pySpline

MDO Lab

May 09, 2022

CONTENTS

`pySpline` is a package containing code for wiking with b-spline `curve`, `surface`, and `volume` objects. The Python interface to all functionality is found in the pySpline.py module. The documentation for this package is summarized below:

BUILDING

For speed purposes, `pySpline` uses a small compiled Fortran library for doing the time consuming computational operations. It is therefore necessary to build this library before using `pySpline`.

pySpline follows the standard MDO Lab build procedure. To start, find a configuration file close to your current setup in:

```
$ config/defaults
```

and copy it to ''config/config.mk''. For example:

```
$ cp config/defaults/config.LINUX_GFORTRAN.mk config/config.mk
```

If you are a beginner user installing the packages on a linux desktop, you should use the `config.LINUX_GFORTRAN.mk` versions of the configuration files. The `config.LINUX_INTEL.mk` versions are usually used on clusters.

Once you have copied the config file, compile *pySpline* ` by running:

```
$ make
```

If everything was successful, the following lines will be printed to the screen (near the end):

```
Testing if module pyspline can be imported...
Module pyspline was successfully imported.
```

If you don't see this, it will be necessary to configure the build manually. To configure manually, open `config/config.mk` and modify options as necessary.

Lastly, to build and install the Python interface, type:

```
pip install .
```

REGRESSION TESTS

pySpline includes a set of built-in regression tests that are used to ensure that the code reproduces a consistent set of reproducible results. It is *HIGHLY* recommended that the regression tests are run after installation on a new system to ensure consistency of the code build.

To run the regression tests go to the following directory:

```
python/reg_tests
```

and run the automatic testing script:

```
python run_reg_tests.py
```

If successful, you should see the following line:

```
pyspline: Success!
```

if you don't, then *something* went wrong in the test. By default an `xxdiff` window will appear that should quickly highlight where the discrepency is.

After running the regression tests, there will be *three* important files to look at:

```
pyspline_reg.ref
pyspline_reg
pyspline_ref.orig
```

- **`pyspline_reg.ref` is the reference file that contains the** 'truth' values that we should reproduce

- `pyspline_ref` is the generated output that is *identical* to the reference output *except* for @value lines where a discrepency has been detected. A graphical diff of `pyspline_reg` and `pyspline_reg.ref` should quickly show where the discrepancies are.

- `pyspline_ref.org` is the *actual* output of the regression run. Generally speaking there will be small discrepancies in the 15 or 16th digit which depends on the compiler. This will trigger a text-based diff program which would make it difficult to determine which values are incorrect in a file with several thousand lines.

Note: Only lines that contain the @value magic string are actually compared by the regression tester.

2.1 Regression Test Script Help

To see a list of options for the regression test script run:

```
python run_reg_test.py --help
```

which will show:

```
usage: run_reg_tests.py [-h] [-mode {train,compare}] [-diff_cmd DIFF_CMD]
                        [-nodiff]

optional arguments:
  -h, --help            show this help message and exit
  -mode {train,compare}
                        Train generates reference file. Compare runs test
  -diff_cmd DIFF_CMD    Command to run for displaying diff. Default: xxdiff
  -nodiff               Suppress displaying the comparison if not successful
```

EXAMPLES

Several examples from the `python/examples` directory are included here for reference.

3.1 Curve example

```python
# This is a test script to test the functionality of the
# pySpline curve class

# External modules
import numpy as np

# First party modules
from pyspline import Curve

# Get some Helix-like data

n = 100
theta = np.linspace(0.0000, 2 * np.pi, n)
x = np.cos(theta)
y = np.sin(theta)
z = np.linspace(0, 1, n)
print("Helix Data")
curve = Curve(x=x, y=y, z=z, k=4, Nctl=16, niter=100)
curve.writeTecplot("helix.dat")

# Load naca0012 data
print("Naca 0012 data")
x, y = np.loadtxt("naca0012", unpack=True)
curve = Curve(x=x, y=y, k=4, Nctl=11, niter=500)
curve.writeTecplot("naca_data.dat")

# Projection Tests
print("Projection Tests")
x = [0, 2, 3, 5]
y = [-2, 5, 3, 0]
z = [0, 0, 0, 0]
curve1 = Curve(x=x, y=y, z=z, k=4)

curve1.writeTecplot("curve1.dat")
```

```python
x = [-2, 5, 2, 1]
y = [5, 1, 4, 2]
z = [3, 0, 1, 4]

curve2 = Curve(x=x, y=y, z=z, k=4)
curve2.writeTecplot("curve2.dat")

# Get the minimum distance distance between a point and each curve
x0 = [4, 4, 3]
s1, D1 = curve1.projectPoint(x0, s=0.5)
val1 = curve1(s1)  # Closest point on curve
s2, D2 = curve2.projectPoint(x0, s=1.0)
val2 = curve2(s2)  # Closest point on curve

# Output the data
f = open("projections.dat", "w")
f.write('VARIABLES = "X", "Y","Z"\n')
f.write("Zone T=curve1_proj I=2 \n")
f.write("DATAPACKING=POINT\n")
f.write("%f %f %f\n" % (x0[0], x0[1], x0[2]))
f.write("%f %f %f\n" % (val1[0], val1[1], val1[2]))

f.write("Zone T=curve2_proj I=2 \n")
f.write("DATAPACKING=POINT\n")
f.write("%f %f %f\n" % (x0[0], x0[1], x0[2]))
f.write("%f %f %f\n" % (val2[0], val2[1], val2[2]))

# Get the minimum distance between the two curves
s, t, D = curve1.projectCurve(curve2)
val1 = curve1(s)
val2 = curve2(t)

f.write("Zone T=curve1_curve2 I=2 \n")
f.write("DATAPACKING=POINT\n")
f.write("%f %f %f\n" % (val1[0], val1[1], val1[2]))
f.write("%f %f %f\n" % (val2[0], val2[1], val2[2]))
```

