Numerical Analysis – Lab 6

1 Hermite interpolation

Implementation of Newton's formula for Hermite interpolation Recall that for Hermite interpolation we need only to replace the first divided differences at the duplicated points with the given derivatives. More generally, we allow at some points only the function values but not the derivatives are specified. The table of divided differences can be constructed as follows:

```
function [X,A] = hermitediv(x,y,yp,ip)
%yp =y'(x(ip))
i = 1:numel(x); [i,I] = sort([i ip]);
X = x(i); Y = y(i); X = X(:); Y = Y(:);
n = numel(X);
A = zeros(n,n) + 0*X(1);
A(:,1) = Y(:);
A(:,2) = [0; diff(y(:))./diff(x(:)); yp(:)];
A(:,2) = A(I,2);
for k = 3:n
    A(k:end,k) = (A(k:end,k-1)-A(k-1:end-1,k-1))./(X(k:end)-X(1:end-k+1));
end
```

With the table of divided differences, we then evaluate the interpolating polynomial at any given points, as well as the derivatives thereof. This can be done with Horner's rule. For example, Newton's formula results a polynomial in the form:

$$p_3(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2)$$

where x_0, x_1, x_2, x_3 may have duplicates for the Hermite interpolation. Horner's rule computes

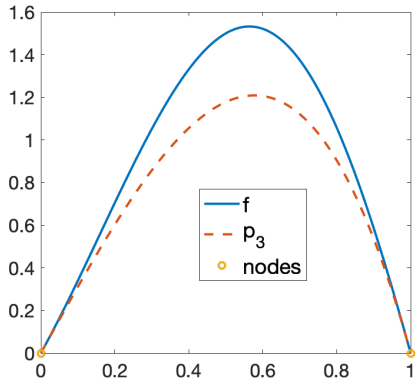
$$\begin{aligned}
p_0(x) &= a_3, & p_0'(x) &= 0 \\
p_1(x) &= a_2 + (x - x_2)p_0(x), & p_1'(x) &= p_0(x) + (x - x_2)p_0'(x) \\
p_2(x) &= a_1 + (x - x_1)p_1(x), & p_2'(x) &= p_1(x) + (x - x_1)p_1'(x) \\
p_3(x) &= a_0 + (x - x_0)p_2(x), & p_3'(x) &= p_2(x) + (x - x_0)p_2'(x).
\end{aligned}$$

The recursive process is implemented as follows:

```
function [yy,yyp] = newtoninterph(x,a,xx)
% Newton interpolation from the divided differences 'a'.
% Return also the derivative values.
n = numel(x)-1;
if n~=numel(a)-1
    error('newtoninterph: x and a are of different length!');
end
yy = a(end)*ones(size(xx)); yyp = 0*yy;
for k=n:-1:1
    yyp = yy+(xx-x(k)).*yyp;
    yy = a(k)+(xx-x(k)).*yy;
end
```

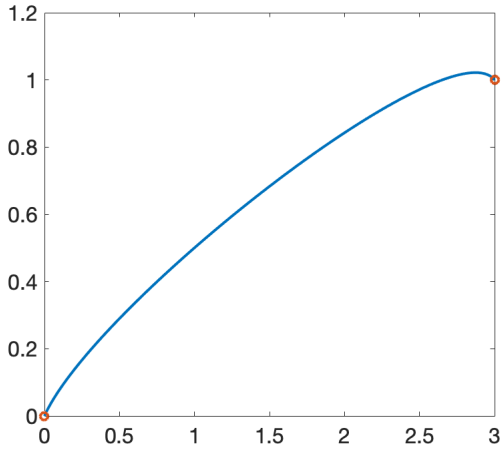
Let us try the Hermite interpolation for $f(x) = (1+x)\sin(\pi x)$ in the nodes $\{0, 1\}$. Clearly, the polynomial preserves the tangents of f at the nodes.

```
x = [0; 1];
f = @(x) (1+x).*sin(pi*x);
fp = @(x) sin(pi*x)+pi*cos(pi*x).*(x+1)
xx = 0:0.001:1;
y = f(x); yp = fp(x);
ip = 1:2;
[X, A] = hermitediv(x,y,yp,ip);
[yy,yyp] = newtoninterph(X,diag(A),xx);
plot(xx,f(xx),xx,yy,'--',x,y,'o','linewidth',2)
```



