The bpolynomial package*

Stephan Hennig†

December 12, 2007

Abstract

The MetaPost package bpolynomial helps plotting polynomial and root functions. It provides macros to calculate one-segment Bézier curves exactly matching a given cubic polynomial or square or cubic root function. Additionally, tangents on all functions and derivatives of polynomials can be calculated.

Contents

1 Introduction

MetaPost has a variable type `path` that can be used for drawing smooth and visualy pleasing curves. When plotting graphs, the problem users are confronted with is how to define a suitable path representing a given function $f(x)$?

The splines package by Dan Luecking provides macros for drawing smooth piece-wise Bézier curves through arbitrary sample points of a function. [6] Since Bézier curves are polynomials of degree three, for cubic polynomials we can do better and find a matching one-segment Bézier curve. This package eases the task of finding a Bézier curve corresponding to a given function of the following types:

$$f(x) = ax^3 + bx^2 + cx + d \tag{1}$$

$$f(x) = u\sqrt{x+v} + w \tag{2}$$

$$f(x) = u\sqrt[3]{x+v} + w \tag{3}$$

*This document describes bpolynomial v0.5, last revised 2007/12/12.

†stephanhennig@arcor.de

2 Usage

The bpolynomial package provides two flavours of user interfaces:

- An advanced interface, that first requires a function to be defined and then provides access to the Bézier curve corresponding to the function, its tangent or derivative in a given intervall.
- A plain interface, that gives instant access to the Bézier curve matching a given function in an intervall.

The plain interface is less powerful, but might be convenient in certain circumstances. It is briefly described in section 2.5. The average user most likely wants to use the advanced interface, so we will discuss that first.

2.1 Plotting polynomials

2.1.1 Defining polynomials

As was said before, calculating the Bézier curve of a polynomial function first requires a function to be defined. The macro for defining a polynomial

$$f(x) = ax^3 + bx^2 + cx + d \tag{4}$$

is called

`newBPolynomial`

and takes five arguments. First argument is a suffix and the remaining four arguments are the four coefficients of the polynomial function a, b, c, d.

A definition of a polynomial

$$f(x) = 2x^3 + 0x^2 - 3x - 1 \tag{5}$$

associated with suffix `f` exemplary looks like this

```
newBPolynomial.f(2, 0, -3, -1);
```

The suffix argument `f` serves as an identifier for deriving names of other macros, that are defined by `newBPolynomial`. To be more precise, command

`newBPolynomial.<suffix>`

defines three new macros

```
<suffix>.getPath
<suffix>.eval
<suffix>.getTangent
```

that do the real work. These macros are described in the following sections.

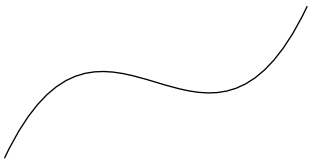

Figure 1: A cubic polynomial.

2.1.2 Getting the Bézier curve of a polynomial

Once a polynomial ⟨suffix⟩ is defined, the macro to request a Bézier curve matching the function on an intervall $[x_l, x_r]$ is

```
<suffix>.getPath
```

This macro takes two argument, the intervall boundaries x_l and x_r, and returns a polynomial shaped Bézier curve corresponding to function ⟨suffix⟩. Command `<suffix>.getPath` can be called as often as required with varying intervall boundaries and always returns a path corresponding to the new intervall.

Let's have a look at an example. Plotting our polynomial $f(x)$ on the intervall $(-2, 2)$ can be done with the following code (figure 1).

```
newBPolynomial.f(2, 0, -3, -1);
draw f.getPath(-2, 2) transformed T;
```

Hint: Since the bpolynomial package never uses ⟨suffix⟩ as a complete identifier, you can use that as the name of a path variable to store the path returned by `<suffix>.getPath` for later drawing. Any other path (array) variable serves the same purpose, though.

```
newBPolynomial.f(2, 0, -3, -1);
path f;
f := f.getPath(-2, 2);
draw f transformed T;
```

Note, since the base unit of MetaPost is a big point ($1\,\mathrm{bp} = \frac{1}{72}\,\mathrm{in}$) in most cases functions have to be scaled to a proper size before plotting. It is *not* recommended, however, to apply scaling to polynomial coefficients, since current MetaPost versions can't handle large numbers very well.[1] Instead, scaling should be applied to paths during `draw` operations. If not stated otherwise, in this manual, scaling is applied by an affine transform

```
T = identity xscaled 10mm yscaled 1mm;
```

[1] At the time of writing the latest release is MetaPost v1.002.

