Math 128a - Programming Project 2 - Answers

We provide

1. our implementations of the subroutines you had to supply,
2. a discussion of periodic splines, and
3. outputs and discussion.

- **FindCorners.m** is self-explanatory. It depends on the function **IntAngle.m**, which given three lists of points $p1$,$p2$,$p3$, returns the angle between the vectors $(p2 - p1)$ and $(p2 - p3)$, measured between 0 and π. It works by writing each 2-dimensoinal vector as a complex complex number (write them as $r_{21}e^{i\theta_{21}}$ and $r_{23}e^{i\theta_{23}}$ in polar coordinates), takes their ratio $(r_{21}/r_{23}e^{i(\theta_{21}-\theta_{23})})$, and extracts the "phase" of this complex number $(\theta_{21} - \theta_{23})$ using the built-in Matlab **angle** function.

```
% ------------------------  FindCorners.m  ----------------------
%
% function [CornersFound] = FindCorners(x,y,threshhold,closed)
%
% inputs: x(1:n),y(1:n) are the coordinates of n points on a
%         curve,
%         0 < threshold < pi is the angle defining a corner: if
%         the interior angle theta between succesive segments in
%         the polygon satisfies theta<=pi-threshhold then
%         the point is declared a corner,
%         closed = 1 if the curve is to be considered a closed curve,
%                  0 if not.
% Note: at this point we do not assume x(n)==x(1) or y(n)==y(1) for
% closed curves.
%
% outputs: CornersFound: a list, in order, of indices into x,y indicating
%          points identified as corners.
%
% This function depends on the function "IntAngle"
%

function [CornersFound] = FindCorners(x,y,threshhold,closed)

CornersFound=[];
if (threshhold<=0)|(threshhold>=pi)
  display('Error, threshhold out of range'); return;
end
if length(x)~=length(y)
  display('Error, x,y must have same length'); return;
end
```

```
n=length(x);
if closed==1 % locally extend x and y at both ends
  x=[x(n),x,x(1)];y=[y(n),y,y(1)];
end
m = length(x);
theta = IntAngle([x(1:m-2);y(1:m-2)],[x(2:m-1);y(2:m-1)],[x(3:m);y(3:m)]);
if closed==0
  CornersFound=[1]; offset=1;
else
  CornersFound=[]; offset=0;
end
for k=1:m-2
  if theta(k)<=pi-threshhold
    CornersFound = [CornersFound,k+offset];
  end
end
if closed==0
  CornersFound = [CornersFound,n];
end
```

```
% ------------------  IntAngle.m  -------------------
%
% function [theta] = IntAngle(p1,p2,p3)
%
% inputs: p1(1:2;1:n),p2(1:2;1:n),p3(1:2;1:n) lists of triples of
%         points
% outputs: theta(1:n) list of the interior angles given by the triples:
%
% the interior angle is the angle from the vector (p2-p1) to the vector
% (p2-p3), measured without respect to orientation so as to be between
% 0 and pi, where pi means the two vectors are collinear but point in
% opposite directions and 0 means they are collinear but point in the
% same direction.
%

function [theta] = IntAngle(p1,p2,p3)

p1_to_p2 = p2-p1;
p3_to_p2 = p2-p3;
z_p1_to_p2 = p1_to_p2(1,:) + i.*p1_to_p2(2,:);
z_p3_to_p2 = p3_to_p2(1,:) + i.*p3_to_p2(2,:);
theta = abs(angle(z_p3_to_p2./z_p1_to_p2));
```