Key frame interpolation. To create an annimation from a few pictures, it is needed to generate the pictures at the intermediate times between two consecutive scenes. Essentially, a coordinate x_0 at the inital time $t = 0$ is known, as well as x_1 at the final time $t = 1$. A continuous function $x(t)$ is sought to satisfy $x(0) = x_0$ and $x(1) = x_1$. The simplest model for $x(t)$ assumes a constant velocity, which gives $x(t) = (1-t)x_0 + tx_1$. A better model specifies also the initial and final velocities: $x'(0) = x'_0$ and $x'(1) = x'_1$, which gives rise to the Hermite interpolation problem. For example, we would like to animate a basketball being shooted. For simplicity, suppose the x coordinates remains the same, the initial (y, z)-coordinates are $(0, 0)$ and the final $(3, 1)$, and the initial velocity in (y, z)-coordinates is $(1, 1)$ and the the final $(0.9, -0.5)$. We make a video as follows. You can checkout "basket.mp4" after running the code. You may try to adjust the time step size, the initial and final speeds to make the video more real. A better solution, not to be discussed here, would be a simulation based on physics!

```
t = [0; 1]; y = [0; 3]; z = [0; 1];
yp = [1; 0.9]; zp = [1; -0.5]; ip = 1:2;
tt = 0:0.01:1;
[T, Ay] = hermitediv(t,y,yp,ip);
[yy,yyp] = newtoninterph(T,diag(Ay),tt);
[T, Az] = hermitediv(t,z,zp,ip);
[zz,zzp] = newtoninterph(T,diag(Az),tt);
plot(yy,zz,y,z,'o','linewidth',2);
vfile = VideoWriter('basket.mp4','MPEG-4');
open(vfile);
for i = 1:length(tt)
   plot(yy(i),zz(i),'o'); axis([0 3 0 1.2]);
   drawnow; writeVideo(vfile,getframe);
end
close(vfile);
```



A realistic application would be more complicated, in which an object may change its profile during the motion; see the video linked on the cover of lecture 6 slides. That requires a representation of the profile e.g. by piecewise straight line segments or periodic cubic splines, and identification of the same material points at the initial and final times.

Osculating splines for motion interpolation (optional). The goal here is very similar to the key frame interpolation, but suppose the initial and finishing accelerations are also known. For simplicity, people still prefer to use cubic polynomials in this case. So the 6 conditions from the positions, velocities and accelerations at the initial and final times are too many for a single cubic polynomial. One solution is to split the time interval into 3 pieces and look for the cubic splines which require totally 12 conditions for interpolation. Indeed, the continuity of positions, velocities and accelerations at the 2 intermediate nodes in the interval provide perfectly the additional 6 conditions. Let $\mathbf{r} = \mathbf{r}(t) = (x(t), y(t))$ be the coordinates at time t and suppose $t \in [0, 1]$, $\mathbf{r}(0) = \mathbf{0}$, $\mathbf{r}(1) = (x_1, y_1)$, $\mathbf{r}'(0) = (x'_0, y'_0)$, $\mathbf{r}'(1) = (x'_1, y'_1)$,

