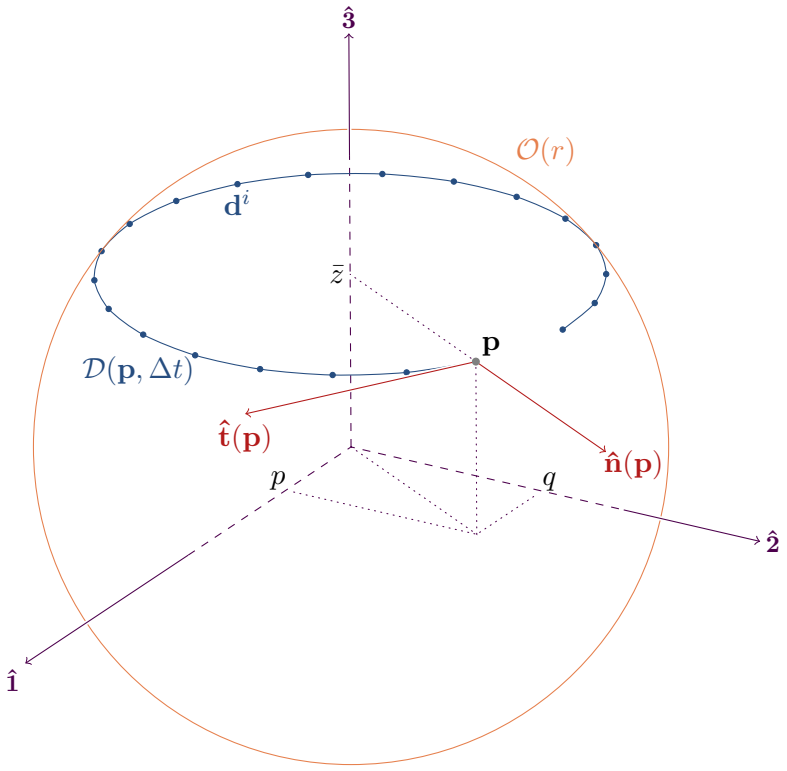
Parametrisation of Irregular Paths on Surfaces Embedded in $\mathbb{R}^3$

Paul Kotschy

8 September 2016

Compiled on March 20, 2025



Abstract

It is straightforward[1] to conceptualise both the embedding of a surface in some real space, and the existence of a path on that embedded surface. But this straightforwardness belies the difficulty in parametrising such paths with a single real variable. In fact, such parametrisation is possible only for either the simplest paths, or for paths for which one or more exploitable global property is known.

This study offers a systematic approach for addressing the difficulty in parametrising arbitrary paths. The approach involves "stepping off the global vantage point" of the surface landscape, and becoming "locally and subjectively immersed" in the surface, the path, and the path's trajectory. Specifically, attempts at finding global parametrisations of paths in three-dimensional space will give way to finding a connected set of locally-derived parametrisations, all of which are regular and tractable.

[1] paul.kotschy@gmail.com

Contents

1 Introduction

It is relatively straightforward to conceptualise the embedding of a surface in some space. Indeed, a panoramic view of an undulating countryside is exactly that. It is a view of a two-dimensional real surface embedded in our familiar real three-dimensional world. The surface, i.e., the countryside, is two-dimensional because any point on the surface must have its altitude constrained if the point is to be located on the surface. That is, the point requires only two numbers for its unique identification, namely, its length and breadth separation from some reference point.

It is also straightforward to conceptualise a path on the surface embedded in the space. From my high-up vantage point, looking out at the countryside, I notice a road winding its way, sometimes around hills, and sometimes up and over. I quickly realise that the path, i.e., the road, must be a one-dimensional geometrical object in our familiar world, because any point on the path must not only have its altitude be constrained, but its length and breadth must also be related to each other if the point is to remain on the path. That is, the point on the path requires only a single real number for its unique identification.

As I watch a distant motorcar being driven along that road, I realise that the car is simply following a trajectory in space as constrained by the path. And as the car moves through space, I imagine that single number changing. Some point on the road, some value. Some other point, some other value. And so on.

I wonder, is it possible to happily parametrise the entire path in this way with a single varying number? In other words, is it possible to take an holistic and objective view of a path on a surface embedded in a space, and to parametrise the entire path with a single real variable? Yes, but only for either the simplest paths, or for paths for which one or more exploitable global property is known. For most paths, however, it is not possible. I shall call them *irregular*.

This study offers a systematic approach for addressing the difficulty in parametrising irregular paths. It derives from "stepping off the global vantage point" and becoming "locally and subjectively immersed" in the surface, the path, and the path's trajectory. Specifically, attempts at finding global parametrisations of irregular paths in three-dimensional space will give way to finding a connected set of locally-derived parametrisations, all of which are regular.

By sacrificing global objectivity in this way, additional insights to the local geometry on the surface are obtained. These insights will show that some of the irregularities are not intrinsic to the path itself, nor to the embedded surface, but arise during attempts to parametrise the path. Indeed, I believe that the presence of irregularities hint at our deeper, more fundamental, inability to describe our world globally and objectively. We must be more content with local views.

The difficulty in parametrising irregular paths is shown by way of example in Section 2. In Section 3, the abovementioned approach is applied to parametrise, in a regular way, the contour paths of an embedded sphere. The same is done in Section 4 for the embedded "hump". In the appendix, a practical application of the insights obtained in this study is addressed. The combined use of the TEX and the Ti*k*Z typesetting systems, and my `PKREALVECTOR` C object class,[1] to produce the three-dimensional schematic diagrams in this document, is described.

2 Limits to global path parametrisation

Introductory texts on vector calculus[2, 3, 4] introduce the notion of a parametrised path (or curve) in $\mathbb{R}^3$ either as the vector-valued mapping

$$C : \mathbb{R} \to \mathbb{R}^3 \tag{1}$$

or as the set of points

$$\mathcal{C} = \{(x, y, z) \mid x = x(t), y = y(t), z = z(t),\ x, y, z, t \in \mathbb{R}\} \tag{2}$$

Each such (x, y, z) triple is called a point on $\mathcal{C}$ or a member of $\mathcal{C}$, and may be represented geometrically by a position vector derived from the (x, y, z) triple as

$$\mathbf{x}(t) = x(t)\hat{\mathbf{1}} + y(t)\hat{\mathbf{2}} + z(t)\hat{\mathbf{3}}, \quad t \in \mathbb{R} \tag{3}$$

where $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}, \hat{\mathbf{3}}\}$ is the usual orthonormal vector basis for E^3. What (1) and (2) do is to establish a relationship between points in a subset of $\mathbb{R}^3$. If we provide some value for the parameter t, then (1) will identify a unique point as a member of (2), and a unique position vector $\mathbf{x}(t)$. And as t varies, it traces out a corresponding variation in the position vector.

The notion of a parametrised path as expressed above is a "global" one. I try to visualise the entire space ($\mathbb{R}^3$) at once, while observing the path "snake" its way through the space. I am a passive observer of the path. I have no actual subjective experience of it.

Unfortunately, as I shall illustrate here, (1) and (2) capture the imagination in this way for none but the simplest of paths. Stated differently, not every path in $\mathbb{R}^3$ admits a single global parametrisation of the forms (1) and (2) which spans the full set $\mathcal{C}$, with each member deriving a corresponding position vector as per (3). The difficulty arises when there is a one-to-many relationship between the path traversal parameter t, say, and one or more of the coordinates $x(t)$, $y(t)$ or $z(t)$. To show this, consider the unit circle in $\mathbb{R}^2$ centred at the origin.

Unit circle. The governing equation of the unit circle in the $\hat{\mathbf{1}}\hat{\mathbf{2}}$ plane is, simply,

$$x^2 + y^2 = 1 \tag{4}$$

A segment of the full path on this unit circle may be parametrised with t as

$$\mathbf{x}(t) = t\hat{\mathbf{1}} + \sqrt{1 - t^2}\,\hat{\mathbf{2}}, \quad -1 \leq t \leq 1 \tag{5}$$

The parametrisation works because the pair $(t, \sqrt{1 - t^2})$, $-1 \leq t \leq 1$, satisfies the condition (4). The segment is shown in Figure 1.