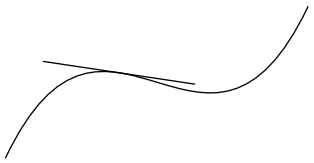

Figure 2: Cubic polynomial with a tangent.

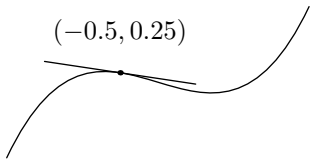

Figure 3: And a labelled point.

2.1.3 Getting a tangent on a polynomial

Another macro set up during a function definition is

```
<suffix>.getTangent
```

This macro returns a path tangent to a function ⟨suffix⟩ at a specific point. Arguments are an x-coordinate, where the tangent is placed, and two values ϵ_l, ϵ_r that specify the neighbourhood around x. The tangent path then covers the intervall $[x + \epsilon_l, x + \epsilon_r]$. This syntax has been choosen to make it easy to move a tangent along a function, keeping its neighbourhood at a fixed size.

As an example, the following code draws a tangent that touches f at $x = -0.5$ with a neighbourhood $\epsilon = \pm 1$ (figure 2).[2]

```
newBPolynomial.f(2, 0, -3, -1);
draw f.getPath(-2, 2) transformed T;
draw f.getTangent(-0.5)(-1, 1) transformed T;
```

2.1.4 Evaluating polynomials

Additionally to getting the Bézier curve or tangent of a polynomial, polynomial functions can also be evaluated numerically at a specific location. Defining a function ⟨suffix⟩ sets up a macro

```
<suffix>.eval
```

that takes as argument an x-coordinate and returns the function value at that location. As an example, labelling an arbitrary point on f can be done as follows (figure 3).

```
dotlabeldiam := 2bp;
labeloffset := 10bp;
```

[2]Both types of arguments to `f.getTangent`—x and ϵ values—have been put in separate pairs of parentheses to make the code more readable. Even if the second pair of parentheses looks like a coordinate of type `pair`—it isn't. For MetaPost this syntax is equivalent to

```
draw f.getTangent(-0.5, -1, 1) transformed T;
```

```
newBPolynomial.f(2, 0, -3, -1);
draw f.getPath(-2, 2) transformed T;
x := -0.5;
show (x, f.eval(x));
draw f.getTangent(x)(-1, 1) transformed T;
dotlabel.top(btex $(-0.5, 0.25)$ etex, (x, f.eval(x)) transformed T);
```

For simplicity, the label has been provided explicitly in this example (after reading the coordinates off the `log` file). But it is also possible to attach the correct coordinates automatically with the help of the MetaPost package LaTeXMP and LaTeX package numprint. While the former helps passing dynamically generated text from MetaPost to LaTeX, the latter can be used to format and round numbers. [2, 7].

2.2 Plotting square and cubic roots

Two other types of functions can be described by Bézier curves: square and cubic roots[3]. The bpolynomial package provides support for square and cubic root functions of the following type

$$f_{sqr}(x) = u\sqrt{x+v} + w \tag{6}$$

$$f_{cub}(x) = u\sqrt[3]{x+v} + w \tag{7}$$

through macros

```
newBSqrRoot
newBCubRoot
```

These macros are similar to `newBPolynomial`. Both macros take as arguments a suffix ⟨suffix⟩ and the three parameters u, v and w of the respective function. They define three macros

```
<suffix>.getPath
<suffix>.eval
<suffix>.getTangent
```

that can be used the same way as for polynomial functions.

There is one notable difference between polynomial and root functions. While polynomials z^k are defined for all arguments $z \in \mathbb{R}$, roots $\sqrt[k]{z}$ are only defined for non-negative arguments $z \in \mathbb{R}_+$ and therefore the functions from equations 6 and 7 are only defined for

$$x \geq -v. \tag{8}$$

In case the arguments to macros `<suffix>.getPath`, `<suffix>.getTangent` or `<suffix>.eval` violate equation 8 all macros write a warning to the log file,

[3]The reason square and cubic roots can be expressed in terms of Bézier curves is that both are inverse functions of cubic polynomials. More information can be found in appendix A.2.

