`PHquintic` Software Library: User Manual

Bohan Dong and Rida T. Farouki
Department of Mechanical and Aerospace Engineering,
University of California, Davis, CA 95616, USA

1 Software packages

Two software packages have been prepared. The first package, found in the file `PHquintic.c`, is written in plain `C` language and offers a set of modular functions that execute each of the basic PH curve construction and analysis computations described in the paper. The function prototypes are specified below. This package allows a software developer to "pick and choose" those functions of primary interest in a specific application context, and incorporate them in an existing software system with minimum effort. The main program accompanying these functions provides some sample data and function calls to test the various functions, but no graphics capability.

The second package, contained in the zip file `InteractivePHquintic`, is implemented in `C++` for compatibility with the MFC and OpenGL libraries, and provides interactive graphical construction, manipulation, and analysis capabilities (these libraries are required to compile and run the package). The unzipped package can be compiled and run by opening a `VC++` project file in Microsoft Visual Studio. The basic functions in `PHquintic.c` are included in this package, but the interfaces are modified to meet the requirements of `C++` and the MFC and OpenGL libraries. This package offers a more visual and intuitive user interface, but is perhaps less well–suited to the purpose of porting individual functions to an existing software system.

2 Function prototypes

The prototype for each function in `PHquintic.c` is described below.

2.1 PH quintic representation

The data defining a single PH quintic segment is encapsulated in the following `C` struct.

```
struct PHquintic {
complex double p[6] ; /* Bezier control points of PH quintic */
complex double w[3] ; /* Bernstein coefficients of w polynomial */
double sigma[5] ; /* parametric speed Bernstein coefficients */
double s[6] ; /* arc length Bernstein coefficients */
} ;
```

The complex arrays `p[6]` and `w[3]` store the Bézier control points $\mathbf{p}_0, \ldots, \mathbf{p}_5$ of the curve, and coefficients $\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2$ of the quadratic polynomial $\mathbf{w}(t)$. The real arrays `sigma[5]` and `s[6]` store the coefficients of the parametric speed and arc length polynomials, $\sigma(t)$ and $s(t)$. Note that this specification is redundant, since $\mathbf{p}_1, \ldots, \mathbf{p}_5$, $\sigma_0, \ldots, \sigma_4$, and $s_0, \ldots, s_5$ can be determined from $\mathbf{p}_0$ and $\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2$ alone. However, pre–computing the entire contents of a `PHquintic` struct can save considerable effort in subsequent usage of the PH quintic segment it defines.

2.2 PH quintic offset curve

The prototype for the function that computes the offset to a PH quintic segment is as follows. The struct `curve` passes the data that defines the PH quintic to the function, the value `d` defines the signed[1] offset distance, and the homogeneous coordinates of the control points for the degree 9 rational offset curve are returned in the arrays `W[]`, `X[]`, `Y[]`.

```
void PHquintic_offset( double d ,
                       struct PHquintic *curve ,
                       double W[] , double X[] , double Y[] )
```

2.3 PH quintic elastic bending energy

The prototype for the function that computes the bending energy of a PH quintic segment is as follows. The struct `curve` passes the data defining the PH quintic to the function, and the computed bending energy is returned as the value of the function.

```
double PHquintic_energy( struct PHquintic *curve )
```

2.4 First–order Hermite interpolants

The prototype for the function that constructs a PH quintic segment is as follows. The complex variables `p0`, `p1`, `p4`, `p5` pass the initial and final pairs of control points to the function, and the data defining the constructed PH quintic are returned in the struct `curve`.

[1] The offset distance is positive to the *right* of the curve $\mathbf{r}(t)$, relative to the sense of increasing parameter t.

```
void construct_PHquintic( complex double p0 ,
                          complex double p1 ,
                          complex double p4 ,
                          complex double p5 ,
                          struct PHquintic *curve )
```

2.5 C^2 PH quintic spline curves

Although the algorithms to compute open and closed C^2 PH quintic splines have much in common, they also differ in numerous details. To avoid the use of many conditional statements accommodating the end conditions, they are implemented in separate functions. The prototypes are as follows.

```
void open_PHquintic_spline( int n ,
                            complex double q[] ,
                            struct PHquintic spline[] )

void closed_PHquintic_spline( int n ,
                              complex double q[] ,
                              struct PHquintic spline[] )
```

Here the integer `n` defines the number of interpolation points (namely, `n`+1), passed to the function through the array of complex values `q[]` — these points are labelled `q[0]`,..., `q[n]`. The `n` PH quintic segments defining the constructed spline curve are returned through the struct array `spline[]` — these segments are labelled `spline[1]`,...,`spline[n]`.

