Medial-Based Morphing for 2D Binary Objects

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Created: September 25, 2000
Last Modified: March 2, 2008

Contents

1 Introduction **3**

2 Differential Geometry of Curves **3**

3 Curvature-Based Interpolation of Curves **4**

4 Medial Axes **5**

5 Boundary Information **6**

6 Implementation **8**

6.1 Core_Base Class 8

6.1.1 Numerical Differentiation 8

6.1.2 Numerical Integration 9

6.1.3 Inverting Normalized Arc Length 10

6.1.4 Other Methods 11

6.2 Core_Continuous Class 11

6.3 Core_Interpolant Class 11

6.4 Drawing the Cores 12

6.5 Test Driver 13

6.6 Planned Modifications 13

7 Source Code **14**

7.1 core.h 14

7.2 core.c 16
7.3 coredraw.h 23
7.4 coredraw.c 24
7.5 interp.c 26

1 Introduction

The following development gives an algorithm for morphing 2D binary objects by computing the medial axes for the objects and interpolating the axes by averaging their curvatures.

2 Differential Geometry of Curves

Let $(x(q), y(q))$ be a parameterized curve for $q \in [a, b]$. Let s be the arc length measured along the curve from initial position $(x(a), y(a))$ to a point $(x(q), y(q))$; then

$$s(q) = \int_a^q \sqrt{\left(\frac{dx}{dq}\right)^2 + \left(\frac{dy}{dq}\right)^2}\, dq.$$

The total length of the curve is $L = s(b)$. The speed at which a particle travels along the curve is

$$\frac{ds}{dq} = \sqrt{\left(\frac{dx}{dq}\right)^2 + \left(\frac{dy}{dq}\right)^2}.$$

Define *normalized arc length* to be $p = s/L$ where $p \in [0, 1]$. This allows us to compare two curves using a common parameter. Note that p depends on q, say $p = f(q)$ where f is invertible, so $q = f^{-1}(p)$. Thus, every curve can be reparameterized by normalized arc length:

$$(x(q), y(q)) = (x \circ f^{-1}(p), y \circ f^{-1}(p)) = (\bar{x}(p), \bar{y}(p)).$$

The curvature of curve $(x(q), y(q))$ is defined as

$$\kappa(q) = \frac{\frac{dx}{dq}\frac{d^2y}{dq^2} - \frac{dy}{dq}\frac{d^2x}{dq^2}}{\left(\left(\frac{dx}{dq}\right)^2 + \left(\frac{dy}{dq}\right)^2\right)^{3/2}}.$$

Let $\bar{\kappa}(p) = \kappa \circ f^{-1}(q)$. Since curvature is intrinsic to the curve, the same formula holds for curvature in terms of parameter p:

$$\bar{\kappa}(p) = \frac{\frac{d\bar{x}}{dp}\frac{d^2\bar{y}}{dp^2} - \frac{d\bar{y}}{dp}\frac{d^2\bar{x}}{dp^2}}{\left(\left(\frac{d\bar{x}}{dp}\right)^2 + \left(\frac{d\bar{y}}{dp}\right)^2\right)^{3/2}}.$$

If $(x(s), y(s))$ is a paremeterization by arc length s, then the tangent vector $\mathbf{T}(s) = (x'(s), y'(s))$ has unit length. Let $\mathbf{N}(s)$ be a unit normal whose orientation to $\mathbf{T}(s)$ is constant. The Frenet-Serret formulas relate tangent and normal by

$$\frac{d\mathbf{T}}{ds} = \kappa(s)\mathbf{N} \text{ and } \frac{d\mathbf{N}}{ds} = -\kappa(s)\mathbf{T},$$

where $\kappa(s)$ is the curvature. If the parameterization is by normalized arc length p where L is the total length of the curve, then

$$\frac{d\mathbf{T}}{dp} = L\bar{\kappa}(p)\mathbf{N} \text{ and } \frac{d\mathbf{N}}{dp} = -L\bar{\kappa}(p)\mathbf{T}.$$

3 Curvature-Based Interpolation of Curves

Let $(x_i(q_i), y_i(q_i))$ be parameterized curves for $q_i \in [a_i, b_i]$ for $i = 0, 1$. Let L_i denote the arc lengths of the curves. The curves can be reparameterized by normalized arc length p where

$$p = f_i(q_i) = \frac{1}{L_i} \int_{a_i}^{q_i} \sqrt{\left(\frac{dx_i}{dq_i}\right)^2 + \left(\frac{dy_i}{dq_i}\right)^2}\, dq_i.$$

The curves are then represented by

$$(x_i(q_i), y_i(q_i)) = (x_i \circ f_i^{-1}(p), y_i \circ f_i^{-1}(p)) = (\bar{x}_i(p), \bar{y}_i(p)).$$

Let $\kappa_i(q_i) = \bar{\kappa}_i(p)$ denote the curvatures of the curves.

A key idea in the morphing algorithm is the construction of a curve if its curvature function is specified together with an initial location and orientation. That is, if $\bar{\kappa}(p)$ for $p \in [0, 1]$ is known, and if an initial parameter $p_0 \in [0, 1]$ is selected together with initial location (ℓ_0, ℓ_1) and initial direction (d_0, d_1), then the curve is uniquely determined as the solution to the Frenet-Serret differential equations

$$\begin{array}{rcll} \frac{d\bar{x}}{dp} & = & u, & \bar{x}(p_0) = \ell_0 \\ \frac{d\bar{y}}{dp} & = & v, & \bar{y}(p_0) = \ell_1 \\ \frac{du}{dp} & = & -L\bar{\kappa}(p)v, & u(p_0) = d_0 \\ \frac{dv}{dp} & = & L\bar{\kappa}(p)u, & v(p_0) = d_1 \end{array}$$

where $\mathbf{T} = (u, v)$ and $\mathbf{N} = (-v, u)$.

We will interpolate our two initial curves by taking an average of their curvatures and solving this system of equations. The interpolation parameter is $t \in [0, 1]$. The interpolated curve at time t will have curvature

$$\bar{\kappa}_t(p) = (1 - t)\bar{\kappa}_0(p) + t\bar{\kappa}_1(p).$$

A simple choice for initial location is the average of the starting locations on the original two curves

$$(\ell_0, \ell_1) = (1 - t)(x_0(a_0), y_0(a_0)) + t(x_1(a_1), y_1(a_1)).$$

Similarly the initial direction is the average

$$(d_0, d_1) = (1 - t)\left(\frac{dx_0(a_0)}{dq_0}, \frac{dy_0(a_0)}{dq_0}\right) + t\left(\frac{dx_1(a_1)}{dq_1}, \frac{dy_1(a_1)}{dq_1}\right).$$