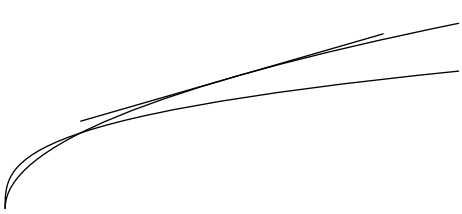

Figure 4: Square and cubic roots with a tangent.

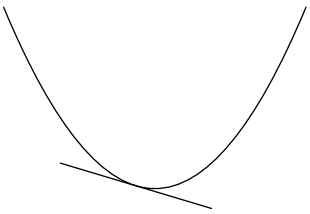

Figure 5: First derivative of a cubic polynomial with tangent.

but go on with their calculations replacing the arguments in question (or the resulting range boundaries) by $-v$.

The following example plots two functions $s(x) = \sqrt{x}$ and $c(x) = \sqrt[3]{x}$ with a tangent on s at $x = 3$ (figure 4).

```
T := identity scaled 10mm;
newBSqrRoot.s(1,0,0);
newBCubRoot.c(1,0,0);
draw s.getPath(0,6) transformed T;
draw c.getPath(0,6) transformed T;
draw s.getTangent(3)(-2, 2) transformed T;
```

2.3 Dealing with derivatives

Since plotting function often involves plotting derivatives of functions, too, the bpolynomial package provides support for plotting derivatives of polynomials and tangents thereof. Unfortunately, derivatives of root functions cannot be described by Bézier curves, so those are not supported.

In section 2.1.1 it was said macro `newBPolynomial` defines three new macros. But this is not the full story. In fact, the command

`newBPolynomial.<suffix>`

defines twelve macros, three of them we already know, `<suffix>.getPath`, `<suffix>.eval`, `<suffix>.getTangent`. The remaining nine macros are similar, but correspond to the first, second and third derivative of a polynomial ⟨suffix⟩, resp. To access these macros just add the required number of prime characters to the suffix name (three at maximum). For instance, to get the path corresponding to the first derivative of polynomial ⟨suffix⟩ call

`<suffix>'.getPath`

and to get a tangent on the first derivative call

`<suffix>'.getTangent`.

In total these are the macros defined by `newBPolynomial.<suffix>`:

```
<suffix>.getPath          <suffix>''.getPath
<suffix>.eval             <suffix>''.eval
<suffix>.getTangent       <suffix>''.getTangent
<suffix>'.getPath         <suffix>'''.getPath
<suffix>'.eval            <suffix>'''.eval
<suffix>'.getTangent      <suffix>'''.getTangent
```

As an example, the following code draws a tangent on the first derivative of a polynomial `f` (figure 5).

```
newBPolynomial.f(2, 0, -3, -1);
draw f'.getPath(-2, 2) transformed T;
draw f'.getTangent(-0.25)(-1, 1) transformed T;
```

2.4 Accessing function parameters

The parameters passed to one of the function defining macros are saved for internal calculations. These variables can be accessed by users, too, but should not be changed.

Defining a polynomial with

```
newBPolynomial.<suffix>(<a>, <b>, <c>, <d>)
```

defines variables

```
<suffix>.a                <suffix>''.a
<suffix>.b                <suffix>''.b
<suffix>.c                <suffix>''.c
<suffix>.d                <suffix>''.d
<suffix>'.a               <suffix>'''.a
<suffix>'.b               <suffix>'''.b
<suffix>'.c               <suffix>'''.c
<suffix>'.d               <suffix>'''.d
```

and sets them to the value of the corresponding coefficient.

For root functions the following variables are used:

```
<suffix>.u
<suffix>.v
<suffix>.w
```

Additionally, variables

```
<suffix>.a
<suffix>.b
<suffix>.c
<suffix>.d
```

contain the coefficients of the inverse polynomial function.

2.5 The plain interface

The bpolynomial package provides a plain interface, too. This interface avoids the necessity to define a function prior to calculating paths, but is limited to calculating Bézier curves of explicitly given polynomials and root functions—tangents and derivatives are not supported. The plain interface consists of three macros

```
getBezierFromPolynomial
getBezierFromSqrRoot
getBezierFromCubRoot
```

that take as arguments the parameters of the polynomial or root function plus the range the path should cover.

For instance, the first example could as well have been drawn with the following code

```
draw getBezierFromPolynomial(2, 0, -3, -1)(-2, 2) transformed T;
```

All three macros call a macro `bpolynomial_getBezierFromPolynomial` behind the scenes, that does the necessary calculations and, in fact, is the heart of this package. The mathematics behind that macro are described in appendix A.