3.2 Surface example

```python
# This is a test script to test the functionality of the
# pySpline surface
# External modules
import numpy as np

# First party modules
from pyspline import Curve, Surface

# Create a generic surface
nu = 20
```

Chapter 3. Examples

```python
nv = 20
u = np.linspace(0, 4, nu)
v = np.linspace(0, 4, nv)
[V, U] = np.meshgrid(v, u)
Z = np.cos(U) * np.sin(V)
surf = Surface(x=U, y=V, z=Z, ku=4, kv=4, Nctlu=5, Nctlv=5)
surf.writeTecplot("surface.dat")

n = 100
theta = np.linspace(0.0000, 2 * np.pi, n)
x = np.cos(theta) - 1
y = np.sin(theta) + 1
z = np.linspace(0, 1, n) + 2
curve = Curve(x=x, y=y, z=z, k=4, Nctl=16, niter=100)
curve.writeTecplot("helix.dat")

u, v, s, D = surf.projectCurve(curve, Niter=100, eps1=1e-10, eps2=1e-10, u=1,
→v=1, s=1)

print(u, v, s, D)
print(curve(s))
print(surf(u, v))
```

3.3 Volume example

```python
# This simple script test some of the volume functionality in pySpline
# Standard Python modules
import time

# External modules
import numpy as np

# First party modules
from pyspline import Volume

X = np.zeros((2, 2, 2, 3))
X[0, 0, 0, :] = [0, 0, 0]
X[1, 0, 0, :] = [1.1, -0.1, -0.1]
X[1, 1, 0, :] = [0.9, 1.05, 0.2]
X[0, 1, 0, :] = [-0.1, 1.1, 0]

X[0, 0, 1, :] = [-0.1, 0.1, 1.5]
X[1, 0, 1, :] = [1.2, -0.2, 1.8]
X[1, 1, 1, :] = [1.2, 1.0, 2]
X[0, 1, 1, :] = [-0.2, 1.3, 2.1]

vol = Volume(X=X, ku=2, kv=2, kw=2, Nctlu=2, Nctlv=2, Nctlw=2)
vol.writeTecplot("vol.dat", orig=True)

# Generate random data
```

```python
M = 10000
Y = np.zeros((M, 3))
for i in range(M):
    Y[i, 0] = np.random.random() * 0.5 + 0.25
    Y[i, 1] = np.random.random() * 0.5 + 0.25
    Y[i, 2] = np.random.random() * 0.5 + 0.25


timeA = time.time()
u, v, w, D = vol.projectPoint(Y)
print("Time to project %d points: %f seconds:" % (M, time.time() - timeA))
```

CURVE

class pyspline.pyCurve.**Curve**(***kwargs*)

Create an instance of a b-spline curve. There are two ways to initialize the class

- **Creation**: Create an instance of the spline class directly by supplying the required information. kwargs MUST contain the following information: `k, t, coef`.

- **LMS/Interpolation**: Create an instance of the spline class by using an interpolating spline to given data points or a LMS spline. The following keyword argument information is required:

  1. k Spline Order

  2. **X real array size (N, nDim) of data to fit. OR**

      1. `x` (1D) and `s` for 1D

      2. `x` (1D) and `y` (1D) for 2D spatial curve

      3. `x` (1D) and y` (1D) and `z` 1D for 3D spatial curve

  3. `s` real array of size (N). Optional for nDim >= 2

Parameters

k [{2, 3, 4}] Order for spline. A spline with order n has at most C^{n-2} continuity. The $n - 1$ derivative will be piecewise constant.

nCtl [int] Number of control points. If this is specified then LMS will be used instead of interpolation.

t [array, list] Knot vector. Used only for creation. Must be size `nCtl + k`

coef [array, size (nCtl, nDim)] Coefficients to use. Only used for creation. The second dimension determine the spatial order of the spline

X [arary, size (N, ndim)] Full array of data to interpolate/fit

x [array, size (N)] Just x coordinates of data to fit

y [array, size (N)] Just y coordinates of data to fit

z [array, size (N)] Just z coordinates of data to fit

s [array, size(N)] Optional parameter to use. Not required for nDim >=2

nIter [int] The number of Hoscheks parameter correction iterations to run. Only used for LMS fitting.

weight [array, size(N)] A array of weighting factors for each fitting point. A value of -1 can be used to exactly constrain a point. By default, all weights are 1.0

> **deriv** [array, size (len(derivPtr), nDim)] Array containing derivative information the user wants to use at a particular point. **EXPERIMENTAL**

> **derivPtr** [int, array, size (nDerivPtr)] Array of indices pointing to the index of points in X (or x,y,z), for which the user has supplied a derivative for in `deriv`. **EXPERIMENTAL**

> **derivWeights** [array size(nDerivPtr)] Optional array of the weighting to use for derivatives. A value of -1 can be used to exactly constrain a derivative. **EXPERIMENTAL**

Examples

```
>>> x = [0, 0.5, 1.0]
>>> y = [0, 0.25, 1.0]
>>> s = [0., 0.5, 1.0]
>>> # Spatial interpolated seg (k=2 makes this a straight line)
>>> line_seg = Curve(x=x, y=y, k=2)
>>> # With explicit parameter values
>>> line_seg = Curve(x=x, y=y, k=2, s=s)
>>> #Interpolate parabolic curve
>>> parabola = Curve(x=x, y=y, k=3)
>>> #Interpolate parabolic curve with parameter values
>>> parabola = Curve(x=x, y=y, k=3, s=s)
```

calcGrevillePoints()

Calculate the Greville points

calcInterpolatedGrevillePoints()

Compute greville points, but with additional interpolated knots

computeData(*recompute=False*)

Compute discrete data that is used for the Tecplot Visualization as well as the data for doing the brute-force checks

> **Parameters**

> > **recompute** [bool] If True, recompute the data even if it has already been computed.

getDerivative(*s*)

Evaluate the derivatie of the spline at parametric position, s

> **Parameters**

> > **s** [float] Parametric location for derivative

> **Returns**

> > **ds** [array] The first derivative. This is an array of size nDim

getLength()

Compute the length of the curve using the Euclidean Norm

> **Returns**

> > **length** [float] Physical length of curve

getSecondDerivative(*s*)

Evaluate the second derivatie of the spline at parametric position, s

> **Parameters**

> **s** [float] Parametric location for derivative

> **Returns**

>> **d2s** [array] The second derivative. This is an array of size nDim

getValue(*s*)

> Evaluate the spline at parametric position, s

>> **Parameters**

>>> **s** [float or array] Parametric position(s) at which to evaluate the curve.