Other choices for initial location and direction can depend on the particular application. For example, if the curves have associated density functions, then the average of the centers of mass could be used. To compute the center of mass for a single curve $(x(q), y(q))$ for $q \in [a, b]$ with density $r(s)$, let L be the length of the curve. The arc length for the center is

$$s_c = \frac{\int_0^L sr(s)\, ds}{\int_0^L r(s)\, ds} = \frac{\int_a^b s(q)r(s(q))s'(q)\, dq}{\int_a^b r(s(q))s'(q)\, dq}.$$

4 Medial Axes

Let $B \subset \mathbb{R}^2$ be a connected and compact set. The *symmetry axis transform* or *medial axis* of B is the set of triples (x, y, r) such that

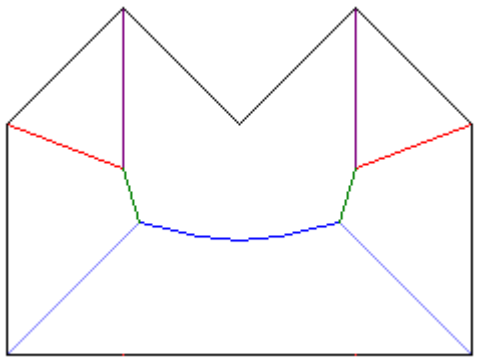
- the maximal disk contained in B centered at (x, y) with radius $r > 0$ intersects the boundary of B, ∂B, in at least two points, or
- the maximal disk contained in B centered at (x, y) has radius $r = 0$.

Such points (x, y) are called *medial points*. The radii r are measures of how wide the object is at medial points. A medial point is

- an *end point* if its maximal disk is tangent to ∂B in exactly one point (or in a continuum of points forming a circular arc),
- a *normal point* if its maximal disk is tangent to ∂B in exactly two points,
- a *branch point* if its maximal disk is tangent to ∂B in three or more points,
- a *filament point* if its maximal disk has radius 0.

The medial points can be divided into curvilinear segments by separating the axis at branch points. Each segment can be parameterized and used in an interpolation. The following picture illustrates medial axes for an object with a polygonal boundary. For polygonal objects, the medial axes consist of straight line segments or parabolic arcs (caused by concave vertices). Figure 1 shows the medial axes for a sawtooth polygon.

Figure 1. Medial axis for a sawtooth polygon.



---

5 Boundary Information

The following discussion deals with an object B whose medial axis is a single curvilinear segment. Let the axis be parametrized by arc length, say $(x(s), y(s))$, with corresponding radius function $r(s)$, where $s \in [0, L]$. Let the maximal disks be denoted $D(s) = D(\mathbf{x}(s), r(s))$. The medial axis is

$$\text{Medial}(B) = \bigcup_{s\in[0,L]} (x(s), y(s), r(s))$$

and the set B is reconstructed by

$$B = \bigcup_{s\in[0,L]} D(s).$$

In this sense, Medial(B) may be considered a compressed representation of B.

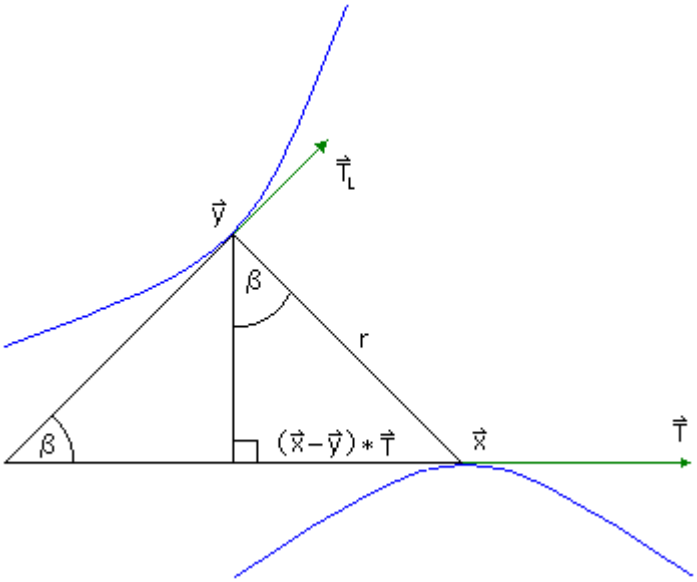
The tangent and normal vectors for the medial axis are

$$\mathbf{T} = \frac{d\mathbf{x}}{ds} = (\cos(\alpha), \sin(\alpha)) \quad \text{and} \quad \mathbf{N} = (-\sin(\alpha), \cos(\alpha)).$$

The tangent vector defines the angle α, called the *axis angle*, measured with respect to the positive x-axis. For a curve which is parametrized by arc length, we know that $d\mathbf{T}/ds = \kappa(s)\mathbf{N}$ where $\kappa(s)$ is the curvature of the curve. For the medial axis, the curvature is therefore given by $\kappa(s) = \frac{d\alpha}{ds}$.

Let $\mathbf{T}_L(s)$ and $\mathbf{T}_R(s)$ denote the tangent vectors to the left and right boundaries of the object where the maximal disk $D(s)$ touches. Intuitively, $\mathbf{T}(s)$ bisects the angle between these two boundary tangents. Let the half-angle be denoted by $\beta(s)$, called the *object angle*. Let $\mathbf{y}(s)$ be the induced parameterization of the left boundary. Figure 2 illustrates.

Figure 2. The relationship of boundary curves to the medial axis.



The radius function is given by $r^2 = |\mathbf{x} - \mathbf{y}|^2$. Taking derivatives, we have

$$r\frac{dr}{ds} = (\mathbf{x} - \mathbf{y}) \cdot \left(\frac{d\mathbf{x}}{ds} - \frac{d\mathbf{y}}{ds} \right) = (\mathbf{x} - \mathbf{y}) \cdot \mathbf{T}$$

where the last equality holds since $d\mathbf{y}/ds$ is tangent to the boundary and $\mathbf{x} - \mathbf{y}$ is normal to the boundary implying that their dot product is zero.

The projection of $\mathbf{x} - \mathbf{y}$ onto $\mathbf{T}$ has length $(\mathbf{x} - \mathbf{y}) \cdot \mathbf{T}$. Using the smaller right-triangle indicated in the figure, we have

$$\sin(\beta) = \frac{(\mathbf{x} - \mathbf{y}) \cdot \mathbf{T}}{r}.$$

The last two displayed equations imply

$$\frac{dr}{ds} = \sin(\beta(s)).$$

We had shown that $d\alpha/ds$ is the curvature of the medial axis. Using an analogy, we define the *object curvature* as $-d\beta/ds$. The minus sign is a convention to make a positive object curvature correspond to a convex boundary. Using the second derivative of the radius function, one can show that

$$-\frac{d\beta}{ds} = \frac{-d^2r/ds^2}{\sqrt{1 - (dr/ds)^2}},$$

so the medial axis information allows us to compute the object curvature.