$\mathbf{r}''(0) = (x_0'', y_0'')$, $\mathbf{r}''(1) = (x_1'', y_1'')$. The interpolation problem for

$$x(t) = \begin{cases} x_0' t + \frac{1}{2} x_0'' t^2 + d_1 t^3 & \text{if } t \in [0, t_1] \\ a_2 + b_2(t - t_1) + c_2(t - t_1)^2 + d_2(t - t_1)^3 & \text{if } t \in [t_1, t_2] \\ x_1 + x_1'(t-1) + \frac{1}{2} x_1''(t-1)^2 + d_3(t-1)^3 & \text{if } t \in [t_2, 1] \end{cases}$$

is to find d_1,a_2, b_2, c_2, d_2, and d_3 such that

$$\begin{aligned} x_0' t_1 + \frac{1}{2} x_0'' t_1^2 + d_1 t_1^3 &= a_2 \\ x_0' + x_0'' t_1 + 3 d_1 t_1^2 &= b_2 \\ x_0'' + 6 d_1 t_1 &= 2 c_2 \\ a_2 + b_2(t_2 - t_1) + c_2(t_2 - t_1)^2 + d_2(t_2 - t_1)^3 &= x_1 + x_1'(t_2 - 1) + \frac{1}{2} x_1''(t_2 - 1)^2 + d_3(t_2 - 1)^3 \\ b_2 + 2 c_2(t_2 - t_1) + 3 d_2(t_2 - t_1)^2 &= x_1' + x_1''(t_2 - 1) + 3 d_3(t_2 - 1)^2 \\ 2 c_2 + 6 d_2(t_2 - t_1) &= x_1'' + 6 d_3(t_2 - 1). \end{aligned}$$

We can substitute the first three equations into the last three equations to get

$$\begin{aligned} (t_1^3 - 3 t_1^2 t_2 + 3 t_1 t_2^2) d_1 + (t_2 - t_1)^3 d_2 + (1 - t_2)^3 d_3 &= -x_0' t_2 - \frac{1}{2} x_0'' t_2^2 + x_1 + x_1'(t_2 - 1) + \frac{1}{2} x_1''(t_2 - 1)^2 \\ (-3 t_1^2 + 6 t_1 t_2) d_1 + 3(t_2 - t_1)^2 d_2 - 3(1 - t_2)^2 d_3 &= -x_0' - t_2 x_0'' + x_1' + x_1''(t_2 - 1) \\ 6 t_1 d_1 + 6(t_2 - t_1) d_2 + 6(1 - t_2) d_3 &= -x_0'' + x_1''. \end{aligned}$$

From the above system we can solve out d_1, d_2, d_3 and then get a_2, b_2, c_2. The idea is implemented as follows. First, solved out the coefficients:

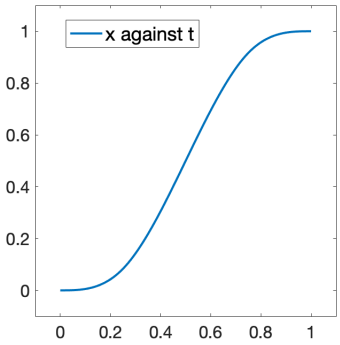
```
function [a, b, c, d] = motionspline(t12,x1,xp01,xpp01)
t1 = t12(1); t2 = t12(2);
xp0 = xp01(1); xp1 = xp01(2);
xpp0 = xpp01(1); xpp1 = xpp01(2);
A = [t1^3-3*t1^2*t2+3*t1*t2^2  (t2-t1)^3  (1-t2)^3;
     -3*t1^2+6*t1*t2  3*(t2-t1)^2  -3*(1-t2)^2;
     6*t1  6*(t2-t1)  6*(1-t2)];
rhs = [-xp0*t2-1/2*xpp0*t2^2+x1+xp1*(t2-1)+1/2*xpp1*(t2-1)^2;
       -xp0-t2*xpp0+xp1+xpp1*(t2-1);
       -xpp0 + xpp1];
d = A\rhs;
a = xp0*t1+1/2*xpp0*t1^2+d(1)*t1^3;
b = xp0+xpp0*t1+3*d(1)*t1^2;
c = 1/2*xpp0 + 3*d(1)*t1;
```