>> **Returns**

>>> **values** [array of size nDim or an array of size (N, 3)] The evaluated points. If a single scalar s was given, the result with be an array of length nDim (or a scalar if nDim=1). If a vector of s values were given it will be an array of size (N, 3) (or size (N) if ndim=1)

insertKnot(*u*, *r*)

> Insert at knot in the curve at parametric position u

>> **Parameters**

>>> **u** [float] Parametric position to split at

>>> **r** [int] Number of time to insert. Should be > 0.

>> **Returns**

>>> **actualR** [int] The number of times the knot was **actually** inserted

>>> **breakPt** [int] Index in the knot vector of the new knot(s)

projectCurve(*inCurve*, *nIter=25*, *eps=1e-10*, ***kwargs*)

> Find the minimum distance between this curve (self) and a second curve passed in (inCurve)

>> **Parameters**

>>> **inCurve** [pySpline.curve objet] Other curve to use

>>> **nIter** [int] Maximum number of Newton iterations to perform.

>>> **eps** [float] Desired parameter tolerance.

>>> **s** [float] Initial guess for curve1 (this curve class)

>>> **t** [float] Initial guess for inCurve (curve passed in)

>> **Returns**

>>> **s** [float] Parametric position on curve1 (this class)

>>> **t** [float] Parametric position on curve2 (inCurve)

>>> **D** [float] Minimum distance between this curve and inCurve. It is equivalent to ||self(s) - inCurve(t)||_2.

projectCurveMultiSol(*inCurve*, *nIter=25*, *eps=1e-10*, ***kwargs*)

> Find the minimum distance between this curve (self) and a second curve passed in (inCurve). Try to find as many local solutions as possible

>> **Parameters**

>>> **inCurve** [pySpline.curve objet] Other curve to use

>>> **nIter** [int] Maximum number of Newton iterations to perform.

> > **eps** [float] Desired parameter tolerance.
> >
> > **s** [float] Initial guess for curve1 (this curve class)
> >
> > **t** [float] Initial guess for inCurve (curve passed in)
>
> > **Returns**
> >
> > > **s** [array] Parametric position(s) on curve1 (this class)
> > >
> > > **t** [float] Parametric position)s_ on curve2 (inCurve)
> > >
> > > **D** [float] Minimum distance(s) between this curve and inCurve. It is equivalent to ||self(s) - inCurve(t)||_2.

projectPoint(*x0*, *nIter=25*, *eps=1e-10*, ***kwargs*)

> Perform a point inversion algorithm. Attempt to find the closest parameter values to the given points x0.
>
> > **Parameters**
> >
> > > **x0** [array] A point or list of points in nDim space for which the minimum distance to the curve is sought.
> > >
> > > **nIter** [int] Maximum number of Newton iterations to perform.
> > >
> > > **eps** [float] Desired parameter tolerance.
> >
> > **Returns**
> >
> > > **s** [float or array] Solution to the point inversion. s are the parametric locations yielding the minimum distance to points x0
> > >
> > > **D** [float or array] Physical distances between the points and the curve. This is simply ||curve(s) - X0||_2.

recompute(*nIter*, *computeKnots=True*)

> Run iterations of Hoscheks Parameter Correction on the current curve
>
> > **Parameters**
> >
> > > **nIter** [int] The number of parameter correction iterations to run
> > >
> > > **computeKnots** [bool] Flag whether or not the knots should be recomputed

reverse()

> Reverse the direction of this curve

splitCurve(*u*)

> Split the curve at parametric position u. This uses the [*insertKnot()*](#) routine
>
> > **Parameters**
> >
> > > **u** [float] Parametric position to insert knot.
> >
> > **Returns**
> >
> > > **curve1** [pySpline curve object] Curve from s=[0, u]
> > >
> > > **curve2** [pySpline curve object] Curve from s=[u, 1]

Notes

curve1 and curve2 may be None if the parameter u is outside the range of (0, 1)

windowCurve(*uLow*, *uHigh*)

Compute the segment of the curve between the two parameter values

> **Parameters**
>
> > **uLow** [float] Lower bound for the clip
> >
> > **uHigh** [float] Upper bound for the clip

writeIGES_directory(*handle*, *Dcount*, *Pcount*, *twoD=False*)

Write the IGES file header information (Directory Entry Section) for this curve.

DO NOT MODIFY ANYTHING HERE UNLESS YOU KNOW **EXACTLY** WHAT YOU ARE DOING!

writeIGES_parameters(*handle*, *Pcount*, *counter*)

Write the IGES parameter information for this curve.

DO NOT MODIFY ANYTHING HERE UNLESS YOU KNOW **EXACTLY** WHAT YOU ARE DOING!

writeTecplot(*fileName*, *curve=True*, *coef=True*, *orig=True*)

Write the curve to a tecplot .dat file

> **Parameters**
>
> > **fileName** [str] File name for tecplot file. Should have .dat extension
> >
> > **curve** [bool] Flag to write discrete approximation of the actual curve
> >
> > **coef** [bool] Flag to write b-spline coefficients
> >
> > **orig** [bool] Flag to write original data (used for fitting) if it exists

SURFACE

class pyspline.pySurface.**Surface**(*recompute=True, **kwargs*)

> Create an instance of a b-spline surface. There are two ways to initialize the class

- **Creation**: Create an instance of the Surface class directly by supplying the required information: kwargs MUST contain the following information: `ku, kv, tu, tv, coef`.