Note that the tangent and normal vectors to the left boundary are

$$\mathbf{T}_L = (\cos(\alpha + \beta), \sin(\alpha + \beta)) \quad \text{and} \quad \mathbf{N}_L = (-\sin(\alpha + \beta), \cos(\alpha + \beta)).$$

We want to determine the *boundary curvature*. We can write the left boundary as

$$\mathbf{y}(s) = \mathbf{x}(s) + r(s)\mathbf{N}_L(s).$$

However, the parametrization for $\mathbf{y}$ is not necessarily one of arc length. Suppose that $\mathbf{y}(t)$ is an arc length parametrization; then

$$\mathbf{T}_L(t) = \frac{d\mathbf{y}(t)}{dt} = \frac{d\mathbf{y}(s)}{ds}\frac{ds}{dt}.$$

and,

$$\begin{aligned}
\tfrac{d\mathbf{y}(s)}{ds} &= \mathbf{T} + r\tfrac{d\mathbf{N}_L}{ds} + \dot{r}\mathbf{N}_L \\
\tfrac{dt}{ds}\mathbf{T}_L &= (\cos\beta)\mathbf{T}_L + (-\sin\beta)\mathbf{N}_L - r(\dot{\alpha} + \dot{\beta})\mathbf{T}_L + \dot{r}\mathbf{N}_L \\
&= [\cos\beta - r(\dot{\alpha} + \dot{\beta})]\mathbf{T}_L,
\end{aligned}$$

so

$$\frac{dt}{ds} = \cos\beta - r\left(\frac{d\alpha}{ds} + \frac{d\beta}{ds}\right).$$

Finally,

$$\kappa_L(t) = \frac{d\alpha}{dt} + \frac{d\beta}{dt} = \frac{\dot{\alpha} + \dot{\beta}}{dt/ds} = \frac{\dot{\alpha} + \dot{\beta}}{cos\beta - r(\dot{\alpha} + \dot{\beta})}.$$

6 Implementation

I have implemented the interpolation using C++ classes. These are found in the files `core.h` and `core.c`. The class representing the original curves to be interpolated is named `Core_Continuous`. An object of this class is created from functions and a parameter interval for the parameterized curve. The class representing an interpolated curve is named `Core_Interpolant`. An object of this class is typically created from two objects of class `Core_Continuous` and a morphing time. However, in order to allow for more general settings where one might want to interpolate the interpolants, I have created an abstract base class `Core_Base` to take advantage of the polymorphism allowed by C++.

6.1 Core_Base Class

I found it convenient to save the interval $[a, b]$ endpoints as data members `ca` and `cb`. I also have a data member `clength` which stores the total length of the curve. The interpolation requires numerical differentiation, numerical integration, and root finding. Each numerical method has some user-selected parameters. These are stored in the data members `cb_ode_maxiter`, `cb_integral_order`, and `cb_invlen_param`.

6.1.1 Numerical Differentiation

The numerical differentiation is used to solve the Frenet-Serret system of equations. The method to do this is

```
void ode_solve (float p0, float p1, Vector &pos, Vector &dir);
```

where `p0` and `p1` are in the interval $[0, 1]$ with `p0` being the starting parameter for the system. The initial position is `pos` and the initial direction is `dir`, both of type `Vector` which is a struct of two floats, x and y.

The routine is a 4-step Runge-Kutta algorithm. You can look up the theory in any standard numerical analysis text book. At each step you will notice that the right-hand sides of the assignments are exactly the values of the right-hand sides in the Frenet-Serret system. The solution vector (x, y, u, v) is represented by 1D arrays with 4 elements. For example,

```
xder[0] = clength*xin[2];          // x' = L u
xder[1] = clength*xin[3];          // y' = L v
xder[2] = -clength*curv*xin[3];    // u' = -L K(p) v
xder[3] = clength*curv*xin[2];     // v' = L K(p) u
```

The number of iterations of the solver is user-defined and is stored in data member `cb_ode_maxiter`. This can be set by the method `ode_maxiter (int i)` immediately after a core object is constructed. The default number of iterations is (arbitrarily) 100.

The output of the solver consists of the final position `pos` and the final direction `dir`.

6.1.2 Numerical Integration

The numerical integration is used to compute the arc length via integration. It is also used in the center of mass calculations when these are required for an initial position. The integration method is

```
void integral (float a, float b, float (Core_Base::*f)(float));
```

where `a` and `b` are the limits of integration and `f` is a pointer to the function to be integrated. One of the deficiencies of C++ is the lack of a mechanism to allow class member functions to be passed to routines without specifying the class type. A member function always has an implicit first parameter which is a pointer to the calling object. As a result the member function could not be passed to a generic routine such as

```
void integral (float a, float b, float (*f)(float));
```

since strict type checking shows that `float (*f)(float)` and `float (*f)(object-pointer,float)` are distinct signatures.

I require integration in both classes `Core_Continuous` and `Core_Interpolant`. To avoid implementing integration in each class, the abstract base class `Core_Base` allows me to implement the algorithm once. But now all functions to be integrated must be members of the base class. In particular, for center of mass calculations the methods

```
float mass_integrand (float q);
float moment_integrand (float q);
```

are declared in the base class even though the center of mass calculations are performed in the derived classes. Also, routines to calculate length and speed are needed by the mass and moment integrands, so these also must be declared. At first it seems like the abstract base class knows nothing of how the derived classes will compute length and speed. Fortunately, C++ allows virtual functions whose pointer values are properly initialized to the derived classes functions when these derived classes are constructed. Moreover, I have made most of the base class methods *pure virtual functions* (syntactically done by placing `= 0` after the method), for example,

```
virtual float speed (float q) = 0;
virtual float length (float q) = 0;
```

Pure virtual functions *must* be implemented in any derived class. The presence of pure virtual functions also prevents an explicit construction of an object of class `Core_Base`.

The numerical integration routine is a Romberg routine whose order is specified by the user. The order is stored as data member `cb_integral_order` and can be set by the method `integral_order (int i)`. The default order is (arbitrarily) 5. The theory of Romberg integration routines can be found in any standard numerical analysis text book. The return value of the routine is the approximate integral value.

6.1.3 Inverting Normalized Arc Length

For a curve $(x(q), y(q))$ with $q \in [a, b]$, I defined normalized arc length as

$$p = \frac{1}{L} \int_a^q \sqrt{\left(\frac{dx}{dq}\right)^2 + \left(\frac{dy}{dq}\right)^2}\, dq \quad \in [0, 1].$$

Many of the calculations involve selecting a p value and finding the corresponding q value. For example, the curvatures of the interpolated core were defined by

$$\bar{\kappa}(p) = (1-t)\bar{\kappa}_0(p) + t\bar{\kappa}_1(p) = (1-t)\kappa \circ f_0(q_0) + t\kappa \circ f_1(q_1).$$

Given a $p \in [0, 1]$, to find $\bar{\kappa}(p)$ required two inversions to compute $q_0 \in [a_0, b_0]$ and $q_1 \in [a_1, b_1]$. The inversions are numerically calculated by root finding using a standard Newton's algorithm. The class method is

```
float invlen (float p);
```

and is implemented in the base class.

