Nitro documentation

January 13, 2019

1 Input file

An input file (e.g. <input.txt>) includes the following items:

- spaceDim
  Space dimension, specifically spaceDim = 2 for planar curves and spaceDim = 3 for space curves and surfaces.

- degree (for surface)
  For tensor-product surfaces, uDeg and vDeg denote the degree of the surface in the two directions (θ^1, θ^2) respectively.

- geometryFileName
  The file name of the curve or surface.

- lineFileName
  The file name of the line.

- method
  The method to be used for solving the generalised eigenvalue problem in intersection computation. Four methods available are REDUCTION (pencil reduction method), SQUARE (square submatrix method), SVD (SVD preconditioning method) and QR (QR preconditioning method).

- inversion
  The inversion method. Two methods available are SVD and EIGENVEC (using the eigenspace in the generalised eigenvalue problem).

The variables of the input file are read with the following lines in the code,

```
// read input file
const std::string inputFile( argv[1] );
std::ifstream inp( inputFile.c_str() );
lineCurve::Parameters parameters(inp); {or lineSurface::Parameters parameters(inp);}
inp.close();

// read, record and retrieve input variables
std::ifstream geometryf( parameters.geometryFileName.c_str() );
const int spaceDim = parameters.spaceDim;
```

2 Geometry files

Geometry files (for curves, surfaces and lines) use the .obj file format, which contains the vertices of the geometry. A general format of a geometry file <geometry.obj> is as follows,

$$\begin{array}{cccc} \text{v} & x_1^1 & x_1^2 & x_1^3 \\ \text{v} & x_2^1 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ \text{v} & x_n^1 & x_n^2 & x_n^3 \end{array}$$

For Lagrange curves and surfaces, the vertices in input files are those interpolating points (known points), and the vertices are arranged in the way which will be described later. The curve and surface is read via the following lines,

```
// read geometry vertices
typedef nitro::input::Point Point;
std::vector<Point> points;
nitro::input::readGeometry(geometryf, points, spaceDim);
```

For the intersecting line, the two vertices are the two end points of the line segment. Different choices of end points of the line only affect the line parameters ξ^* of intersection points, not the coordinates $\boldsymbol{x}^*$ nor the surface parameters $\boldsymbol{\theta}^*$. The line is read via the following lines,

```
// create a line object
typedef nitro::input::Line Line;
Line line;

// read line from the input file
std::ifstream linef( parameters.lineFileName.c_str() );
nitro::input::readLine(linef, line, spaceDim);
```

2.1 Curves

It is straightforward for the vertex indexing of a curve. For example, the Lagrange curve represented by the example curve <curve.obj> in "lineCurve" is depicted as Figure 1.

Point ID	Coordinates ($\boldsymbol{x}_i$)	Parameter (θ)
1	$(0, 0, 0)$	0
2	$(1, 1, 0)$	$1/3$
3	$(2, -0.5, 0)$	$2/3$
4	$(4, 0, 0)$	1

Table 1: Vertex indexing of the Lagrange curve.

A curve object is defined with the points and converted to an equivalent monomial curve via the following lines,

```
// create a Lagrange curve object
const nitro::curve curveBasis = nitro::LAGRANGE;
typedef nitro::Curve< curveBasis, Point > Geometry;
```

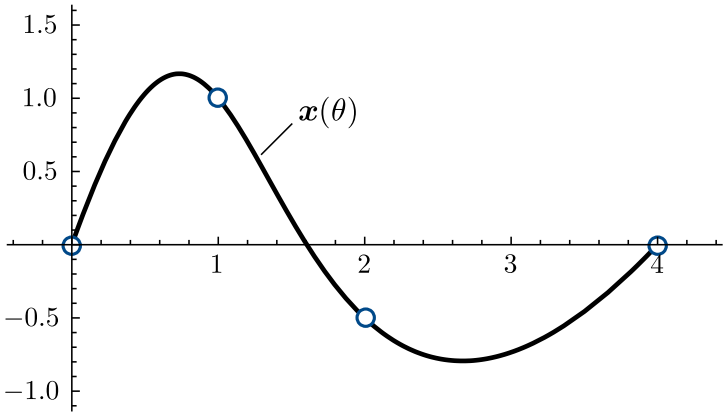

Figure 1: The Lagrange curve with interpolating points listed in Table 1.

```
Geometry curve(spaceDim);
curve.setCoefficients(points);

// get coefficients of the equivalent monomial curve
std::vector<Point> coeffsMonomial;
curve.getMonomialCoeffs(coeffsMonomial);

// create the equivalent monomial curve
typedef nitro::Curve< nitro::MONOMIAL, Point > Monomial;
Monomial geometryMonomial(spaceDim);
geometryMonomial.setCoefficients(coeffsMonomial);
```

2.2 Tensor-product surfaces

The vertex indexing of a tensor-product surface follows the manner as shown in Figure 2, which denotes a tensor-product surface of bi-degree $(3, 3)$ (uDeg = 3, vDeg = 3).