- **LMS/Interpolation**: Create an instance of the Surface class by using an interpolating spline to given data points or a LMS spline. The following keyword argument information is required:

  1. `ku` and `kv` Spline Orders

  2. **X real array size (Nu, Nv, nDim) of data to fit. OR**

     1. `x` (2D) and `y` (2D) for 2D surface interpolation

     2. `x` (3D) and `y` (3D) and `z` (3) for 3D surface

  3. `u`, `v` real array of size (Nu, Nv). Optional

  4. `nCtlu`, `nCtlv` Specify number of control points. Only for LMS fitting.

Parameters

> **ku** [int] Spline order in u
>
> **kv** [int] Spline order in v
>
> **nCtlu** [int] Number of control points in u
>
> **nCtlv** [int] Number of control points in v
>
> **coef** [array, size (nCtlu, nCtl, nDim)] b-spline coefficient array.
>
> **tu** [array, size(nCtlu + ku)] knot array in u
>
> **tv** [array, size(nCtlv + kv)] knot array in v
>
> **X** [array, size (Nu, Nv, ndim)] Full data array to fit
>
> **x** [array, size (Nu, Nv)] Just x data to fit/interpolate
>
> **y** [array, size (Nu, Nv)] Just y data to fit/interpolate
>
> **u** [array, size (Nu, Nv)] Explict u parameters to use. Optional.
>
> **v** [array, size (Nu, Nv)] Explict v parameters to use. Optional.
>
> **scaledParams** [bool] default is to use u,v for parameterization. If true use u,v as well. If false, use U,V.
>
> **nIter** [int] Number of Hoscheks parater corrections to run

Notes

The orientation of the nodes, edges and faces is the same as the **bottom** surface as described in `Volume` documentation.

calcKnots()

    Determine the knots depending on if it is inerpolated or an LMS fit

calcParameterization()

    Compute a spatial parameterization

computeData(*recompute=False*)

    Compute discrete data that is used for the Tecplot Visualization as well as the data for doing the brute-force checks

        Parameters

            recompute [bool] If True, recompute the data even if it has already been computed.

getBasisPt(*u*, *v*, *vals*, *istart*, *colInd*, *lIndex*)

    This function should only be called from pyGeo The purpose is to compute the basis function for a u, v point and add it to pyGeo's global dPt/dCoef matrix. vals, row_ptr, col_ind is the CSR data and lIndex in the local -> global mapping for this surface

getBounds()

    Determine the extents of the surface

        Returns

            xMin [array of length 3] Lower corner of the bounding box

            xMax [array of length 3] Upper corner of the bounding box

getDerivative(*u*, *v*)

    Evaluate the first derivatvies of the spline surface

        Parameters

            u [float] Parametric u value

            v [float] Parametric v value

        Returns

            deriv [array size (2,3)] Spline derivative evaluation at u,vall points u,v. Shape depend on the input.

getOrigValueCorner(*corner*)

    Return the original data for he spline at the corner if it exists

        Parameters

            corner [int] Corner index 0<=corner<=3

        Returns

            value [float] Original value at corner

getOrigValuesEdge(*edge*)

    Return the endpoints and the mid-point value for a given edge.

        Parameters

            edge [int] Edge index 0<=edge<=3

Returns

> **startValue** [array size nDim] Original value at start of edge
>
> **midValue** [array size nDim] Original value at mid point of edge
>
> **endValue** [array size nDim] Original value at end of edge.

getSecondDerivative(*u*, *v*)

> Evaluate the second derivatvies of the spline surface

deriv = [(d^2)/(du^2) (d^2)/(dudv)] [(d^2)/(dudv) (d^2)/(dv^2)]

> **Parameters**
>
> > **u** [float] Parametric u value
> >
> > **v** [float] Parametric v value
>
> **Returns**
>
> > **deriv** [array size (2,2,3)] Spline derivative evaluation at u,vall points u,v. Shape depend on the input.

getValue(*u*, *v*)

> Evaluate the spline surface at parametric positions u,v. This is the main function for spline evaluation.
>
> **Parameters**
>
> > **u** [float, array or matrix (rank 0, 1, or 2)] Parametric u values
> >
> > **v** [float, array or matrix (rank 0, 1, or 2)] Parametric v values
>
> **Returns**
>
> > **values** [varies] Spline evaluation at all points u,v. Shape depend on the input. If u,v are scalars, values is array of size nDim. If u,v are a 1D list, return is (N,nDim) etc.

getValueCorner(*corner*)

> Evaluate the spline spline at one of the four corners
>
> **Parameters**
>
> > **corner** [int] Corner index 0<=corner<=3
>
> **Returns**
>
> > **value** [float] Spline evaluated at corner

getValueEdge(*edge*, *s*)

> Evaluate the spline at parametric distance s along edge 'edge'
>
> **Parameters**
>
> > **edge** [int] Edge index 0<=edge<=3
> >
> > **s** [float or array] Parameter values to evaluate
>
> **Returns**
>
> > **values** [array size (nDim) or array of size (N,nDim)] Requested spline evaluated values

insertKnot(*direction*, *s*, *r*)

    Insert a knot into the surface along either u or v.

        Parameters

            direction [str] Parameteric direction to insert. Either 'u' or 'v'.

            s [float] Parametric position along 'direction' to insert

            r [int] Desired number of times to insert.

        Returns

            r [int] The **actual** number of times the knot was inserted.

projectCurve(*inCurve*, *nIter=25*, *eps=1e-10*, ***kwargs*)

    Find the minimum distance between this surface (self) and a curve (inCurve).

        Parameters

            inCurve [pySpline.curve objet] Curve to use

            nIter [int] Maximum number of Newton iterations to perform.

            eps [float] Desired parameter tolerance.

            s [float] Initial solution guess for curve

            u [float] Initial solution guess for parametric u position

            v [float] Initial solution guess for parametric v position

        Returns

            u [float] Surface parameter u yielding min distance to point x0

            u [float] Surface parameter v yielding min distance to point x0

            s [float] Parametric position on curve yielding min distance to point x0

            D [float] Minimum distance between this surface and curve. is equivalent to ||surface(u,v) - curve(s)||_2.

projectPoint(*x0*, *nIter=25*, *eps=1e-10*, ***kwargs*)

    Perform a point inversion algorithm. Attempt to find the closest parameter values (u,v) to the given points x0.