Some exceptions are trapped by the initial inversion code:

```
if ( clength == 0.0 )  return 0.0;
if ( p <= 0.0 )  return ca;
if ( p >= 1.0 )  return cb;
```

The first exception does not occur when interpolating two curves. However, I have made the code general enough to allow mapping a curve to a single point which has zero length. The inversion always returns $q = 0$ even though the single point is given by $(x(q), y(q)) \equiv (x_0, y_0)$ for all $q \in [a, b]$. (I allow $a < b$ for singleton points.) The second and third exceptions handle the problems when the input p value is computed with some small numerical errors, in which case p is theoretically 0 (1) but evaluates to $0 - \epsilon$ $(1 + \epsilon)$ for small $\epsilon > 0$.

In theory the inversion always has a unique solution. That is, if $p = f(q)$, then $q = f^{-1}(p)$ for exactly one q. The function for Newton's method is $F(q) = f(q) - p$ for a specified p. Since $f'(q) \in (0, 1)$ for $q \in (a, b)$, the sequence of theoretical iterates is guaranteed to converge. But just in case of some numerical errors, I have a limit on the number of iterations. This limit is stored in data member `cb_invlen_maxiter` and can be set by method `invlen_maxiter (int i)`. The default value is (arbitrarily) 50.

The root finding is straightforward. The initial iterate is the midpoint of the q domain, $q_0 = (a + b)/2$. The iterates are defined by

$$q_{n+1} = q_n + \frac{F(q_n)}{F'(q_n)}, \quad n \geq 0.$$

In the code I simply compute $F(q_n)$ and compare it to a tolerance. If smaller than the tolerance, the current q value is returned as the inverted value; otherwise I update the q value and repeat the test. The tolerance is stored in data member `cb_invlen_tolerance` and can be set by method `invlen_tolerance (float toler)`. The default value is `1e-06`.

6.1.4 Other Methods

I have provided a couple of methods for reading the data members. The methods to read the q domain endpoints are

```
float max_q ()  { return ca; }
float min_q ()  { return cb; }
```

and the method

```
float curve_length ()  { return clength; }
```

reads the total length of the curve. Note that `clength` must be initialized by the derived classes.

The remaining methods are all pure virtual functions and must be implemented by each derived class. Note that the input float variables to the routines are labeled `q` if the routine requires the original parameter values or labeled `p` if the routine requires normalized arc length values. The method names are self-explanatory. The radius function is the one mentioned in the medial axis construction.

6.2 Core_Continuous Class

The class `Core_Continuous` requires the user to specify the parameterization of the curve, including its first and second derivatives. These derivatives are used for calculation of speed, length, and curvature. The user also specifies the radius function.

The constructor saves pointers to the functions and saves them as data members `cx[3]`, `cy[3]` for location and derivatives, and as `cradius` for the radius function. The total length of the curve is computed and stored in `clength` (part of the base class).

Position, direction, radius, and speed are computed in the straightforward way (evaluation of the functions passed by the user). Length is computed by calling the integration routine. Curvature is also computed in a straightforward way. The only problem arises when speed is 0, in which case curvature must be computed as a limit. I avoid the technical problems here for now and just return zero curvature when speed is zero (which is a possible error).

The mass is calculated in the straightforward way by integration. The p value at which the center of mass occurs if also computed in a straightforward way, except when the core is a single point. For single points return $p = 0.5$ to indicate that the center of mass occurs at the "center" of the curve (which happens to be the single point). This assignment makes sense if you consider the morphing of a single point with a straight line. (Try it yourself!)

6.3 Core_Interpolant Class

The class `Core_Interpolant` requires the user to pass two other cores. I use the polymorphism of C++ in that the input types are class `Core_Base`. This allows the user to pass objects of type `Class_Continuous`, `Class_Interpolant`, or of any other derived class. Of course, if an interpolated object is passed, the

interpolations become nested and probably slow in calculating. The user must also pass the morphing time $t \in [0, 1]$.

The constructor saves the morphing time as member `ctime` and saves references to the cores as `ccore0` and `ccore1` for future use. The length of the interpolated curve is computed as the average of the core lengths and stored in `clength` (part of the base class).

The initial p value is saved as member `cinit_param` and has default value 0. It can be changed by method `initial_parameter (float p)` immediately after creation of an object. The initial position is stored as `cinit_pos` and has default as the average of the initial points of the two cores. This can also be changed, using method `initial_position (Vector pos)`. The initial direction is stored as `cinit_dir` and has default value as the average value (in the sense of rotations) of the initial directions of the two cores. Note that the initial angle between the direction vectors `dir0` and `dir1` is given by

```
cs_angle = dir0.x*dir1.x+dir0.y*dir1.y;  // dir0 "dot" dir1
sn_angle = dir0.x*dir1.y-dir1.x*dir0.y;  // length(dir0 "cross" dir1)
angle = ctime*atan2(sn_angle,cs_angle);
```

The return value of `atan2` is in the interval $[-\pi, \pi]$, so the initial angle always amounts to a rotation away from `dir0` towards `dir1`.

The radius, curvature, and p value for the center of mass are computed as linear interpolations of the corresponding original core values. Speed and length are also linear interpolations, but note that the original core speed and length functions require q values rather than p values, so the inversion routines must be called first to compute from p to q values.

6.4 Drawing the Cores

I have a class, `CoreDraw`, which allows me to draw cores as white curves on a black background in an image. The code is found in `coredraw.h` and `coredraw.c`. The class uses my image library for image manipulation. In particular, it uses 2-dimensional images of short integers, called `Image_SHORT`. You can replace the image handling by any of your own code which manipulates 2-dimensional arrays.

The constructor takes as input a bounding box in which the core will occur. You should make sure the box is big enough, perhaps by analyzing what you think your cores will look like during construction. The parameter `meshsize` is the number of steps taken while drawing the core. Between each pair of consecutive mesh points on the core, I use Bresenham's line drawing algorithm to approximate the core. The larger the mesh size, the better the approximation to the core. The input values are stored in data members.

The image to be drawn in is `cd_im` and is a 256×256 image of short ints (you really only need a bit image here). The image size is defined in the source file. You can change it, but need to recompile. Another parameter could be passed to the constructor to allow the user to specify the image size, or a method could be built to change this size.

Method `CoreDraw& clear()` sets all pixels to 0 in the image. The returned reference value just allows you to call the methods all in the same statement (see the sample main driver). The method `void save (char *filename)` saves the drawn image to the selected file.