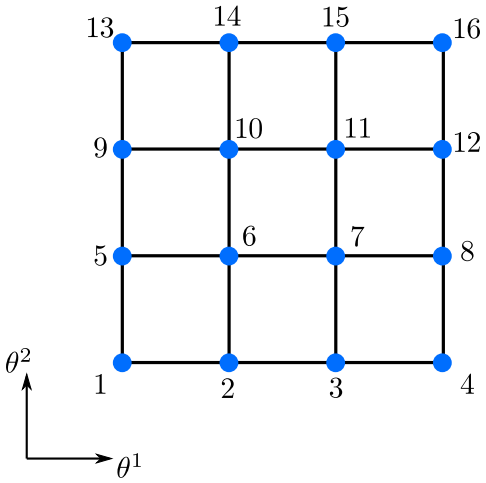

Figure 2: Vertex indexing of the tensor-product surface.

A surface object is defined with the points and converted to an equivalent monomial surface via the following lines,

```
// create a tensor-product Lagrange surface
const nitro::surface surfaceBasis = nitro::TENSORLAGRANGE;
typedef nitro::Surface< surfaceBasis, Point > Geometry;
Geometry surface(uDeg, vDeg);
surface.setCoefficients(points);

// get coefficients of the equivalent monomial surface
std::vector¡Point¿ coeffsMonomial;
surface.getMonomialCoeffs(coeffsMonomial);

// create the equivalent monomial surface
typedef nitro::Surface< nitro::TENSORMONOMIAL, Point > Monomial;
Monomial geometryMonomial(uDeg, vDeg);
geometryMonomial.setCoefficients(coeffsMonomial);
```

3 Moving lines / planes

The curve or surface is implicitly represented with moving lines or planes. For the purpose of generality, the moving lines or planes are computed based on the equivalent monomial counterpart, which is achieved with the following lines,

```
// create moving line object
typedef nitro::movingLines::MovingLine< Geometry::dim, Monomial > MovingLines;
MovingLines movingLines;
.....
// alternatively, create moving lines with user-defined degrees
MovingLines movingLines(q_g); // for curves

MovingLines movingLines(q_g^1, q_g^2); // for tensor-product surfaces
.....
// compute moving lines
movingLines.registerGeometry(geometryMonomial);
movingLines.setRankTolerance(1.e-6);
movingLines.computeCoefficients();
```

Note that the default constructor takes the minimum degree necessary for the auxiliary vector, i.e. $q_g = q_x - 1$ for curves, and $q_g^1 = 2q_x^1 - 1, q_g^2 = q_x^2 - 1$ (or by symmetry $q_g^1 = q_x^1 - 1, q_g^2 = 2q_x^2 - 1$) for tensor-product surfaces. The moving lines / planes can also be defined with user-defined degrees q_g for curves or (q_g^1, q_g^2) for surfaces,

4 Intersection compuation

The intersection between a line and a curve or surface can be finally computed with an "intersector" as follows,

```
// select intersection computation method and assign computation tolerance
std::string method = parameters.method;
```

```
const double tol = 1.e-7;

// create intersector object
typedef nitro::lineIntersector::LineIntersector< MovingLines, Line > LineIntersector;
LineIntersector lineIntersector( movingLines, line, method, tol );

// solve generalised eigenvalue problem
lineIntersector.solveEigenValue();

// inversion process
std::string inversion = parameters.inversion;
lineIntersector.computeIntersection(inversion);

// Finally, get all the intersection points
typedef LineIntersector::IntersectPoint IntersectPoint;
std::vector¡IntersectPoint¿ intersectPoints;
lineIntersector.giveIntersection(intersectPoints);
```

Note that currently four methods are available for computing generalised eigenvalue problem: REDUCTION, SQUARE, SVD and QR.

When REDUCTION method is used, the inversion process can only be performed with SVD. Using the eigenspace (EIGENVEC) for the inversion process is applicable when one of SQUARE, SVD and QR methods is selected. It would be better to choose SVD for the inversion process in terms of accuracy when multiple pre-images of intersection points exist.