- **ClosedSpline2d.m**. This function first computes the necessary cumulative arclengths s and then invokes the periodic spline algorithm twice to compute the coefficients of the splines describing x and y as *periodic functions* of arclength. It calls **PeriodicSpline.m**, which implements the periodic spline algorithm described below. The only subtlety is in the maintenance of indices "mod n".

```
% ---------------------  ClosedSpline2d.m  -----------------------
%
% function [xc,yc,s] = ClosedSpline2d(x,y)
%
% inputs x(1:n),y(1:n). Require that x(n)==x(1), y(n)==y(1)
% outputs xc(1:4;1:n-1),yc(1:4;1:n-1), s
%
% Computes the coefficients for the 2-d closed curve Spline as
% described in the programming assignment. First computes
% the arclengths s(1:n) and then applies PeriodicSpline to
% both (s,x) and (s,y) to get coefficient matrices xc and yc
% (see ClosedSpline.m)
%

function [xc,yc,s] = ClosedSpline2d(x,y)

n = length(x);
if length(y)~=n
  c=[]; display('Error, x and y must have the same lengths'); return;
end
s=[0];
for i=2:n
  s=[s,s(i-1)+sqrt((x(i)-x(i-1))^2+(y(i)-y(i-1))^2)];
end
[xc]=PeriodicSpline(s,x);
[yc]=PeriodicSpline(s,y);
```

```
% --------------  PeriodicSpline.m  ---------------------
%
% function [c] = PeriodicSpline(t,y)
%
% inputs: t(1:n+1), y(1:n+1) : require that y(n+1)==y(1)
% outputs: c(1:4;1:n)
%
% following the notation in the book combined with the notation
% in the programming assignment, computes a "periodic spline"
% with "knots" at t(1),...,t(n+1) with y-values y(1),...,y(n+1)
% and returns the coefficients in the
% 4-by-n matrix c. See also NaturalSpline.m
%

function [c] = PeriodicSpline(t,y)

n = length(t)-1; c=[];
if n<1 display('error: need at least two knots'); return; end
% set up indices mod n - these look a little weird because
% matlab indices start at 1.
ind = 1:n;
indplus1 = mod(ind,n)+1;
indminus1 = mod(ind-2,n)+1;
if length(y)~=n+1
  display('error: t and y have incompatible lengths'); return;
end
% calculate values top of p. 358
h = t(2:n+1)-t(1:n); % h(i)=t(i+1)-t(i), 1<=i<=n
u = 2*(h(ind)+h(indminus1)); % u(i) as in the book, indices mod n
b = (6./h).*(y(indplus1)-y(ind)); % b(i) as in book, indices mod n
v = b(ind)-b(indminus1); % v(i) as in book, indices mod n
% put into a matrix S(1:n;1:n) and compute z(1:n)
S = diag(u,0)+diag(h(1:n-1),1)+diag(h(1:n-1),-1)+...
    diag(h(n),n-1)+diag(h(n),1-n);
z = (S\(v'))';
% calculate the coefficients and end
c3 = z./(6.*h);
c2 = z(indplus1)./(6.*h);
c1 = (y(indplus1)./h) - (z(indplus1).*h)/6;
c0 = (y(1:n)./h) - (z.*h)/6;
c = [c0;c1;c2;c3];
```

Discussion of periodic splines:

The way to think about achieving the correct smoothness when going around a closed curve is to imagine the curve parametrized by two *periodic* functions x and y of arclength. Thus we reduce to the problem of coming up with coefficients describing a periodic spline function, given infinitely many knots and values satisfying various periodicity conditions. Thus, abstractly, we are given infinitely many knots

$$\ldots < t_{-1} < t_0 < t_1 < \ldots < t_{n-1} < t_n < t_{n+1} < \ldots$$

and infinitely many values

$$\ldots y_{-1}, y_0, y_1, \ldots, y_{n-1}, y_n, y_{n+1}, \ldots$$

satisfying the following periodicity conditions:

$$\begin{aligned} y_{i+n} &= y_i \\ t_{i+n} &= t_i + T \end{aligned}$$

for *all* integers i and some given period T. We are trying to find a function $S(x)$, defined by infinitely many cubic polynomials $S_i(x)$, satisfying the following relations:

$$\begin{aligned} S(x) &= S_i(x) \quad \text{when } x \in [t_i, t_{i+1}] \\ S_{i-1}(t_i) &= y_i = S_i(t_i) \\ S'_{i-1}(t_i) &= S'_i(t_i) \\ S''_{i-1}(t_i) &= S''_i(t_i) \\ S(x+T) &= S(x) \end{aligned}$$

Again we let $z_i = S''(t_i)$. The entire analysis in the books then works (allowing for infinitely many variables) except that we drop the restrictions that i be between 0 and n and that $z_0 = 0 = z_n$. We get to equation 10 on p377, which now holds for *all* i. Also the definitions of h_i, u_i, b_i, v_i all make sense in this infinite context, so we get:

$$h_{i-1}z_i + u_i z_i + h_i z_{i+1} = v_i$$

for all integers i. Note that $t_{i+n} = t_i$ implies that $h_{i+n} = h_i$ and that $u_{i+n} = u_i$. Also $S(x+T) = S(x)$ implies that $S''(x+T) = S''(x)$ which implies that $z_{i+n} = z_i$, which in turn implies that $b_{i+n} = b_i$ and $v_{i+n} = v_i$. Thus in fact we only need to determine $z_1, \ldots, z_n$. We will use this and our various periodicity facts to reduce the infinite set of equations to a finite set and then to a matrix equation. We write down every equation involving $z_1, \ldots, z_n$ and shift all indices to the range $1, \ldots, n$:

$$\begin{aligned} h_{-1}z_{-1} + u_0 z_0 + h_0 z_1 &= v_0 \quad i = 0 \text{ becomes :} \\ h_{n-1}z_{n-1} + u_n z_n + h_n z_1 &= v_n \\ h_0 z_0 + u_1 z_1 + h_1 z_2 &= v_1 \quad i = 1, \text{becomes :} \\ h_n z_n + u_1 z_1 + h_1 z_2 &= v_1 \end{aligned}$$

$$
\begin{aligned}
h_1 z_1 + u_2 z_2 + h_2 z_3 &= v_2 \quad i = 2, \\
&\vdots \\
h_{n-2} z_{n-2} + u_{n-1} z_{n-1} + h_{n-1} z_n &= v_{n-1} \quad i = n-1 \\
h_{n-1} z_{n-1} + u_n z_n + h_n z_{n+1} &= v_n \quad i = n, \text{becomes :} \\
h_{n-1} z_{n-1} + u_n z_n + h_n z_1 &= v_n \\
h_n z_n + u_{n+1} z_{n+1} + h_{n+1} z_{n+2} &= v_{n+1} \quad i = n+1, \text{becomes :} \\
h_n z_n + u_1 z_1 + h_1 z_2 &= v_1
\end{aligned}
$$

Then we notice that we can get rid of the first and last equations since they are repeated elsewhere to get n equations in n unknowns which looks, in matrix form, like:

$$
\begin{bmatrix}
u_1 & h_1 & & & & & h_n \\
h_1 & u_2 & h_2 & & & & \\
 & h_2 & u_3 & h_3 & & & \\
 & & \cdot & \cdot & \cdot & & \\
 & & & \cdot & \cdot & \cdot & \\
 & & & & h_{n-2} & u_{n-1} & h_{n-1} \\
h_n & & & & & h_{n-1} & u_n
\end{bmatrix}
\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \cdot \\ \cdot \\ z_{n-1} \\ z_n \end{bmatrix}
=
\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \cdot \\ \cdot \\ v_{n-1} \\ v_n \end{bmatrix}
$$

Then we solve this matrix for $z_1, \ldots z_n$ and proceed to compute coefficients exactly as with the natural spline, remembering that $z_{i+n} = z_i$.

- **NSpline2d.m** This function first computes the cumulative arclengths s and then invokes the natural spline algorithm twice to compute the coefficients of the splines describing x and y as functions of arclength. It calls **NaturalSpline.m**. which implements the natural spline algorithm described in Kincaid and Cheney, pp 350-354. The only difference from the book is that indices start at 1 instead of 0. In a few places we insert dummy entries at the beginnings of lists just to make the indexing match the book as closely as possible.

```
% --------------------  NSpline2d.m  -------------------
%
% function [cx,cy,s] = NSpline2d(x,y)
%
% inputs x(1:n),y(1:n)
% outputs cx(1:4;1:n-1),cy(1:4;1:n-1), s
%
% Computes the coefficients for the 2-d Natural Spline as
% described in the programming assignment. First computes
% the arclengths s(1:n) and then applies NaturalSpline to
% both (s,x) and (s,y) to get coefficient matrices cx and cy
% (see NaturalSpline.m)
%

function [cx,cy,s] = NSpline2d(x,y)

n = length(x);
if length(y)~=n
  c=[]; display('Error, x and y must have the same lengths'); return;
end
s=[0];
for i=2:n
  s=[s,s(i-1)+sqrt((x(i)-x(i-1))^2+(y(i)-y(i-1))^2)];
end
[cx]=NaturalSpline(s,x);
[cy]=NaturalSpline(s,y);
```

```
% --------------  NaturalSpline.m  ------------------------
%
% function [c] = NaturalSpline(t,y)
%
% inputs: t(1:n), y(1:n)
% outputs: c(1:4;1:n-1)
%
% following the notation in the book (except indices start at 1)
% combined with the notation
% in the programming assignment, computes the "natural spline"
% with "knots" at t(1),...,t(n) with y-values y(1),...,t(n)
% (see pp. 350-354), and returns the coefficients in the
% 4-by-(n-1) matrix c.
%
% Thus the computed spline function is given as follows:
%    when x is in the interval [t(i),t(i+1)], then
%       Spline(x) = c(4,i)*(t(i+1)-x)^3 + c(3,i)*(x-t(i)) + ...
%                   c(2,i)*(x-t(i)) + c(1,i)*(t(i+1)-x)
%
function [c] = NaturalSpline(t,y)

n = length(t); c=[];
if n<2 display('error: need at least two knots'); return; end
if length(y)~=n
  display('error: t and y have different lengths'); return;
end
% calculate values top of p. 358
h = t(2:n)-t(1:n-1); % h(i)=t(i+1)-t(i), 1<=i<=n-1
u = [0,2*(h(2:n-1)+h(1:n-2))]; % u(i)=2*(h(i)+h(i-1)), 2<=i<=n-1, u(1)=0
b = (6./h).*(y(2:n)-y(1:n-1)); % b(i)=(6/h(i))*(y(i+1)-y(i)), 1<=i<=n-1
v = [0,b(2:n-1)-b(1:n-2)]; % v(i)=b(i)-b(i-1), 2<=i<=n-1, v(1)=0
% put into a matrix S(1:n-2;1:n-2) (bottom p. 377) and compute z(1:n)
S = diag(u(2:n-1),0)+diag(h(2:n-2),1)+diag(h(2:n-2),-1);
z = [0,(S\(v(2:n-1)'))',0];
% calculate the coefficients and end
c3 = z(1:n-1)./(6.*h);
c2 = z(2:n)./(6.*h);
c1 = (y(2:n)./h) - (z(2:n).*h)/6;
c0 = (y(1:n-1)./h) - (z(1:n-1).*h)/6;
c = [c0;c1;c2;c3];
```