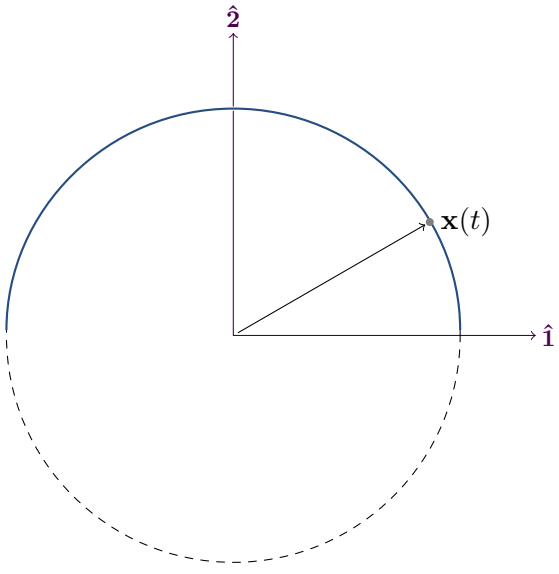


Figure 1: Upper segment of the unit circle centred at the origin.

But the parameter t in (5) cannot span the full path because for some $-1 \leq u \leq 1$, the point $(u, -\sqrt{1 - u^2})$ which is located on the lower segment of the path also satisfies the condition (4), and there is no value t in (5) which can cover that point. Of course, you might be quick to quip that the new parametrisation

$$\mathbf{x}(t) = \cos((1 + t)\pi)\hat{\mathbf{1}} + \sin((1 + t)\pi)\hat{\mathbf{2}}, \quad -1 \leq t \leq 1 \tag{6}$$

overcomes the difficulty because (6) now spans the full path. But this parametrisation happens to exploit some global property of the path, namely that $\cos^2\theta + \sin^2\theta = 1$ for all θ. That is, we must already be aware of the path's global behaviour before we can parametrise it. But for many (indeed, most) paths, such global properties are simply not available. Furthermore, knowledge of this global behaviour discourages any consideration of the path's local behaviour.

This work explores local behaviour. The work shows by way of examples that much can be gained by analysing a path's local context by "subjective immersion" in the path's trajectory on a corresponding surface embedded in $\mathbb{R}^3$. And provided we are in this way prepared to sacrifice some objectivity, the difficulties encountered in a strictly global view vanish. Specifically, *attempts at finding global parametrisations of irregular paths in* $\mathbb{R}^3$ *give way to a systematic approach to finding a connected set of locally-derived parametrisations of an irregular path, all of which are regular.* Furthermore, in keeping with the spirit of local analysis, we shall resist exploiting any global properties, such as $\cos^2\theta + \sin^2\theta = 1$.

3 Contour paths on an embedded sphere

Global perspective. We begin, heuristically, with a sphere embedded in $\mathbb{R}^3$ of radius r:

$$\mathcal{O}(r) = \{(x, y, z) \mid x^2 + y^2 + z^2 = r^2\} \tag{7}$$

In keeping with (5), a globally-centric parametrisation of the $\bar{z}$-contour path of $\mathcal{O}$ is

$$\mathbf{c}(t; r, \bar{z}) \;=\; t\hat{\mathbf{1}} \,\pm\, \sqrt{r^2 - \bar{z}^2 - t^2}\,\hat{\mathbf{2}} \,+\, \bar{z}\hat{\mathbf{3}}, \quad -\sqrt{r^2 - \bar{z}^2} \le t \le \sqrt{r^2 - \bar{z}^2} \tag{8}$$

This is actually two parametrisations, one for the '+' square root, and one for the '−' square root. Suppose however that we are concerned with the segment of the $\bar{z}$-contour path beginning at position $\mathbf{A} = \sqrt{(r^2 - \bar{z}^2)/2}\,\hat{\mathbf{1}} - \sqrt{(r^2 - \bar{z}^2)/2}\,\hat{\mathbf{2}} + \bar{z}\hat{\mathbf{3}}$, passing through $\mathbf{B} = \sqrt{r^2 - \bar{z}^2}\,\hat{\mathbf{1}} + 0\hat{\mathbf{2}} + \bar{z}\hat{\mathbf{3}}$, and ending at $\mathbf{C} = \sqrt{(r^2 - \bar{z}^2)/2}\,\hat{\mathbf{1}} + \sqrt{(r^2 - \bar{z}^2)/2}\,\hat{\mathbf{2}} + \bar{z}\hat{\mathbf{3}}$, as shown in Figure 2. This $\overline{\mathbf{ABC}}$ segment cannot be parametrised using just one of the parametrisations in (8). It requires both, with $\mathbf{A} = \mathbf{c}^-(\sqrt{(r^2 - \bar{z}^2)/2}; r, \bar{z})$, $\mathbf{B} = \mathbf{c}^-(\sqrt{r^2 - \bar{z}^2}; r, \bar{z}) = \mathbf{c}^+(\sqrt{r^2 - \bar{z}^2}; r, \bar{z})$, and $\mathbf{C} = \mathbf{c}^+(\sqrt{(r^2 - \bar{z}^2)/2}; r, \bar{z})$.

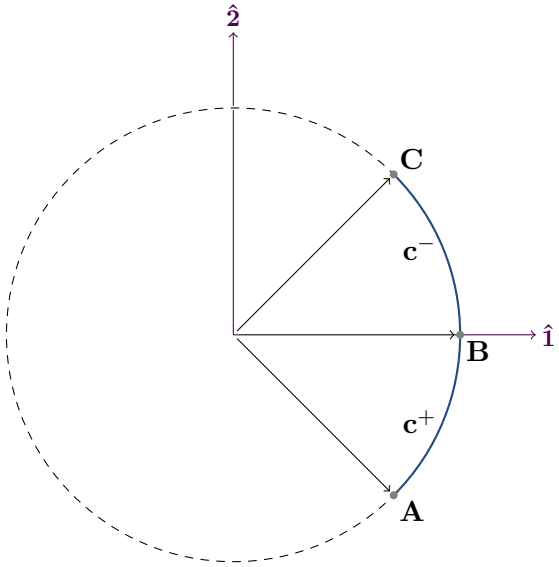


Figure 2: Path segment $\overline{\mathbf{ABC}}$ on a circle centred at the origin.

It is obviously desirable to obtain a single parametrisation of $\overline{\mathbf{ABC}}$. Doing so would place all such path segments on $\mathcal{O}$ on an equal conceptual footing, showing that the irregular character of the **B** position in Figure 2 is entirely a result of the limitation of the global parametrisation (8). There is nothing intrinsically different between **B**, **A** and **C**.

Local perspective. To obtain a desired single parametrisation of $\overline{\mathbf{ABC}}$, we must place ourselves on the path at some specified position

$$\mathbf{p} \equiv \mathbf{c}(p; r, \bar{z}) = p\hat{\mathbf{1}} + q(p, r, \bar{z})\hat{\mathbf{2}} + \bar{z}\hat{\mathbf{3}}$$

Once $\mathbf{c}(p; r, \bar{z})$ has been used to fix $\mathbf{p}$, $\mathbf{p}$'s components will be deemed to be constants. So in the ensuing analysis, instead of regarding q as a function of p, r, and $\bar{z}$, it will be considered a constant parameter, along with p and $\bar{z}$:

$$\mathbf{p} = p\hat{\mathbf{1}} + q\hat{\mathbf{2}} + \bar{z}\hat{\mathbf{3}}$$

From this local position on $\mathcal{O}$, we shall describe our local view of the landscape, and thereby obtain a local parametrisation.