The method

```
CoreDraw& draw (Core_Base& core);
```

maps the float bounding box to an integral one in the image. The normalized arc length p is sampled uniformly. At the two sampled values the core positions are computed and a "line" is drawn between the positions. A Bresenham's algorithm is used, but instead of drawing single pixels, disks are drawn of the appropriate radius. Method

```
void line(long x0, long y0, long r0, long x1, long y1, long r1);
```

takes as input two core points and corresponding radii (in integral coordinates) and draws this line of disks connecting them. The radius for each drawn disk is an interpolated radius of the input radii. The disks are drawn by

```
void disk (long xc, long yc, long radius);
```

where `xc` and `yc` are the coordinates for the disk center and `radius` is the integral radius for the disk. This routine uses Bresenham's circle drawing algorithm for solid disks.

6.5 Test Driver

The test driver (interp.c) has two morphing examples. The first example morphs the counterclockwise spiral $(x(q), y(q)) = (q\cos q, q\sin q)$, $q \in [0, 5\pi/2]$ to its reflection through the y-axis. The radius function is constant for this example. The morphing takes the first spiral, unwinds it into a straight line (halfway through the process), and then rewinds it into the reflected spiral.

The second example takes a Y-shaped object consisting of a semicircle and a straight line segment and morphs it into a horizontal line segment. The Y-shaped object has constant radius function, but the horizontal line segment has a linearly varying radius, so the object looks like a cone with cap. Note that the two object have topologically different axes. I map the line segment in the Y-shaped object to a single point on the horizontal line. Thus, I morph two times, a curve to curve, then a curve to point.

6.6 Planned Modifications

I intend on building a class `Core_Spline` which takes as input two lists of points. Each point list will be internally fitted with a spline curve. The curves and their derivatives take the place of the function pointers passed to `Core_Continuous`. Thus, the user can simply specify a list of points and not have to construct a parametrization explicitly.

If this interpolation scheme becomes popular, then a user-interface is needed to allow registration of two (possibly) topologically distinct medial axes.

The ideas here apply to medial curves for 3D objects, which is a restricted generalization of medial axes. Now one must average curvatures and torsions. The Frenet-Serret formulas for 3D are given by

$$\begin{bmatrix} \mathbf{T}' \\ \mathbf{N}' \\ \mathbf{B}' \end{bmatrix} = \begin{bmatrix} 0 & \kappa(s) & 0 \\ -\kappa(s) & 0 & \tau(s) \\ 0 & -\tau(s) & 0 \end{bmatrix} \begin{bmatrix} \mathbf{T} \\ \mathbf{N} \\ \mathbf{B} \end{bmatrix}$$

where $\tau(s)$ is the torsion of the curve and where $\mathbf{B} = \mathbf{T} \times \mathbf{N}$. This system has a unique solution when $\kappa(s)$ and $\tau(s)$ are specified and initial position, tangent, and normal are given.

The ideas also generalize to medial surfaces (the natural extension of medial axes). But now one must average the two principal curvature functions and the angles between the principal direction vectors.

7 Source Code

7.1 core.h

```
#ifndef CORE_H
#define CORE_H

#include <math.h>

typedef float (*Function)(float);
typedef struct { float x, y; } Vector;

//-----------------------------------------------------------------------------
class Core_Base
{
protected:
    float ca, cb;   // parameterization defined on [ca,cb]
    float clength;  // total length of curve
    int cb_ode_maxiter, cb_integral_order, cb_invlen_maxiter;
    float cb_invlen_tolerance;

    void ode_solve (float p0, float p1, Vector &pos, Vector &dir);
    float integral (float a, float b, float (Core_Base::*f)(float));
    float mass_integrand (float q) {
        float p = length(q)/clength;
        return radius(p)*speed(q);
    }
    float moment_integrand (float q) {
        float p = length(q)/clength;
        return length(q)*radius(p)*speed(q);
    }
public:
    Core_Base (float a, float b) {
        ca = a;
        cb = b;
        clength = 0;
        cb_ode_maxiter = 100;
        cb_integral_order = 5;
        cb_invlen_maxiter = 50;
        cb_invlen_tolerance = 1e-06;
```

```
    }
    Core_Base (Core_Base &cb) { *this = cb; }

    void  ode_maxiter (int i) { cb_ode_maxiter = i; }
    void  integral_order (int i) { cb_integral_order = i; }
    void  invlen_maxiter (int i) { cb_invlen_maxiter = i; }
    void  invlen_tolerance (float tol)  { cb_invlen_tolerance = tol; }
    float invlen (float p);

    float min_q () { return ca; }
    float max_q () { return cb; }
    float curve_length () { return clength; }

    virtual float speed (float q) = 0;
    virtual float length (float q) = 0;

    virtual Vector position (float p) = 0;
    virtual Vector direction (float p) = 0;
    virtual float  radius (float p) = 0;
    virtual float  curvature (float p) = 0;

    virtual float  mass () = 0;
    virtual float  mass_normal_arclength () = 0;
};
//-----------------------------------------------------------------------------
class Core_Continuous : public Core_Base
{
private:
    // core specified by (x(q),y(q),radius(q)) for q in [a,b]
    Function cx[3];     // x(q), x'(q), x"(q)
    Function cy[3];     // y(q), y'(q), y"(q)
    Function cradius;   // radius(q)
    // p : [a,b] -> [0,1] by p = length(a,q)/length(a,b)
public:
    Core_Continuous () : Core_Base(0,0) {;}
    Core_Continuous (float a, float b, Function xjet[], Function yjet[],
        Function radiusjet);

    float speed (float q);
    float length (float q);

    Vector position (float p);
    Vector direction (float p);
    float  radius (float p);
    float  curvature (float p);

    float  mass ();
    float  mass_normal_arclength ();
```

```
};
//-----------------------------------------------------------------------------
class Core_Interpolant : public Core_Base
{
private:
    float ctime;
    Core_Base &ccore0, &ccore1;

    float cinit_param;
    Vector cinit_pos, cinit_dir;
public:
    Core_Interpolant (float time, Core_Base &core0, Core_Base &core1);

    void initial_parameter (float p);
    void initial_position (Vector pos);
    void initial_direction (Vector dir);

    float speed (float q);
    float length (float q);

    Vector position (float p);
    Vector direction (float p);
    float  radius (float p);
    float  curvature (float p);

    float  mass ();
    float  mass_normal_arclength ();
};
//-----------------------------------------------------------------------------

#endif
```

7.2 core.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "core.h"

//=============================================================================
void Core_Base::
ode_solve (float p0, float p1, Vector &pos, Vector &dir)
{
    float rstep = (p1-p0)/cb_ode_maxiter, rstep2 = rstep/2, rstep6 = rstep/6;
    float xder[4], xt[4], xtder[4], xmder[4];

    float xin[4] = { pos.x, pos.y, dir.x, dir.y }, xout[4];
```

```
    float p = p0;
    for (int iter = 0; iter < cb_ode_maxiter; iter++, p += rstep) {
        // first step
        float curv = curvature(p+rstep2);
        xder[0] = clength*xin[2];
        xder[1] = clength*xin[3];
        xder[2] = -clength*curv*xin[3];
        xder[3] = clength*curv*xin[2];
        for (int i = 0; i < 4; i++)
            xt[i] = xin[i] + rstep2*xder[i];