- **EvalSpline.m** This evaluates a cubic spline at a list of x-values given the coefficients in a matrix c and the list of knots $t_1, \ldots t_n$. For efficiency's sake I required that the x-values be in increasing order. Values of x less than t_1 are evaluated using the first cubic and values greater than t_n are evaluated using the last cubic. Note that, if the coefficients are intended to represent a periodic function this will not know that, but that as long as $t_1 \leq x \leq t_n$ that doesn't matter anyway.

```
% --------------  EvalSpline.m  ----------------------------
%
% function [y] = EvalSpline(c,t,x)
%
% inputs: c(1:4;1:n-1), t(1:n), x(1:m)
% outputs: y(1:m)
%
% Evaluates the spline function given by coefficients in
% the 4-by-(n-1) matrix c with knots t(1),...,t(n) at the
% points x to output y-values.
%
% Thus the computed spline function is given as follows:
%    when x is in the interval [t(i),t(i+1)], then
%       Spline(x) = c(4,i)*(t(i+1)-x)^3 + c(3,i)*(x-t(i))^3 + ...
%                   c(2,i)*(x-t(i)) + c(1,i)*(t(i+1)-x)

% For efficiency's sake, we require that x(1)<x(2)<...x(m)
%

function [y] = EvalSpline(c,t,x)

m = length(x); n = length(t); y=[];
% check for errors
if n<=1
  display('Error, need at least two knots'); return
end
if size(c)~=[4 (n-1)]
  display('Error, wrong dimensions for c'); return
end
if (m>1)&(any(x(2:m)-x(1:m-1)<=0))
  display('Error, need all x(i)<x(i+1)'); return
end
% k = count along x, i = index of the current interval
i = 1;
for k=1:m
  % increment i until we are in the correct interval, but
  % don't go past i=n-1
  while (x(k)>t(i+1))&(i<n-1), i=i+1; end
```

```
  % and calculate y=spline(x(k)), add to our list.
  y = [y, (c(4,i)*(t(i+1)-x(k))^3+c(3,i)*(x(k)-t(i))^3+...
           c(2,i)*(x(k)-t(i))+c(1,i)*(t(i+1)-x(k)))];
end
```

Complexity analysis:
Let n be the number of points.

- Preparing the lists of points and corners takes just $O(n)$ steps involving several simple passes through the list of points.
- Computing the cumulative arclengths is also just $O(n)$, involving a single pass through the list of points.
- Preparing the matrices for the spline algorithm is again $O(n)$, and solving the matrix equations using matlab's matrix division (assuming it does Gaussian elimination) is $O(n^3)$.
- Evaluating the spline at N points involves $O(N)$ steps. There is a little subtlety because you have to find the write interval in which to evaluate the spline for each point, but the way I implemented this, assuming that the points are in order along the curve, and assuming that $N >> n$, simply involves an occasional increment of the spline index as you move through the points.
- Thus plotting and computing error bounds are both $O(N)$ where N is the number of points to be plotted or to be used for estimating the error bounds.