From a global perspective of the path, $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}, \hat{\mathbf{3}}\}$ is an obvious choice of orthonormal vector basis spanning $\mathbb{R}^3$. Indeed, it is with this basis that the sphere's geometry has been specified, because any point on $\mathcal{O}$ (or, point as a member of $\mathcal{O}$) may be represented geometrically by one of the positions

$$\mathbf{x}(x, y) = x\hat{\mathbf{1}} + y\hat{\mathbf{2}} + \sqrt{r^2 - x^2 - y^2}\,\hat{\mathbf{3}}, \quad \text{or}$$
$$\mathbf{x}(x, y) = x\hat{\mathbf{1}} + y\hat{\mathbf{2}} - \sqrt{r^2 - x^2 - y^2}\,\hat{\mathbf{3}}$$

Globally, our position of reference is obviously the origin. However, once we shift our position from the origin to $\mathbf{p}$, $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}, \hat{\mathbf{3}}\}$ is no longer an obvious choice of basis. Instead, a reasonable choice would be: the unit tangent vector at $\mathbf{p}$; a unit vector at $\mathbf{p}$ perpendicular to the unit tangent vector but still aligned with the $\hat{\mathbf{1}}\hat{\mathbf{2}}$ plane; and a unit vector perpendicular to both, namely, the $\hat{\mathbf{3}}$ vector. To calculate the unit tangent vector at $\mathbf{p}$ on the z-contour path, from (7):

$$2y(x)\frac{\mathrm{d}y}{\mathrm{d}x} = -2x \quad \Rightarrow \quad \frac{\mathrm{d}y}{\mathrm{d}x} = -\frac{x}{y(x)} = \mp\frac{x}{\sqrt{r^2 - z^2 - x^2}}$$

The derivative $\mathrm{d}z/\mathrm{d}x$ vanishes because along the $\bar{z}$-contour path $\mathbf{c}(t; r, \bar{z})$, z is constant. So

$$\frac{\mathrm{d}\mathbf{c}(t; r, \bar{z})}{\mathrm{d}t} = \hat{\mathbf{1}} - \frac{t}{y(t)}\hat{\mathbf{2}} + 0\hat{\mathbf{3}}$$
$$\left|\frac{\mathrm{d}\mathbf{c}(t; r, \bar{z})}{\mathrm{d}t}\right| = \frac{\sqrt{r^2 - \bar{z}^2}}{|y(t)|}$$

The unit tangent vector at the $\mathbf{c}(t; r, \bar{z})$ position along the path is therefore

$$\hat{\mathbf{t}}(\mathbf{c}(t; r, \bar{z})) = \frac{\mathrm{d}\mathbf{c}(t; r, \bar{z})}{\mathrm{d}t} \Big/ \left|\frac{\mathrm{d}\mathbf{c}(t; r, \bar{z})}{\mathrm{d}t}\right| = \frac{y(t)\hat{\mathbf{1}} - t\hat{\mathbf{2}} + 0\hat{\mathbf{3}}}{\sqrt{r^2 - \bar{z}^2}} \tag{9}$$

It is immediately evident that the unit vectors perpendicular to $\hat{\mathbf{t}}$ at the same position and aligned with the $\hat{\mathbf{1}}\hat{\mathbf{2}}$ plane are $(\pm t\hat{\mathbf{1}} \pm y(t)\hat{\mathbf{2}} + 0\hat{\mathbf{3}})/\sqrt{r^2 - \bar{z}^2}$.

But I wish here to appeal to a systematic approach to calculating the normal vector[2]—an approach which is not restricted to paths in $\mathbb{R}^3$. Since $\mathbf{c}(t; r, \bar{z})$ is the $\bar{z}$-contour path on the embedded surface $\mathcal{O}$, the function gradient $\nabla_{(x,y)}z(x, y)$ is orthogonal to the level curve corresponding to $\mathbf{c}(t; r, \bar{z})$, namely, the level curve $\mathbf{c}(t; r, \bar{z}) - \bar{z}\hat{\mathbf{3}}$. That is, a normal vector at the $\mathbf{c}(t; r, \bar{z})$ position is

$$\mathbf{n}(\mathbf{c}(t; r, \bar{z})) = \nabla_{(x,y)}z(t, y(t)) = \frac{\partial z(t, y(t))}{\partial x}\hat{\mathbf{1}} + \frac{\partial z(t, y(t))}{\partial y}\hat{\mathbf{2}}$$
$$= \mp\frac{t}{\bar{z}}\,\hat{\mathbf{1}} \mp \frac{y(t)}{\bar{z}}\,\hat{\mathbf{2}} + 0\hat{\mathbf{3}}$$
$$|\mathbf{n}(\mathbf{c}(t; r, \bar{z}))| = \frac{\sqrt{r^2 - \bar{z}^2}}{|\bar{z}|}$$

One of the unit normal vectors is therefore

$$\hat{\mathbf{n}}(\mathbf{c}(t; r, \bar{z})) = \frac{t\hat{\mathbf{1}} + y(t)\hat{\mathbf{2}} + 0\hat{\mathbf{3}}}{\sqrt{r^2 - \bar{z}^2}} \tag{10}$$

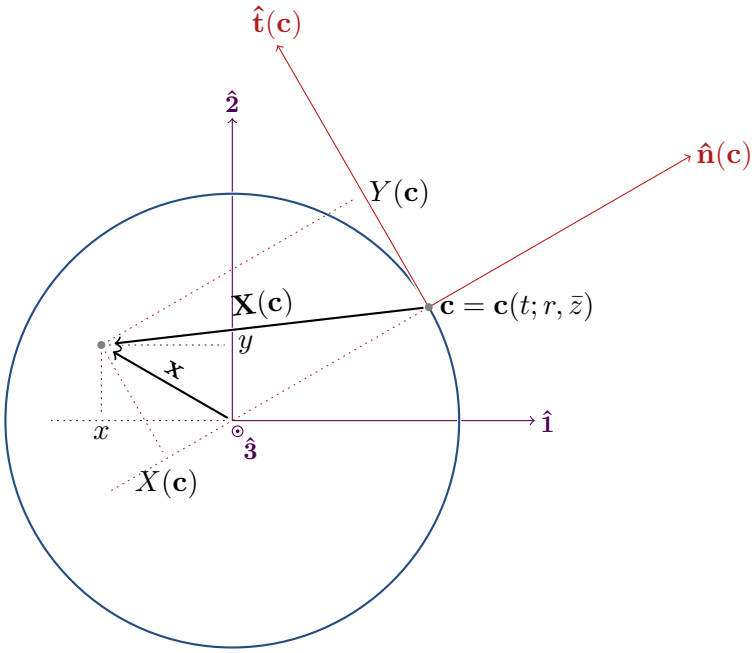


Figure 3: Local orthonormal vector basis $\{\hat{\mathbf{n}}(\mathbf{c}), \hat{\mathbf{t}}(\mathbf{c}), \hat{\mathbf{3}}\}$ at position $\mathbf{c} = \mathbf{c}(t; r, \bar{z})$ on the $\bar{z}$-contour path of the sphere $\mathcal{O}$. A position in $\mathbb{R}^3$ is represented as $\mathbf{x} = x\hat{\mathbf{1}} + y\hat{\mathbf{2}} + z\hat{\mathbf{3}}$ and as $\mathbf{X}(\mathbf{c}) = X(\mathbf{c})\hat{\mathbf{n}}(\mathbf{c}) + Y(\mathbf{c})\hat{\mathbf{t}}(\mathbf{c}) + Z(\mathbf{c})\hat{\mathbf{3}}$.

as expected. The unit vectors $\hat{\mathbf{t}}$ and $\hat{\mathbf{n}}$ are positioned conceptually to pass through $\mathbf{c}(t; r, \bar{z})$, as shown in Figure 3.

From our global perspective, any point in $\mathbb{R}^3$ may be represented by the position vector

$$\mathbf{x} = x\hat{\mathbf{1}} + y\hat{\mathbf{2}} + z\hat{\mathbf{3}}$$

But from our local perspective, located at the position $\mathbf{c} = \mathbf{c}(t; r, \bar{z})$ on the path, the same point may be represented by

$$\mathbf{X}(\mathbf{c}) = X(\mathbf{c})\hat{\mathbf{n}}(\mathbf{c}) + Y(\mathbf{c})\hat{\mathbf{t}}(\mathbf{c}) + Z(\mathbf{c})\hat{\mathbf{3}} \tag{11}$$

The $\mathbf{c}$ and $\mathbf{x}$ position vectors are specified relative to the global origin. Graphically, their tails are positioned at the origin. But the $\mathbf{X}$ position vector is specified relative to a local origin at the global position $\mathbf{c}(t; r, \bar{z})$, as shown in Figure 3. It therefore follows that

$$\mathbf{X} = \mathbf{x} - \mathbf{c} \tag{12}$$

The expressions in (9), (10), (11) and (12) hold for any value of t satisfying the constraint in (8). And when $t = p$, the position is on the $\bar{z}$-contour is $\mathbf{p}$, so

$$\frac{1}{\sqrt{r^2 - \bar{z}^2}} X(p\hat{\mathbf{1}} + q\hat{\mathbf{2}}) + \frac{1}{\sqrt{r^2 - \bar{z}^2}} Y(q\hat{\mathbf{1}} - p\hat{\mathbf{2}}) + Z\hat{\mathbf{3}} = x\hat{\mathbf{1}} + y\hat{\mathbf{2}} + z\hat{\mathbf{3}} - p\hat{\mathbf{1}} - q\hat{\mathbf{2}} - \bar{z}\hat{\mathbf{3}}$$

from which we obtain

$$\begin{aligned} x &= \frac{p}{\sqrt{r^2 - \bar{z}^2}} X + \frac{q}{\sqrt{r^2 - \bar{z}^2}} Y + p \\ y &= \frac{q}{\sqrt{r^2 - \bar{z}^2}} X - \frac{p}{\sqrt{r^2 - \bar{z}^2}} Y + q \\ z &= Z + \bar{z} \end{aligned} \tag{13}$$

