

Quick Reference

Links

[Using the Spline Library](#)

[Functions by Category](#)

[Function Definitions](#)

[Glossary](#)

Spline Library Version: [1-0](#)

ETAC Programming Language Version: [1-1](#)

ETAC Interpreter Version: [4-0-6-ena](#)

Spline Library

Inclusion File: [etacSplineFunctions.etac](#)

Data Object Name: [etacSplineFunctions](#)

Number of Functions: [17](#)

Release Date: [1 December 2025](#)

Spline ETAC Script Library

1 December 2025

Copyright © **Victor Vella** (2025)
All rights reserved.



Other Related ETAC Documents

ETAC_Preliminaries.pdf	Preliminaries before using ETAC
ETACOverview.pdf	An Overview of ETAC
ETACProgLang(Official).pdf	The Official ETAC Programming Language
ETACVisualInteractionSystem.pdf	ETAC: Visual Interaction System
RunETAC.chm	Run ETAC Scripts Help
ETACWithCPP.pdf	ETAC: Interacting with C++
ETACCompiler.pdf	The ETAC Compiler
ETACCompiler.chm	ETAC Compiler Help
ETACErrorCodes.pdf	ETAC Compilation and Run-time Error Codes
FunctionsETACScriptLib.pdf	Functions ETAC Script Library
PathScriptFnts.pdf	Path ETAC Script Library
VISShapePathFnts.pdf	VIS Shape Path Functions
VISPointSetFnts.pdf	VIS Point Set Functions

Legal Information

ETAC and  (the ETAC logo) are unregistered trademarks (™) of Victor Vella for *computer software incorporating an implementation of a computer programming language*. There may be other owners of the “ETAC” trademark used for other purposes.

This document is copyright © by Victor Vella (2025). All rights reserved. Permission is hereby granted to make any number of exact electronic copies of this document without any remuneration whatsoever. Permission is also granted to make annotated electronic copies of this document for personal use only. Except for the permissions granted, and apart from any fair dealing as permitted under the relevant Copyright Act, no part of this document may be reproduced or transmitted in any form or by any means without the express permission of the author. The copyright of this document shall remain entirely with the original copyright holder.

The author of this document shall not be liable for any direct or indirect consequences arising with respect to the use of all or any part of the information in this document, even if such information is inaccurate or in error. The information in this document is subject to change without notice.

Contents

Quick Reference.....	i
Contents.....	iv
Document Conventions.....	v
1. Introduction.....	1
2. Features of an ETAC Script Library.....	1
3. Using an ETAC Script Library.....	1
4. Spline ETAC Script Library Reference.....	3
4.1 Functions by Category.....	4
4.2 Function Summary.....	4
4.3 Function Definitions.....	5
Bibliography.....	25
Glossary.....	26

Document Conventions

The following symbolic conventions are used in this document.

Symbol	Meaning
$\langle x \rangle$	separates x as a unit of information from the surrounding text.
$x \cdots$	middle ellipsis means zero, one, or more of the same kind as x .
$x y$	means that only x or y applies, but not both (could have more than two options).
$\dots$	ellipsis represents omitted text (as usual).
<i>text</i>	maroon coloured italic text is a link to the text's definition.
text	bold green text is a link into the document.
♦	indicates the end of a block of text.

Spline ETAC Script Library

This document is for version **1-0-4-ena** of the Spline ETAC Script Library for version **1-1** of the **ETAC™ Programming Language** implemented in RunETAC.exe and AppETAC.dll version **4-0-6-ena**. The source file of the library is in etacSplineFunctions.etac.

(Australian English)

1. Introduction

ETAC™ (pronounced: E-tack) is a syntactically simple but extremely versatile dictionary and stack based interpreted script programming language. The ETAC programming language is not based on any other programming language. Familiarity with the ETAC programming language is required to fully understand this document. See the document ETACOverview.pdf for an overview of the language, and also the document ETACProgLang(Official).pdf for the official definition of the language.

The Spline ETAC Script Library (referred to as the “Spline Library”) is an *ETAC script library* containing various functions involving the relationships among *Bezier paths*, *B-spline paths*, and *sample points*.

2. Features of an ETAC Script Library

An *ETAC script library* (ESL) is an *ETAC text script* file containing *ETAC function* definitions. The *ETAC functions* are constructed in such a way that the same function is allocated only once within a *main ETAC session*, even when the same ESL file is included more than once.

An ETAC programmer can arrange for an ESL so that only the desired *ETAC functions*, and their dependencies, are allocated within a *main ETAC session*. This feature is useful when only a few functions of an ESL containing a large number of *ETAC function* definitions are required in an ETAC program. By default, all of the *ETAC functions* in an ESL are included in an ETAC program.

The included functions of an ESL are typically allocated within a named *data object* having a name based on the name of the ESL.

3. Using an ETAC Script Library

An *ETAC script library* (ESL) exists in an *ETAC text script* file, and can be included in any *ETAC text script* as an inclusion file (using the `<:include>` pre-processor inclusion command). An ESL is typically included at the top of the *ETAC text script*. The processing of the ESL inclusion file causes a new named *data object* to be created containing the desired *ETAC function* allocations; the *data object* name is based on the name of the ESL. For example, the *data object* name into which the desired functions of the Spline Library are included is called `etacSplineFunctions`.

Each *ETAC function* within an ESL is identified by a pre-processor definition name beginning with the initials of the ESL name followed by the name of the particular function. For example, if an ESL function name is `xxxBSplineSPts`, then the corresponding pre-processor definition name would be `@XXX_BSPLINE_SPTS`, where `xxx` and `XXX` are the initials of the ESL name (for the Spline Library, `XXX` is `SF`, so the function name identifiers are `sfBSplineSPts` and `@SF_BSPLINE_SPTS`, respectively).

The pre-processor definition name of each *ETAC function* within an ESL is called the *ESL function identifier* (for example, @SF_BSPLINE_SPTS is an *ESL function identifier*). To explicitly include a particular ESL function within some *ETAC text script*, the corresponding ESL function identifier must be defined before the ESL inclusion file is included. For example, to include the sfBSplineSPTS function of the Spline Library into *ETAC text script*, <::define @SF_BSPLINE_SPTS> must be present before <::include "etacSplineFunctions.etac">. If the included function calls other ESL functions, then those other ESL functions are automatically included as well.

Including an ESL inclusion file without specifying the definition of any *ESL function identifier* of that ESL results in all of the functions in that ESL being included. For example, specifying only <::include "etacSplineFunctions.etac"> without defining any of the *ESL function identifiers* of the Spline Library beforehand (in the same *ETAC text script*) results in all of the *ETAC functions* within the Spline Library to be included into the *data object* etacSplineFunctions.

If an *ESL function identifier* is defined more than once for the same ESL inclusion file, then the corresponding *ETAC function* is allocated only once within the *main ETAC session*, even when the same ESL inclusion file is included more than once.

In summary, to use an *ETAC script library*, the following are required:

- The file name of the ESL, which must be included before any of the ESL functions are used.
- A list of *ESL function identifiers* (corresponding to the desired ESL functions), which must be defined before the inclusion of the ESL file. The identifiers are obtained from the ESL source file or documentation.
- The name of the ESL *data object* (obtained from the ESL source file or documentation), which is automatically created by the inclusion of the ESL.
- A variable used to access the ESL *data object*. The desired ESL functions are accessed via that variable, which is assigned from the return value of the global @NewData *ETAC function*.

The following example is an illustration of how to use the Spline Library in a main ETAC program.

Using the Spline Library

```
[*::10ETAC-SOURCE-V1::*]
[* ETAC main program file illustrating how to use the Spline Library. *]
::include "TACGlobal.PTAC" [* Required by the Spline Library. *]

...

::define @SF_BSPLINE_SPTS [* Allows allocation of the sfBSplineSPts() function. *]
::define @SF_FIT_UBSPLINE3 [* Allows allocation of the sfFitUBSpline3() function. *]
::define @SF_STD_BEZ_SPLINE [* Allows allocation of the sfStdBezSpline() function. *]
::include "etacSplineFunctions.etac" [* Spline Library inclusion file. *]
[* etacSplineFunctions.etac creates the etacSplineFunctions data object containing
the three functions. *]

...

start_local; [* Create a local dictionary for the main program. *]
[* Other program variables allocated here. *]

    [* Allocate 'SFnts' variable for accessing ESL functions. *]
    SFnts :- @NewData("etacSplineFunctions");

    [* PROGRAM *]
    ...

    Seq := SFnts.sfBSplineSPts([...]); [* Call to ESL function. *]

    ...

end_local;
```

In the example above, three *ETAC functions*, `sfBSplineSPts()`, `sfFitUBSpline3()`, and `sfStdBezSpline()`, defined in the Spline Library are to be used in the program (the three functions are for illustrative purposes only). The corresponding *ESL function identifiers* of those functions are defined (via `::define`) so that the three functions are automatically allocated in the `etacSplineFunctions` *data object* when the Spline Library inclusion file, `etacSplineFunctions.etac`, is included in the program. The three *ESL function identifiers* must be defined before the inclusion of the Spline Library inclusion file. If no *ESL function identifiers* were defined, then all the *ETAC functions* of the Spline Library would have been allocated in the `etacSplineFunctions` *data object*, even though only some of those functions are used in the program. Those three functions will not be allocated again within the `etacSplineFunctions` *data object*, even if they were to be specified for inclusion in another *ETAC session*.