Thus the runtime is either dominated by the matrix division or by the plotting and computing of error bounds, depending on how n^3 and N compare. In my program N was 1000 for error bounds and 500 for plotting.

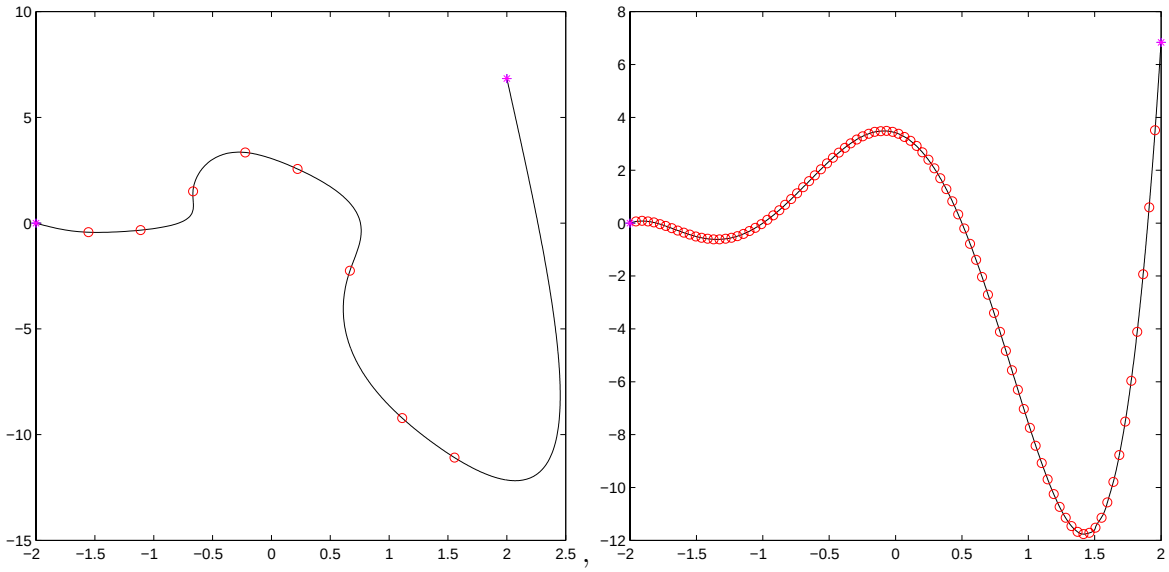
Results and Analysis:

```
Case 1: Begin testing Quintic

Results for Quintic
    #points        # corners      error
   1.0000e+01    2.0000e+00    7.9093e+01
   3.0000e+01    2.0000e+00    5.2997e-01
   5.0000e+01    2.0000e+00    8.7667e-02
   7.0000e+01    2.0000e+00    6.9607e-02
   9.0000e+01    2.0000e+00    5.0585e-02
```

Comments: The error decreases slowly as the number of points increases. We are approximating a quinitic by a cubic on each interval, so we don't expect fast convergence. There are exactly 2 corners, the ends of the curve.

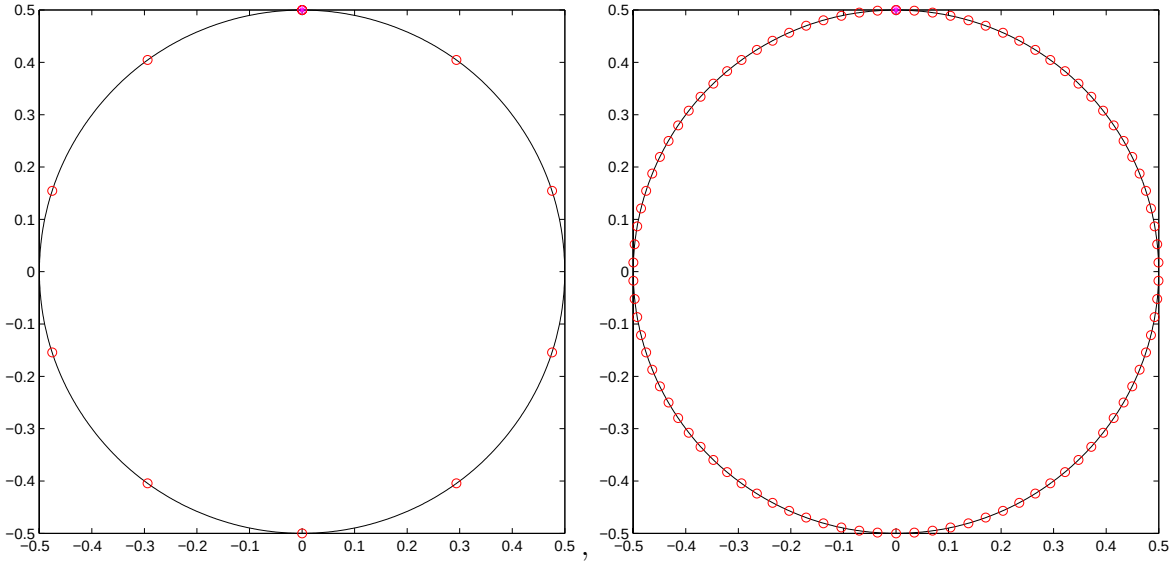
We show quintics with 10 and 90 points each below.



```
Case 2: Begin testing Circle

Results for Circle
    #points       # corners     error
   1.0000e+01             0   2.2358e-04
   3.0000e+01             0   2.5329e-06
   5.0000e+01             0   3.2598e-07
   7.0000e+01             0   8.4692e-08
   9.0000e+01             0   3.0968e-08
```