So if we are located at the position $\mathbf{p}$ on the $\bar{z}$-contour path, then any arbitrary local position $\mathbf{X}$ identifies with the arbitrary global position $\mathbf{x}$, and their respective representations are connected by (13). And in particular, if $\mathbf{x}$ also happens to correspond to a point on $\mathcal{O}$ (Eq. (7)), then

$$\left(\frac{p}{\sqrt{r^2 - \bar{z}^2}} X + \frac{q}{\sqrt{r^2 - \bar{z}^2}} Y + p\right)^2 + \left(\frac{q}{\sqrt{r^2 - \bar{z}^2}} X - \frac{q}{\sqrt{r^2 - \bar{z}^2}} Y + q\right)^2 + (Z + \bar{z})^2 = r^2$$

Expanding the terms and after some algebraic manipulation:

$$\left(X + \sqrt{r^2 - \bar{z}^2}\right)^2 + Y^2 + (Z + \bar{z})^2 = r^2$$

We conclude that if we are located at $\mathbf{p} = \mathbf{c}(p; r, \bar{z})$ on the $\bar{z}$-contour path of the embedded sphere $\mathcal{O}$ in $\mathbb{R}^3$, as shown in Figure 4, then our local "experience" of the geometry is *not* that of the globally embedded $\mathcal{O}$. Instead, our experience is that of the embedded sphere

$$\bar{\mathcal{O}}(r, \bar{z}) = \{\, (X, Y, Z) \mid \left(X + \sqrt{r^2 - \bar{z}^2}\right)^2 + Y^2 + (Z + \bar{z})^2 = r^2 \,\}$$

To be sure, $\mathcal{O}$ and $\bar{\mathcal{O}}$ are obviously the same geometrical object embedded in $\mathbb{R}^3$. The former is just a globally objective view of it, while the latter is a locally subjective view taken from the position $\mathbf{p}$.

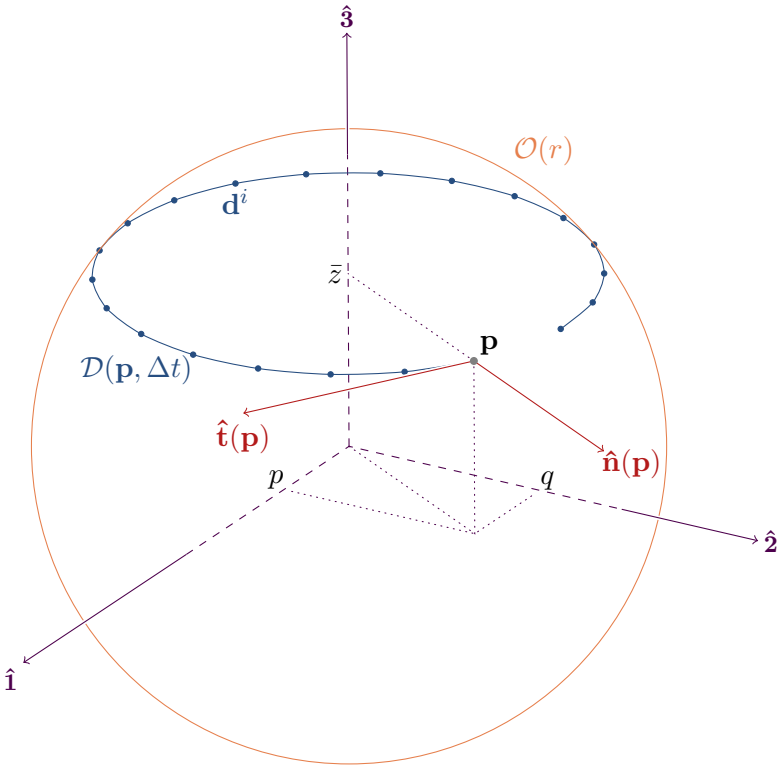


Figure 4: Sphere $\mathcal{O}(r)$ specified by (7), showing a subset of the contour path set $\mathcal{D}(\mathbf{p}, \Delta t)$ passing through the position $\mathbf{p}$. The contour path was calculated using (17) on page 10, and drawn computationally. (Refer to the annotated listing of the C source file, `spherefigure.c`, in the appendix on page 16.)

Local parametrisation. An important observation now is:

> *Provided that the local position* $\mathbf{X}$ *is sufficiently close to the local origin* $\mathbf{0}$*, a regular local parametrisation of the* $\overline{\mathbf{0X}}$ *segment of a contour path is readily available, irrespective of where on the global contour path* $\mathbf{0}$ *may be located.*

For example, the irregular global parametrisation of the $\overline{\mathbf{ABC}}$ segment (Figure 2) can be replaced with a local regular parametrisation of $\overline{\mathbf{ABC}}$. Under the $\{\hat{\mathbf{n}}(\mathbf{c}), \hat{\mathbf{t}}(\mathbf{c}), \hat{\mathbf{3}}\}$ vector basis, all positions on the global $\bar{z}$-contour path must have $Z = 0$. So while we are located at $\mathbf{p}$, $\bar{\mathcal{O}}$ admits an obvious local parametrisation of the $\bar{z}$-contour path through $\mathbf{p}$ as

$$\mathbf{D}(s; \mathbf{p}) = s\hat{\mathbf{n}}(\mathbf{p}) \pm \sqrt{r^2 - \bar{z}^2 - \left(s + \sqrt{r^2 - \bar{z}^2}\right)^2}\,\hat{\mathbf{t}}(\mathbf{p}) + 0\hat{\mathbf{3}}, \quad -\sqrt{r^2 - \bar{z}^2} \le s \le 0$$

To parametrise the local path $\mathbf{D}$ with a parameter whose domain is $[0,1]$, and such that $\mathbf{D}=\mathbf{0}$ when the parameter's value is 0, consider the transformation $s=-2\sqrt{r^2-\bar{z}^2}\,t$, giving

$$\mathbf{D}(t;\mathbf{p}) = 2\sqrt{r^2-\bar{z}^2}\left(-t\hat{\mathbf{n}}(\mathbf{p}) \pm \sqrt{t(1-t)}\,\hat{\mathbf{t}}(\mathbf{p}) + 0\hat{\mathbf{3}}\right), \quad 0\le t\le 1 \tag{14}$$

Locally-centric global reparametrisation. Eq. (14) is a local counterpart of (8). We may use (14) to reparametrise the global perspective of the $\bar{z}$-contour path, but limited to a path segment near $\mathbf{p}$. Substituting (14) into (13):

$$\begin{aligned}
x &= \frac{p}{\sqrt{r^2-\bar{z}^2}}\left(-2\sqrt{r^2-\bar{z}^2}\,t\right) \pm \frac{q}{\sqrt{r^2-\bar{z}^2}}\left(2\sqrt{r^2-\bar{z}^2}\sqrt{t(1-t)}\right)+p\\
&= (1-2t)p \pm 2\sqrt{t(1-t)}\,q\\
y &= \frac{q}{\sqrt{r^2-\bar{z}^2}}\left(-2\sqrt{r^2-\bar{z}^2}\,t\right) \mp \frac{p}{\sqrt{r^2-\bar{z}^2}}\left(2\sqrt{r^2-\bar{z}^2}\sqrt{t(1-t)}\right)+q\\
&= (1-2t)q \mp 2\sqrt{t(1-t)}\,p\\
z &= \bar{z}
\end{aligned}$$

We are free to choose the '+' or the '−' parametrisation near $\mathbf{p}$. In what follows, I choose the '−' one. A locally-centric reparametrisation of the $\bar{z}$-contour path on $\mathcal{O}$ is therefore

$$\boxed{\begin{aligned}
\mathbf{d}(t;\mathbf{p}) = &\left((1-2t)(\mathbf{p}\cdot\hat{\mathbf{1}}) \pm 2\sqrt{t(1-t)}\,(\mathbf{p}\cdot\hat{\mathbf{2}})\right)\hat{\mathbf{1}}\\
&+\left(\mp 2\sqrt{t(1-t)}\,(\mathbf{p}\cdot\hat{\mathbf{1}}) + (1-2t)(\mathbf{p}\cdot\hat{\mathbf{2}})\right)\hat{\mathbf{2}}\\
&+(\mathbf{p}\cdot\hat{\mathbf{3}})\,\hat{\mathbf{3}}, \quad 0\le t\le 1
\end{aligned}} \tag{15}$$