        Parameters

            x0 [array] A point or list of points in nDim space for which the minimum distance to the curve is sought.

            nIter [int] Maximum number of Newton iterations to perform.

            eps [float] Desired parameter tolerance.

            u [float or array of length x0] Optional initial guess for u parameter

            v [float or array of length x0] Optional initial guess for v parameter

        Returns

            u [float or array] Solution to the point inversion. u are the u-parametric locations yielding the minimum distance to points x0

            v [float or array] Solution to the point inversion. v are the v-parametric locations yielding the minimum distance to points x0

D [float or array] Physical distances between the points and the curve. This is simply ||surface(u,v) - X0||_2.

recompute()

Recompute the surface if any data has been modified

setEdgeCurves()

Create curve spline objects for each of the edges

splitSurface(*direction*, *s*)

Split surface into two surfaces at parametric location s

> **Parameters**
>
> > **direction** [str] Parameteric direction along which to split. Either 'u' or 'v'.
> >
> > **s** [float] Parametric position along 'direction' to split
>
> **Returns**
>
> > **surf1** [pySpline.surface] Lower part of the surface
> >
> > **surf2** [pySpline.surface] Upper part of the surface

windowSurface(*uvLow*, *uvHigh*)

Create a surface that is windowed by the rectangular parametric range defined by uvLow and uvHigh.

> **Parameters**
>
> > **uvLow** [list or array of length 2] (u,v) coordinates at the bottom left corner of the parameteric box
> >
> > **uvHigh** [list or array of length 2] (u,v) coordinates at the top left corner of the parameteric box
>
> **Returns**
>
> > **surf** [pySpline.surface] A new surface defined only on the interior of uvLow -> uvHigh

writeDirections(*handle*, *isurf*)

Write out and indication of the surface direction

writeIGES_directory(*handle*, *Dcount*, *Pcount*)

Write the IGES file header information (Directory Entry Section) for this surface

writeIGES_parameters(*handle*, *Pcount*, *counter*)

Write the IGES parameter information for this surface

writeTecplot(*fileName*, *surf=True*, *coef=True*, *orig=True*, *directions=False*)

Write the surface to a tecplot .dat file

> **Parameters**
>
> > **fileName** [str] File name for tecplot file. Should have .dat extension
> >
> > **surf** [bool] Flag to write discrete approximation of the actual surface
> >
> > **coef: bool** Flag to write b-spline coefficients
> >
> > **orig** [bool] Flag to write original data (used for fitting) if it exists
> >
> > **directions** [bool] Flag to write surface direction visualization

writeTin(*handle*)

Write the pySpline surface to an open handle in .tin format

VOLUME

class `pyspline.pyVolume.`**Volume**(*recompute=True*, ***kwargs*)

Create an instance of a b-spline surface. There are two ways to initialize the class

- **Creation**: Create an instance of the Volume class directly by supplying the required information: kwargs MUST contain the following information: `ku, kv, kw, tu, tv, tw, coef`.

- **LMS/Interpolation**: Create an instance of the Volume class by using an interpolating spline to given data points or a LMS spline. The following keyword argument information is required:

  1. `ku` and `kv` and `kw` Spline Orders

  2. **X real array size (Nu, Nv, Nw, nDim) of data to fit. OR**

     1. `x` (3D) and `y` (3D) `z` (3D) 3D volume interpolation/fitting

  3. `u`, `v`, `w` real array of size (Nu, Nv, Nw). Optional

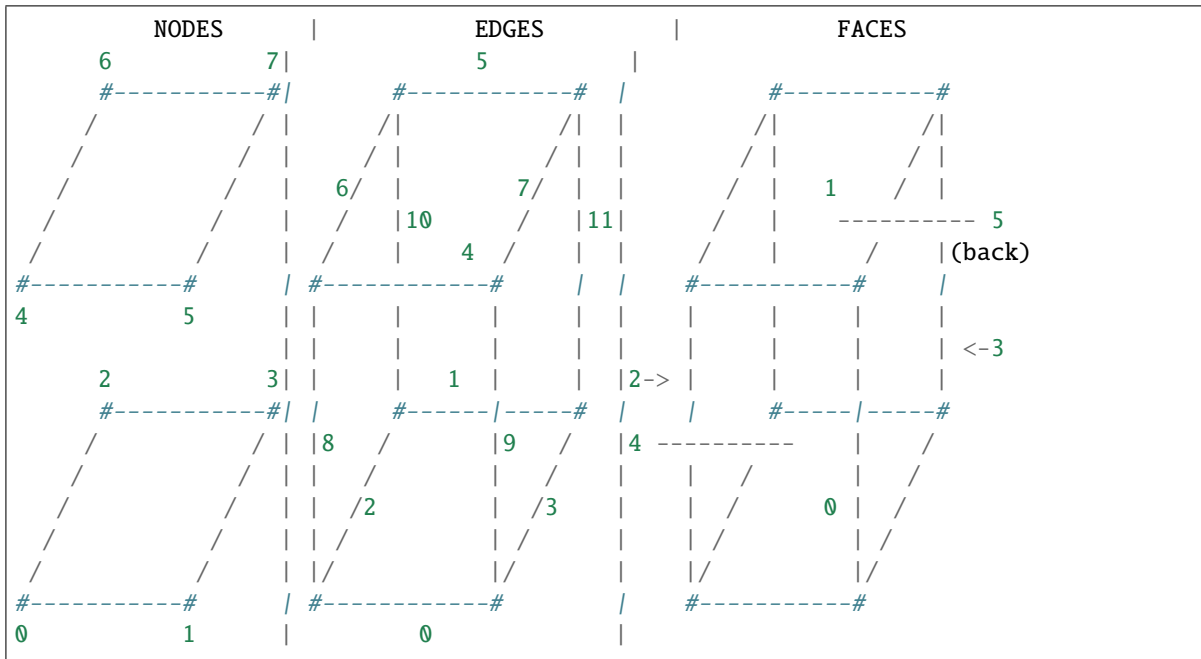 4. `nCtlu`, `nCtlv`, `nCtlw`. Specify number of control points. Only for LMS fitting.

