 'biunit', ns)

# print out the computed interpolation errors
print(tabulate(E, tablefmt="pipe", headers="keys", missingval='-', floatfmt='.1e'))
```

d	n	equispaced	BLP	warp & blend	RSV-lebgls	$\boldsymbol{R}^d_{\boldsymbol{X}}$
2	6	3.6e-04	2.6e-04	2.4e-04	2.4e-04	2.2e-04
2	9	2.7e-07	2.4e-07	1.7e-07	1.6e-07	1.6e-07
2	12	7.9e-11	7.3e-11	3.6e-11	1.5e-11	3.6e-11
2	15	5.7e-14	1.5e-14	8.4e-15	5.1e-15	8.7e-15
2	18	4.1e-13	1.4e-14	1.6e-14	5.8e-15	4.6e-15
3	6	1.1e-03	8.4e-04	8.1e-04	-	7.8e-04
3	9	9.1e-07	1.6e-06	1.3e-06	-	1.1e-06
3	12	4.0e-10	1.1e-09	7.4e-10	-	4.6e-10
3	15	1.2e-13	3.6e-13	1.8e-13	-	9.0e-14
3	18	6.3e-13	9.9e-14	-	-	4.6e-14

```
print("")
```

```
print(r'Table: Interpolation errors $\|I_X(f_A) - f_A\|_\infty$\label{tab:interperr_smooth}'
```

Table: Interpolation errors $\|I_X(f_A) - f_A\|_\infty$

```
print("")

# Use the maximum interpolation error calculator on a Runge-like function
```

```
def agnesi2(x):
    radius = np.linalg.norm(x, axis=-1)
    return 1/(1 + 25*radius**2)

def agnesi3(x):
    radius = np.linalg.norm(x, axis=-1)
    return 1/(1 + 60*radius**2)

E['equispaced'] = err_table(2, eqs, agnesi2, 'equilateral', ns) \
    + err_table(3, eqs, agnesi3, 'equilateral', ns)
E['BLP'] = err_table(2, blp, agnesi2, 'equilateral', ns) \
    + err_table(3, blp, agnesi3, 'equilateral', ns)
E['warp \& blend'] = err_table(2, war, agnesi2, 'equilateral', ns) \
```

```
    + err_table(3, war, agnesi3, 'equilateral', ns[:-1]) + [None]
E['RSV-lebgls'] = err_table(2, rsv2, agnesi2, 'equilateral', ns) \
    + [None] * len(ns)
E[r'${\RR}^d_{\XX}$'] = err_table(2, rec, agnesi2, 'equilateral', ns) \
    + err_table(3, rec, agnesi3, 'equilateral', ns)

# print out the computed interpolation errors
print(tabulate(E, tablefmt="pipe", headers="keys", missingval='-', floatfmt='.1e'))
```

d	n	equispaced	BLP	warp & blend	RSV-lebgls	$\boldsymbol{R}^d_{\boldsymbol{X}}$
2	6	4.5e-01	3.0e-01	3.1e-01	3.0e-01	3.1e-01
2	9	6.6e-01	2.4e-01	1.7e-01	1.7e-01	1.7e-01
2	12	1.1e+00	2.6e-01	9.8e-02	7.9e-02	9.9e-02
2	15	1.9e+00	3.0e-01	6.2e-02	4.4e-02	6.8e-02
2	18	3.1e+00	3.5e-01	2.7e-01	2.3e-02	4.9e-02
3	6	6.5e-01	6.9e-01	7.1e-01	-	7.4e-01
3	9	4.1e-01	4.9e-01	5.1e-01	-	5.6e-01
3	12	1.0e+00	1.6e+00	7.7e-01	-	2.3e-01
3	15	1.9e+00	2.4e+00	9.0e-01	-	1.4e-01
3	18	4.5e+00	4.3e+00	-	-	1.3e-01

```
print("")
```

```
print(r'Table: Interpolation errors $\|I_X(f_B) - f_B\|_\infty$\label{tab:interperr_runge}')
```

Table: Interpolation errors $\|I_X(f_B) - f_B\|_\infty$