If we represent the basis vectors with:

$$\hat{\mathbf{1}}=\begin{bmatrix}1\\0\\0\end{bmatrix}, \quad \hat{\mathbf{2}}=\begin{bmatrix}0\\1\\0\end{bmatrix}, \quad \hat{\mathbf{3}}=\begin{bmatrix}0\\0\\1\end{bmatrix} \tag{16}$$

then a matrix representation of the reparametrisation is

$$\begin{aligned}
\mathbf{d}(t;\mathbf{p}) &= \begin{bmatrix}1-2t & \pm 2\sqrt{t(1-t)} & 0\\ \mp 2\sqrt{t(1-t)} & 1-2t & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}p\\q\\\bar{z}\end{bmatrix}\\
&= \mathbf{E}(t)\mathbf{p}
\end{aligned}$$

Consider again the $\overline{\mathbf{ABC}}$ path segment on the $\bar{z}$-contour, shown in Figure 2. If we set $p=\sqrt{(r^2-\bar{z}^2)/2}$ and $q=-\sqrt{(r^2-\bar{z}^2)/2}$, so $\mathbf{A}=p\hat{\mathbf{1}}-p\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}}$. Then from (15)

$$\begin{aligned}
\mathbf{d}^-(t;\mathbf{p}) = &\,p\left((1-2t)+2\sqrt{t(1-t)}\right)\hat{\mathbf{1}}\\
&+p\left(-(1-2t)+2\sqrt{t(1-t)}\right)\hat{\mathbf{2}}\\
&+\bar{z}\hat{\mathbf{3}}, \quad 0\le t\le 1
\end{aligned}$$

And specifically

$$\begin{aligned}
\mathbf{d}^-(0;\mathbf{p}) &= p\hat{\mathbf{1}}-p\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}} = \mathbf{A}\\
\mathbf{d}^-(\tfrac{1}{2}(1-\tfrac{1}{\sqrt{2}});\mathbf{p}) &= \sqrt{r^2-\bar{z}^2}\,\hat{\mathbf{1}}+0\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}} = \mathbf{B}\\
\mathbf{d}^-(\tfrac{1}{2};\mathbf{p}) &= p\hat{\mathbf{1}}+p\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}} = \mathbf{C}\\
\mathbf{d}^-(1;\mathbf{p}) &= -p\hat{\mathbf{1}}+p\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}}
\end{aligned}$$

This shows clearly that this single locally-centric reparametrisation happily spans $\overline{\mathbf{ABC}}$!

An important result now is:

For an arbitrarily large segment of the contour path passing through the position $\mathbf{p}$ *located on the sphere* $\mathcal{O}(r)$*, there exists a set* $\mathcal{D}(\mathbf{p}, \Delta t)$ *of connected regular parametrisations suggested by* (15)*.*

$$\begin{aligned}\mathcal{D}(\mathbf{p},\Delta t) = \Big\{\, \mathbf{d}(t;\mathbf{d}^i) = &\left[(1-2t)(\mathbf{d}^i\cdot\hat{\mathbf{1}}) \;\pm\; 2\sqrt{t(1-t)}\,(\mathbf{d}^i\cdot\hat{\mathbf{2}})\right]\hat{\mathbf{1}} \\ &+\left[(1-2t)\,(\mathbf{d}^i\cdot\hat{\mathbf{2}}) \;\mp\; 2\sqrt{t(1-t)}\,(\mathbf{d}^i\cdot\hat{\mathbf{1}})\right]\hat{\mathbf{2}} \\ &+(\mathbf{p}\cdot\hat{\mathbf{3}})\;\hat{\mathbf{3}} \\ &\mid \mathbf{d}^i = \mathbf{d}(\Delta t;\mathbf{d}^{i-1});\; \mathbf{d}^0=\mathbf{p};\; i=1,2,3,\ldots;\; 0\le t\le 1\,\Big\}\end{aligned} \tag{17}$$

In Figure 4, a segment of the contour path is shown. The segment represents a subset of $\mathcal{D}(\mathbf{p}, \Delta t)$. The segment was calculated using (17) and drawn computationally. Refer to the annotated listing of the C source file, `spherefigure.c`, in the appendix on page 16.

4 Contour paths on an embedded hump

Global perspective. The $\bar{z}$-contours of the $\mathcal{O}$ sphere (Eq. (7)) are, trivially, a family of circles of radius $\sqrt{r^2-\bar{z}^2}$. In Section 3, analysis of their contour paths showed the benefit of a local and subjective immersion in a path's trajectory (Eq. (14)). In this section, a less trivial—but still analytically tractable—surface embedded in $\mathbb{R}^3$ is considered, namely the "hump" of height h and centred at $a\hat{\mathbf{1}}+b\hat{\mathbf{2}}+0\hat{\mathbf{3}}$:

$$\mathcal{H}(h,a,b) = \left\{(x,y,z)\mid z=\frac{h}{(x-a)^2+(y-b)^2+1}\right\} \tag{18}$$

A subset of $\mathcal{H}(h,a,b)$ is shown in Figure 5 on page 13. An obvious globally-centric parametrisation of $\mathcal{H}$'s $\bar{z}$-contour path is

$$\mathbf{c}(t;h,a,b,\bar{z}) \;=\; t\hat{\mathbf{1}} \;+\; \left(b\pm\sqrt{h/\bar{z}-(t-a)^2-1}\right)\hat{\mathbf{2}} \;+\; \bar{z}\hat{\mathbf{3}}, \tag{19}$$

with $a-\sqrt{(h-\bar{z})/\bar{z}}\le t\le a+\sqrt{(h-\bar{z})/\bar{z}}$.

Again, this is actually two parametrisations, one for '+' and one for '−'. Two global parametrisations are needed for the full contour path. The path is irregular. Suppose that we are interested in the segment of the $\bar{z}$-contour path beginning at position $\mathbf{A}=(a+\sqrt{(h-\bar{z})/2\bar{z}})\hat{\mathbf{1}}+(b-\sqrt{(h-\bar{z})/2\bar{z}})\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}}$, passing through $\mathbf{B}=(a+\sqrt{(h-\bar{z})/\bar{z}})\hat{\mathbf{1}}+b\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}}$, and ending at $\mathbf{C}=(a+\sqrt{(h-\bar{z})/2\bar{z}})\hat{\mathbf{1}}+(b+\sqrt{(h-\bar{z})/2\bar{z}})\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}}$. Once again, the segment cannot be parametrised using just one of the parametrisations in (19). It requires both, with $\mathbf{A}=\mathbf{c}^-(a+\sqrt{(h-\bar{z})/2\bar{z}})$, $\mathbf{B}=\mathbf{c}^-(a+\sqrt{(h-\bar{z})/\bar{z}})=\mathbf{c}^+(a+\sqrt{(h-\bar{z})/\bar{z}})$, and $\mathbf{C}=\mathbf{c}^+(a+\sqrt{(h-\bar{z})/2\bar{z}})$.

Local perspective. To obtain a desired single parametrisation of $\overline{\mathbf{ABC}}$, we must place ourselves on the path at some specified position

$$\mathbf{p}=\mathbf{c}(p;h,a,b,\bar{z})=p\hat{\mathbf{1}}+q(p,h,a,b,\bar{z})\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}}=p\hat{\mathbf{1}}+q\hat{\mathbf{2}}+\bar{z}\hat{\mathbf{3}}$$

where, once fixed by $\mathbf{c}(p;h,a,b,\bar{z})$, $\mathbf{p}$'s components will be deemed to be fixed.