Parameters

  ku [int] Spline order in u

  kv [int] Spline order in v

  kw [int] Spline order in w

  nCtlu [int] Number of control points in u

  nCtlv [int] Number of control points in u

  nCtlw [int] Number of control points in w

  coef [array, size (nCtlu, nCtl, nDim)] b-spline coefficient array.

  tu [array, size(nCtlu + ku)] knot array in u

  tv [array, size(nCtlv + kv)] knot array in v

  tw [array, size(nCtlw + kw)] knot array in w

  X [array, size (Nu, Nv, Nw, ndim)] Full data array to fit

  x [array, size (Nu, Nv)] Just x data to fit/interpolate

  y [array, size (Nu, Nv, Nw)] Just y data to fit/interpolate

  z [array, size (Nu, Nv, Nw)] Just z data to fit/interpolate

  u [array, size (Nu, Nv, Nw)] Explict u parameters to use. Optional.

  v [array, size (Nu, Nv, Nw)] Explict v parameters to use. Optional.

w [array, size (Nu, Nv, Nw)] Explict w parameters to use. Optional.

nIter [int] Number of Hoscheks parater corrections to run

recompute [bool] Specifies whether the actual fitting is completed.

Notes

The orientation of the nodes, edges and faces for the volumes is given below:

```
          NODES        |         EDGES         |          FACES
      6           7|           5           |
      #----------#|         #-----------#   |            #----------#
     /          / |        /|          /|   |           /|         /|
    /          /  |       / |         / |   |          / |        / |
   /          /   |    6 /  |      7 /  |   |         /  |   1   /  |
  /          /    |     /  |10      /  |11|  |        /  |       ---------- 5
 /          /     |    /   |    4  /   |  |  |       /   |      /     |(back)
#----------#      |   #-----------#   |  |  |      #----------#      |
4          5      |   | |         |   |  |  |      |    |      |      |
                  |   | |         |   |  |  |      |    |      |      | <-3
   2          3|  |   | |   1     |   |  |2->|     |    |      |      |
   #----------#|  |   #------|-----#   |  |  |     #-----|-----#
  /          / |  |8 /       9    /    |4 ---------      |     /
 /          /  |  | |  /           |   /    |    |    |  /     |    /
/          /   |  | | /2           |  /3    |    | |  /     0  |   /
```

calcKnots()

Determine the knots depending on if it is inerpolated or an LMS fit

calcParameterization()

Compute distance based parametrization. Use the fortran function for this

computeData(*recompute=False*)

Compute discrete data that is used for the Tecplot Visualization as well as the data for doing the brute-force checks

> **Parameters**
>
> > **recompute** [bool] If True, recompute the data even if it has already been computed.

getBasisPt(*u, v, w, vals, istart, colInd, lIndex*)

This function should only be called from pyBlock The purpose is to compute the basis function for a u, v, w point and add it to pyBlocks's global dPt/dCoef matrix. vals, rowPtr, colInd is the CSR data and lIndex in the local -> global mapping for this volume

getBounds()

Determine the extents of the volume

> **Returns**
>
> > **xMin** [array of length 3] Lower corner of the bounding box
> >
> > **xMax** [array of length 3] Upper corner of the bounding box

getMidPointEdge(*edge*)

> Get the midpoint of the edge using the original data.
>
> > **Parameters**
> >
> > > **edge** [int] Edge index. Must be 0<edge<11.
> >
> > **Returns**
> >
> > > **midpoint** [array of length nDim] Mid point of edge

getMidPointFace(*face*)

> Get the midpoint of the face using the original data.
>
> > **Parameters**
> >
> > > **face** [int] Face index. Must be 0, 1, 2, 3, 4 or 5
> >
> > **Returns**
> >
> > > **midpoint** [array of length nDim] Mid point of face

getOrigValueCorner(*corner*)

> Get the value of the original spline data on the corner if it exists
>
> > **Parameters**
> >
> > > **corner** [int] Index of corner, 0<=corner<=7
> >
> > **Returns**
> >
> > > **value** [float] Original data on corner.

getOrigValuesFace(*face*)

> For a given face index, face, return the 4 corners and the values of the midpoints of the 4 edges on that face.
>
> > **Parameters**
> >
> > > **face** [int] Index of face, 0<=face<=5
> >
> > **Returns**
> >
> > > **coords** [array of size (8, ndim)] The first 4 entries are the corner, and the last 4 are the midpoints.

getValue(*u*, *v*, *w*)

> Get the value at the volume points(s) u, v, w. This is the main evaluation routine for the volume object.
>
> > **Parameters**
> >
> > > **u** [scalar, vector or matrix or tensor of values] u position
> > >
> > > **v** [scalar, vector or matrix or tensor of values] v position
> > >
> > > **w** [scalar, vector or matrix or tensor of values] w position
> >
> > **Returns**
> >
> > > **values** [scalar, vector, matrix or tensor of values] The spline evaluation at (u, v, w)

getValueCorner(*corner*)

> Get the value of the volume spline on the corner.
>
> > **Parameters**
> >
> > > **corner** [int] Index of corner, 0<=corner<=7

Returns

value [float] Volume spline evaluation at corner.

getValueEdge(*edge*, *s*)

Get the value at the volume points(s) u, v, w

Parameters

edge [int] Index of edge. Must be between 0 and 11.

s [float or array] Parameter position(s) along edge to evaluate.