```
print("")
```

The relative sizes of the interpolation errors in Tables 5 and 5 more or less correspond to the relative sizes of the Lebesgue constants. For the more difficult function f_B the recursive nodes $\boldsymbol{R}^3_{\boldsymbol{X}}$ continue to converge at $n = 18$ when the other node sets have already begun to diverge.

6 Conclusion

How and by whom should the nodes $\boldsymbol{R}^d_{\boldsymbol{X}}$ defined by the recursive rule (1) be used? The comparisons in this paper have made the case that is the best explicit construction rule thus far, because of its simplicity, its symmetry, and its performance in the metrics that matter to finite element construction (but not for producing asymptotically convergent interpolants). It does not outperform the Warburton's *warp & blend* node family in 2D, so software already using those would not benefit from switching, but its performance is superior to all other explicit node families in 3D, particularly for $n \geq 7$. Likewise, where

implicitly defined node families—such as Rapetti, Sommariva, and Vianello's LEBGLS nodes—have been computed and published, they are superior to the $\boldsymbol{R}^d_{\boldsymbol{X}}$ node family, especially in 2D for $n \geq 10$. But at the time of this writing the tetrahedron has not received nearly as much attention as the triangle, and so this new node family is the best available in 3D.

In Section 3.1 it was argued that the edge trace property was useful in aligning nodes with tensor-product cells in hybrid meshes. In 3D, a further necessary condition is for the traces on triangular facets to align with neighboring nodes pyramid cells. There is no perfect analogue for the overdetermined projections (4) in the pyramid, so the development of a matching node construction for the pyramid is left for future work, where it would have to be compared against existing node sets such as those developed by Chan and Warburton [CW15].

7 Acknowledgements

The methods in this work were discovered in the course of research funded by grant #DE-SC0016140 from the U.S. Department of Energy's Office of Advanced Scientific Research. The author also thanks the anonymous referees who suggested additional references and improvements to this work.

References

[AAD11] M. Ainsworth, G. Andriamaro, and O. Davydov. "Bernstein–Bézier Finite Elements of Arbitrary Order and Optimal Assembly Procedures". In: *SIAM Journal on Scientific Computing* 33.6 (2011), pp. 3087–3109. ISSN: 1095-7197. DOI: 10.1137/11082539x.

[AK20] L. Allen and R. C. Kirby. "Structured Inversion of the Bernstein Mass Matrix". In: *SIAM Journal on Matrix Analysis and Applications* 41.2 (2020), pp. 413–431. DOI: 10.1137/19m1284166.

[Blo+12] T. Bloom, L. P. Bos, J.-P. Calvi, and N. Levenberg. "Polynomial interpolation and approximation in C^d". In: *Annales Polonici Mathematici* 106 (2012), pp. 53–81. ISSN: 1730-6272. DOI: 10.4064/ap106-0-5.

[Blo+92] T. Bloom, L. Bos, C. Christensen, and N. Levenberg. "Polynomial Interpolation of Holomorphic Functions in C and C^n". In: *Rocky Mountain Journal of Mathematics* 22.2 (1992), pp. 441–470. ISSN: 0035-7596. DOI: 10.1216/rmjm/1181072740.

[BP06] M. G. Blyth and C. Pozrikidis. "A Lobatto interpolation grid over the triangle". In: *IMA Journal of Applied Mathematics* 71.1 (2006), pp. 153–169. ISSN: 0272-4960. DOI: 10.1093/imamat/hxh077.

[CB95] Q. Chen and I. Babuška. "Approximate optimal points for polynomial interpolation of real functions in an interval and in a triangle". In: *Computer Methods in Applied Mechanics and Engineering* 128.3-4 (1995), pp. 405–417. ISSN: 0045-7825. DOI: 10.1016/0045-7825(95)00889-6.

[CB96] Q. Chen and I. Babuška. “The optimal symmetrical points for polynomial interpolation of real functions in the tetrahedron”. In: *Computer Methods in Applied Mechanics and Engineering* 137.1 (1996), pp. 89–94. ISSN: 0045-7825. DOI: 10.1016/0045-7825(96)01051-1.

[CW15] J. Chan and T. Warburton. “A Comparison of High Order Interpolation Nodes for the Pyramid”. In: *SIAM Journal on Scientific Computing* 37.5 (2015), A2151–A2170. DOI: 10.1137/141000105.

