MetaPost path resolution isolated

Abstract
A new interface in MPLib version 1.800 allows one to resolve path choices programmatically, without the need to go through the MetaPost input language.

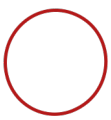
Metapost path solving

As we all know, MetaPost is pretty good at finding pleasing control points for paths. What all of you may know is that besides drawing on a picture, MetaPost can also display the found control points in the log file.

Some illustration at this point is useful. Here is the MetaPost path input source of a very simple path (as well as a visualisation of the path):

```
tracingchoices := 1;
path p;
p := (0,0) ..(10,10) ..(10,-5) ..cycle;
```

And here is what MetaPost outputs in the log file:



```
Path at line 5, before choices:
(0,0)
 ..(10,10)
 ..(10,-5)
 ..cycle

Path at line 5, after choices:
(0,0)..controls (-1.8685,6.35925) and (4.02429,12.14362)
 ..(10,10)..controls (16.85191,7.54208) and (16.9642,-2.22969)
 ..(10,-5)..controls (5.87875,-6.6394) and (1.26079,-4.29094)
 ..cycle
```

A more complex path of course creates more output, so:

```
p := (0,0)..(2,20){curl1}..{curl1}(10, 5)..controls (2,2) and (9,4.5)..
  (3,10)..tension 3 and atleast 4 .. (1, 14){2,0} .. {0,1}(5,-4) ;
```

produces:

```
Path at line 7, before choices:
(0,0){curl 1}
 ..{curl 1}(2,20){curl 1}
 ..{curl 1}(10,5)..controls (2,2) and (9,4.5)
 ..(3,10)..tension 3 and atleast4.5
 ..{4096,0}(1,14){4096,0}
 ..{0,4096}(5,-4)

Path at line 7, after choices:
(0,0)..controls (0.66667,6.66667) and (1.33333,13.33333)
 ..(2,20)..controls (4.66667,15) and (7.33333,10)
 ..(10,5)..controls (2,2) and (9,4.5)
 ..(3,10)..controls (2.34547,10.59998) and (0.48712,14)
 ..(1,14)..controls (13.40117,14) and (5,-35.58354)
 ..(5,-4)
```

But what if ...

But what if you want to use that functionality outside of MetaPost, for instance in a C program?

> You will have to compile MPLib into your program; then create a Metapost language input string; execute it; and parse the log result.

All of that is not very appealing. It would be much better ...

> if you could compile MPLib into your program; create a path programmatically; and then run the Metapost path solver directly; automatically updating the original path.

And that is what the current version of MPLib will allow you to do.

How it works

Once again, it is easiest to show you what to do by using a source code example:

```
#include "mplib.h"

int main (int argc, char ** argv)  {
        MP mp ;
        MP_options * opt = mp_options () ;
        opt -> command_line = NULL;
        opt -> noninteractive = 1 ;
        mp = mp_initialize ( opt ) ;
        my_try_path(mp);
        mp_finish ( mp ) ;
        free(opt);
        return 0;
}
```

Most of the example code above is exactly what one needs to do anything with MPlib programmatically. The only new line is the line calling `my_try_path(mp)`:

```
void my_try_path(MP mp) {
        mp_knot first, p, q;
        first = p = mp_append_knot(mp,NULL,0,0);
        q = mp_append_knot(mp,p,10,10);
        p = mp_append_knot(mp,q,10,-5);
        mp_close_path_cycle(mp, p, first);
        if (mp_solve_path(mp, first)) {
            mp_dump_solved_path(mp, first);
        }
        mp_free_path(mp, first);
}
```

This function uses a new type (`mp_knot`) as well as a bunch of new library functions in MPlib that exist since version 1.800.

- `mp_append_knot()` creates a new knot, appends it to the path that is being built, and returns it as the new tail of the path.
- `mp_close_path_cycle()` mimics `cycle` in the Metapost language.
- `mp_solve_path()` finds the control points of the path. `solve_path` does not alter the state of the used MPlib instance in away, it only modifies its argument path.
- `mp_dump_solved_path()` *user defined function, see below for its definition*
- `mp_free_path()` releases the used memory.

`mp_dump_solved_path` uses even more new functions. First let us look at its definition:

```
#define SHOW(a,b) mp_number_as_double(mp,mp_knot_##b(mp,a))
void mp_dump_solved_path (MP mp, mp_knot h) {
    mp_knot p, q;
    p = h;
    do {
        q = mp_knot_next(mp,p);
        printf ("(%g,%g)..controls (%g,%g) and (%g,%g)",
                SHOW(p,x_coord), SHOW(p,y_coord), SHOW(p,right_x),
                SHOW(p,right_y), SHOW(q,left_x), SHOW(q,left_y));
        p = q;
        if (p != h || mp_knot_left_type(mp,h) != mp_endpoint)
            printf ("\n ..");
    } while (p != h);
    if (mp_knot_left_type(mp,h) != mp_endpoint)
        printf("cycle");
    printf ("\n");
}
```

Somewhat hidden in the source above is that there is another new type: `mp_number`, the data structure representing a numerical value inside MPlib.