Returns

values [array] Array of values evaluated along edge.

insertKnot(*direction*, *s*, *r*)

Insert a knot into the volume along either u, v, w:

Parameters

direction [str] Parameteric direction to insert. Either 'u', 'v', or 'w'

s [float] Parametric position along 'direction' to insert

r [int] Desired number of times to insert.

Returns

r [int] The **actual** number of times the knot was inserted.

projectPoint(*x0*, *nIter=25*, *eps=1e-10*, ***kwargs*)

Project a point x0 or points x0 onto the volume and return parametric positions

Parameters

x0 [array of length 3 or array of size (N, 3)] Points to embed in the volume. If the points do not **actually** lie in the volume, the closest point is returned

nIter [int] Maximum number of Newton iterations to perform

eps [float] Tolerance for the Newton iteration

u [float or array of len(X0)] Optional initial guess for u position.

v [float or array of len(X0)] Optional initial guess for v position.

w [float or array of len(X0)] Optional initial guess for w position.

Returns

u [float or array of length N] u parametric position of closest point

v [float or array of length N] v parametric position of closest point

w [float or array of length N] w parametric position of closest point

D [float or array of length N] Distance between projected point and closest point If the points are 'inside' the volume, D should be less than eps.

recompute()

Recompute the volume if any driving data has been modified

setEdgeCurves()

Create edge spline objects for each edge

setFaceSurfaces()

> Create face spline objects for each of the faces

writeTecplot(*fileName*, *vols=True*, *coef=True*, *orig=False*)

> Write the volume to a tecplot data file.
>
> > **Parameters**
> >
> > > **fileName** [str] Tecplot filename. Should end in .dat
> > >
> > > **vols** [bool] Flag specifiying whether the interpolated volume should be used. This is usually True if you want to get an approximation of the entire volume.
> > >
> > > **coef** [bool] Flag specifiying if the control points are to be plotted
> > >
> > > **orig** [bool] Flag specifiying if original data (used for fitting) is to be included. If on original data exists, this argument is ignored.

UTILS

exception pyspline.utils.**Error**(*message*)

> Format the error message in a box to make it clear this was a explicitly raised exception.

pyspline.utils.**bilinearSurface**(**args*)

> This is short-cut function to create a bilinear surface.
>
> Args can contain:
>
> 1. `x` array of size(4, 3). The four corners of the array arranged in the coordinate direction orientation:

```
2          3
+---------+
|         |
|         |
|         |
+---------+
0          1
```

> 2. `p1`, `p2`, `p3`, `p4`. Individual corner points in CCW Ordering:

```
3          2
+---------+
|         |
|         |
|         |
+---------+
0          1
```

pyspline.utils.**checkInput**(*inputVal*, *inputName*, *dataType*, *dataRank*, *dataShape=None*)

> This is a generic function to check the data type and sizes of inputs in functions where the user must supply proper values. Since Python does not do type checking on Inputs, this is required
>
> **Parameters**
>
> > **input** [int, float, or complex] The input argument to check
> >
> > **inputName** [str] The name of the variable, so it can be identified in an Error message
> >
> > **dataType** [str] Numpy string dtype code
> >
> > **dataRank** [int] Desired rank. 0 for scalar, 1 for vector 2 for matrix etc
> >
> > **dataShape** [tuple] The required shape of the data. Scalar is denoted by () Vector is denoted by (n,) where n is the shape Matrix is denoted by (n, m) where n and m are rows/columns
>
> **Returns**

> **output** [various] The input transformed to the correct type and guaranteed to the desired size and shape.

pyspline.utils.**closeTecplot**(*f*)

> Close Tecplot file opened with openTecplot()

pyspline.utils.**line**(**args*, ***kwargs*)

> This is a short cut function to create a line as a pySpline Curve object.
>
> Args can contain: (pick one)
>
> 1. `X` array of size(2, ndim). The two end points
>
> 2. `x1`, `x2` The two end points (each of size ndim)
>
> 3. `x`, dir=direction. A point and a displacement vector
>
> 4. `x1`, dir=direction, length=length. As 3. but with a specific length

pyspline.utils.**line_plane**(*ia*, *vc*, *p0*, *v1*, *v2*)

> Check a line against multiple planes. Solve for the scalars α, β, γ such that
>
> $$i_a + \alpha \times v_c = p_0 + \beta \times v_1 + \gamma \times v_2$$
>
> $$i_a - p_0 = \begin{bmatrix} -v_c & v_1 & v_2 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$
>
> $$\alpha \geq 0 : \text{The point lies above the initial point}$$
> $$\alpha < 0 : \text{The point lies below the initial point}$$
>
> The domain of the triangle is defined by
>
> $$\beta + \gamma = 1$$
>
> and
>
> $$0 < \beta, \gamma < 1$$
>
> > **Parameters**
> >
> > > **ia** [ndarray[3]] initial point
> > >
> > > **vc** [ndarray[3]] search vector from initial point
> > >
> > > **p0** [ndarray[3, n]] vector to triangle origins
> > >
> > > **v1** [ndarray[3, n]] vector along first triangle direction
> > >
> > > **v2** [ndarray[3, n]] vector along second triangle direction
> >
> > **Returns**
> >
> > > **sol** [real ndarray[6, n]] Solution vector—parametric positions + physical coordinates
> > >
> > > **nSol** [int] Number of solutions
> > >
> > > **pid** [int ndarray[n]]

pyspline.utils.**openTecplot**(*fileName*, *ndim*)

> A Generic function to open a Tecplot file to write spatial data.
>
> > **Parameters**
> >
> > > **fileName** [str] Tecplot filename. Should have a .dat extension.

ndim [int] Number of spatial dimensions. Must be 1, 2 or 3.