By placing ourselves on the path, the $\{\hat{\mathbf{1}},\hat{\mathbf{2}},\hat{\mathbf{3}}\}$ orthonormal vector basis is no longer an obvious choice of basis. By shifting our position from the global origin $\mathbf{0}$ to $\mathbf{p}=\mathbf{c}(p;h,a,b,\bar{z})$, and by preparing to follow the path's trajectory, an obvious choice of basis becomes $\{\hat{\mathbf{n}}(\mathbf{p}),\hat{\mathbf{t}}(\mathbf{p}),\hat{\mathbf{3}}\}$. As before, $\hat{\mathbf{t}}(\mathbf{p})$ is the unit tangent vector along the path at position $\mathbf{p}$, and $\hat{\mathbf{n}}(\mathbf{p})$ is a unit vector at $\mathbf{p}$

perpendicular to $\hat{\mathbf{t}}(\mathbf{p})$ but aligned with the $\hat{\mathbf{i}}\hat{\mathbf{2}}$ plane. To calculate the unit tangent vector at $\mathbf{p}$ on the z-contour path, from (18):

$$2(y(x)-b)\frac{\mathrm{d}y}{\mathrm{d}x}+2(x-a)=0 \quad \Rightarrow \quad \frac{\mathrm{d}y}{\mathrm{d}x}=-\frac{x-a}{y(x)-b}, \quad y\neq 0$$

The derivative $\mathrm{d}z/\mathrm{d}x$ vanishes because z is constant along the $\bar{z}$-contour path $\mathbf{c}(t;h,a,b,\bar{z})$. So

$$\frac{\mathrm{d}\mathbf{c}(p;h,a,b,\bar{z})}{\mathrm{d}t}=\hat{\mathbf{i}}-\left(\frac{p-a}{q-b}\right)\hat{\mathbf{2}}+0\hat{\mathbf{3}}$$
$$\left|\frac{\mathrm{d}\mathbf{c}(p;h,a,b,\bar{z})}{\mathrm{d}t}\right|=\sqrt{1+\left(\frac{p-a}{q-b}\right)^2}=\frac{1}{|q-b|}\sqrt{\frac{h-\bar{z}}{\bar{z}}}$$

A unit tangent vector at $\mathbf{p}$ is therefore

$$\begin{aligned}\hat{\mathbf{t}}(\mathbf{p}) \;&=\; \frac{\mathrm{d}\mathbf{c}(p;h,a,b,\bar{z})}{\mathrm{d}t}\Big/\left|\frac{\mathrm{d}\mathbf{c}(p;h,a,b,\bar{z})}{\mathrm{d}t}\right| \\ &=\sqrt{\frac{\bar{z}}{h-\bar{z}}}\Big((q-b)\hat{\mathbf{i}}\;-\;(p-a)\hat{\mathbf{2}}\;+\;0\hat{\mathbf{3}}\Big)\end{aligned} \tag{20}$$

Since $\mathbf{c}(t;h,a,b,\bar{z})$ is a contour path, $\nabla_{(x,y)}z(x,y)$ is orthogonal to the level curve $\mathbf{c}(t;h,a,b,\bar{z})-\bar{z}\hat{\mathbf{3}}$. That is, a normal vector at $\mathbf{p}$ is $\mathbf{n}(\mathbf{p})=\nabla_{(x,y)}z(p,q)$. It is then easy to verify that a unit normal vector at $\mathbf{p}$ is

$$\hat{\mathbf{n}}(\mathbf{p}) \;=\; \sqrt{\frac{\bar{z}}{h-\bar{z}}}\Big((p-a)\hat{\mathbf{i}}\;+\;(q-b)\hat{\mathbf{2}}\;+\;0\hat{\mathbf{3}}\Big) \tag{21}$$

The set $\{\hat{\mathbf{n}}(\mathbf{p}),\hat{\mathbf{t}}(\mathbf{p}),\hat{\mathbf{3}}\}$ forms an orthonormal vector basis at $\mathbf{p}$. From our global perspective, any point in $\mathbb{R}^3$ may be represented by the position vector

$$\mathbf{x}=x\hat{\mathbf{i}}+y\hat{\mathbf{2}}+z\hat{\mathbf{3}}$$

But from our local perspective, located at $\mathbf{p}$, the same point may be represented by

$$\mathbf{X}(\mathbf{p}) \;=\; X(\mathbf{p})\hat{\mathbf{n}}(\mathbf{p})+Y(\mathbf{p})\hat{\mathbf{t}}(\mathbf{p})+Z(\mathbf{p})\hat{\mathbf{3}} \;=\; \mathbf{x}-\mathbf{p}$$

Applying (20) and (21) gives

$$\sqrt{\frac{\bar{z}}{h-\bar{z}}}\Big(X\big((p-a)\hat{\mathbf{i}}+(q-b)\hat{\mathbf{2}}\big)\;+\;Y\big((q-b)\hat{\mathbf{i}}-(p-a)\hat{\mathbf{2}}\big)\Big)\;+\;Z\hat{\mathbf{3}} \;=\; (x-p)\hat{\mathbf{i}}+(y-q)\hat{\mathbf{2}}+(z-\bar{z})\hat{\mathbf{3}}$$

From this we obtain

$$\begin{aligned}x&=\sqrt{\frac{\bar{z}}{h-\bar{z}}}\Big((p-a)X+(q-b)Y\Big)+p\\ y&=\sqrt{\frac{\bar{z}}{h-\bar{z}}}\Big((q-b)X-(p-a)Y\Big)+q\\ z&=Z+\bar{z}\end{aligned} \tag{22}$$

So if we are located at the global position $\mathbf{p}=\mathbf{c}(p;h,a,b,\bar{z})$ on $\mathcal{H}$, and are embedded in the $\bar{z}$-contour path's trajectory, thereby enabling the preparation of an appropriate local vector basis, then any local $\mathbf{X}$ position identifies with the global $\mathbf{x}$ position, and their respective representations are connected by (22). But if $\mathbf{x}$ also happens to correspond to a point on $\mathcal{H}$ then its coordinates must be constrained to satisfy (18):

$$\begin{aligned}\frac{h}{Z+\bar{z}} \;&=\; (x-a)^2+(y-b)^2+1\\ &=\left[\sqrt{\frac{\bar{z}}{h-\bar{z}}}\Big((p-a)X+(q-b)Y\Big)+p-a\right]^2\\ &+\left[\sqrt{\frac{\bar{z}}{h-\bar{z}}}\Big((q-b)X-(p-a)Y\Big)+q-b\right]^2\\ &+\;1\end{aligned}$$

By expanding the terms and exploiting the fact that $(p-a)^2+(q-b)^2+1 = h/\bar{z}$, we obtain

$$Z(X,Y) = \frac{h}{\left(X+\sqrt{(h-\bar{z})/\bar{z}}\right)^2 + Y^2 + 1} \; - \; \bar{z}$$

We conclude that if we are located at $\mathbf{p} = \mathbf{c}(p;h,a,b,\bar{z})$ on the $\bar{z}$-contour path of the embedded hump $\mathcal{H}$ in $\mathbb{R}^3$, as shown in Figure 5, then our local "experience" of the geometry has changed from that of $\mathcal{H}$ to that of $\bar{\mathcal{H}}$:

$$\bar{\mathcal{H}}(h,\bar{z}) = \left\{ (X,Y,Z) \mid Z = \frac{h}{\left(X+\sqrt{(h-\bar{z})/\bar{z}}\right)^2 + Y^2 + 1} \; - \; \bar{z} \right\}$$

Local parametrisation. This is a local counterpart of (18). Under the $\{\hat{\mathbf{n}}(\mathbf{p}), \hat{\mathbf{t}}(\mathbf{p}), \hat{\mathbf{3}}\}$ vector basis, all positions on the global $\bar{z}$-contour path must have $Z = 0$. So while we are located at $\mathbf{p}$, $\bar{\mathcal{H}}$ admits an obvious local parametrisation of the $\bar{z}$-contour path through $\mathbf{p}$ as

$$\mathbf{D}(s;\mathbf{p}) = s\hat{\mathbf{n}}(\mathbf{p}) + \sqrt{(h-\bar{z})/\bar{z} - \left(s+\sqrt{(h-\bar{z})/\bar{z}}\right)^2}\,\hat{\mathbf{t}}(\mathbf{p}) + 0\hat{\mathbf{3}}$$

provided that $(h-\bar{z})/\bar{z} - \left(s+\sqrt{(h-\bar{z})/\bar{z}}\right)^2 \geq 0$, that is, $-2\sqrt{(h-\bar{z})/\bar{z}} \leq s \leq 0$. To parametrise the local path $\mathbf{D}$ with a parameter whose domain is $[0,1]$, and such that $\mathbf{D} = \mathbf{0}$ when the parameter's value is 0, consider the transformation $s = -2\sqrt{(h-\bar{z})/\bar{z}}\; t$, giving

$$\mathbf{D}(t;\mathbf{p}) = 2\sqrt{\frac{h-\bar{z}}{\bar{z}}}\left(-t\hat{\mathbf{n}}(\mathbf{p}) \; + \; \sqrt{t(1-t)}\,\hat{\mathbf{t}}(\mathbf{p}) \; + \; 0\hat{\mathbf{3}}\right), \quad 0 \leq t \leq 1 \tag{23}$$