The functions `tridiag_open()` and `tridiag_closed()` are called to solve the tridiagonal system arising in each Newton–Raphson iteration. The inputs to these functions are the dimension `n` of the system, arrays `a[]`, `b[]`, `c[]` defining the lower, main, and upper diagonal matrix elements, and the array `d[]` of right–hand side values. The solutions are returned in the array `x[]`.

The function `beval()` is another basic utility, that receives as input the degree `n` and array of Bernstein coefficients `b[]` of a polynomial, and an independent variable value `t`, and returns the polynomial value computed by the de Casteljau algorithm. The parameter `MAXDEGREE` sets that maximum polynomial degree that `beval()` can accommodate.

3 Interactive implementation

The interactive implementation allows the user to input the point data that defines a single PH quintic segment or a C^2 PH quintic interpolating spline by mouse, and to modify the resulting curve in real time by using the mouse to move these points. Key properties of the resulting PH curves (arc length, bending energy, etc.) are reported, and offset curves can also be constructed. For applications in which the point data must be precisely specified, the user can type in the point coordinates.

Individual PH quintic curve segments are defined by the `PHquintic` struct, and PH quintic splines are defined as arrays of these structs. Class `PlanarPH` defines basic functions for computing and analyzing these curves. To ensure a high level of precision, all complex variables in the class are type double.

```
PlanarPH(const CPointPH m_pt[]);
PlanarPH(const CPointPH m_pt[], int index);
```

The overloading constructors (for a single PH quintic Hermite interpolant and a PH quintic spline, respectively) convert the point coordinates specified by the user from type `CPointPH` to complex double. The `CPoint` type captured in the window display is long int, so the point type is redefined as `CPointPH`, which gives a representation of plane coordinates in type double.

The `beval()` function is used to plot PH curves. This function is over–loaded to evaluate polynomials with *complex* Bernstein coefficients, to obtain the curve points directly. The friend function is a non–member function, but can access the private and protected members. Friend utility functions are typically used to allow mutual access to private or protected members of different classes. It also allows other classes to invoke functions using a concise syntax — e.g., `iter = getIter(pph)`, rather than `iter = pph.getIter()`.

```
friend double beval(int n, const double b[], double t);
friend complex<double> beval(int n, const complex<double> b[],
                                                    double t);
```

The function `Spline()` computes a C^2 PH quintic spline interpolating `num`+1 points labelled $0, \ldots,$ `num` under specified end conditions. The coordinates of these points are stored in the complex array `q[]`. The data that defines the `num` PH quintic segments of the constructed spline curve are returned in the `PHquintic` struct array `spline[]`. The Boolean parameter `closed` specifies

the end conditions (cubic end spans or periodic end conditions, for open and closed curves, respectively). To obtain a closed C^2 spline, the user types "c" after entering the last point with the mouse. The functions `tridiag_open()` and `tridiag_closed()` solve the tridiagonal systems for these two cases.

```
void Spline(BOOL closed);
friend void tridiag_open(...);
friend void tridiag_closed(...);
```

The following functions define various interfaces for passing data. External functions must call `getCtrlPt` to obtain the complex values that define the control points. `getIter` returns the number of Newton–Raphson iterations employed in the spline construction, stored in the variable `iter`. Similarly, the functions `getParaSpeed`, `getArcLength`, and `getEnergy` compute the parametric speed, arc length, and bending energy.

```
friend complex<double> getCtrlPt(const PlanarPH &pph, int i, int j);
friend int getIter(const PlanarPH &pph);
friend double getParaSpeed(const PlanarPH &pph, int i, double t);
friend double getArcLength(const PlanarPH &pph, int i, double t);
friend double getEnergy(const PlanarPH &pph, int j);
```