3 Examples

This section contains some more elaborate examples. The code of all examples is copied here, so that you can easily compare it with the resulting figures. If you want to play around with the code you can also find it in file `examples.mp`, that actually contains the code of all examples in this manual.

One additional note in advance: While playing with figure 8, the author noticed a subtle rendering bug in Adobe Reader 7.0.9, that caused the graphs and the filled area not matching exactly. In print or with GSview 4.8 and Ghostscript 8.60 everything looked fine. The problem is actually unrelated to the filling, but there seem to be some numeric issues in the Bézier curve rendering algorithm of Adobe Reader. A work-around is to modifiy problematic paths, *e. g.*, by slightly changing their plotting range.

The first example demonstrates how bpolynomial and John Hobby's graph package[3] can be used together to plot polynomials in a coordinate system. The `draw` command has just to be replaced by `gdraw`. The latter command additionally clips paths to the boundaries of the coordinate system (see figure 6). Finally, in this example a table of points is printed to the console and `log` file.

```
path f, g;
  xmin := -7; xmax := 7;
  ymin := -7; ymax := 7;
  %%% Define polynomials f and g.
```

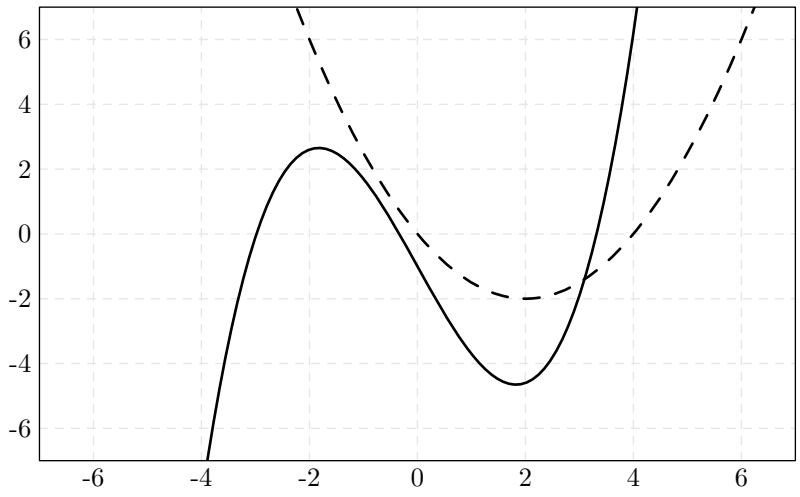

Figure 6: Packages bpolynomial and graph interacting.

```
newBPolynomial.f(0.3, 0, -3, -1);
f := f.getPath(xmin, xmax);
newBPolynomial.g(0, 0.5, -2, 0);
g := g.getPath(xmin, xmax);
%%% Draw graph.
draw begingraph(10cm, 6cm);
  setrange(xmin,ymin, xmax,ymax);
  autogrid(grid.bot, grid.lft) dashed evenly withcolor .9white;
  drawoptions(withpen pencircle scaled 1bp);
  gdraw f;
  gdraw g dashed evenly scaled 2;
  drawoptions();
endgraph;
show f;
%%% Write table with some points of f to log file.
show "Polynomial: " & decimal f.a & "x^3 + " &
  decimal f.b & "x^2 + " & decimal f.c & "x + " & decimal f.d;
for x=-5 upto 5:
  show (x, f.eval(x));
endfor
```

Note command `show f` that writes path `f` to the `log` file. Inspecting that we can easily verify `f` consists of just one path segment:

```
(-7,-82.90105)..controls (-2.33333,108.90105) and (2.33333,-110.90105)
 ..(7,80.90105)
```

In the next example, a cubic polynomial f is plotted together with its derivatives f', f'' and f'''. Additionally, for all four functions the tangents are drawn