        // second step
        xtder[0] = clength*xt[2];
        xtder[1] = clength*xt[3];
        xtder[2] = -clength*curv*xt[3];
        xtder[3] = clength*curv*xt[2];
        for (i = 0; i < 4; i++)
            xt[i] = xin[i] + rstep2*xtder[i];

        // third step
        xmder[0] = clength*xt[2];
        xmder[1] = clength*xt[3];
        xmder[2] = -clength*curv*xt[3];
        xmder[3] = clength*curv*xt[2];
        for (i = 0; i < 4; i++) {
            xt[i] = xin[i] + rstep*xmder[i];
            xmder[i] += xtder[i];
        }

        // fourth step
        curv = curvature(p+rstep);
        xtder[0] = clength*xt[2];
        xtder[1] = clength*xt[3];
        xtder[2] = -clength*curv*xt[3];
        xtder[3] = clength*curv*xt[2];
        for (i = 0; i < 4; i++)
            xout[i] = xin[i] + rstep6*(xder[i]+xtder[i]+2*xmder[i]);

        // re-initialize for next pass
        for (i = 0; i < 4; i++)
            xin[i] = xout[i];
    }

    pos.x = xin[0];
    pos.y = xin[1];
    dir.x = xin[2];
    dir.y = xin[3];
}
```

```
//------------------------------------------------------------------------
float Core_Base::
integral (float a, float b, float (Core_Base::*f)(float))
{
    float *rom[2];
    rom[0] = new float[cb_integral_order];
    rom[1] = new float[cb_integral_order];
    float h = b-a;

    rom[0][0] = h*((this->*f)(a)+(this->*f)(b))/2;
    for (int i=2,ipower=1; i <= cb_integral_order; i++, ipower *= 2, h /= 2) {
        float sum = 0;
        for (int j = 1; j <= ipower; j++)
            sum += (this->*f)(a+h*(j-0.5));

        rom[1][0] = (rom[0][0]+h*sum)/2;
        for (int k = 1, kpower = 4; k < i; k++, kpower *= 4)
            rom[1][k] = (kpower*rom[1][k-1] - rom[0][k-1])/(kpower-1);

        for (j = 0; j < i; j++)
            rom[0][j] = rom[1][j];
    }
    float result = rom[0][cb_integral_order-1];
    delete[] rom[1];
    delete[] rom[0];

    return result;
}
//------------------------------------------------------------------------
float Core_Base::
invlen (float p)
{
    if ( clength == 0.0 )  return 0.0;
    if ( p <= 0.0 )  return ca;
    if ( p >= 1.0 )  return cb;

    float q = (ca+cb)/2;  // initial guess
    for (int i = 0; i < cb_invlen_maxiter; i++) {
        float difference = length(q)/clength-p;
        if ( fabs(difference) < cb_invlen_tolerance )
            return q;

        q -= clength*difference/speed(q);
    }

    printf("exceeded maximum iterations in method 'param'\n");
    exit(-1);
    return 0;
```

```
}
//============================================================================
Core_Continuous::
Core_Continuous (float a, float b, Function xjet[], Function yjet[],
    Function radiusjet) : Core_Base(a,b)
{
    for (int i = 0; i <= 2; i++) {
        cx[i] = xjet[i];
        cy[i] = yjet[i];
    }
    cradius = radiusjet;
    clength = length(cb);
}
//----------------------------------------------------------------------------
Vector Core_Continuous::
position (float p)
{
    float q = invlen(p);
    float x0 = cx[0](q), y0 = cy[0](q);
    Vector pos = { x0, y0 };
    return pos;
}
//----------------------------------------------------------------------------
Vector Core_Continuous::
direction (float p)
{
    float q = invlen(p);
    float x1 = cx[1](q), y1 = cy[1](q), len = sqrt(x1*x1+y1*y1);
    if ( len > 0 ) {
        x1 /= len;
        y1 /= len;
    }
    Vector dir = { x1, y1 };
    return dir;
}
//----------------------------------------------------------------------------
float Core_Continuous::
radius (float p)
{
    float q = invlen(p);
    return cradius(q);
}
//----------------------------------------------------------------------------
float Core_Continuous::
speed (float q)
{
    float x1 = cx[1](q), y1 = cy[1](q), sp = sqrt(x1*x1+y1*y1);
    return sp;
```

```
}
//-------------------------------------------------------------------------
float Core_Continuous::
length (float q)
{
    return integral(ca,q,&Core_Base::speed);
}
//-------------------------------------------------------------------------
float Core_Continuous::
curvature (float p)
{
    float q = invlen(p);
    float x1 = cx[1](q), x2 = cx[2](q), y1 = cy[1](q), y2 = cy[2](q);
    float denom = pow(x1*x1+y1*y1,1.5);
    if ( denom > 0 )
        return (x1*y2-y1*x2)/denom;
    else
        return 0;  // could be nonzero in the limit
}
//-------------------------------------------------------------------------
float Core_Continuous::
mass ()
{
    return integral(ca,cb,&Core_Base::mass_integrand);
}
//-------------------------------------------------------------------------
float Core_Continuous::
mass_normal_arclength ()
{
    float p;

    if ( clength > 0 ) {
        float tmoment = integral(ca,cb,&Core_Base::moment_integrand);
        float tmass = integral(ca,cb,&Core_Base::mass_integrand);
        p = tmoment/(clength*tmass);
    }
    else  // for singleton points, assume mass is "midway"
        p = 0.5;

    return p;
}
//=========================================================================
Core_Interpolant::
Core_Interpolant (float time, Core_Base &core0, Core_Base &core1)
: Core_Base(0,1), ccore0(core0), ccore1(core1)
{
    ctime = time;
    clength = (1-ctime)*ccore0.curve_length()+ctime*ccore1.curve_length();
```

```
    // initial p parameter
    cinit_param = 0;

    // initial position
    Vector pos0 = ccore0.position(0), pos1 = ccore1.position(0);
    cinit_pos.x = (1-ctime)*pos0.x+ctime*pos1.x;
    cinit_pos.y = (1-ctime)*pos0.y+ctime*pos1.y;