Somewhere in the program, after the inclusion of the Spline Library, a replicate of the `etacSplineFunctions` *data object* is created by the global `@NewData` function, and allocated to a variable, `SFnts`, to access the *ETAC functions* within the *data object*. The `etacSplineFunctions` *data object* is replicated (via `@NewData`) rather than used directly (via `@Data`) so as to keep the original *ETAC functions* within the *data object* unchanged; *ETAC functions* within an ESL may contain initialised persistent local data which needs to remain as originally initialised. Using an ESL *data object* directly may cause undesirable changes in any persistent local data within that *data object*.

4. Spline ETAC Script Library Reference

This section contains information about the Spline Library. The Spline Library exists in the `etacSplineFunctions.etac` inclusion file. When that file is included (via `::include`) in *ETAC text script*, a *data object* named `etacSplineFunctions` is automatically created containing the desired functions. Those functions can then be assessed via a replicate of that *data object*.

4.1 Functions by Category

The following is a list of the Spline Library functions organised by category.

Bezier Path to Sample Points

`sfBezSplineSPts()`

B-spline Path to Sample Points

`sfBSplineSPts()` ♦ `sfUBSpline3SPts()`

B-spline Path to Bezier Path

`sfBSpline3ToBez()` ♦ `sfUBSpline3ToBez()`

Sample Points to Bezier Path

`sfFitBez()` ♦ `sfFitExactBez()` ♦ `sfFitExactQBez()` ♦ `sfStdBezSpline()`

Sample Points to B-spline Path

`sfFitUBSpline3()` ♦ `sfFitUBSpline3C()`

Other

`sfBezCurvePoint()` ♦ `sfBSplineBasisFnt()` ♦ `sfBSplineBasisSPts()` ♦
`sfRepeatPoints()` ♦ `sfSubBezCPts()` ♦ `sfUBS3BlendFnts()`

4.2 Function Summary

The table below contains an alphabetical list of the Spline Library functions.

Function Summary for the Spline Library

Function	Description
<code>sfBezCurvePoint()</code>	Returns a point on a Bezier curve.
<code>sfBezSplineSPts()</code>	Creates <i>sample points</i> on a Bezier spline.
<code>sfBSpline3ToBez()</code>	Converts a cubic B-spline to a <i>Bezier path</i> .
<code>sfBSplineBasisFnt()</code>	Returns the value on a B-spline basis function curve.
<code>sfBSplineBasisSPts()</code>	Creates <i>sample points</i> on a B-spline basis function curve.
<code>sfBSplineSPts()</code>	Creates <i>sample points</i> on a B-spline.
<code>sfFitBez()</code>	Converts <i>sample points</i> to a <i>Bezier path</i> approximating an interpolated uniform cubic B-spline.
<code>sfFitExactBez()</code>	Returns the <i>Bezier path</i> of a cubic spline that interpolates given <i>sample points</i> .
<code>sfFitExactQBez()</code>	Returns the <i>Bezier path</i> of a quadratic spline that interpolates given <i>sample points</i> .
<code>sfFitUBSpline3()</code>	Converts <i>sample points</i> to a <i>B-spline path</i> approximating an interpolated uniform cubic B-spline.
<code>sfFitUBSpline3C()</code>	Converts <i>sample points</i> to a closed <i>B-spline path</i> approximating an interpolated uniform cubic B-spline.
<code>sfRepeatPoints()</code>	Repeats a point at a position within a path a number of times.
<code>sfStdBezSpline()</code>	Returns the <i>Bezier path</i> of a standard Bezier spline from <i>sample points</i> .
<code>sfSubBezCPts()</code>	Returns the four control <i>points</i> of a sub-Bezier curve.
<code>sfUBS3BlendFnts()</code>	Creates “blending functions” for a uniform cubic B-spline.
<code>sfUBSpline3SPts()</code>	Creates <i>sample points</i> on a uniform cubic B-spline.
<code>sfUBSpline3ToBez()</code>	Converts a uniform cubic B-spline to a <i>Bezier path</i> .

4.3 Function Definitions

The following boxes contain a description of all the Spline Library functions. The functions need to be called in the context of the `etacSplineFunctions` *data object*. The corresponding *ESL function identifier* is shown at the top right of each box.

sfBezCurvePoint()			@SF_BEZ_CURVE_POINT
sfBezCurvePoint (<i>pP0 pP1 pP2 pP3 pT</i>) → <i>x-val y-val</i>			
<i>pP0</i>	A decimal sequence stack object.		
<i>pP1</i>	A decimal sequence stack object.		
<i>pP2</i>	A decimal sequence stack object.		
<i>pP3</i>	A decimal sequence stack object.		
<i>pT</i>	A decimal stack object.		
<i>x-val</i>	A decimal stack object.		
<i>y-val</i>	A decimal stack object.		

Details

Returns a point (*x-val*, *y-val*) on a cubic Bezier curve specified by four control points (*pP0* to *pP3*) at a given value (*pT*) between 0.0 and 1.0.

pP0 is the beginning terminal control *point* of the Bezier curve where $t = 0.0$.

pP1 is the first control *point* of the Bezier curve.

pP2 is the second control *point* of the Bezier curve.

pP3 is the end terminal control *point* of the Bezier curve where $t = 1.0$.

pT is the t parameter of the Bezier curve. The value of *pT* typically ranges from 0.0 to 1.0.

x-val is the X-coordinate of the value on the Bezier curve at *pT*.

y-val is the Y-coordinate of the value on the Bezier curve at *pT*.

The value of *pT* may be outside the range 0.0 to 1.0. However, in that case, the returned point (*x-val*, *y-val*) will not be on the Bezier curve along the path from *pP0* to *pP3*.

Other Information

sfBezSplineSPts() ♦

sfBezSplineSPts()			@SF_BEZ_SPLINE_SPTS
sfBezSplineSPts (<i>pDist pCPts</i>) → <i>points</i>			
<i>pDist</i>	A decimal stack object.		
<i>pCPts</i>	A decimal sequence stack object.		
<i>points</i>	A decimal sequence stack object.		

Details

Creates *sample points* (*points*) on a cubic Bezier spline defined by a *Bezier path* (*pCPts*).

$pDist$ is the approximate distance between the points of the returned *sample points*. The value must be greater than zero.

$pCPts$ is a *Bezier path* defining a Bezier spline on which *sample points* are to be returned.

points are *sample points* on the Bezier spline defined by $pCPts$.

Other Information

sfBezCurvePoint() ♦

sfBSpline3ToBez()

sfBSpline3ToBez ($pCPts$ $pKnots$) $\rightarrow$ <i>bez-path</i>		@SF_BSPLINE3_TO_BEZ
$pCPts$	A decimal sequence stack object.	
$pKnots$	A decimal sequence stack object.	
<i>bez-path</i>	A decimal sequence stack object.	

Details

Converts a cubic B-spline defined by a *B-spline path* ($pCPts$) and a *knot sequence* ($pKnots$) to a *Bezier path* (*bez-path*).

$pCPts$ is the *B-spline path* defining a cubic B-spline.

$pKnots$ is a *knot sequence* as part of the B-spline definition.

bez-path is the *Bezier path* of a Bezier spline identical to the B-spline.

This function implements the deBoor algorithm (polar form) for calculating cubic B-spline values for the given control points ($pCPts$). The returned *Bezier path* can be used to display the B-spline.

Other Information

sfUBSpline3ToBez() ♦

sfBSplineBasisFnt()

sfBSplineBasisFnt (pT pI $pDeg$ $pKnots$) $\rightarrow$ <i>value</i>		@SF_BSPLINE_BASIS_FNT
pT	A decimal stack object.	
pI	An integer stack object.	
$pDeg$	An integer stack object.	
$pKnots$	A decimal sequence stack object.	
<i>value</i>	A decimal stack object.	

Details

Returns the value (*value*) on a B-spline basis function curve, $B_{pI, pDeg}$, for a specified t (pT).

pT is the argument t for the basis function.

pI is the parameter i for the basis function.

$pDeg$ is the parameter d for the basis function.

$pKnots$ is the *knot sequence* used with the basis function.

value is the value of the B-spline basis function curve, $B_{i,d}(t)$.