Locally-centric global reparametrisation. Eq. (23) is a local counterpart of (19). We may use (23) to reparametrise the global perspective of the $\bar{z}$-contour path, but limited to a path segment near $\mathbf{p}$. Substituting (23) into (22):

$$\begin{aligned}
x &= \sqrt{\bar{z}/(h-\bar{z})}\left((p-a)(-2\sqrt{(h-\bar{z})/\bar{z}}\;t) + (q-b)(2\sqrt{(h-\bar{z})/\bar{z}}\,\sqrt{t(1-t)})\right) + p \\
&= (1-2t)(p-a) + 2\sqrt{t(1-t)}(q-b) + a \\
y &= \sqrt{\bar{z}/(h-\bar{z})}\left((q-b)(-2\sqrt{(h-\bar{z})/\bar{z}}\;t) - (p-a)(2\sqrt{(h-\bar{z})/\bar{z}}\,\sqrt{t(1-t)})\right) + q \\
&= (1-2t)(q-b) - 2\sqrt{t(1-t)}(p-a) + b \\
z &= \bar{z}
\end{aligned}$$

A locally-centric reparametrisation of the $\bar{z}$-contour path on $\mathcal{H}$ is therefore

$$\boxed{\begin{aligned}
\mathbf{d}(t;\mathbf{p}) = &\left((1-2t)(\mathbf{p}\cdot\hat{\mathbf{1}} - a) + 2\sqrt{t(1-t)}(\mathbf{p}\cdot\hat{\mathbf{2}} - b) + a\right)\hat{\mathbf{1}} \\
&+ \left((1-2t)(\mathbf{p}\cdot\hat{\mathbf{2}} - b) - 2\sqrt{t(1-t)}(\mathbf{p}\cdot\hat{\mathbf{1}} - a) + b\right)\hat{\mathbf{2}} \\
&+ (\mathbf{p}\cdot\hat{\mathbf{3}})\,\hat{\mathbf{3}}, \quad 0 \leq t \leq 1
\end{aligned}} \tag{24}$$

This parametrisation is guaranteed to be regular over the stipulated domain for t.

Representing the basis vectors as in (16), a matrix representation of the reparametrisation becomes

$$\begin{aligned}
\mathbf{d}(t;\mathbf{p}) &= \begin{bmatrix} 1-2t & 2\sqrt{t(1-t)} & 0 \\ -2\sqrt{t(1-t)} & 1-2t & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} p \\ q \\ \bar{z} \end{bmatrix} + \begin{bmatrix} t2 & -2\sqrt{t(1-t)} & 0 \\ 2\sqrt{t(1-t)} & 2t & 0 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} a \\ b \\ h \end{bmatrix} \\
&= \mathbf{E}(t)\mathbf{p} + \mathbf{F}(t)\mathbf{a}
\end{aligned}$$

where $\mathbf{a}$ is the apex position of $\mathcal{H}$, as shown in Figure 5. Following (17), for an arbitrarily large segment of the contour path passing through the position $\mathbf{p}$ located on the hump $\mathcal{H}$, a set $\mathcal{D}(\mathbf{p}, \Delta t)$ of connected regular parametrisations is suggested by (24) to be:

$$\begin{aligned}\mathcal{D}(\mathbf{p},\Delta t) = \Big\{\, \mathbf{d}(t;\mathbf{d}^i) = &\Big[(1-2t)(\mathbf{d}^i\cdot\hat{\mathbf{1}}-a)+2\sqrt{t(1-t)}(\mathbf{d}^i\cdot\hat{\mathbf{2}}-b)+a\Big]\,\hat{\mathbf{1}}\\ &+\Big[(1-2t)(\mathbf{d}^i\cdot\hat{\mathbf{2}}-b)-2\sqrt{t(1-t)}(\mathbf{d}^i\cdot\hat{\mathbf{1}}-a)+b\Big]\,\hat{\mathbf{2}}\\ &+(\mathbf{p}\cdot\hat{\mathbf{3}})\,\hat{\mathbf{3}}\\ \big|\ \mathbf{d}^i = \mathbf{d}(\Delta t;\mathbf{d}^{i-1});\ \mathbf{d}^0 = \mathbf{p};\ i = 1,2,3,\ldots;\ 0\le t\le 1 \,\Big\}&\end{aligned} \tag{25}$$

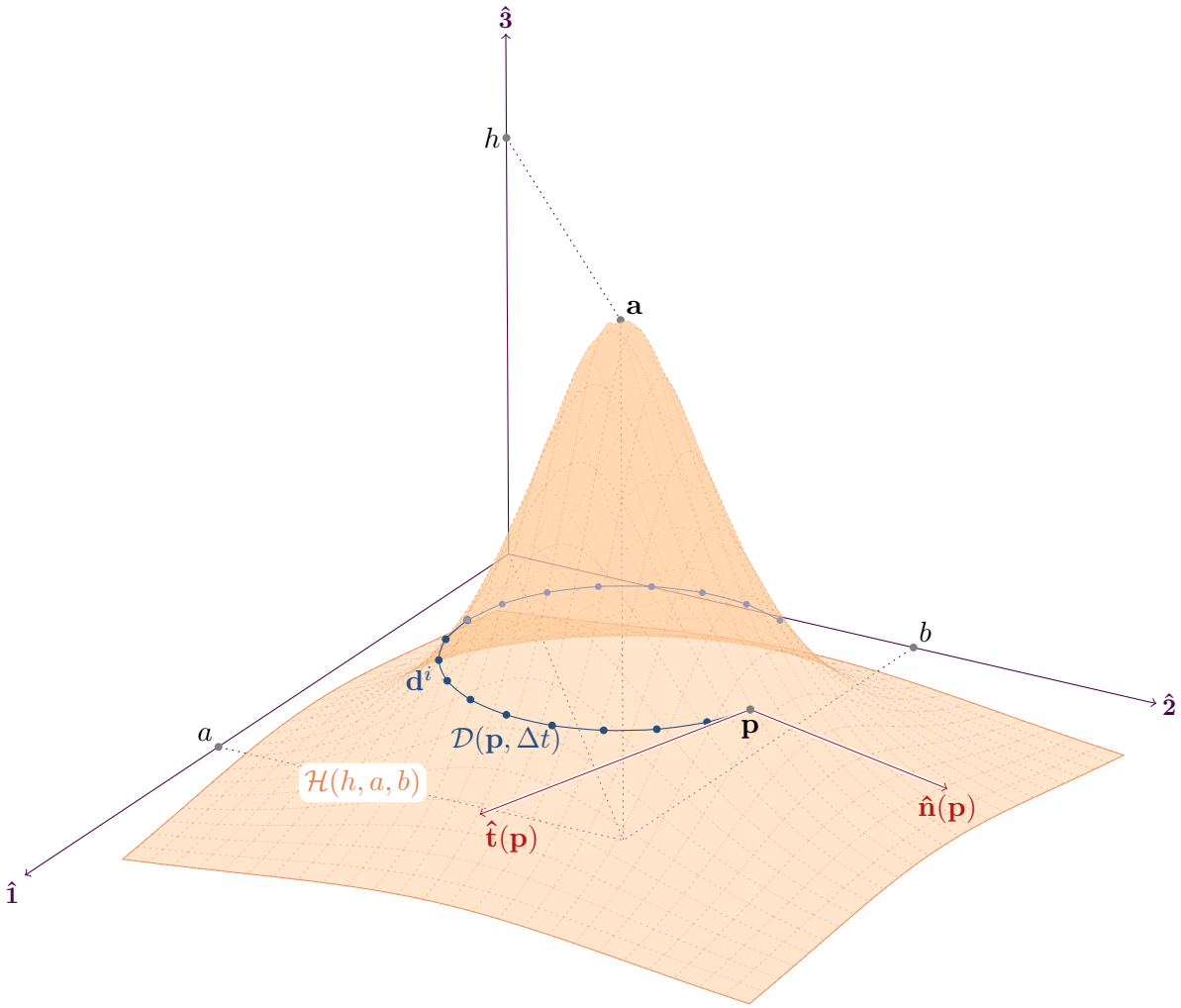


Figure 5: Hump $\mathcal{H}(h,a,b)$ specified by (18), showing a subset of the contour path set $\mathcal{D}(\mathbf{p},\Delta t)$, and the local basis vectors $\hat{\mathbf{t}}(\mathbf{p})$ and $\hat{\mathbf{n}}(\mathbf{p})$. The contour path was calculated using (25), and drawn computationally. (Refer to the annotated listing of the C source file, `humpfigure.c`, in the appendix on page 26.)

