Macros to Compute Splines*

Dan Luecking

2009/10/23

1 Introduction

A cubic spline through a set of points is a curve obtained by joining each point to the next with a cubic parametrized curve, where adjoining cubics must have matching first and second derivative at their common point.

It is possible for METAPOST to compute the necessary controls. Unfortunately, the controls are not uniquely determined unless the curve is required to be closed. For open curves, there is need for two additional equations at the end points. A 'relaxed spline' is produced if we require that the second derivative is $(0, 0)$ at those points. For a closed curve, the equality of the first and second derivatives at the common beginning/ending point gives the needed additional equations.

Note that this equates *time* derivatives, so this works best when points are relatively evenly spaced and so the speed is relatively uniform. If points are differently spaced then the relatively slower speed between closely spaced points allows sharper turns without large second derivatives. Curves produced tend to have a more natural look, and relaxed splines are most suitable for smoothing data that is obtained by taking observations at evenly spaced times. Still, the technique is somewhat unstable when points are closely spaced, for example when a small change in the position of one point can produce a large change in its direction when viewed from another point.

2 The Code

Start with version control information.

```
⟨*package⟩
if known splines_fileversion: endinput fi;
string splines_fileversion;
splines_fileversion := "2009/10/23, v0.3";
message "Loading splines.mp " & splines_fileversion;

```

`list_to_array` Now for a command that takes a variable name `(suffix arr)` and copies a list

*This file has version number v0.3, last revised 2009/10/23.

of pairs to `arr1`, `arr2`, etc.. The suffix must be declared by the calling program so that `arr` is numeric but `arr[]` are pairs, for example by "`save arr; pair arr[];`".

```
def list_to_array (suffix arr) (text list) =
  arr := 0;
  for _itm = list :
    arr[incr arr] := _itm;
  endfor
enddef;

```

`compute_spline` In this command we generate the equations common to all cubic splines: the equality of derivatives at all interior points. This command accepts three suffixes: `points`, `pr`, and `po` which should represent previously declared arrays of pairs. `points` is the array of points to be connected, and must be known. Arrays `pr` and `po` *must* be unknown and will hold the computed control points. See the code of `mkrelaxedspline` for an example of how this was arranged for the variables `rs_pr` and `rs_po`.

Contrary to the previous (unreleased) version, `compute_spline` takes a boolean argument (`closed`) and appends the same equations at the first (= last) point if the boolean is true.

`mkrelaxedspline` `mkclosedspline` The first of these macros appends the necessary additional equations to get zero second derivatives at the ends. The second simply calls `compute_spline` with the boolean set to true. Both return the computed path. In theory the knowledgeable user can call `compute_spline (false)`, append a choice of equations for the ends, and then call `mksplinepath (false)`.

`dospline` This version accepts a list of pairs and produces a spline through them. It simply stores the list in an array and calls the appropriate version that operates on an array.

```
def compute_spline (expr closed) (suffix points, pr, po) =
  % interior equations:
  for j= 2 upto points - 1 :
    % equate first derivatives:
    po[j] + pr[j] = 2 points[j];
    %   and second derivatives:
    pr[j+1] + 2 pr[j] = 2 po[j] + po[j-1];
  endfor
  % for a closed curve, the first and last are also interior:
  if closed:
   po 1 + pr 1 = 2 points 1;
   po[points] + pr[points] = 2 points[points];
   pr 2 + 2 pr 1 = 2 po 1 + po[points];
   pr 1 + 2 pr[points] = 2 po[points] + po[points-1];
  fi
enddef;

vardef mksplinepath (expr closed) (suffix points, pr, po) =
  points1..controls po1 and
```

```
33   for j = 2 upto points if not closed: -1 fi:
34     pr[j]..points[j]..controls po[j] and
35   endfor
36     if closed: pr 1..cycle else: pr[points]..points[points] fi
37 enddef;
38
39 vardef mkrelaxedspline (suffix pnts) =
40   save rs_pr, rs_po;
41   pair rs_po[], rs_pr[];
42   % Equate second derivative to zero at both end points
43   rs_pr 2 + pnts 1 = 2 rs_po 1 ;
44   pnts[pnts] + rs_po[pnts-1] = 2 rs_pr[pnts];
45   compute_spline (false) (pnts, rs_pr, rs_po);
46   mksplinepath   (false) (pnts, rs_pr, rs_po)
47 enddef;
48
49 vardef mkclosedspline (suffix pnts) =
50   save cs_pr, cs_po;
51   pair cs_pr[], cs_po[];
52   compute_spline (true) (pnts, cs_pr, cs_po);
53   mksplinepath   (true) (pnts, cs_pr, cs_po)
54 enddef;
55
56 vardef dospline (expr closed) (text the_list) =
57   save _sp; pair _sp[];
58   list_to_array (_sp) (the_list);
59   if closed :
60     mkclosedspline (_sp)
61   else:
62     mkrelaxedspline (_sp)
63   fi
64 enddef;
65
```

The above computations produce a 2-dimensional spline. A 1-dimensional cubic spline would be a function $f(t)$ with numeric values rather than pair values. Such are often used to interpolate functions. That is, given pairs (x_j, y_j), and assuming they lie on the graph of some function (generally unknown), fill in the graph with $y = f(x)$ where f is a cubic function of x in each interval $x_j < x < x_{j+1}$, making sure that the resulting graph is as smooth as possible at the points x_j.