The basis functions, $B_{i,d}$, for a d^{th} degree B-spline and a *knot sequence* $[t_{-d}, \dots, t_{n+1}]$ are defined by the Cox-deBoor recursive equations as follows.

$$B_{i,0}(t) = \begin{cases} 1 & [t_i \leq t < t_{i+1}] \\ 0 & [\text{otherwise}] \end{cases}$$

$$B_{i,d}(t) = \left(\frac{t - t_i}{t_{i+d} - t_i} \right) B_{i,d-1}(t) + \left(\frac{t_{i+d+1} - t}{t_{i+d+1} - t_{i+1}} \right) B_{i+1,d-1}(t)$$

$[t \in \mathbb{R}] [i = -d, \dots, (n+1); i \in \mathbb{Z}]$

Other public documentation needs to be consulted for more information on B-spline basis functions.

Using the basis functions defined above, a B-spline, $\mathbf{S}$, of degree d is defined as follows.

$$\mathbf{S}(t) = \sum_{j=0}^n B_{j-d,d}(t) \mathbf{P}_j$$

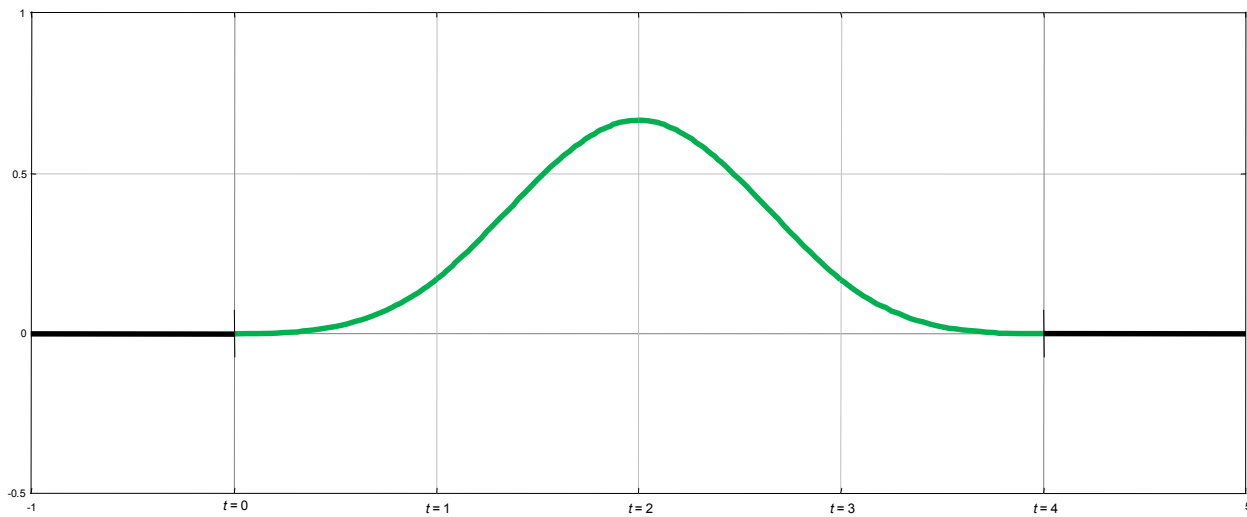
$[t_0 \leq t \leq t_{n+1-d}; t \in \mathbb{R}] [d \in \mathbb{Z}^+; n \geq d]$

where $\mathbf{P}_j$ is the j^{th} point of the *B-spline path* corresponding to the B-spline.

Note that the `sfBSplineBasisFnt()` function directly implements the specified basis function, and accordingly, is not efficient to use for calculating a large number of points on a B-spline.

Illustration

The following shows the graph of the basis function $B_{0,3}$ for a uniform cubic B-spline.



Other Information

`sfBSplineBasisSPts()` ♦

sfBSplineBasisSPts()

sfBSplineBasisSPts(<i>pDens pDeg pT1 pT2 pI pKnots</i>) → <i>points</i>	
@SF_BSPLINE_BASIS_SPTS	
<i>pDens</i>	An integer or decimal stack object.
<i>pDeg</i>	An integer stack object.
<i>pT1</i>	A decimal stack object.
<i>pT2</i>	A decimal stack object.
<i>pI</i>	A decimal stack object.
<i>pKnots</i>	A decimal sequence stack object.
<i>points</i>	A decimal sequence stack object.

Details

Creates *sample points* (*points*) on a B-spline basis function curve.

pDens indicates the density of the returned *sample points*.

pDeg is the parameter d for the basis function.

pT1 is the argument t_1 for the basis function.

pT2 is the argument t_2 for the basis function.

pI is the parameter i for the basis function.

pKnots is the *knot sequence* used with the basis function.

points are the *sample points* on the B-spline basis function curve.

This function creates *sample points* between t_1 and t_2 for a B-spline basis function curve, $B_{i,d}(t)$, for a d^{th} degree B-spline and a *knot sequence* $[t_{-d}, \dots, t_{n+1}]$. The function calls **sfBSplineBasisFnt()** repeatedly for the specified values of t in increments of $1/pDens$.

Additional Information

sfBSplineBasisFnt() ♦

sfBSplineSPts()

sfBSplineSPts(<i>pDens pDeg pCPts pKnots</i> <i>?</i>) → <i>points</i>	
@SF_BSPLINE_SPTS	
<i>pDens</i>	An integer or decimal stack object.
<i>pDeg</i>	An integer stack object.
<i>pCPts</i>	A decimal sequence stack object.
<i>pKnots</i>	A decimal sequence stack object.
<i>points</i>	A decimal sequence stack object.

Details

Creates *sample points* (*points*) on any B-spline defined by a *B-spline path* (*pCPts*), the degree of the B-spline (*pDeg*), and a *knot sequence* (*pKnots*).

pDens indicates the density of the returned *sample points*.

pDeg is the degree of the B-spline.

pCPts is the *B-spline path* for the B-spline.

pKnots is the *knot sequence* for the B-spline. If a NULL stack object (?) is used instead of *pKnots*, then a uniform B-spline is assumed.

points are the *sample points* on the B-spline defined by *pCPts*.

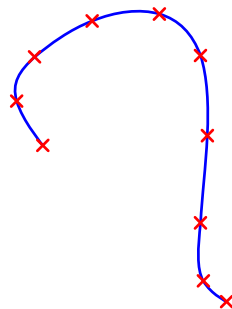
This is an efficient implementation of the deBoor algorithm for any B-spline. The algorithm returns a sample of points (*points*) on the specified B-spline with control points *pCPts* and of degree *pDeg*. *pDens* determines the density of the *sample points*. The higher the value of *pDens* the closer are the points in the *sample points*.

Example

The following illustration shows how the **sfBSplineSPts()** function can be used.

```
(1) BezPath := sfFitExactBez(? sfBSplineSPts(3 4 [1.0, 0.0, 3.0, 5.0,
-2.0, 5.9, 4.7, 9.3, 2.0, 1.9, 4.7, 3.5, 5.3, 4.0] ?));
```

In example (1), the *sample points* of a uniform B-spline of degree 4 are obtained and passed to the function **sfFitExactBez()** which returns a *Bezier path* corresponding to the B-spline. The *Bezier path* can be used directly to create a VIS graphics object representing the B-spline, as shown below.



The blue curve is the B-spline, and the red crosses indicate the *sample points* returned from the **sfBSplineSPts()** function. Note that the blue curve is an interpolation of the *sample points* and may not be perfectly identical to the specified B-spline.

Other Information

sfUBSpline3SPts() ♦

sfFitBez()

sfFitBez(<i>pPoints</i> <i>pFact</i> ?) → <i>bez-path</i>		@SF_FIT_BEZ
<i>pPoints</i>	A decimal sequence stack object.	
<i>pFact</i>	A decimal stack object.	
<i>bez-path</i>	A decimal sequence stack object.	

Details

Converts *sample points* (*pPoints*) to a *Bezier path* (*bez-path*) approximating an interpolated uniform cubic B-spline to a specified accuracy (*pFact*).

pPoints are given *sample points* to be approximately interpolated.

pFact indicates the accuracy to which the *sample points* are interpolated. This value can be negative or zero. A NULL stack object (?) in place of *pFact* typically gives the best fit (with a value of 1.353).

bez-path is the *Bezier path* of a Bezier spline that approximately interpolates the *sample points*.

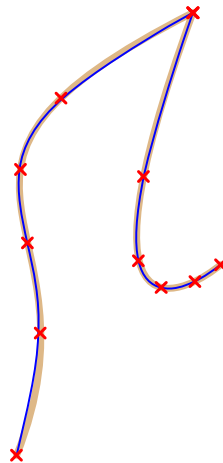
This function converts *sample points* (*pPoints*) to the control points (*bez-path*) of the Bezier spline equivalent to a uniform cubic B-spline that approximately interpolates the *sample points*. A value of 1.0 for *pFact* will result in the *sample points* being the control points of a uniform cubic B-spline identical to the Bezier spline. A smaller (positive) value of *pFact* than 1.0 will result in a smoother curve; a higher value will result in a looser curve.