Then, use the coefficients to evaluate the splines:

```
function x = motionspline_eval(t12,x1,xp01,xpp01,a,b,c,d,t)
x = zeros(size(t));
t1 = t12(1); t2 = t12(2);

i = (t>=0) & (t<=t1);
x(i) = xp01(1)*t(i) + 1/2*xpp01(1)*t(i).^2 + d(1)*t(i).^3;

i = (t>=t1) & (t<=t2);
x(i) = a + b*(t(i)-t1) + c*(t(i)-t1).^2 + d(2)*(t(i)-t1).^3;

i = (t>=t2) & (t<=1);
x(i) = x1 + xp01(2)*(t(i)-1) + 1/2*xpp01(2)*(t(i)-1).^2 + d(3)*(t(i)-1).^3;
```

We try the natural move on a line with zero velocity and zero accleration at both the initial and final positions.

```
t12 = [0.25 0.75]; x1 = 1; xp01 = [0 0]; xpp01 = [0 0];
[a, b, c, d] = motionspline(t12,x1,xp01,xpp01);
t = 0:0.01:1;
x = motionspline_eval(t12,x1,xp01,xpp01,a,b,c,d,t);
plot(t,x,'linewidth',2); figure();
for j=1:length(t)
   plot(x(j),0,'o'); axis([-0.2 1.2 -0.1 0.1]); drawnow;
end
```



We can see the point gradually accelerates in the starting phase and decelerates in the finishing phase.

2 Bézier curves (optional)

To draw a curve between two points is a very basic step in graph or shape design. In lieu of completely drawing by hand, it is very helpful for the designer to specify some key information and let the computer generate the curve automatically and rapidly. The tangents of the curve at two ends are essential information to be determined by a designer. More precisely, let the curve be $\mathbf{r} = \mathbf{r}(t) = (x(t), y(t))$. The key information is $\mathbf{r}(0)$, $\mathbf{r}(1)$ and $\mathbf{r}'(0)$, $\mathbf{r}'(1)$, which determines by the Hermite interpolation a unique cubic curve (i.e. both $x(t)$, $y(t)$ are cubic interpolating polynomials). However, it is not intuitive for a designer to tell the value of $\mathbf{r}'(0)$ or $\mathbf{r}'(1)$.

It was in the 1950's that the mathematician Paul de Casteljau working for the French automaker Citroën and the engineer Pierre Bézier working for the French automaker Renault got independently the idea of using Bernstein's polynomials to design automobile bodies.

"At that time, the shape of a mechanical part was mainly defined by lines and circles, because draftsmen and designers used straight edges and compasses. Other surfaces were broadly expressed by cross-sections ..."

Pierre Etienne Bézier (1910-1999)

"Nobody in 1958 could have foreseen that one day mathematicians would be looking for the equation of an artistic or aesthetic form. Until then, they limited themselves to draw the representative curves of the equations ..."

Paul de Faget de Casteljau (1930-2022)

Recall that Bernstein's polynomial for approximating $f \in C[0,1]$ is

$$\sum_{k=0}^{n} \binom{n}{k} (1-t)^{n-k} t^k f\left(\frac{k}{n}\right).$$

Let us now replace $f(k/n)$ with the k-th point $\mathbf{r}_k = (x_k, y_k)$. Then

$$\mathbf{B}_n(t) = \sum_{k=0}^{n} \binom{n}{k} (1-t)^{n-k} t^k \mathbf{r}_k$$

is the parametric curve for $t \in [0,1]$, named after Bézier. In particular, when $n = 1$, the Bézier curve

$$\mathbf{B}_1(t) = (1-t)\mathbf{r}_0 + t\mathbf{r}_1$$

is the straight line segment connecting the points $\mathbf{r}_0$ and $\mathbf{r}_1$. When $n = 2$, the Bézier curve

$$\mathbf{B}_2(t) = (1-t)^2\mathbf{r}_0 + 2(1-t)t\mathbf{r}_1 + t^2\mathbf{r}_2$$

is a parabola segment connecting $\mathbf{r}_0$ and $\mathbf{r}_2$, and moreover

$$\mathbf{B}_2'(t) = 2(1-t)(\mathbf{r}_1 - \mathbf{r}_0) + 2t(\mathbf{r}_2 - \mathbf{r}_1)$$