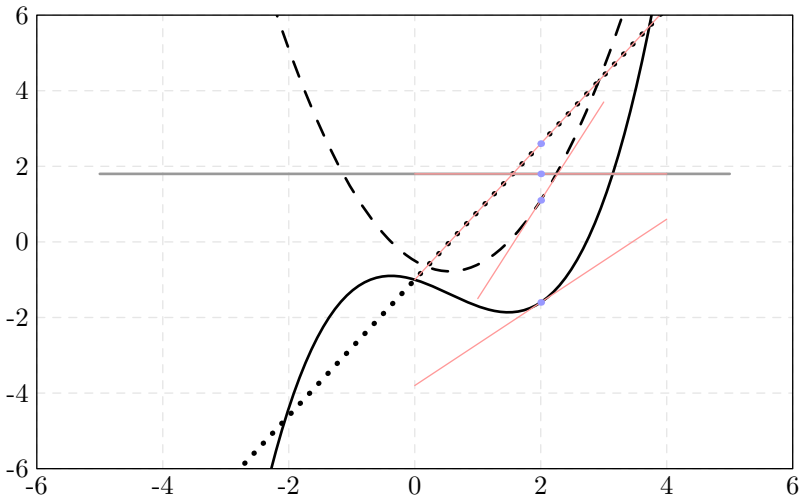

Figure 7: A cubic polynomial with derivatives and tangents.

at $x = 2$. Admittedly, the plot is a little bit crowded. But it should only serve as an example (figure 7).

```
xmin := -6; xmax := 6;
ymin := -6; ymax := 6;
newBPolynomial.f(0.3, -0.5, -0.5, -1);
draw begingraph(10cm, 6cm);
  setrange(xmin,ymin, xmax,ymax);
  autogrid(grid.bot, grid.lft) dashed evenly withcolor .9white;
  drawoptions(withpen pencircle scaled 1bp);
  %%% Draw f and its derivatives f', f'', f'''.
  gdraw f.getPath(xmin, xmax);
  gdraw f'.getPath(xmin, xmax) dashed evenly scaled 2;
  gdraw f''.getPath(xmin, xmax) dashed withdots
    withpen pencircle scaled 2bp;
  gdraw f'''.getPath(-5, 5) withcolor .6white;
  %%% Draw tangents and mark points.
  x := 2;
  drawoptions(withcolor (1, 0.6, 0.6));
  gdraw f.getTangent(x)(-2, 2);
  gdraw f'.getTangent(x)(-1, 1);
  gdraw f''.getTangent(x)(-2, 2);
  gdraw f'''.getTangent(x)(-2, 2);
  drawoptions(withcolor (0.6, 0.6, 1));
  dotlabeldiam := 2.5bp;
  gdotlabel("", (x, f.eval(x)));
  gdotlabel("", (x, f'.eval(x)));
  gdotlabel("", (x, f''.eval(x)));
```

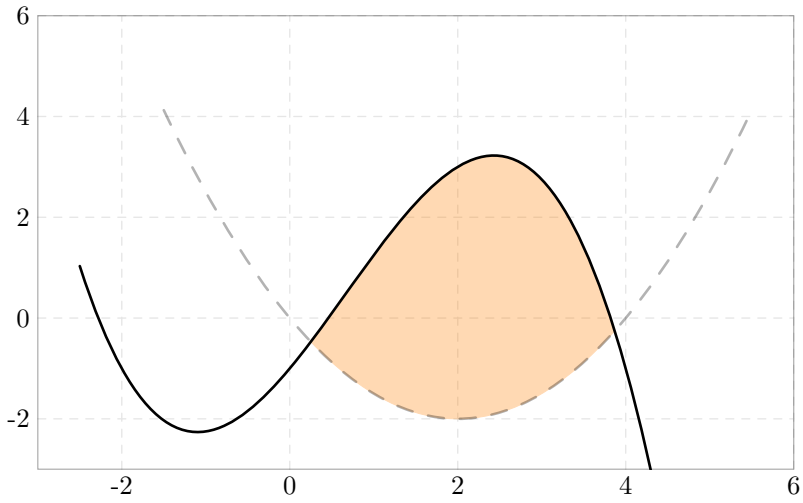

Figure 8: Applying a transparent fill to an area enclosed by two functions.

```
    gdotlabel("", (x, f'''.eval(x)));
    drawoptions();
  endgraph;
```

How about some eye candy, *e. g.*, transparency effects? In the next example the area enclosed by two functions is filled with a transparent colour (figure 8).