Examples

The following illustrations show how the **sfFitBez()** function can be used.

```
(1) KnotSeq := [0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2];  
    CPts := [1.0, 0.0, 3.0, 5.0, -2.0, 5.9, 4.7, 9.3, 2.0, 1.9, 4.7,  
            3.5, 5.3, 4.0]; SPts := sfBSplineSPts(5 3 CPts KnotSeq);  
    BezPath := sfFitBez(SPts ?);
```

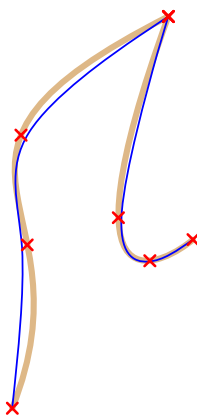
In example (1), a cubic B-spline is created for reference using the specified control points (CPts) and *knot sequence* (KnotSeq). Some *sample points* (SPts) are created on the B-spline via the **sfBSplineSPts()** function. The **sfFitBez()** function is then called on the *sample points* to approximately interpolate them. BezPath contains the *Bezier path* of a Bezier spline that ideally would be identical to the B-spline reference. However, the **sfFitBez()** function does not necessarily interpolate the *sample points* exactly (the **sfFitExactBez()** function does interpolate *sample points* exactly, but is less efficient). The diagram below illustrates the situation.



The thick brown curve is the cubic B-spline reference. The red crosses indicate the *sample points* (SPts) on that B-spline. The blue thin curve is the curve that approximately interpolates the *sample points* using the default value for *pFact*. It will be noticed that the blue curve does not exactly go through all the *sample points*, meaning that interpolation is approximate. However it is very close to being an exact interpolation. (The sharp corner on the curve is a result of a triplicate sample point due to the particular specification of the *knot sequence*.)

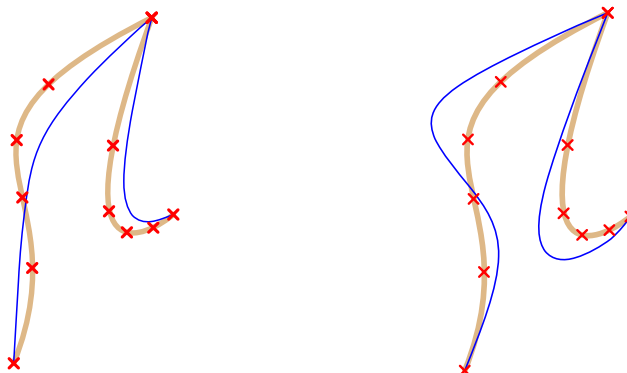
```
(2) KnotSeq := ...; CPts := ...; SPts := sfBSplineSPts(3 3 CPts KnotSeq);  
    BezPath := sfFitBez(SPts ?);
```

In example (2), the control points (CPts) and *knot sequence* (KnotSeq) are the same as for example (1). However, the number of *sample points* has been reduced. The diagram below shows the result; the interpolation is less accurate.



```
(3) KnotSeq := ...; CPts := ...; SPTS := sfBSplineSPTS(5 3 CPts KnotSeq);
    BezPath := sfFitBez(SPTS 0.0);
```

In example (3), the control points (CPts), *knot sequence* (KnotSeq), and number of *sample points* are the same as for example (1). However, the value of *pFact* is set to 0.0. The diagram below on the left shows the result; the approximation is smoother. The diagram on the right has a *pFact* value of 3.0. Negative values of *pFact* can be specified, but any value too far from the default (1.353) may produce ungainly results.



Other Information

sfFitExactBez() ▀ **sfFitUBSpline3()** ♦

sfFitExactBez()

sfFitExactBez (<i>pTPts</i> ? <i>pPoints</i>) → <i>bez-path</i>		@SF_FIT_EXACT_BEZ
<i>pTPts</i>	A decimal sequence stack object.	
<i>pSPTS</i>	A decimal sequence stack object.	
<i>bez-path</i>	A decimal sequence stack object.	

Details

Returns the *Bezier path* (*bez-path*) of a cubic spline that interpolates a sequence of *sample points* (*pPoints*).

pTPts is a TAC sequence that determines the start and end point tangents of the cubic spline. The sequence contains two end control points of the form: $[x_1, y_1, x_2, y_2]$. An optional fifth element (integer) of the sequence determines whether the resulting spline is closed, as described below. If a NULL stack object (?) is used instead of *pTPts*, then both control points are assumed to be anchored to the corresponding end of the spline.

pPoints are the given *sample points* which form the basis of interpolation.

bez-path is the *Bezier path* of a Bezier spline that interpolates the *sample points*. This implies that the Bezier spline will go through the *sample points*.

This function returns the Bezier control points (*bez-path*) of a cubic spline that interpolates the specified *sample points* (*pPoints*). The start and end point tangents of the cubic spline are determined by a sequence containing the two end control points (*pTPts*). An optional fifth element of the sequence has the following meaning:

- 0 The cubic spline is not closed. This is the default.
- 1 An internal copy of the *sample points* (*pPoints*) is closed in the following way before the cubic spline is determined. The first of the *sample points* is copied and appended to the last, and the second control point in *pTPts* is modified to maintain continuity $C^{(1)}$ at the junction.
- 2 The *Bezier path* of the cubic spline is closed in the following way before being returned as *bez-path*. The 2nd then the 1st control point in *pTPts*, followed by the first of the *sample points*, are copied and appended to the end of the *Bezier path*.

Any coordinate of the control points in *pTPts* can be anchored to either corresponding end point coordinate of the *sample points* (*pPoints*) by setting the appropriate coordinate in *pTPts* to a NULL stack object (?). For example, if *pTPts* contains [?, -3.5, 7.2, ?] and the *sample points* are [1.8, 2.9, ..., -6.7, 11.6], then *pTPts* is literally modified to be [1.8, -3.5, 7.2, 11.6]. If a NULL stack object (?) is used instead of *pTPts*, then *pTPts* is assumed to be [1.8, 2.9, -6.7, 11.6], ie: the first and last points of the *sample points*.

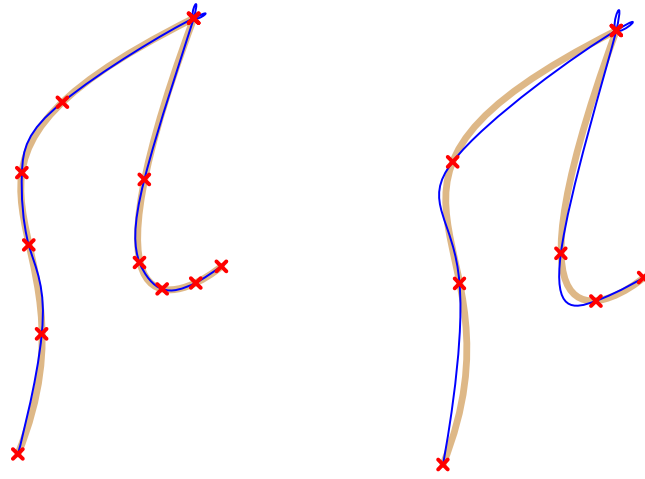
Coordinates within *pTPts* may be modified by this function if close option 1 is specified, or a coordinate is a NULL stack object (?). The resulting spline is twice continuously differentiable ($C^{(2)}$), except possibly at the end points of the *sample points*.

Examples

The following illustrations show how the **sfFitExactBez()** function can be used.

```
(1) KnotSeq := [0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2];
    CPTs := [1.0, 0.0, 3.0, 5.0, -2.0, 5.9, 4.7, 9.3, 2.0, 1.9, 4.7,
    3.5, 5.3, 4.0]; SPTs := sfBSplineSPTs(5 3 CPTs KnotSeq);
    BezPath := sfFitExactBez(? SPTs);
```

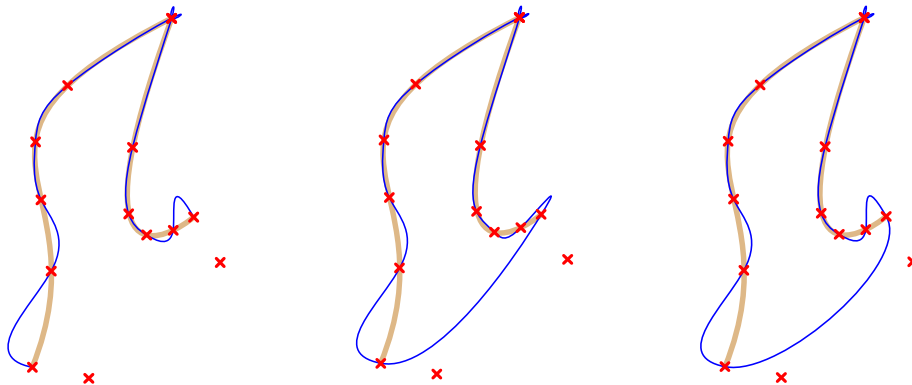
In example (1), a cubic B-spline is created for reference using the specified control points (CPTs) and *knot sequence* (KnotSeq). Some *sample points* (SPTs) are created on the B-spline via the **sfBSplineSPTs()** function. The **sfFitExactBez()** function is then called on the *sample points* to interpolate them. BezPath contains the *Bezier path* of a Bezier spline that ideally would be identical to the B-spline reference. However, the **sfFitExactBez()** function may produce a different curve than the original due to the requirement that the produced Bezier spline be twice continuously differentiable. The diagram below on the left illustrates the situation. The diagram on the right has less *sample points*.