    // initial direction
    Vector dir0 = ccore0.direction(0), dir1 = ccore1.direction(0);
    float cs_angle = dir0.x*dir1.x+dir0.y*dir1.y;
    float sn_angle = dir0.x*dir1.y-dir1.x*dir0.y;
    float angle = ctime*atan2(sn_angle,cs_angle);
    cs_angle = cos(angle);
    sn_angle = sin(angle);
    cinit_dir.x = cs_angle*dir0.x-sn_angle*dir0.y;
    cinit_dir.y = sn_angle*dir0.x+cs_angle*dir0.y;
}
//---------------------------------------------------------------------------
void Core_Interpolant::
initial_parameter (float p)
{
    if ( 0 <= p && p <= 1 )
        cinit_param = p;
    else
        cinit_param = 0.0;
}
//---------------------------------------------------------------------------
void Core_Interpolant::
initial_position (Vector pos)
{
    cinit_pos = pos;
}
//---------------------------------------------------------------------------
void Core_Interpolant::
initial_direction (Vector dir)
{
    float len = sqrt(dir.x*dir.x+dir.y*dir.y);
    if ( len > 0 ) {
        dir.x /= len;
        dir.y /= len;
    }
    cinit_dir = dir;
}
//---------------------------------------------------------------------------
Vector Core_Interpolant::
position (float p)
```

```
{
    if ( p == cinit_param )
        return cinit_pos;

    Vector pos = cinit_pos, dir = cinit_dir;
    ode_solve(cinit_param,p,pos,dir);
    return pos;
}
//--------------------------------------------------------------------------
Vector Core_Interpolant::
direction (float p)
{
    if ( p == cinit_param )
        return cinit_dir;

    Vector pos = cinit_pos, dir = cinit_dir;
    ode_solve(cinit_param,p,pos,dir);
    float len = sqrt(dir.x*dir.x+dir.y*dir.y);
    if ( len > 0 ) {
        dir.x /= len;
        dir.y /= len;
    }

    return dir;
}
//--------------------------------------------------------------------------
float Core_Interpolant::
radius (float p)
{
    return (1-ctime)*ccore0.radius(p)+ctime*ccore1.radius(p);
}
//--------------------------------------------------------------------------
float Core_Interpolant::
speed (float p)
{
    float q0 = ccore0.invlen(p), q1 = ccore1.invlen(p);
    return (1-ctime)*ccore0.speed(q0)+ctime*ccore1.speed(q1);
}
//--------------------------------------------------------------------------
float Core_Interpolant::
length (float p)
{
    float q0 = ccore0.invlen(p), q1 = ccore1.invlen(p);
    float result = (1-ctime)*ccore0.length(q0)+ctime*ccore1.length(q1);
    return result;
}
//--------------------------------------------------------------------------
float Core_Interpolant::
```

```
curvature (float p)
{
    return (1-ctime)*ccore0.curvature(p)+ctime*ccore1.curvature(p);
}
//-------------------------------------------------------------------------
float Core_Interpolant::
mass ()
{
    return (1-ctime)*ccore0.mass()+ctime*ccore1.mass();
}
//-------------------------------------------------------------------------
float Core_Interpolant::
mass_normal_arclength ()
{
    float mna0 = ccore0.mass_normal_arclength();
    float mna1 = ccore1.mass_normal_arclength();
    return (1-ctime)*mna0+ctime*mna1;
}
//=========================================================================
```

7.3 coredraw.h

```
#ifndef COREDRAW_H
#define COREDRAW_H

#include "core.h"
#include "images.h"

class CoreDraw
{
private:
    float cd_xmin, cd_ymin, cd_xmax, cd_ymax;
    float cd_xrange, cd_yrange;
    int cd_meshsize;
    Image_SHORT cd_im;

    void disk (long xc, long yc, long radius);
    void line (long x0, long y0, long r0, long x1, long y1, long r1);
public:
    CoreDraw(float xmin, float ymin, float xmax, float ymax, int meshsize=10);

    CoreDraw& draw (Core_Base &core);
    CoreDraw& clear ();
    void save (char *filename);
};

#endif
```

7.4 coredraw.c

```
#include "coredraw.h"
static Message msg("coredraw.cpp");

#define IMSIZE 256L
#define IMSIZEM (IMSIZE-1)

//============================================================================
CoreDraw::
CoreDraw (float xmin, float ymin, float xmax, float ymax, int meshsize)
: cd_im(Coordinate(2,IMSIZE))
{
    if ( (cd_xmin = xmin) >= (cd_xmax = xmax) )
        msg.fatal("CoreDraw - xmin < xmax is required");
    if ( (cd_ymin = ymin) >= (cd_ymax = ymax) )
        msg.fatal("CoreDraw - ymin < ymax is required");
    if ( (cd_meshsize = meshsize) < 2 )
        msg.fatal("CoreDraw - mesh size must be at least 2");

    cd_xrange = cd_xmax - cd_xmin;
    cd_yrange = cd_ymax - cd_ymin;
}
//----------------------------------------------------------------------------
void CoreDraw::
disk (long xc, long yc, long radius)
{
    // Bresenham's circle drawing algorithm with horizontal line fill
    for (long x = 0, y = radius, decision = 3-2*y; x <= y; x++) {
        for (long i = -x; i <= x; i++) {
            cd_im(xc+i,yc+y) = 0xFF;
            cd_im(xc+i,yc-y) = 0xFF;
        }
        for (i = -y; i <= y; i++) {
            cd_im(xc+i,yc+x) = 0xFF;
            cd_im(xc+i,yc-x) = 0xFF;
        }

        if ( decision >= 0 ) {
            decision += 4*(x-y)+6;
            y--;
        }
        else
            decision += 4*x+2;
    }
}
//----------------------------------------------------------------------------
void CoreDraw::
```

```
line (long x0, long y0, long r0, long x1, long y1, long r1)
{
    // Bresenham's line drawing algorithm with disks drawn instead of pixels
    long dx = x1-x0, dy = y1-y0;
    if ( dx == 0 && dy == 0 ) {
        disk(x0,y0,r0);
        return;
    }
    long sx = 2*( (dx > 0) ) - 1, sy = 2*( (dy > 0) ) - 1;
    long ax = 2*(dx = labs(dx)), ay = 2*(dy = labs(dy));

    if ( dx >= dy ) {
        for (long dec = ay-dx, x = x0, y = y0;  ; dec += ay, x += sx) {
            long radius = labs((r0*(x1-x)+r1*(x-x0))/dx);
            disk(x,y,radius);
            if ( x == x1 )
                break;
            if ( dec >= 0 ) {
                dec -= ax;
                y += sy;
            }
        }
    }
    else {
        for (long dec = ax-dy, x = x0, y = y0;  ; dec += ax, y += sy) {
            long radius = labs((r0*(y1-y)+r1*(y-y0))/dy);
            disk(x,y,radius);
            if ( y == y1 )
                break;
            if ( dec >= 0 ) {
                dec -= ay;
                x += sx;
            }
        }
    }
}
//------------------------------------------------------------------------
CoreDraw& CoreDraw::
draw (Core_Base &core)
{
    Vector point = core.position(0);
    long x0 = (long)(IMSIZEM*(point.x-cd_xmin)/cd_xrange);
    long y0 = (long)(IMSIZEM*(point.y-cd_ymin)/cd_yrange);
    long r0 = (long)(IMSIZEM*(core.radius(0)/cd_xrange));

    for (int i = 1; i <= cd_meshsize; i++) {
        float p = i/(float)cd_meshsize;
```