Note, advanced PDF features like transparency and shadings are provided by package metafun. [1] Figures using such features have to be converted to stand-alone PDF files with the `mptopdf` utility before including them into a LaTeX document.

```
path f, g, A;
  xmin := -3; xmax := 6;
  ymin := -3; ymax := 6;
  newBPolynomial.f(-0.25, 0.5, 2, -1);
  newBPolynomial.g(0, 0.5, -2, 0);
  f := f.getPath(-2.5, 5.5);
  g := g.getPath(-1.5, 5.5);
  %%% Find area between f and g.
  A := buildcycle(g, reverse f);
  draw begingraph(10cm, 6cm);
    setrange(xmin,ymin, xmax,ymax);
    autogrid(grid.bot, grid.lft) dashed evenly withcolor .9white;
    %%% Fill area with transparent colour.
    gfill A withcolor transparent (1, .3, (1, 0.5, 0));
    drawoptions(withpen pencircle scaled 1bp);
    gdraw f;
```

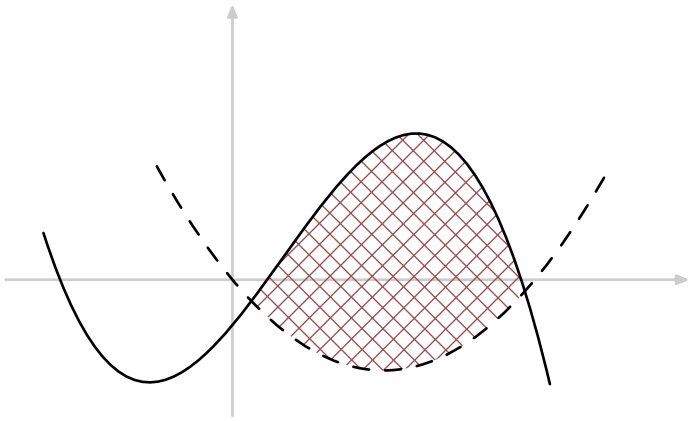

Figure 9: Hatching an area enclosed by two functions.

```
    gdraw g dashed evenly scaled 2;
    drawoptions();
  endgraph;
```

As an alternative to solid fills areas can be emphasized by hatch patterns. Hatching support is provided by package hatching. [4] Unfortunately, packages graph and hatching do not work together. Therefore, in the next example a simple coordinate system had to be drawn manually (figure 9).

```
path f, g, A;
  T := identity xscaled 10mm yscaled 6mm;
  %%% Draw coordinate system.
  xmin := -3; xmax := 6;
  ymin := -3; ymax := 6;
  drawoptions(withpen pencircle scaled 1bp withcolor 0.8white);
  drawarrow ((xmin,0)--(xmax,0)) transformed T;
  drawarrow ((0,ymin)--(0,ymax)) transformed T;
  newBPolynomial.f(-0.25, 0.5, 2, -1);
  newBPolynomial.g(0, 0.5, -2, 0);
  f := f.getPath(-2.5, 4.2);
  g := g.getPath(-1, 5);
  A := buildcycle(g, reverse f);
  %%% Fill area with pattern.
  drawoptions();
  hatchoptions(withcolor (0.6, 0.3, 0.3));
  hatchfill A transformed T
    withcolor (-45, 2mm, -0.5bp) withcolor (45, 2mm, -0.5bp);
  drawoptions(withpen pencircle scaled 1bp);
  draw f transformed T;
  draw g transformed T dashed evenly scaled 2;
```

A Calculating Bézier curves

A.1 Polynomials

A Bézier curve $P(t)$ with end points $A = (x_A, y_A)$ and $D = (x_D, y_D)$ and control points $B = (x_B, y_B)$ and $C = (x_C, y_C)$ is defined as [5]

$$P(t) = \begin{pmatrix} x \\ y \end{pmatrix}(t) = (1-t)^3 A + 3(1-t)^2 t B + 3(1-t)t^2 C + t^3 D, \quad 0 \le t \le 1. \tag{9}$$

This equation can be rewritten as

$$P(t) = A + 3(B-A)t + 3(C-2B+A)t^2 + (D-3C+3B-A)t^3, \quad 0 \le t \le 1. \tag{10}$$

An arbitrary function $y = f(x)$ can be written in parameter form as

$$F(t) = \begin{pmatrix} x \\ y \end{pmatrix}(t) = \begin{pmatrix} x(t) \\ f\big(x(t)\big) \end{pmatrix}, \quad t \in \mathbb{R} \tag{11}$$

with parameter t.

For a polynomial function

$$f(x) = ax^3 + bx^2 + cx + d, \quad x \in [x_0, x_1] \tag{12}$$

we have

$$x(t) = x_0 + (x_1 - x_0)t, \quad 0 \le t \le 1 \tag{13}$$

and hence

$$F(t) = \begin{pmatrix} x_0 + (x_1 - x_0)t \\ ax(t)^3 + bx(t)^2 + cx(t) + d \end{pmatrix}, \quad 0 \le t \le 1. \tag{14}$$

Writing F(t) down explicitly is left as an exercise for the interested reader.