The thick brown curve is the cubic B-spline reference. The red crosses indicate the *sample points* (SPts) on that B-spline. The blue thin curve is the curve that interpolates the *sample points* using the default value for *pTPts*. It will be noticed that the blue curve goes through all the *sample points*, meaning that the interpolation is exact. Also, there is some overshoot at the top sample point due to it being tripled because of the particular specification of the *knot sequence* and the requirement that the produced Bezier spline be twice continuously differentiable.

(2) KnotSeq := ...; CPTs := ...; SPts := sfBSplineSPts(5 3 CPTs KnotSeq);
BezPath := sfFitExactBez([2.5, -0.3, 6.0, 2.8, 0] SPts);

In example (2), the control points (CPTs), *knot sequence* (KnotSeq), and number of *sample points* are the same as for example (1). However, *pTPts* contains two points and an optional fifth element. The two points are also shown along with the *sample points*. The three diagrams below show the result of three values of the fifth element, 0, 1, and 2, respectively from left to right.



It will be noticed that in the first and last diagram, the tangent of the blue curve at the terminals are respectively the same.

Other Information

sfFitBez() ▀ **sfFitUBSpline3()** ♦

sfFitExactQBez()

sfFitExactQBez (*pPoints*) → *bez-path*

@SF_FIT_EXACT_Q_BEZ

pPoints A decimal sequence stack object.

bez-path A decimal sequence stack object.

Details

Returns the *Bezier path* (*bez-path*) of a quadratic spline that interpolates a sequence of *sample points* (*pPoints*).

pPoints is are given *sample points* which form the basis of interpolation.

bez-path is the *Bezier path* of a quadratic Bezier spline that interpolates the *sample points*. This implies that the Bezier spline will go through the *sample points*.

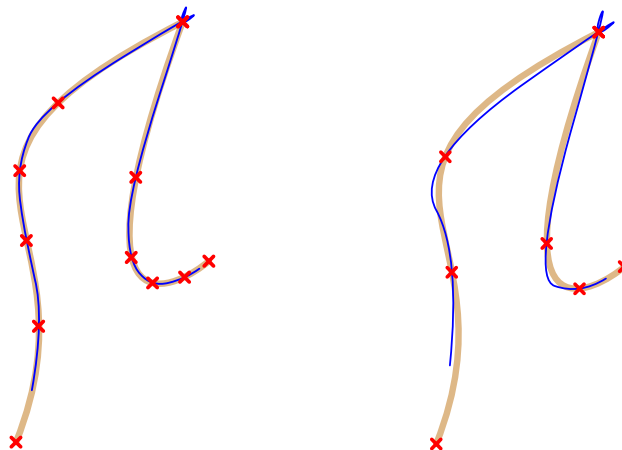
This function returns the Bezier control points (*bez-path*) of a quadratic spline that interpolates the specified *sample points* (*pPoints*). The quadratic spline does not start or end at the first or last of the *sample points* unless either one is repeated once to start or end at that point (the **sfRepeatPoints()** function can be used to repeat points in a path). NOTE: this function requires ETAC interpreter version 4-0 or above.

Examples

The following illustrations show how the **sfFitExactQBez()** function can be used.

```
(1) KnotSeq := [0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2];
    CPts := [1.0, 0.0, 3.0, 5.0, -2.0, 5.9, 4.7, 9.3, 2.0, 1.9, 4.7,
    3.5, 5.3, 4.0]; SPts := sfBSplineSPts(5 3 CPts KnotSeq);
    BezPath := sfFitExactQBez(SPts);
```

In example (1), a cubic B-spline is created for reference using the specified control points (CPts) and *knot sequence* (KnotSeq). Some *sample points* (SPts) are created on the B-spline via the **sfBSplineSPts()** function. The **sfFitExactQBez()** function is then called on the *sample points* to interpolate them. BezPath contains the *Bezier path* of a quadratic Bezier spline that ideally would be identical to the B-spline reference. However, the **sfFitExactQBez()** function may produce a different curve than the original due to the fact that the produced curve is quadratic. The diagram below on the left illustrates the situation. The diagram on the right has less *sample points*.

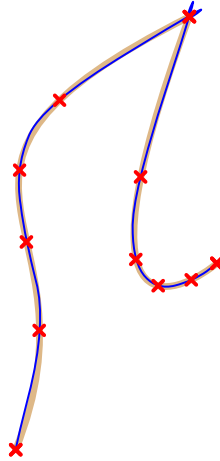


The thick brown curve is the cubic B-spline reference. The red crosses indicate the *sample points* (SPts) on that B-spline. The blue thin curve is the curve that interpolates the *sample points*. It

will be noticed that the blue curve goes through all the *sample points*, except the end points, meaning that the interpolation is exact. Also, there is some overshoot at the top sample point due to it being tripled because of the particular specification of the *knot sequence* and the requirement that the produced Bezier spline be quadratic.

```
(2) KnotSeq := ...; CPts := ...; SPts := sfBSplineSPts(5 3 CPts KnotSeq);
void sfRepeatPoints(0 1 sfRepeatPoints(1 1 SPts));
BezPath := sfFitExactQBez (SPts);
```

In example (2), the control points (CPts), *knot sequence* (KnotSeq), and number of *sample points* are the same as for example (1). However, the first and last of the *sample points* are repeated, resulting in the quadratic spline terminating at the end points, as shown below.



It is noteworthy that, in many cases, a quadratic interpolation results in a better fit to the original spline than a cubic interpolation.

Other Information

`sfFitExactBez()` ▪ `sfFitBez()` ▪ `sfFitUBSpline3()` ▪ `sfRepeatPoints()` ♦

sfFitUBSpline3()

<code>sfFitUBSpline3(pPoints pFact ?) → path</code>		@SF_FIT_UBSPLINE3
<code>pPoints</code>	A decimal sequence stack object.	
<code>pFact</code>	A decimal stack object.	
<code>path</code>	A decimal sequence stack object.	

Details

Converts *sample points* (`pPoints`) to a *B-spline path* (`path`) approximating an interpolated uniform cubic B-spline to a specified accuracy (`pFact`).

`pPoints` are given *sample points* to be approximately interpolated.

`pFact` indicates the accuracy to which the *sample points* are interpolated. This value can be negative or zero. A NULL stack object (?) in place of `pFact` gives the best fit (with a value of 1.353).

`path` is the *B-spline path* of a uniform cubic B-spline that approximately interpolates the *sample points*.

This function converts *sample points* (`pPoints`) to the control points (`path`) of a uniform cubic B-spline that approximately interpolates the *sample points*. The B-spline does not start or end at the first or last of the *sample points* unless either one is repeated twice more to start or end at that

point (the **sfRepeatPoints()** function can be used to repeat points in a path). A value of 1.0 for *pFact* will result in the returned control points being the same as the *sample points*. A smaller (positive) value of *pFact* than the default will result in a smoother curve; a higher value will result in a looser curve.

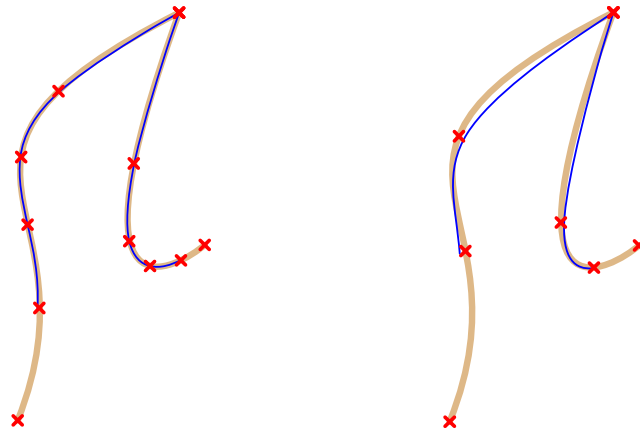
This function has the same functionality as does the **sfFitBez()** function, except that this function returns a *B-spline path* rather than a *Bezier path*, and the B-spline does not begin at the end points of *pPoints*.

Examples

The following illustrations show how the **sfFitUBSpline3()** function can be used.

```
(1) KnotSeq := [0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2];
    CPts := [1.0, 0.0, 3.0, 5.0, -2.0, 5.9, 4.7, 9.3, 2.0, 1.9, 4.7,
    3.5, 5.3, 4.0]; Spts := sfBSplineSpts(5 3 CPts KnotSeq);
    Path := sfFitUBSpline3(Spts ?);
```

In example (1), a cubic B-spline is created for reference using the specified control points (CPts) and *knot sequence* (KnotSeq). Some *sample points* (Spts) are created on the B-spline via the **sfBSplineSpts()** function. The **sfFitUBSpline3()** function is then called on the *sample points* to approximately interpolate them. Path contains the *B-spline path* of a cubic B-spline that ideally would be identical to the B-spline reference. However, the **sfFitUBSpline3()** function does not necessarily interpolate the *sample points* exactly (the **sfFitExactBez()** function does interpolate *sample points* exactly, but is less efficient). The diagram below on the left illustrates the situation. The diagram on the right has less *sample points*.