```
        point = core.position(p);
        long x1 = (long)(IMSIZEM*(point.x-cd_xmin)/cd_xrange);
        long y1 = (long)(IMSIZEM*(point.y-cd_ymin)/cd_yrange);
        long r1 = (long)(IMSIZEM*(core.radius(p)/cd_xrange));

        line(x0,y0,r0,x1,y1,r1);
        x0 = x1;
        y0 = y1;
        r0 = r1;
    }

    return *this;
}
//-----------------------------------------------------------------------------
CoreDraw& CoreDraw::
clear ()
{
    cd_im = 0;
    return *this;
}
//-----------------------------------------------------------------------------
void CoreDraw::
save (char *filename)
{
    cd_im.save(filename);
}
//-----------------------------------------------------------------------------
```

7.5 interp.c

```
#include <stdio.h>
#include "images.h"
#include "core.h"
#include "coredraw.h"

float pi = 4.0*atan(1.0);

#if 1
// ----- object 0 = counterclockwise spiral
float a0 = 0, b0 = 5*pi/2;
float X0 (float q) { return q*cos(q); }
float Y0 (float q) { return q*sin(q); }
float X0d (float q) { return -q*sin(q)+cos(q); }
float Y0d (float q) { return q*cos(q)+sin(q); }
float X0dd (float q) { return -q*cos(q)-2*sin(q); }
float Y0dd (float q) { return -q*sin(q)+2*cos(q); }
float R0 (float q) { return 0.5; }
```

```
// ----- object 1 = clockwise spiral
float a1 = 0, b1 = 5*pi/2;
float X1 (float q) { return -q*cos(q); }
float Y1 (float q) { return q*sin(q); }
float X1d (float q) { return q*sin(q)-cos(q); }
float Y1d (float q) { return q*cos(q)+sin(q); }
float X1dd (float q) { return q*cos(q)+2*sin(q); }
float Y1dd (float q) { return -q*sin(q)+2*cos(q); }
float R1 (float q) { return 0.5; }
#endif

#if 0
// ----- object 0 -----
// core 0
float a00 = 0, b00 = 1;
float X00 (float q) { return cos(pi*q*q); }
float Y00 (float q) { return sin(pi*q*q); }
float X00d (float q) { return -2*pi*q*sin(pi*q*q); }
float Y00d (float q) { return 2*pi*q*cos(pi*q*q); }
float X00dd (float q) { float A = pi*q*q;  return -2*pi*(2*A*cos(A)+sin(A)); }
float Y00dd (float q) { float A = pi*q*q;  return -2*pi*(2*A*sin(A)-cos(A)); }
float R00 (float q) { return 0.1; }
// core 1
float a01 = 1, b01 = 1.5;
float X01 (float q) { return 0; }
float Y01 (float q) { return q; }
float X01d (float q) { return 0; }
float Y01d (float q) { return 1; }
float X01dd (float q) { return 0; }
float Y01dd (float q) { return 0; }
float R01 (float q) { return 0.1; }

// ----- object 1 -----
// core 0
float a10 = -1, b10 = 1;
float X10 (float q) { return -q; }
float Y10 (float q) { return 0; }
float X10d (float q) { return -1; }
float Y10d (float q) { return 0; }
float X10dd (float q) { return 0; }
float Y10dd (float q) { return 0; }
float R10 (float q) { return 0.02+0.09*(q+1); }
// core 1
float a11 = 0, b11 = 0;
float X11 (float q) { return 0; }
float Y11 (float q) { return 0; }
float X11d (float q) { return 0; }
```

```
float Y11d (float q) { return 0; }
float X11dd (float q) { return 0; }
float Y11dd (float q) { return 0; }
float R11 (float q) { return 0.11; }
#endif

//============================================================================
main ()
{
#if 1
    Function X0jet[] = { X0, X0d, X0dd };
    Function Y0jet[] = { Y0, Y0d, Y0dd };
    Core_Continuous c0(a0,b0,X0jet,Y0jet,R0);

    Function X1jet[] = { X1, X1d, X1dd };
    Function Y1jet[] = { Y1, Y1d, Y1dd };
    Core_Continuous c1(a1,b1,X1jet,Y1jet,R1);

    // draw cores
    CoreDraw cd(-32,-32,32,32,50);
    const int N = 16;
    char filename[128];

    // core t = 0
    printf("core %f\n",(float)0.0);
    cd.draw(c0).save("core0.im");

    // cores t = i/N, 1 <= t <= N-1
    for (int i = 1; i < N; i++) {
        float t = i/(float)N;
        printf("core %f\n",t);
        Core_Interpolant ci(t,c0,c1);
        sprintf(filename,"core%d.im",i);
        cd.clear().draw(ci).save(filename);
    }

    // core t = 1
    printf("core %f\n",(float)1.0);
    sprintf(filename,"core%d.im",N);
    cd.clear().draw(c1).save(filename);
#endif

#if 0
    Function X00jet[] = { X00, X00d, X00dd };
    Function Y00jet[] = { Y00, Y00d, Y00dd };
    Core_Continuous c00(a00,b00,X00jet,Y00jet,R00);

    Function X01jet[] = { X01, X01d, X01dd };
```

```
    Function Y01jet[] = { Y01, Y01d, Y01dd };
    Core_Continuous c01(a01,b01,X01jet,Y01jet,R01);

    Function X10jet[] = { X10, X10d, X10dd };
    Function Y10jet[] = { Y10, Y10d, Y10dd };
    Core_Continuous c10(a10,b10,X10jet,Y10jet,R10);

    Function X11jet[] = { X11, X11d, X11dd };
    Function Y11jet[] = { Y11, Y11d, Y11dd };
    Core_Continuous c11(a11,b11,X11jet,Y11jet,R11);

    // draw cores
    CoreDraw cd(-2,-2,2,2,25);
    const int N = 16;
    char filename[128];

    // core t = 0
    printf("core %f\n",(float)0.0);
    cd.draw(c00).draw(c01).save("core0.im");

    // cores t = i/N, 1 <= t <= N-1
    for (int i = 1; i < N; i++) {
        float t = i/(float)N;
        printf("core %f\n",t);
        Core_Interpolant ci0(t,c00,c10);

        ci0.initial_parameter(0.5);
        Vector ipos = { 0.0, 1-t };
        ci0.initial_position(ipos);
        Vector idir = { -1.0, 0.0 };
        ci0.initial_direction(idir);

        Core_Interpolant ci1(t,c01,c11);

        sprintf(filename,"core%d.im",i);
        cd.clear().draw(ci0).draw(ci1).save(filename);
    }

    // core t = 1
    printf("core %f\n",(float)1.0);
    sprintf(filename,"core%d.im",N);
    cd.clear().draw(c10).draw(c11).save(filename);
#endif

    return 0;
}
```