Comments: There are no corners in this smooth close curve. The error is very small. We show circles with 10 and 90 points each below.



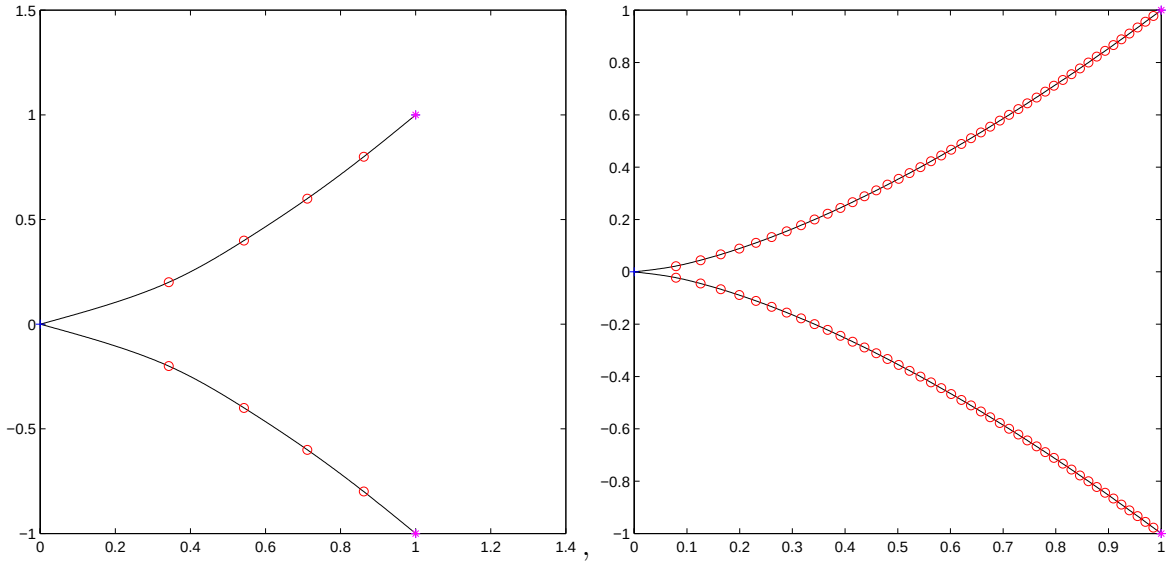
,

```
Case 3: Begin testing Cusp

Results for Cusp
    #points       # corners     error
   1.1000e+01   3.0000e+00   2.8868e-03
   3.1000e+01   3.0000e+00   2.9049e-04
   5.1000e+01   3.0000e+00   1.0063e-04
   7.1000e+01   3.0000e+00   5.0206e-05
   9.1000e+01   3.0000e+00   2.9945e-05
```

Comments: There are 3 corners: the ends of the curve and the cusp itself. Since we place one sample point directly at the cusp, the curve is smooth between sample points and the error is small.

We show cusps with 11 and 91 points each below.