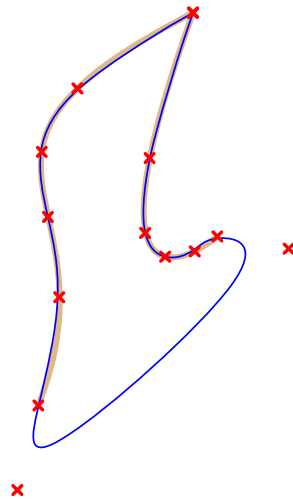


The thick brown curve is the cubic B-spline reference. The red crosses indicate the *sample points* (Spts) on that B-spline. The blue thin curve is the curve that interpolates the *sample points*. It will be noticed that the blue curve goes near the *sample points*, except the end points, meaning that the interpolation is approximate. (The sharp corner on the curve is a result of a triplicate sample point due to the particular specification of the *knot sequence*.) Note that a *B-spline path* is returned from the **sfFitUBSpline3()** function, not a *Bezier path*. The function **sfUBSpline3ToBez()** can be used to convert a *B-spline path* to a *Bezier path* for displaying.

```
(2) KnotSeq := ...; CPts := ...; Spts := sfBSplineSpts(5 3 CPts KnotSeq);
    Spts<=%[1] := [0.5, -2.0]; Spts ++:= [7.0, 3.7];
    Path := sfFitUBSpline3(Spts ?);
    Path ++:= [Path%[1], Path%[2]]; Path ++:= [Path%[3], Path%[4]];
    Path ++:= [Path%[5], Path%[6]];
```

In example (2), the control points (CPts), *knot sequence* (KnotSeq), and number of *sample points* are the same as for example (1). However, in this example, the end points are desired to be included in the approximate interpolation, and the resulting B-spline closed. This is achieved,

firstly, by adding a *point* to the beginning `[0.5, -2.0]`, and one to the end `[7.0, 3.7]` of the original *sample points*. The two points are chosen to give the desired curvature of the resulting B-spline at the ends of the *sample points*. Secondly, if desired, the first three points of the resulting B-spline are copied and appended to the end to close the B-spline. The result is shown below.



The diagram above also shows the two added *points* sitting there all alone.

Note that the `sfFitUBSpline3C()` function is the same as this function but closes the resultant B-spline automatically without the need for adding extra *points*.

Other Information

`sfFitBez()` ▪ `sfFitExactBez()` ▪ `sfUBSpline3ToBez()` ▪ `sfFitUBSpline3C()` ♦

sfFitUBSpline3C()

<code>sfFitUBSpline3C(<i>pPoints</i> <i>pFact</i> ?) → <i>path</i></code>		@SF_FIT_UBSPLINE3C
<i>pPoints</i>	A decimal sequence stack object.	
<i>pFact</i>	A decimal stack object.	
<i>path</i>	A decimal sequence stack object.	

Details

Converts *sample points* (*pPoints*) to a closed *B-spline path* (*path*) approximating an interpolated uniform cubic B-spline to a specified accuracy (*pFact*).

pPoints are given *sample points* to be approximately interpolated.

pFact indicates the accuracy to which the *sample points* are interpolated. This value can be negative or zero. A NULL stack object (?) in place of *pFact* gives the best fit (with a value of 1.353).

path is the *B-spline path* of a uniform cubic B-spline that approximately interpolates the *sample points*. The path is automatically closed to form a closed shape.

This function converts *sample points* (*pPoints*) to the control points (*path*) of a uniform cubic B-spline that approximately interpolates the *sample points*. The B-spline forms a closed shape. A value of 1.0 for *pFact* will result in the returned control points being the same as the *sample points*. A smaller (positive) value of *pFact* than the default will result in a smoother curve; a higher value will result in a looser curve.

This function has the same functionality as does the `sfFitUBSpline3()` function, except that the returned *B-spline path* is automatically closed by this function. It is also possible to explicitly

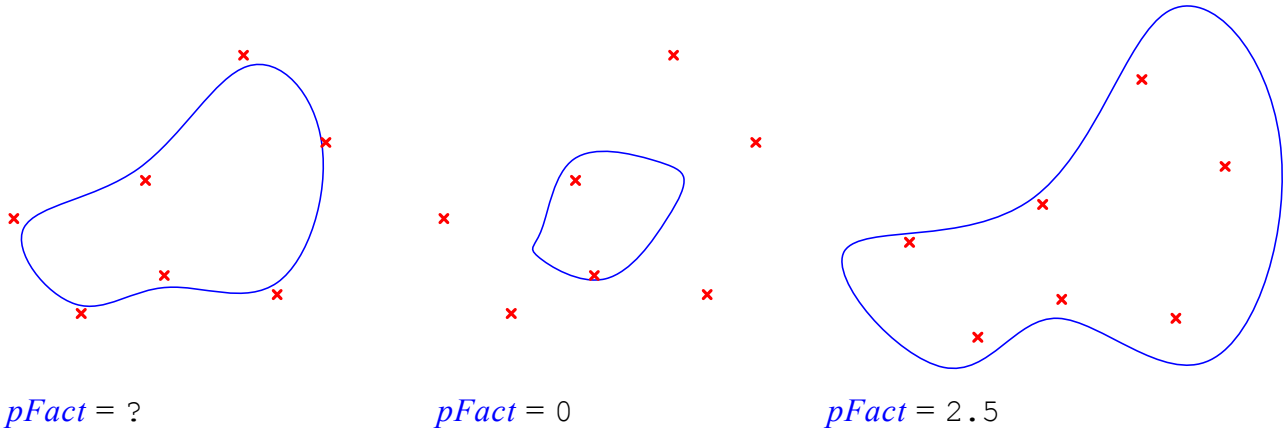
close the *B-spline path* resulting from the `sfFitUBSpline3()` function, but this function may be preferred.

Example

The following illustration shows how the `sfFitUBSpline3C()` function can be used.

(1) `SPTs := [-0.2, 2.5, -2.0, 5.0, 1.5, 6.0, 4.1, 9.3, 6.3, 7.0, 5.0, 3.0, 2.0, 3.5]; Path := sfFitUBSpline3C(SPTs ?);`

In example (1), the `sfFitUBSpline3C()` function is called on the specified *sample points* (SPTs) to approximately interpolate them. Path contains the *B-spline path* of a uniform cubic B-spline that forms a closed shape. The three diagrams below show the result of three values of *pFact*.



The red crosses indicate the specified *sample points* (SPTs). The first of the *sample points* is the lowest one on the left. The blue thin curve is the curve that interpolates the *sample points*. Note that a *B-spline path* is returned from the `sfFitUBSpline3C()` function, not a *Bezier path*. The function `sfUBSpline3ToBez()` can be used to convert a *B-spline path* to a *Bezier path* for displaying.

Other Information

`sfFitBez()` ▫ `sfFitExactBez()` ▫ `sfUBSpline3ToBez()` ▫ `sfFitUBSpline3()` ♦

sfRepeatPoints()

<code>sfRepeatPoints(pPos pCount pPath) → points</code>		@SF_REPEAT_POINTS
<i>pPos</i>	An integer stack object.	
<i>pCount</i>	An integer stack object.	
<i>pPath</i>	A decimal sequence stack object.	
<i>points</i>	A decimal sequence stack object.	

Details

Repeats a point (two coordinates) at a position (*pPos*) within a path (*pPath*) a number of times (*pCount*).

pPos is an index of a point into *pPath*. The first point is 1, etc, the last point is 0, the second last point is -1, etc. No action occurs if *pPos* is out of range.

pCount is the number of times to repeat the point at the specified position (*pPos*).

pPath is a TAC sequence of points in the form: $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point.

points is the same sequence as *pPath* but with the additional points.

Note that the sequence specified by *pPath* is altered and returned by this function.

Examples

The following illustrations show how the **sfRepeatPoints()** function can be used.

```
(1) Path := sfRepeatPoints(2 1 [1.0, 0.0, 3.0, 5.0, -2.0, 5.9]);
(2) Path := sfRepeatPoints(0 2 [1.0, 0.0, 3.0, 5.0, -2.0, 5.9]);
(3) Path := sfRepeatPoints(-2 1 [1.0, 0.0, 3.0, 5.0, -2.0, 5.9]);
```

In example (1), a sequence of points will have the second point repeated once. Path will contain $[1.0, 0.0, 3.0, 5.0, 3.0, 5.0, -2.0, 5.9]$.

In example (2), a sequence of points will have the last point repeated twice. Path will contain $[1.0, 0.0, 3.0, 5.0, -2.0, 5.9, -2.0, 5.9, -2.0, 5.9]$.