The used MPlib library functions are as follows:

- `mp_knot_next()` move to the next knot in the path.
- `mp_knot_x_coord()`, `mp_knot_y_coord()`, `mp_knot_right_x()`, `mp_knot_right_y()`, `mp_knot_left_x()`, `mp_knot_left_y()`
  return the value of a knot field, as a `mp_number` object (the calls to these functions are hidden inside the definition of the `SHOW` macro).
- `mp_knot_left_type()` returns the type of a knot, normally either `mp_endpoint` or `mp_open`.
- `mp_number_as_double()` converts a `mp_number` to `double`.

To satisfy our curiosity, here is the actual output of the example program listed above:

```
(0,0)..controls (-1.8685,6.35925) and (4.02429,12.1436)
 ..(10,10)..controls (16.8519,7.54208) and (16.9642,-2.22969)
 ..(10,-5)..controls (5.87875,-6.6394) and (1.26079,-4.29094)
 ..cycle
```

And that is almost exactly the same as in the log file:

```
(0,0)..controls (-1.8685,6.35925) and (4.02429,12.14362)
 ..(10,10)..controls (16.85191,7.54208) and (16.9642,-2.22969)
 ..(10,-5)..controls (5.87875,-6.6394) and (1.26079,-4.29094)
 ..cycle
```

The output is not perfectly the same because MetaPost itself does not use `mp_number_as_double` and `%g` for printing the `scaled` values that are by default used to represent numerical values.

The difference is not really relevant, since any programmatic use of the path solver should not have to be 100% compatible with the MetaPost programming language.

More complex paths

Of course there are also new functions to the create more complex paths that make use of `curl`, `tension` and/or `direction` specifiers.

Here is how to encode the second MetaPost path from the earlier example:

```
first = p = mp_append_knot(mp,NULL,0,0);
q = mp_append_knot(mp,p,2,20);
```

```
p = mp_append_knot(mp,q,10,5);
if (!mp_set_knotpair_curls(mp, q,p, 1.0, 1.0))
    exit ( EXIT_FAILURE ) ;
q = mp_append_knot(mp,p,3,10);
if (!mp_set_knotpair_controls(mp, p,q, 2.0, 2.0, 9.0, 4.5))
    exit ( EXIT_FAILURE ) ;
p = mp_append_knot(mp,q,1,14);
if (!mp_set_knotpair_tensions(mp,q,p, 3.0, -4.0))
    exit ( EXIT_FAILURE ) ;
q = mp_append_knot(mp,p,5,-4);
if (!mp_set_knotpair_directions(mp, p,q, 2.0, 0.0, 0.0, 1.0))
   exit ( EXIT_FAILURE ) ;
mp_close_path(mp, q, first);
```

Elaborate documentation for these extra functions (and a few more) is in `mplibapi.tex` which is included in the MetaPost distribution.

Lua interface

There is also a Lua interface for use in LuaTEX, which is a bit higher-level

```
<boolean> success = mp.solve_path(<table> knots, <boolean> cyclic)
```

This modifies the `knots` table (which should contain an array of points in a path, with the substructure explained below) by filling in the control points. The boolean `cyclic` is used to determine whether the path should be the equivalent of `--cycle`. If the return value is `false`, there is an extra return argument containing the error string.

On entry, the individual knot tables can contain the six knot field values mentioned above (but typically the `left_{x,y}` and `right_{x,y}` will be missing). `{x,y}_coord` are both required. Also, some extra values are allowed:

`left_tension`	number	A tension specifier
`right_tension`	number	like `left_tension`
`left_curl`	number	A curl specifier
`right_curl`	number	like `left_curl`
`direction_x`	number	x displacement of a direction specifier
`direction_y`	number	y displacement of a direction specifier

Issues to watch out for

All the 'normal' requirements for MetaPost paths still apply using this new interface. In particular

- A knot has either a direction specifier, or a curl specifier, or a tension specification, or explicit control points, with the additional note that tensions, curls and control points are split in a left and a right side (directions apply to both sides equally).
- The absolute value of a tension specifier should be more than 0.75 and less than 4096.0, with negative values indicating '`atleast`'.
- The absolute value of a direction or curl should be less than 4096.0.
- If a tension, curl, or direction is specified, any existing control points will be replaced by the newly computed value.

Taco Hoekwater