In order to plot the constructed PH curves on the screen, OpenGL must first be initialized in the view class of MFC — this is named `PlanarPH_mfcView`, inherited from the base class `CView`.

```
int index, status, pointing;
CPointPH o_ActRF, o_ActRFTrans;
CPointPH ptActRF[Max+1];
CPointPH getAbsCoord(CPointPH pt);
CPointPH getActCoord(CPointPH pt);
double Distance(CPointPH p1, CPointPH p2);
```

The variable `index` (=`num`+1) records the number of input points, and `status` tracks the program mode (plotting, analyzing, translating, etc). The variable `pointing` identifies a point selected by the user after the curve is plotted, and the selected point is highlighted (`pointing` $= -1$ if no point is selected).

The program employs two coordinate systems, absolute and relative, to solve translation and display problems. For absolute coordinates, the window

display selects the left upper window corner as the origin, with a downward ordinate direction. However, OpenGL selects the left lower corner as origin, with an upward ordinate direction. In both cases, the abscissa direction is to the right. The origin of the relative coordinates, in the absolute coordinate system, is defined by `o_ActRF`. All calculations are performed in the relative coordinates, including cursor location and PH curve computations. The array `ptActRF[]` stores the relative coordinates of the input points. By calling the functions `getAbsCoord()` and `getActCoord`, the coordinates of any point can be transformed between the relative and absolute systems. The function `Distance()` returns the distance between two points.

```
PlanarPH pph;
void InitPH();
void DrawLine(int nIndex, int flag);
```

Before rendering, the function `InitPH()` should be called in most cases to initialize `pph`, which is an object of class `PlanarPH`. The plotting function `DrawLine` is called to plot a single point, a whole curve, or an offset, according to whether `flag` is equal to `plotPoint`, `plotSpline`, or `plotOffset`. `nIndex` indicates the number of points to be plotted when `flag` equals `plotPoint`. Otherwise, `nIndex` is the number of the last point which should be plotted.

```
BOOL isMenuPh, isMenuCubic, isMenuPhCtrl, isMenuCubicCtrl;
//Status of Menu
BOOL isLBDown;
CPointPH LBDownPt;
GLfloat margin;
```

The program also offers the ability to compare the C^2 PH quintic spline and the "ordinary" C^2 cubic spline with analogous end conditions, and the user may control the display status of each spline. The variables `isMenuPh`, `isMenuCubic`, `isMenuPhCtrl`, and `isMenuCubicCtrl` monitor the status of the menu. They are TRUE if the corresponding menu is checked, and the program then calls the `Drawline` function to plot the spline curves and their control polygons. `isLBDown` and `LBDownPt` specify the status of the left mouse button, and record the coordinates of the point where it is pressed down.

Color List offers an efficient and flexible means to render the scene in the window. With a pointer to an array that contains red, green, and blue (RGB)

values, OpenGL functions such as `glColor3fv` make the current brush color as needed. Each color in the Color List should be described in RGB format, with three floats between 0 and 1 defining the red, green, and blue values. In order to display the points and lines smoothly on the screen, the program uses OpenGL functions to enable the blending option and smooth option.

As the Graphical User Interface (GUI) for the program, the window has a title bar, menu bar, plotting area, and a status bar. The status bar shows the cursor coordinates before plotting, the iteration number, the arc length and bending energy, and the selected point after plotting. Each menu option has an accelerator (i.e., an alternative keyboard input: see Table 1). In the Edit menu, the user can plot a Hermite interpolant or spline curve, edit multiple points, translate the curve, and construct an offset. The program offers two approaches to editing points: right–clicking on a single point to change it, or selecting Edit Multiple Points in the menu to change all the points.

New Spline	N
Edit Multiple Points	ctrl+E
Translate Spline	T
Offset	O
Hermite	H
PH	P
PH(Ctrl)	ctrl+P
Cubic	C
Parametric Speed	shift+A
Arc Length	A

Table 1: Accelerator keyboard inputs for menu items.