In example (3), a sequence of points will have the third last point repeated once. Path will contain $[1.0, 0.0, 1.0, 0.0, 3.0, 5.0, -2.0, 5.9, -2.0, 5.9]$. ♦

sfStdBezSpline()

sfStdBezSpline(<i>pPoints</i> <i>pSharp</i> ? <i>pAnc1</i> <i>pAnc2</i>) → <i>bez-path</i> @SF_STD_BEZ_SPLINE	
<i>pPoints</i>	A decimal sequence stack object.
<i>pSharp</i>	An integer or decimal stack object.
<i>pAnc1</i>	An integer stack object containing a logical boolean value.
<i>pAnc2</i>	An integer stack object containing a logical boolean value.
<i>bez-path</i>	A decimal sequence stack object.

Details

Returns the *Bezier path* (*bez-path*) of a standard Bezier spline from *sample points* (*pPoints*).

pPoints are given *sample points* which are used to form the basis of a standard Bezier spline. There must be at least four *sample points*. The number of points should be even, otherwise the last point is internally duplicated.

pSharp indicates the sharpness value of the resulting Bezier spline. A value of 0 (zero) will result in a standard Bezier spline. A value of 1 (one) will result in a poly-line. A NULL stack object (?) in place of *pSharp* is the same as having the value 0 (ie: standard Bezier spline). Typical values of *pSharp* are between 0.0 and 1.0.

pAnc1 is true to anchor the start point of the resulting Bezier spline to the first point in *pPoints*; otherwise, the first point of the spline will not necessarily be at the first point in *pPoints*.

pAnc2 is true to anchor the end point of the resulting Bezier spline to the last point in *pPoints*; otherwise, the end point of the spline will not necessarily be at the last point in *pPoints*.

bez-path is the *Bezier path* of a standard Bezier spline based on the specified *sample points*.

This function creates Bezier control points (*bez-path*) of a standard cubic Bezier spline for the given *sample points* (*pPoints*) and sharpness value (*pSharp*).

For a standard Bezier spline ($pSharp = 0$), the inner Bezier control points of all the Bezier curve segments are identical to the given *sample points*, and vice versa (except for the two end *sample points* of the whole spline).

The common outer Bezier control points of each adjacent Bezier curve segment are always midway between the two neighbouring inner *sample points*.

If both *pAnc1* and *pAnc2* are true, the two end Bezier control points of the whole spline are the two end points of *pPoints*.

Examples

The following illustrations show how the `sfStdBezSpline()` function can be used.

(1) `SPTs := [-0.2, 2.5, -2.0, 5.0, 4.1, 9.3, 6.3, 7.0, 5.0, 3.0, 2.0, 3.5]; BezPath := sfStdBezSpline(SPTs ? false false);`

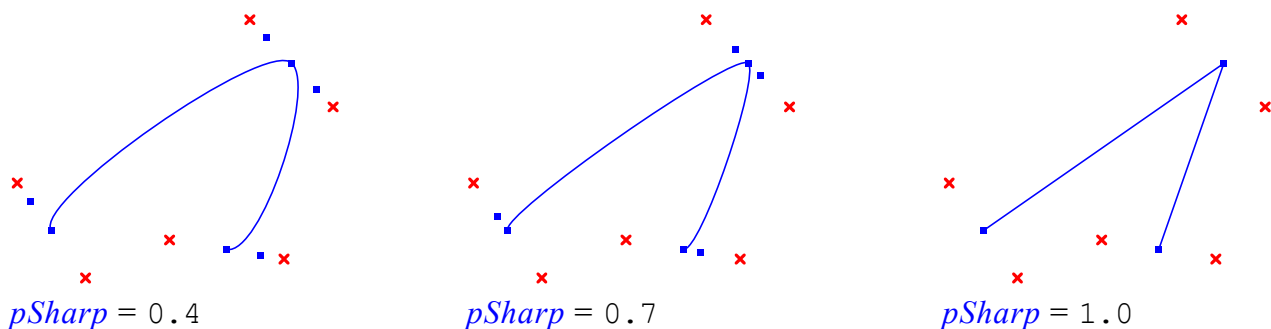
In example (1), a standard Bezier spline is created based on the given *sample points* (SPTs). BezPath contains the *Bezier path* of a standard Bezier spline with unanchored end points (the left diagram below), then with anchored end points (the right diagram). The anchored end points are produced by `sfStdBezSpline(SPTs ? true true)` (not shown above).



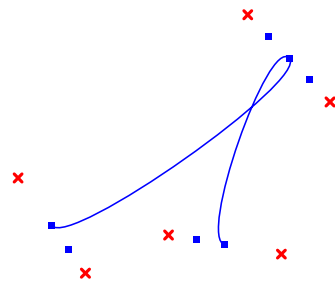
The red crosses indicate the given *sample points* (SPTs). The blue thin curve is the standard Bezier spline produced by the `sfStdBezSpline()` function. The blue boxes represent the control points of the Bezier spline. It will be noticed that the inner points of the *sample points* correspond with the inner control points of each segment of the Bezier spline. Also, the common end points of each Bezier curve segment (there are two segments in this example) are midway between the two neighbouring inner *sample points*.

(2) `SPTs := ...; BezPath := sfStdBezSpline(SPTs 0.4 false false);`

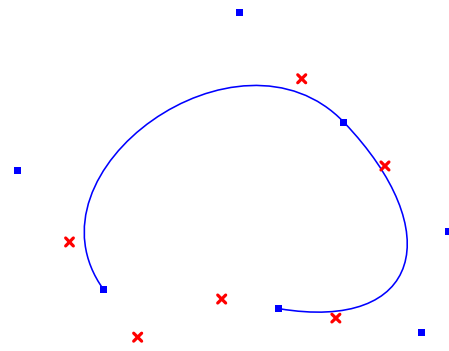
In example (2), the *sample points* (SPTs) are the same as for example (1). *pSharp* is set to 0.4, then to 0.7, then to 1.0 in the diagrams below from left to right respectively.



In the diagrams below, *pSharp* is set to 1.5 (left diagram) then -1.5 (right diagram).



$pSharp = 1.5$



$pSharp = -1.5$

sfSubBezCPTs()

sfSubBezCPTs (pP0 pP1 pP2 pP3 pT0 pT1) → point0 point1 point2 point3					@SF_SUB_BEZ_CPTS
pP0	A decimal sequence stack object.				
pP1	A decimal sequence stack object.				
pP2	A decimal sequence stack object.				
pP3	A decimal sequence stack object.				
pT0	A decimal stack object.				
pT1	A decimal stack object.				
point0	A decimal sequence stack object.				
point1	A decimal sequence stack object.				
point2	A decimal sequence stack object.				
point3	A decimal sequence stack object.				

Details

Returns the four control *points* (*point0*, *point1*, *point2*, *point3*) of a sub-Bezier curve of a specified Bezier curve (*pP0*, *pP1*, *pP2*, *pP3*) between two *t* values (*pT0*, *pT1*).

pP0 to *pP3* are the four control *points* of the specified Bezier curve.

pT0 and *pT1* are the two *t* values (typically between 0.0 and 1.0) on the specified Bezier curve that determine the sub-Bezier curve.

point0 to *point3* are the four control *points* of the sub-Bezier curve.

Bezier curves are mathematically specified as parametric equations defined by the parameter *t* with values between 0.0 and 1.0.

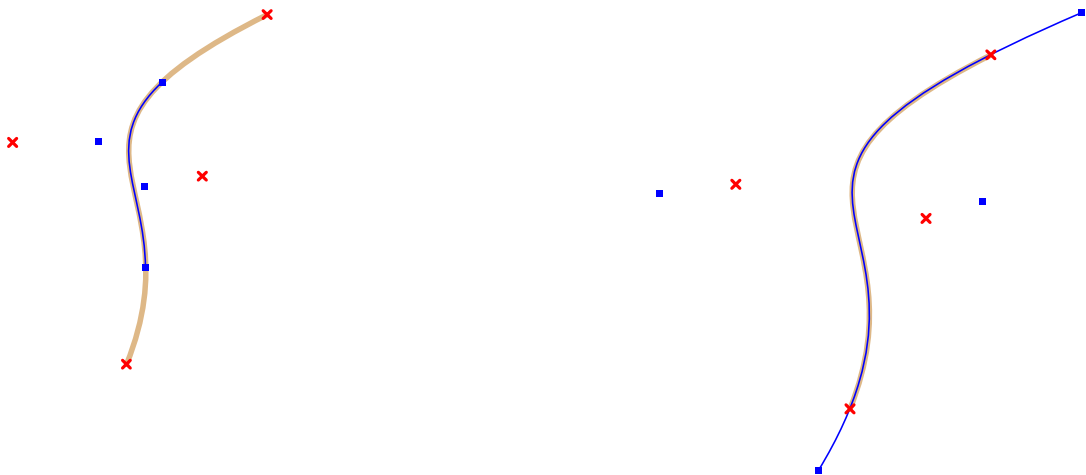
If the value of *pT0* is less than the value of *pT1* then the returned sub-Bezier curve will flow in the opposite direction to the specified Bezier curve. If either of the values of *pT0* or *pT1* is outside the range 0.0 to 1.0, then the corresponding part of the sub-Bezier curve will be outside the specified Bezier curve.