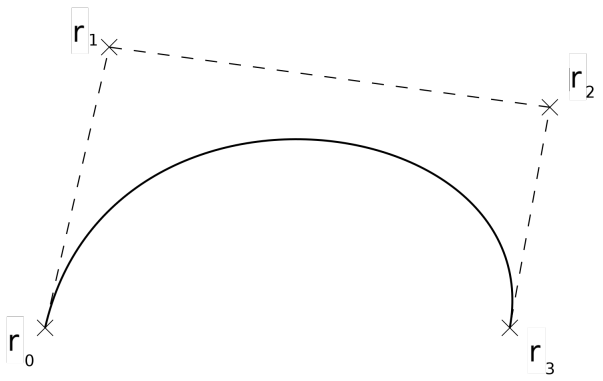
which implies that at $\mathbf{r}_0$ the tangent of the curve is from $\mathbf{r}_0$ to $\mathbf{r}_1$ (and at $\mathbf{r}_2$ from $\mathbf{r}_1$ to $\mathbf{r}_2$). The most used case is $n = 3$, and the cubic Bézier curve

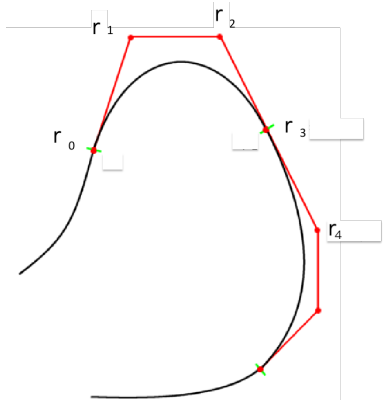
$$\mathbf{B}_3(t) = (1-t)^3\mathbf{r}_0 + 3(1-t)^2t\mathbf{r}_1 + 3(1-t)t^2\mathbf{r}_2 + t^3\mathbf{r}_3$$

is controlled by the 4 points $\mathbf{r}_0$, $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathbf{r}_3$, where $\mathbf{r}_0$ is the starting point, $\mathbf{r}_3$ the finishing point, and the direction (and speed with respect to the parameter t) of the curve at $\mathbf{r}_0$, $\mathbf{r}_3$ are determined by $\mathbf{r}_1$, $\mathbf{r}_2$, respectively:

$$\mathbf{B}_3'(t) = 3(1-t)^2(\mathbf{r}_1 - \mathbf{r}_0) + 6(1-t)t(\mathbf{r}_2 - \mathbf{r}_1) + 3t^2(\mathbf{r}_3 - \mathbf{r}_2),$$

because $\mathbf{B}_3(0) = \mathbf{r}_0$, $\mathbf{B}_3'(0) = 3(\mathbf{r}_1 - \mathbf{r}_0)$, and $\mathbf{B}_3(1) = \mathbf{r}_3$, $\mathbf{B}_3'(1) = 3(\mathbf{r}_3 - \mathbf{r}_2)$. It is then very easy for the designer to manipulate the curve by placing the control points $\mathbf{r}_1$, $\mathbf{r}_2$ without looking into the numbers for derivatives. It is also straightforward to continue one curve with another smoothly (continuous derivative): just put the new control point $\mathbf{r}_4$ along the line of $\mathbf{r}_3$ and $\mathbf{r}_2$.