Finally, setting

$$P(t) = F(t) \tag{15}$$

and sorting the coefficients of the t^k one arrives at the following *original* equation system:

$$\begin{aligned}
x_A &= x_0 && (16)\\
3(x_B - x_A) &= x_1 - x_0 && (17)\\
3(x_C - 2x_B + x_A) &= 0 && (18)\\
x_D - 3x_C + 3x_B - x_A &= 0 && (19)\\
y_A &= ax_0^3 + bx_0^2 + cx_0 + d && (20)\\
3(y_B - y_A) &= 3ax_0^2(x_1 - x_0) + 2bx_0(x_1 - x_0) + c(x_1 - x_0) && (21)\\
3(y_C - 2y_B + y_A) &= 3ax_0(x_1 - x_0)^2 + b(x_1 - x_0)^2 && (22)\\
y_D - 3y_C + 3y_B - y_A &= a(x_1 - x_0)^3 && (23)
\end{aligned}$$

Note, there are only constants on the right-hand side of all equations. That is, this equation system is linear in the eight variables x_A, x_B, x_C, x_D, y_A, y_B, y_C, y_D.

Since MetaPost can solve linear equation systems, hacking equations 16 to 23 into MetaPost code and requesting a path segment

$$(x_A,y_A)..\texttt{controls}\ (x_B,y_B)\ \texttt{and}\ (x_C,y_C)..(x_D,y_D)$$

returns the polynomial shaped curve we are looking for.

Internally, the bpolynomial package does not solve the original equation system, but a *modified* variant, that is numerically slightly more robust.

Equations 16 to 19 can be written down explicitly as

$$x_A = x_0 \tag{24}$$

$$x_B = x_0 + \frac{1}{3}(x_1 - x_0) \tag{25}$$

$$x_C = x_1 - \frac{1}{3}(x_1 - x_0) \tag{26}$$

$$x_D = x_1 \tag{27}$$

Additionally, we know that $D = (x_D, y_D)$ is a point on the polynomial. Therefore, equation 23 of the original system can be replaced by

$$y_D = ax_1^3 + bx_1^2 + cx_1 + d \tag{28}$$

Equations 20 to 22 of the original equation system and the new equations 24 to 28 constitute the modified equation system, that is solved in bpolynomial.

A.2 Square and cubic roots

When requesting the Bézier curve of a root function, the bpolynomial package first calculates the inverse function, then calculates the corresponding path and finally reflects that on the function $f(x) = x$ to get the inverse again.

The inverse functions of the root functions from equations 6 and 7 are

$$\widetilde{f}_{sqr}(y) = \left(\frac{y-w}{u}\right)^2 - v = 0y^3 + \frac{1}{u^2}y^2 + \frac{-2w}{u^2}y + \frac{w^2}{u^2} - v \tag{29}$$

$$\widetilde{f}_{cub}(y) = \left(\frac{y-w}{u}\right)^3 - v = \frac{1}{u^3}y^3 + \frac{-3w}{u^3}y^2 + \frac{3w^2}{u^3}y + \frac{w^3}{u^3} - v \tag{30}$$

which both are clearly polynomials.

Happy TEXing!
Stephan Hennig

References

[1] Hagen, Hans, *metafun*,
http://www.pragma-ade.com/general/manuals/metafun-p.pdf

[2] Harders, Harald, *The numprint package*, 2007,
CTAN:macros/latex/contrib/numprint/numprint.pdf

[3] Hobby, John D., *Drawing graphs with MetaPost*,
http://www.tug.org/docs/metapost/mpgraph.pdf

[4] Jackowski, B., *hatching.mp*,
CTAN:graphics/metapost/contrib/macros/hatching/README

[5] Knuth, Donald E., *The METAFONTbook*, Addison-Wesley, Reading, Massachusetts, 1986, (Computers & Typesetting, C)

[6] Luecking, Dan, *Macros to compute splines*, 2005,
CTAN:graphics/metapost/contrib/macros/splines/splines.pdf

[7] Morawski, Jens-Uwe, *latexMP*, 2005, CTAN:
graphics/metapost/contrib/macros/latexmp/doc/latexmp.pdf