Example

The following illustration shows how the **sfSubBezCPTs ()** function can be used.

```
(1) SPt3 := SPt2 := SPt1 := SPt0 :=
    sfSubBezCpts ([1.0, 0.0] [3.0, 5.0] [-2.0, 5.9] [4.7, 9.3] 0.2 0.8);
```

In example (1), a Bezier curve is specified by the four control *points* (1.0, 0.0), (3.0, 5.0), (−2.0, 5.9), and (4.7, 9.3). The control *points* of a sub-Bezier curve on that specified Bezier curve are desired to be put into the variables SPt0, SPt1, SPt2, and SPt3. The diagram below on the left illustrates the situation. The diagram on the right has a *t* value between −0.1 and 1.1.



The thick brown curve is the specified Bezier curve. The red crosses indicate the specified control *points* of the Bezier curve. The blue boxes represent the returned control *points* of the sub-Bezier curve.

Other Information

sfBezCurvePoint() ♦

sfUBS3BlendFnts()		
sfUBS3BlendFnts	(<i>pDens</i>)	→ <i>output</i> @SF_UBS3_BLEND_FNTS
<i>pDens</i>	An integer stack object.	
<i>output</i>	A decimal sequence stack object.	

Details

Creates “blending functions” (*output*) for a uniform cubic B-spline.

pDens is a measure of the number of *sample points* in each cubic curve segment in the B-spline.

output is a TAC sequence to be used with the **sfUBSpline3Spts()** function.

This function creates pre-calculated data for use with creating *sample points* via the **sfUBSpline3Spts()** function for any *B-spline path* of a uniform cubic B-spline. This function typically needs to be called only once with all calls to the **sfUBSpline3Spts()** function for the same density (*pDens*).

The term “blending function” is a technical term — no actual function is created by this function. Instead, this function produces a sample of outputs from such blending functions in a TAC sequence (*output*).

Note that this function, along with the **sfUBSpline3Spts()** function, is an efficient way of creating *sample points* for a uniform cubic B-spline.

Example

The following illustration shows how the **sfUBS3BlendFnts()** function can be used.

```
(1) Blend := sfUBS3BlendFnts(6); SPts := sfUBSpline3SPts([...] Blend);
```

In example (1), “blending functions” are created (Blend) for any uniform cubic B-spline to be used with the **sfUBSpline3SPts()** function. The **sfUBSpline3SPts()** function returns *sample points* (SPts) for the specified *B-spline path* ([...]) with the density (6) specified for the **sfUBS3BlendFnts()** function. The same blending functions (Blend) can be used with other calls to the **sfUBSpline3SPts()** function with different *B-spline paths* for the same density (6). Note that too high a value for *pDens* may produce wobbly lines by the **sfUBSpline3SPts()** function if the returned *sample points* are converted into curves.

Related Information

sfUBS3BlendFnts() ♦

sfUBSpline3SPts()

sfUBSpline3SPts (<i>pCPts pBlend</i>) → <i>points</i>		@SF_UBSPLINE3_SPTS
<i>pCPts</i>	A decimal sequence stack object.	
<i>pBlend</i>	A decimal sequence stack object.	
<i>points</i>	A decimal sequence stack object.	

Details

Creates *sample points* (*points*) from a uniform cubic B-spline defined by a *B-spline path* (*pCPts*) and “blending functions” (*pBlend*).

pCPts is the *B-spline path* defining a uniform cubic B-spline on which *sample points* are to be returned.

pBlend is the “blending function” data produced from a call to the **sfUBS3BlendFnts()** function.

points are *sample points* on the B-spline defined by *pCPts*.

Note that this function, along with the **sfUBS3BlendFnts()** function, is an efficient way of creating *sample points* for a uniform cubic B-spline.

Example

The following illustration shows how the **sfUBSpline3SPts()** function can be used.

```
(1) Blend := sfUBS3BlendFnts(6); SPts := sfUBSpline3SPts([...] Blend);
```

In example (1), “blending functions” are created (Blend) for any uniform cubic B-spline to be used with the **sfUBSpline3SPts()** function. The **sfUBSpline3SPts()** function returns *sample points* (SPts) for the specified *B-spline path* ([...]) with the density (6) specified for the **sfUBS3BlendFnts()** function. The same blending functions (Blend) can be used with other calls to the **sfUBSpline3SPts()** function with different *B-spline paths* for the same density (6). Note that too high a value for *pDens* may produce wobbly lines by the **sfUBSpline3SPts()** function if the returned *sample points* are converted into curves.

Related Information

sfUBS3BlendFnts() ♦

sfUBSpline3ToBez()

sfUBSpline3ToBez (<i>pCpts</i> <i>pAnc1</i> <i>pAnc2</i>) → <i>bez-path</i>		@SF_UBSPLINE3_TO_BEZ
<i>pCpts</i>	A decimal sequence stack object.	
<i>pAnc1</i>	An integer stack object containing a logical boolean value.	
<i>pAnc2</i>	An integer stack object containing a logical boolean value.	
<i>bez-path</i>	A decimal sequence stack object.	

Details

Converts a uniform cubic B-spline defined by a *B-spline path* (*pCpts*) to a *Bezier path* (*bez-path*).

pCpts is the *B-spline path* defining a uniform cubic B-spline.

pAnc1 is `true` to anchor the start point of the resulting Bezier spline to the first point in *pCpts*; otherwise, the first point of the spline will not necessarily be at the first point in *pCpts*.

pAnc2 is `true` to anchor the end point of the resulting Bezier spline to the last point in *pCpts*; otherwise, the end point of the spline will not necessarily be at the last point in *pCpts*.

bez-path is the *Bezier path* of a Bezier spline identical to the B-spline.

This function converts a uniform cubic B-spline given the *B-spline path* (*pCpts*) to a *Bezier path* (*bez-path*). The returned *Bezier path* can be used to display the B-spline.

Other Information

sfBSpline3ToBez() ♦

Bibliography

The Official ETAC Programming Language copyright © Victor Vella (2025).

An Analysis of B-Splines copyright © Victor Vella (2023).

Glossary

B

B-spline path

A TAC sequence of X and Y coordinates in the form $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point of the *B-spline path*. The points of a *B-spline path* are the control points of a B-spline. A B-spline consists of joined polynomial curve segments. Other public documentation needs to be consulted for the definition of a B-spline.

Bezier path

A TAC sequence of X and Y coordinates in the form $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point of the *Bezier path*. The points of a *Bezier path* are the control and terminal points of a Bezier spline. A Bezier spline consists of joined Bezier curve segments. Only Bezier splines of degree three are considered in this document. Other public documentation needs to be consulted for the definition of a Bezier curve segment. A *Bezier path* must have at least four points (unless it is empty).

D

data object

The container of an ETAC dictionary used as a programmer-defined data structure consisting of stack objects identified by name. The dictionary itself is identified by the name defined by the private pre-processor definition `<_DATA_DICT>`.

E

ESL function identifier

The pre-processor definition name identifying an *ETAC function* within an *ETAC script library*. An *ESL function identifier* is specified by the programmer to include the corresponding *ETAC function* definition into *ETAC text script*.

ETAC function

The container of a special ETAC created procedure that creates a local dictionary then assigns the object stack arguments to that dictionary before calling the programmer-defined procedure.

ETAC script library

An *ETAC text script* file containing *ETAC function* definitions designed in such a way that the same required function is allocated only once within a *main ETAC session*.

ETAC session

The period devoted to the processing of ETAC code by the TAC processor after having been processed by the interpretation part of the ETAC interpreter. New *ETAC sessions* can exist within a given *ETAC session* for different ETAC code. Therefore, a given *ETAC session* can produce a new *ETAC session* (relating to different ETAC code from the given *ETAC session*) so that when the new *ETAC session* ends, the given *ETAC session* resumes.

ETAC text script

ETAC program code that is in human readable and writable text form. A file containing only *ETAC text script* typically has an extension of ‘etac’. Note that the term “ETAC text script” is used in the same sense as the word “code”, as in “ETAC text script code”.

K

knot sequence

A TAC sequence containing an ordered set of non-decreasing decimal numbers (called “knots”) required for the definition of a B-spline. Each knot, after the first d and before the last d knots, corresponds to a point on the spline. d is the degree of the polynomial involved in the spline.

M

main ETAC session

An *ETAC session* and all other new *ETAC sessions* produced directly or indirectly from that *ETAC session*, but not itself produced from any other *ETAC session*. A *main ETAC session* is typically begun via the RunETAC.exe and the AppETAC.dll computer programs.

P

point

A TAC sequence of the form $[x, y]$.

S

sample points

A TAC sequence of X and Y coordinates in the form $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point. The points of the *sample points* could be arbitrary or could be points on a specified spline, depending on the context.