[Dub91] M. Dubiner. “Spectral methods on triangles and other domains”. In: *Journal of Scientific Computing* 6.4 (1991), pp. 345–390. ISSN: 1573-7691. DOI: 10.1007/bf01060030.

[Hei05] W. Heinrichs. “Improved Lebesgue constants on the triangle”. In: *Journal of Computational Physics* 207.2 (2005), pp. 625–638. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2005.02.002.

[Hes98] J. S. Hesthaven. “From Electrostatics to Almost Optimal Nodal Sets for Polynomial Interpolation in a Simplex”. In: *SIAM Journal on Numerical Analysis* 35.2 (1998), pp. 655–676. ISSN: 1095-7170. DOI: 10.1137/s003614299630587x.

[HT00] J. S. Hesthaven and C. H. Teng. “Stable Spectral Methods on Tetrahedral Elements”. In: *SIAM Journal on Scientific Computing* 21.6 (2000), pp. 2352–2380. ISSN: 1095-7197. DOI: 10.1137/s1064827598343723.

[Isa20a] T. Isaac. *recursivenodes: Recursive, parameter-free, explicitly defined interpolation nodes for simplices*. Version 0.1.0. 2020. DOI: 10.5281/zenodo.3675431.

[Isa20b] T. Isaac. *recursivenodes: Recursive, parameter-free, explicitly defined interpolation nodes for simplices*. 2020. URL: https://tisaac.gitlab.io/recursivenodes (visited on 02/21/2020).

[Kir10a] R. C. Kirby. “Fast simplicial finite element algorithms using Bernstein polynomials”. In: *Numerische Mathematik* 117.4 (2010), pp. 631–652. ISSN: 0945-3245. DOI: 10.1007/s00211-010-0327-2.

[Kir10b] R. C. Kirby. “Singularity-free evaluation of collapsed-coordinate orthogonal polynomials”. In: *ACM Transactions on Mathematical Software* 37.1 (2010), pp. 1–16. DOI: 10.1145/1644001.1644006.

[Kir16] R. C. Kirby. “Fast inversion of the simplicial Bernstein mass matrix”. In: *Numerische Mathematik* 135.1 (2016), pp. 73–95. ISSN: 0945-3245. DOI: 10.1007/s00211-016-0795-0.

[Koo75] T. Koornwinder. “Two-Variable Analogues of the Classical Orthogonal Polynomials”. In: *Theory and Application of Special Functions* (1975), pp. 435–495. DOI: 10.1016/b978-0-12-064850-4.50015-x.

[LP06] H. Luo and C. Pozrikidis. “A Lobatto interpolation grid in the tetrahedron”. In: *IMA Journal of Applied Mathematics* 71.2 (2006), pp. 298–313. ISSN: 0272-4960. DOI: 10.1093/imamat/hxh111.

[Pro57] J. Proriol. “Sur une famille de polynomes á deux variables orthogonaux dans un triangle”. In: *Comptes Rendus Hebdomadaires Des Seances De L’Academie Des Sciences* 245.26 (1957), pp. 2459–2461.

[RM10] B. N. Ryland and H. Z. Munthe-Kaas. “On Multivariate Chebyshev Polynomials and Spectral Approximations on Triangles”. In: *Spectral and High Order Methods for Partial Differential Equations* (2010), pp. 19–41. ISSN: 1439-7358. DOI: 10.1007/978-3-642-15337-2_2.

[Rot05] M. J. Roth. “Nodal configurations and Voronoi tessellations for triangular spectral elements”. PhD thesis. University of Victoria, 2005. URL: http://hdl.handle.net/1828/44.

[RSV12] F. Rapetti, A. Sommariva, and M. Vianello. “On the generation of symmetric Lebesgue-like points in the triangle”. In: *Journal of Computational and Applied Mathematics* 236.18 (2012), pp. 4925–4932. ISSN: 0377-0427. DOI: 10.1016/j.cam.2011.11.023.

[SK95] S. J. Sherwin and G. E. Karniadakis. “A new triangular and tetrahedral basis for high-order (hp) finite element methods”. In: *International Journal for Numerical Methods in Engineering* 38.22 (1995), pp. 3775–3802. DOI: 10.1002/nme.1620382204.

[War06] T. Warburton. “An explicit construction of interpolation nodes on the simplex”. In: *Journal of Engineering Mathematics* 56.3 (2006), pp. 247–262. ISSN: 1573-2703. DOI: 10.1007/s10665-006-9086-6.

[Wat87] A. J. Wathen. “Realistic Eigenvalue Bounds for the Galerkin Mass Matrix”. In: *IMA Journal of Numerical Analysis* 7.4 (1987), pp. 449–457. DOI: 10.1093/imanum/7.4.449.

[WPH00] T. Warburton, L. Pavarino, and J. Hesthaven. “A Pseudo-spectral Scheme for the Incompressible Navier–Stokes Equations Using Unstructured Nodal Elements”. In: *Journal of Computational Physics* 164.1 (2000), pp. 1–21. DOI: 10.1006/jcph.2000.6587.