> **Returns**
>
> > **f** [file handle] Open file handle

pyspline.utils.**plane_line**(*ia*, *vc*, *p0*, *v1*, *v2*)

> Check a plane against multiple lines
>
> > **Parameters**
> >
> > > **ia** [ndarray[3, n]] initial point
> > >
> > > **vc** [ndarray[3, n]] search vector from initial point
> > >
> > > **p0** [ndarray[3]] vector to triangle origins
> > >
> > > **v1** [ndarray[3]] vector along first triangle direction
> > >
> > > **v2** [ndarray[3]] vector along second triangle direction
> >
> > **Returns**
> >
> > > **sol** [ndarray[6, n]] Solution vector - parametric positions + physical coordiantes
> > >
> > > **nSol** [int] Number of solutions

pyspline.utils.**searchQuads**(*pts*, *conn*, *searchPts*)

> This routine searches for the closest point on a set of quads for each searchPt. An ADT tree is built and used for the search and subsequently destroyed.
>
> > **Parameters**
> >
> > > **pts** [ndarray[3, nPts]] points defining the quad elements
> > >
> > > **conn** [ndarray[4, nConn]] local connectivity of the quad elements
> > >
> > > **searchPts** [ndarray[3, nSearchPts]] set of points to search for
> >
> > **Returns**
> >
> > > **faceID** [ndarray[nSearchPts]] index of the quad elements, one for each search point
> > >
> > > **uv** [ndarray[2, nSearchPts]] parametric u and v weights of the projected point on the closest quad

pyspline.utils.**tfi2d**(*e0*, *e1*, *e2*, *e3*)

> Perform a simple 2D transfinite interpolation in 3D.
>
> > **Parameters**
> >
> > > **e0** [ndarray[3, Nu]] coordinates along 0th edge
> > >
> > > **e1** [ndarray[3, Nu]] coordinates along 1st edge
> > >
> > > **e2** [ndarray[3, Nv]] coordinates along 2nd edge
> > >
> > > **e3** [ndarray[3, Nv]] coordinates along 3rd edge
> >
> > **Returns**
> >
> > > **X** [ndarray[3 x Nu x Nv]] evaluated points

pyspline.utils.**trilinearVolume**(**args*)

> This is a short-cut function to create a trilinear b-spline volume. It can be created with **x OR** with xmin and xmax.
>
> > **Parameters**

>
> **x** [array of size (2, 2, 2, 3)] Coordinates of the corners of the box.
>
> **xmin** [array of size (3)] The extreme lower corner of the box
>
> **xmax** [array of size (3)] The extreme upper corner of the box. In this case, by construction, the box will be coordinate axis aligned.

pyspline.utils.**writeTecplot1D**(*handle*, *name*, *data*, *solutionTime=None*)

> A Generic function to write a 1D data zone to a tecplot file. Parameters ———- handle : file handle
>
>> Open file handle
>
> **name** [str] Name of the zone to use
>
> **data** [array of size (N, ndim)] 1D array of data to write to file
>
> **SolutionTime** [float] Solution time to write to the file. This could be a fictitious time to make visualization easier in tecplot.

pyspline.utils.**writeTecplot2D**(*handle*, *name*, *data*, *solutionTime=None*)

> A Generic function to write a 2D data zone to a tecplot file. Parameters ———- handle : file handle
>
>> Open file handle
>
> **name** [str] Name of the zone to use
>
> **data** [2D np array of size (nx, ny, ndim)] 2D array of data to write to file
>
> **SolutionTime** [float] Solution time to write to the file. This could be a fictitious time to make visualization easier in tecplot.

pyspline.utils.**writeTecplot3D**(*handle*, *name*, *data*, *solutionTime=None*)

> A Generic function to write a 3D data zone to a tecplot file. Parameters ———- handle : file handle
>
>> Open file handle
>
> **name** [str] Name of the zone to use
>
> **data** [3D np array of size (nx, ny, nz, ndim)] 3D array of data to write to file
>
> **SolutionTime** [float] Solution time to write to the file. This could be a fictitious time to make visualization easier in tecplot.

PYTHON MODULE INDEX

p

projectCurveMultiSol() (*pyspline.pyCurve.Curve method*), 13
projectPoint() (*pyspline.pyCurve.Curve method*), 14
projectPoint() (*pyspline.pySurface.Surface method*), 20
projectPoint() (*pyspline.pyVolume.Volume method*), 26
pyspline.utils
    module, 29

R

recompute() (*pyspline.pyCurve.Curve method*), 14
recompute() (*pyspline.pySurface.Surface method*), 21
recompute() (*pyspline.pyVolume.Volume method*), 26
reverse() (*pyspline.pyCurve.Curve method*), 14

S

searchQuads() (*in module pyspline.utils*), 31
setEdgeCurves() (*pyspline.pySurface.Surface method*), 21
setEdgeCurves() (*pyspline.pyVolume.Volume method*), 26
setFaceSurfaces() (*pyspline.pyVolume.Volume method*), 26
splitCurve() (*pyspline.pyCurve.Curve method*), 14
splitSurface() (*pyspline.pySurface.Surface method*), 21
Surface (*class in pyspline.pySurface*), 17

T

tfi2d() (*in module pyspline.utils*), 31
trilinearVolume() (*in module pyspline.utils*), 31

V

Volume (*class in pyspline.pyVolume*), 23

W

windowCurve() (*pyspline.pyCurve.Curve method*), 15
windowSurface() (*pyspline.pySurface.Surface method*), 21
writeDirections() (*pyspline.pySurface.Surface method*), 21
writeIGES_directory() (*pyspline.pyCurve.Curve method*), 15
writeIGES_directory() (*pyspline.pySurface.Surface method*), 21
writeIGES_parameters() (*pyspline.pyCurve.Curve method*), 15
writeIGES_parameters() (*pyspline.pySurface.Surface method*), 21
writeTecplot() (*pyspline.pyCurve.Curve method*), 15
writeTecplot() (*pyspline.pySurface.Surface method*), 21

writeTecplot() (*pyspline.pyVolume.Volume method*), 27
writeTecplot1D() (*in module pyspline.utils*), 32
writeTecplot2D() (*in module pyspline.utils*), 32
writeTecplot3D() (*in module pyspline.utils*), 32
writeTin() (*pyspline.pySurface.Surface method*), 21