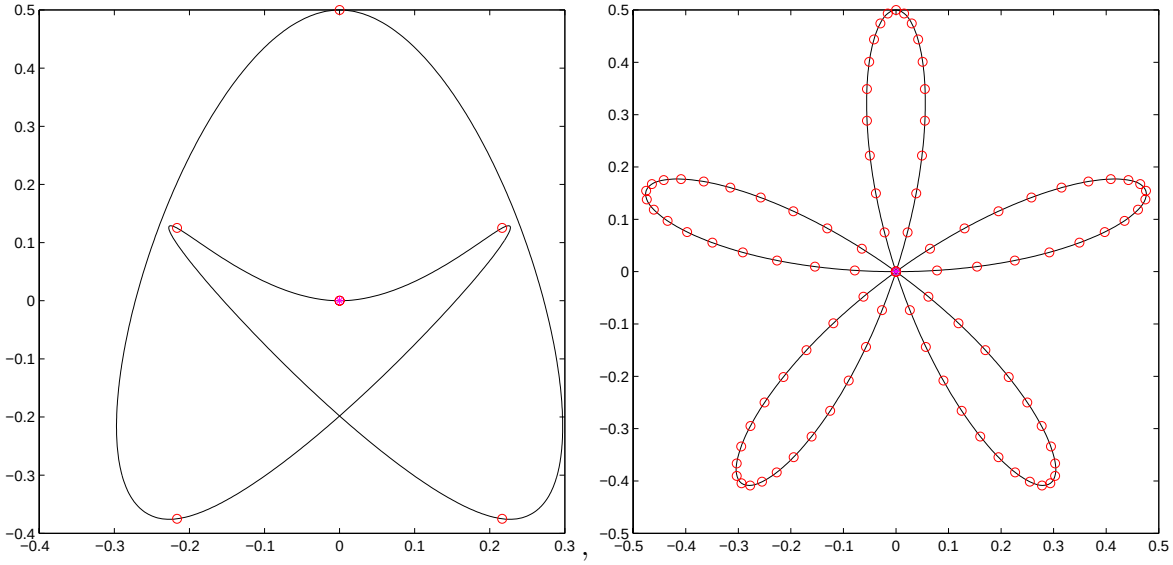


```
Case 4: Begin testing Rose

Results for Rose
    #points       # corners     error
   6.0000e+00             0   4.0910e-01
   7.0000e+00             0   4.7506e-01
   8.0000e+00             0   4.5612e-01
   9.0000e+00             0   4.8268e-01
   1.0000e+01             0   7.8595e-02
   1.5000e+01             0   3.3273e-01
   2.0000e+01             0   1.2690e-01
   4.0000e+01             0   1.0708e-02
   6.0000e+01             0   2.2616e-03
   8.0000e+01             0   6.7530e-04
   1.0000e+02             0   2.3852e-04
```

Comment: Not until we get 10 points that the rose looks at all like its final form, and not until 40 points that it really starting looking close, which is why the early errors are large. There are no corners in this smooth closed curve (self intersection does not count).

We show roses with 6 and 100 points each below.

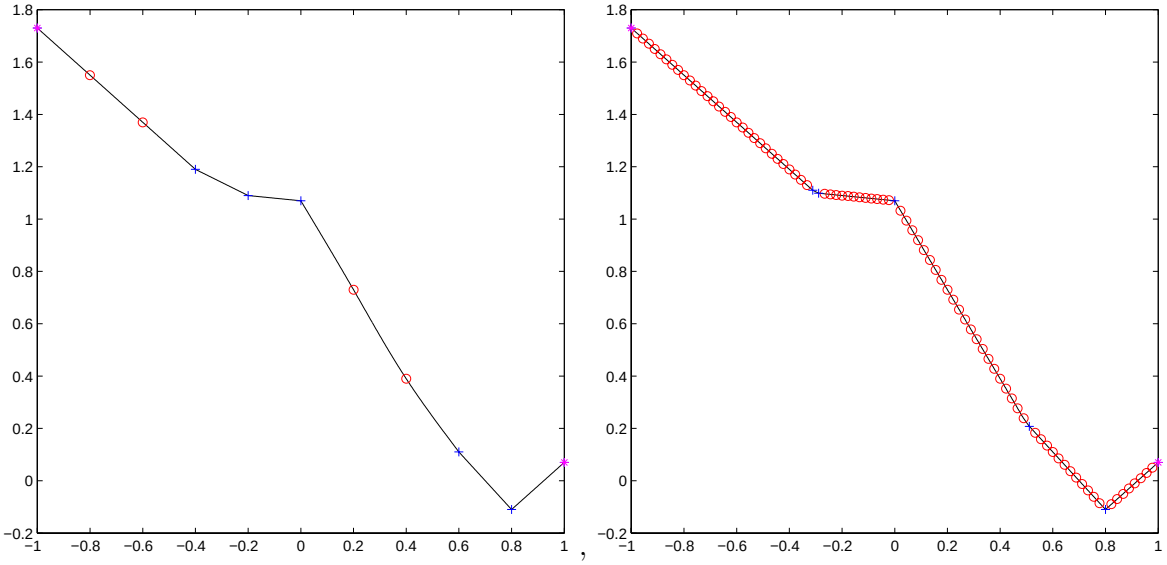


```
Case 5: Begin testing Jagged with corner detection

Results for Jagged with corner detection
    #points       # corners     error
   1.1000e+01   7.0000e+00   3.9504e-02
   2.1000e+01   6.0000e+00   1.5500e-15
   3.1000e+01   7.0000e+00   1.3062e-02
   4.1000e+01   6.0000e+00   3.6915e-15
   5.1000e+01   7.0000e+00   7.5595e-03
   6.1000e+01   6.0000e+00   7.3275e-15
   7.1000e+01   7.0000e+00   5.6522e-03
   8.1000e+01   6.0000e+00   9.7145e-15
   9.1000e+01   7.0000e+00   4.2962e-03
```