The requirements on our 2-dimensional path are the following:

1. The jth link should connect (x_j, y_j) to (x_{j+1}, y_{j+1}).
2. The x-part of that link should increase linearly from x_j to x_{j+1} as t goes from 0 to 1.
3. The y-part should be a cubic $y = f(x)$.
4. The x-derivatives df/dx and d^2f/dx^2 should match at the connecting points.

Two necessary equations for converting between x and t coordinates are:

$$x = x_j + t dx_j \tag{1}$$

(where $dx_j = x_{j+1} - x_j$) and

$$\frac{df}{dt} = \frac{dx}{dt}\frac{df}{dx} = dx_j \frac{df}{dx}. \tag{2}$$

Thus we want to choose controls so that (1) is maintained and so that x-derivatives match. It turns out that this requires controls at

$$\begin{array}{c}(x_j, y_j) - (dx_{j-1}, s_j dx_{j-1})/3 \\ (x_j, y_j) + (dx_j, s_j dx_j)/3\end{array} \tag{3}$$

where s_j is the slope (derivative) at x_j. These control points will produce matching slopes regardless of the values chosen for the s_j. To get matching second derivatives we need the same conditions as in parametric splines. But those equations simplify to the form:

$$s_{j+1}dx_j - 2s_j(dx_j + dx_{j-1}) + s_{j-1}dx_{j-1} = 3y_{j+1} - 3y_{j-1}.$$

As with 2-D splines there can be almost any equations at the end points. For a relaxed spline we equate the second derivatives to 0. To get a periodic function, we equate the slope and second derivative at beginning to those at the end. This makes it possible to put a shifted copy of the graph with starting point at the end of the original and have the same smoothness at that connection as at the other points.

`compute_fcnspline` This issues the equation for the slopes (array `sl` of *unknown* numerics). The array `points` contains the (x, y) values and `dx` is a temporary numeric array which will be overwritten if known.

`mkfcnsplinepath` This simply assembles the path from the information computed by the above equations (and the extra equations given in the calling command).

`mkrelaxedfcnspline` This sets up arrays for the `dx` and `sl` parameters of `compute_fcnspline`, emit the necessary endpoint equations (zero second derivatives) and calls the previous two routines.

`mkperiodicfcnspline` This does the same as the previous command, but the endpoint equations make the first and second derivatives at the start equal to those at the end.

`fcnspline` Finally, this command copies a list of pairs into an array and calls the appropriate command to process them.

```
66 def compute_fcnspline (suffix points, dx, sl) =
67   % Get delta_x:
68   for j = 1 upto points - 1: dx[j] := xpart (points[j+1]-points[j]);
69   endfor
70   for j=2 upto points - 1:
71     sl[j + 1] * dx[j] + 2sl[j]*(dx[j] + dx[j-1]) + sl[j-1]*dx[j-1]
72         = 3*ypart(points[j+1] - points[j-1]);
73   endfor
74 enddef;
75
76 vardef mkfcnsplinepath (suffix points, dx, sl) =
77   points1..controls (points1 + (1, sl1)*dx1/3) and
```

```
    for j = 2 upto points - 1:
      (points[j] - (1, sl[j])*dx[j-1]/3) ..points[j]..
        controls (points[j] + (1,sl[j])*dx[j]/3) and
    endfor
    (points[points] - (1,sl[points])*dx[points-1]/3)..points[points]
enddef;

vardef mkperiodicfcnspline (suffix pnts) =
  save _sl, _dx; numeric _dx[], _sl[];
  compute_fcnspline (pnts, _dx, _sl);
  % periodicity equations:
  _sl 1 = _sl[pnts];
  _sl 2 * _dx 1 + 2 _sl 1 * _dx 1 + 2 _sl[pnts] * _dx[pnts-1]
      + _sl[pnts-1] * _dx[pnts-1]
          = 3 * ypart(pnts[2] - pnts[pnts-1]);
  mkfcnsplinepath  (pnts, _dx, _sl)
enddef;

vardef mkrelaxedfcnspline (suffix pnts) =
  save _sl, _dx; numeric _dx[], _sl[];
  compute_fcnspline (pnts, _dx, _sl);
  % relaxation equations.
  _sl 2 * _dx 1 + 2 _sl1 * _dx 1 = 3 * ypart(pnts2 - pnts1);
  _sl[pnts-1] * _dx[pnts-1] + 2 _sl[pnts] * _dx[pnts-1]
        = 3 * ypart(pnts[pnts] - pnts[pnts-1]);
  mkfcnsplinepath  (pnts, _dx, _sl)
enddef;

vardef fcnspline (expr periodic) (text the_list) =
  save _fs; pair _fs[];
  list_to_array (_fs) (the_list);
  if periodic:
    mkperiodicfcnspline (_fs)
  else:
    mkrelaxedfcnspline (_fs)
  fi
enddef;

⟨/package⟩
```

Index

Numbers refer to the page where the corresponding entry is described.