Figures 1–5 present examples of the program in use. Figure 1 illustrates the construction of a single PH quintic Hermite interpolant from initial and final pairs of control points specified by the user. Figure 2 shows a planar C^2 PH quintic spline interpolating a sequence of points freely selected by the user with the mouse. For this plot, the View menu options PH and PH(Ctrl) have been checked, so the control polygons of each PH quintic spline segment are also shown. Figure 3 presents the same C^2 PH quintic spline, but in this case the View menu options PH and Cubic are checked, to compare the PH spline with the ordinary cubic spline — note the rather poor shape of the latter, as compared to the former.

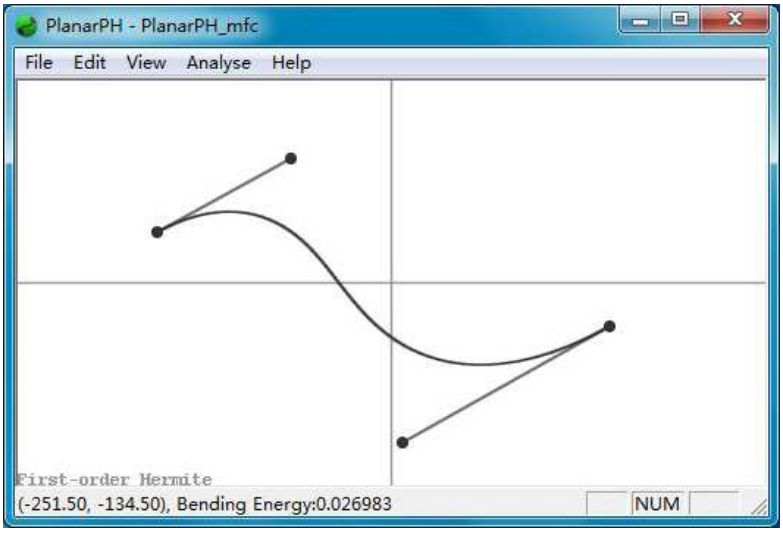

Figure 1: A single PH quintic segment, constructed as a Hermite interpolant from user mouse input specified as initial and final pairs of control points.

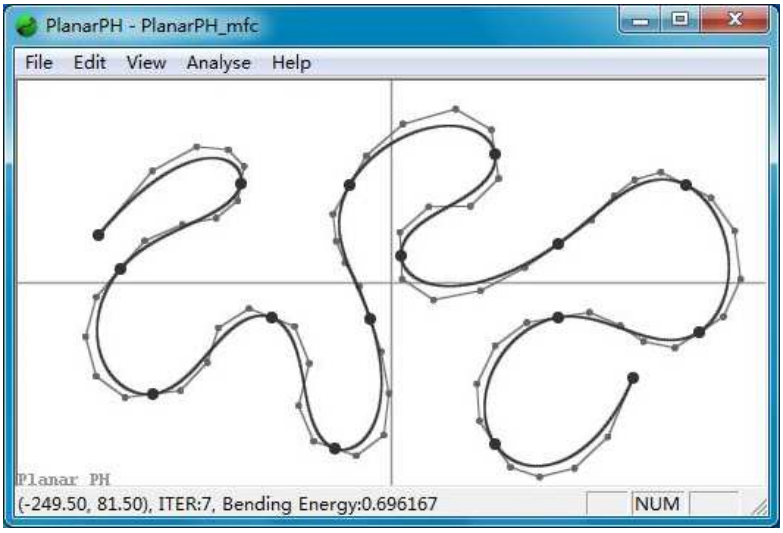

Figure 2: An example of an open C^2 PH quintic spline curve interpolating a sequence of points (large dots) specified interactively with the mouse. The control polygons for each of the PH quintic spline segments are also shown.