Comment: The curve is piecewise linear, so as long as we find the corners exactly, the error should be near 0 (in fact it is around 10^{-15} or roundoff level). In fact when the number of corners detected is the true value 6 (2 end points plus 4 internal corners) this is what happens. This works because the sample point coincide exactly with the true corners. For the other values of n, the sample points do not all coincide with the true corners, so the number of corners detected and the error are higher.

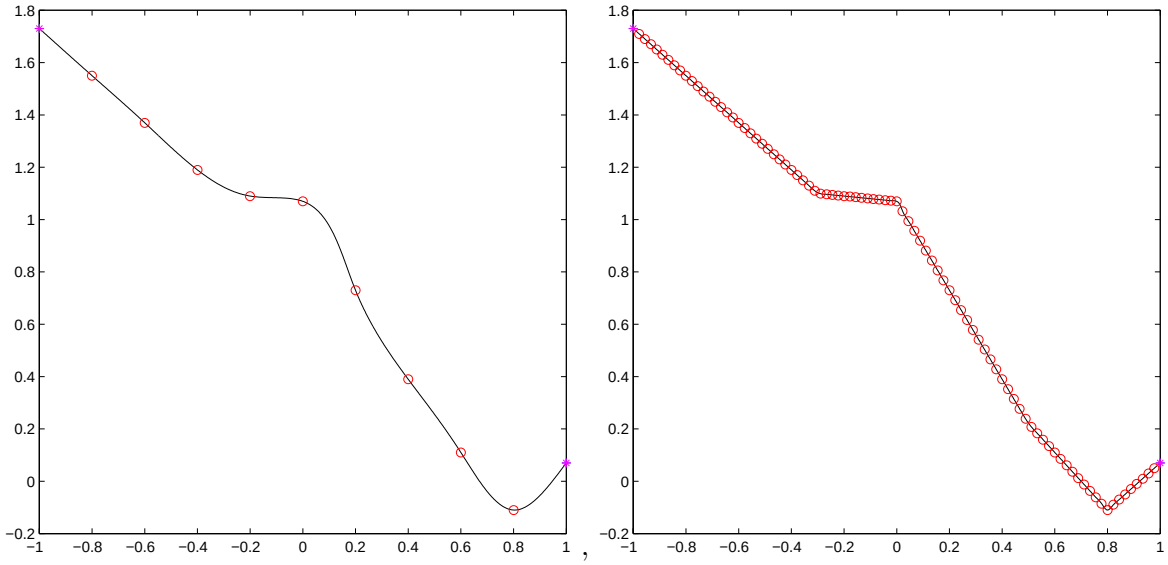
We show jagged lines (with corner detection) with 11 and 91 points each below.



```
Case 6: Begin testing Jagged without corner detection

Results for Jagged without corner detection
    #points       # corners     error
   1.1000e+01   2.0000e+00   7.6430e-02
   2.1000e+01   2.0000e+00   3.5710e-02
   3.1000e+01   2.0000e+00   2.3543e-02
   4.1000e+01   2.0000e+00   1.7684e-02
   5.1000e+01   2.0000e+00   1.4150e-02
   6.1000e+01   2.0000e+00   1.1782e-02
   7.1000e+01   2.0000e+00   1.0099e-02
   8.1000e+01   2.0000e+00   8.8181e-03
   9.1000e+01   2.0000e+00   7.8421e-03
```

Comment: Without corner detection, the error stays large around the corners.
We show jagged lines (without corner detection) with 11 and 91 points each below.



Free Form Art. Appended are 4 figures you might draw using the mouse input, for each possible combination of close/not closed, and with/without corner detection.