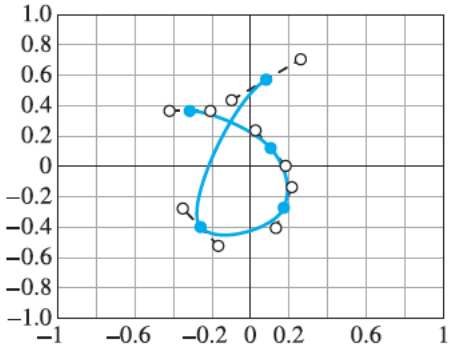




You can try the following program from Timothy Sauer's textbook *Numerical Analysis*.

```
function bezierdraw
plot([-1 1],[0,0],'k',[0 0],[-1 1],'k');hold on
t=0:.02:1;
[x,y]=ginput(1);              % get one mouse click
while(0 == 0)
  [xnew,ynew] = ginput(3);  % get three mouse clicks
  if length(xnew) < 3
    break                     % if return pressed, terminate
  end
  x=[x;xnew];y=[y;ynew];     % plot spline points and control pts
  plot([x(1) x(2)],[y(1) y(2)],'r:',x(2),y(2),'rs');
  plot([x(3) x(4)],[y(3) y(4)],'r:',x(3),y(3),'rs');
  plot(x(1),y(1),'bo',x(4),y(4),'bo');
  bx=3*(x(2)-x(1)); by=3*(y(2)-y(1)); % spline equations ...
  cx=3*(x(3)-x(2))-bx;cy=3*(y(3)-y(2))-by;
```

```
  dx=x(4)-x(1)-bx-cx;dy=y(4)-y(1)-by-cy;
  xp=x(1)+t.*(bx+t.*(cx+t*dx));    % Horner's method
  yp=y(1)+t.*(by+t.*(cy+t*dy));
  plot(xp,yp,'b')                  % plot spline curve
  x=x(4);y=y(4);                 % promote last to first and repeat
end
hold off
```



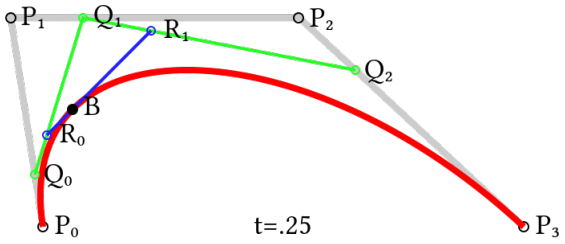
Casteljau discovered a recursive algorithm that is numerically stable for evaluating Bernstein's approximating polynomials and thus the Bézier curves. For example, the cubic Bézier curve $\mathbf{B}_{0,1,2,3}(t)$ with the control points $\mathbf{r}_0$, $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathbf{r}_3$ can be evaluated as follows:

$$\mathbf{B}_{0,1}(t) = (1-t)\mathbf{r}_0 + t\mathbf{r}_1, \quad \mathbf{B}_{1,2}(t) = (1-t)\mathbf{r}_1 + t\mathbf{r}_2, \quad \mathbf{B}_{2,3}(t) = (1-t)\mathbf{r}_2 + t\mathbf{r}_3;$$

$$\mathbf{B}_{0,1,2}(t) = (1-t)\mathbf{B}_{0,1}(t) + t\mathbf{B}_{1,2}(t), \quad \mathbf{B}_{1,2,3}(t) = (1-t)\mathbf{B}_{1,2}(t) + t\mathbf{B}_{2,3}(t);$$

$$\mathbf{B}_{0,1,2,3}(t) = (1-t)\mathbf{B}_{0,1,2}(t) + t\mathbf{B}_{1,2,3}(t).$$

The procedure is illustrated in the following figure



where P_0, P_1, P_2 and P_4 are the control points, Q_0, Q_1 and Q_2 are $\mathbf{B}_{0,1}(t)$, $\mathbf{B}_{1,2}(t)$ and $\mathbf{B}_{2,3}(t)$, respectiely, R_0, R_1 are $\mathbf{B}_{0,1,2}(t)$, $\mathbf{B}_{1,2,3}(t)$, and B is $\mathbf{B}_{0,1,2,3}(t)$.