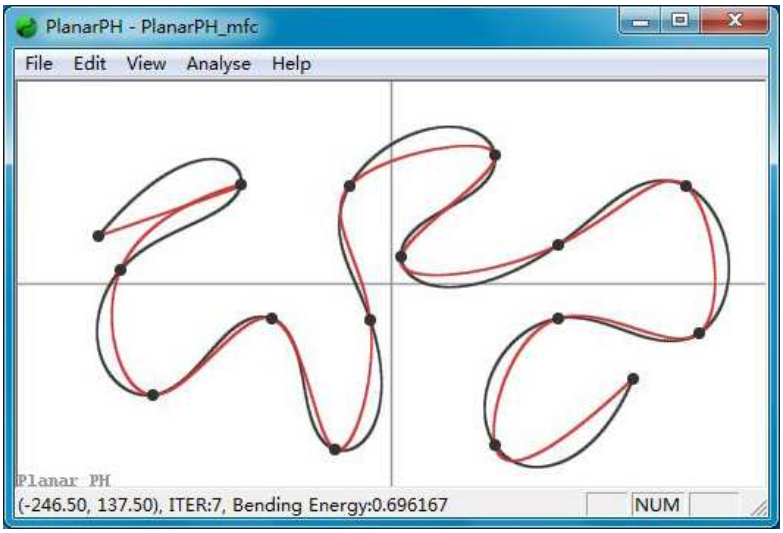

Figure 3: Comparison of the C^2 PH quintic spline (black curve) in Figure 2 and the "ordinary" C^2 cubic spline (red curve) interpolating the same points with uniform parameterizations, and equivalent end conditions (quadratic end spans for the cubic spline, and cubic end spans for the PH quintic spline).

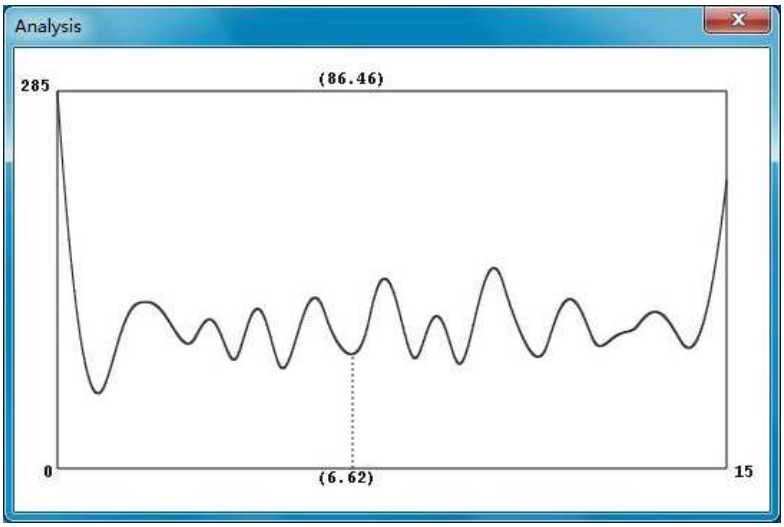

Figure 4: Parametric speed plot for the C^2 PH quintic spline in Figure 2.

In Figure 4, the parametric speed variation for the C^2 PH spline shown in Figure 1 is plotted. Finally, Figure 5 shows the construction of offsets to a C^2 PH quintic spline, for several (positive and negative) values of the offset distance d. The offsets can be cleared from the display by setting $d = 0$.

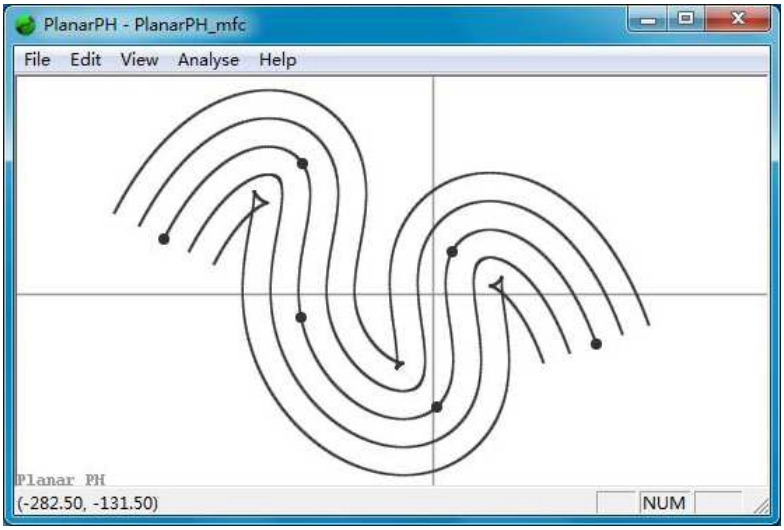

Figure 5: Construction of the rational offsets to a planar C^2 PH quintic spline curve for several — positive and negative — values of the offset distance d.