5 Acknowledgments

As always Mels, thanks for being such a close friend and supportive partner, and for showing an interest in this work. With you, the perspective on my local z-contour path is beholden.

6 Appendix—Computed drawing with LaTeX, Ti*k*Z, pkTi*k*Z and PKREALVECTOR

In this section I demonstrate the combined use of LaTeX, Ti*k*Z, my pkTi*k*Z LaTeX package[5], and my C object class called PKREALVECTOR[1] to produce the three-dimensional schematic diagrams included in this document.

Perhaps not suprisingly, the text in the document was typeset with LaTeX. The figures were typeset with Ti*k*Z. Ti*k*Z is software capability for typesetting graphical content directly in LaTeX. The specification and calculation of the three-dimensional landscapes in the figures were done in the C programming language with the help of my PKREALVECTOR object class. PKREALVECTOR provides a useful coding abstraction for instantiating and manipulating vectors in $\mathbb{R}^n$. For example, PKREALVECTOR's API[2] includes calls to perform the rotational coordinate transformations needed to render on paper a two-dimensional projection of a three-dimensional landscape.

6.1 Embedded sphere

To typeset a figure, the LaTeX source file for this document `\input{}`'s another external LaTeX source file. In the case of Figure 4, the file was named `spherefigure.tex`. The file contains the Ti*k*Z source code instructions to typeset the figure. The file was generated dynamically as the output of the execution of the `spherefigure.run` program, which in turn was created by compiling the C code located in the `spherefigure.c` file. Actually, by virtue of the presence of the `Makefile` file for the UNIX Make system, as listed below, I simply needed to type `make` to create the final PDF-formatted document file, a copy of which you are currently reading.

6.1.1 LaTeX, Ti*k*Z and pkTi*k*Z

To incorporate Ti*k*Z's capabilities during typesetting, I included the following lines of LaTeX code in the preamble of my LaTeX "`.tex`" file:

```
\usepackage{pktikz}
\usetikzlibrary{calc}
%\usetikzlibrary{positioning}
\usetikzlibrary{intersections}
```

Ti*k*Z was customised "globally" for all figures in the document using the following lines of LaTeX code:

```
1  \newcommand*\ud{\text d}
2  \newcommand*\deriv[2]{\frac{\ud #1}{\ud #2}}
3  \newcommand*\derivB[2]{\ud #1/\ud #2}
4  \newcommand*\parDeriv[2]{\frac{\partial #1}{\partial #2}}
5  \newcommand*\abs[1]{\protect\left|#1\protect\right|}
6
7  \newcommand*\realSet{\mathbb{R}}
8  \newcommand*\Rtwo{\realSet^2}
9  \newcommand*\Rthree{\realSet^3}
10 \newcommand*\CcurveSet{\mathcal{C}}
11 \newcommand*\DcurveSet{\mathcal{D}}
12 \newcommand*\sphereSet{\mathcal{O}}
13 \newcommand*\humpSet{\mathcal{H}}
14
15 \newcommand*\one{\pktikzBasisVector{1}}
```

[2] Application Programming Interface

```
\newcommand*\two{\pktikzBasisVector{2}}
\newcommand*\three{\pktikzBasisVector{3}}

\newcommand*\vecx{\pktikzVector{x}}
\newcommand*\vecc{\pktikzVector{c}}
\newcommand*\vecn{\pktikzVector{n}}
\newcommand*\vecA{\pktikzVector{A}}
\newcommand*\vecB{\pktikzVector{B}}
\newcommand*\vecC{\pktikzVector{C}}
\newcommand*\vecp{\pktikzVector{p}}
\newcommand*\vecX{\pktikzVector{X}}
\newcommand*\vecO{\pktikzVector{0}}
\newcommand*\vect{\pktikzVector{t}}
\newcommand*\vecD{\pktikzVector{D}}
\newcommand*\vecd{\pktikzVector{d}}
\newcommand*\veca{\pktikzVector{a}}
\newcommand*\vecE{\pktikzVector{E}}
\newcommand*\vecF{\pktikzVector{F}}

\newcommand*\barz{\bar z}
\newcommand*\mySqrBarz{r^2-\barz^2}
\newcommand*\myRootBarz{\sqrt{\mySqrBarz}}

\newcommand*\abcSegment{\overline{\vecA\vecB\vecC}}

\newcommand*\that{\pktikzUnitVector{t}}
\newcommand*\nhat{\pktikzUnitVector{n}}
\newcommand*\vecComp[2]{\pktikzVector#1\cdot\pktikzBasisVector{#2}}

\newcommand*\sphereContourPosn[1]{\vecc(#1;r,\barz)}
\newcommand*\humpContourPosn[1]{\vecc(#1;h,a,b,\barz)}

%\newcommand*\myHumpRoot{\sqrt{h/\barz-1}}
\newcommand*\myHumpRootBa{\sqrt{\frac{\barz}{h-\barz}}}
\newcommand*\myHumpRootBb{\sqrt{\barz/(h-\barz)}}
\newcommand*\myHumpRootCa{\sqrt{\frac{h-\barz}{\barz}}}
\newcommand*\myHumpRootCb{\sqrt{(h-\barz)/\barz\,}}
\newcommand*\myHumpRootCaSqr{\frac{h-\barz}{\barz}}
\newcommand*\myHumpRootCbSqr{(h-\barz)/\barz}
\newcommand*\myHumpRootD{\sqrt{(h-\barz)/2\barz\,}}

%-------------------------------------------------------------
% For TikZ begins.
%
%
% For TikZ ends.
%-------------------------------------------------------------
```

The file `spherefigure.tex` contains Ti*k*Z code for Figure 4. It resembles:

```
\begin{PkTikzpicture}[scale=1.0]
   %
   % Some coordinates.
   %
   \setPoint{(0,0)}{origin};
   \setPoint{(-4.30673,-2.86112)}{e1};
   \setPoint{(5.40887,-1.24874)}{e2};
   \setPoint{(-0.0288566,5.46849)}{e3};
   \setPoint{(-2.15336,-1.43056)}{a1};
   \setPoint{(3.60591,-0.832491)}{a2};
```

```
      \setPoint{(-0.0203694,3.86011)}{a3};
      \setPoint{(1.65074,1.1287)}{p};
      .
      .
      .
      %
      % 'z'-contour path 'd'.
      %
      \draw[pktikzsurfacepath] plot[mark=*,mark size=1pt] coordinates {
         (1.65074,1.1287)
         (0.73502,0.985538)
         (-0.243457,0.951415)
         .
         .
         .
         (2.79909,1.55235) };
      .
      .
      .
      %
      % The sphere.
      %
      \draw[pktikzsurfacedrawcolor]
         (0,0) circle[radius=4.2]
         (60:4.2) node[pktikzsurfaceedgecolor,above right]{$\sphereSet(r)$};
      .
      .
      .
\end{PkTikzpicture}
```

The `spherefigure.tex` file was incorporated into the body of the text with an `\input{}` LaTeX command, as follows:

```
\begin{figure}[h!]
   \begin{center}
      \input{spherefigure.tex}
   \end{center}
   \caption{...}
   \label{spherefigure}
\end{figure}
```

6.1.2 PKREALVECTOR C object class

The content of the `spherefigure.c` C source file, which was used to create the Ti*k*Z code in `spherefigure.tex`, has been primed to be typeset using the PKTECHDOC "literate programming" LaTeX package.[6] PKTECHDOC makes it possible to closely juxtapose TeX code and non-TeX code both for typesetting and for compilation outside of TeX. A listing of `spherefigure.c` follows:

```
1  #include <pkfeatures.h>
2
3  #include <stddef.h>
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <unistd.h>
7  #include <stdarg.h>
```

```
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "globals.h"
20
21  const char *LOGFNAME = "/tmp/diagram.log";
22
23  static void _printVector( const PKREALVECTOR *v )
24  {
25     printf( "Vector %s = ( ", pkRealVectorGetName(v) );
26     pkRealVectorPrintf( v, "__COMPONENTVALUE__", ", " );
27     puts(" )");
28     return;
29  }
```

The `_sphere()` private function below simply returns the value $|z(x, y)|$ of the constitutive equation for the sphere $\mathcal{O}$ (Eq. (7)).

```
30  static PKMATHREAL _sphere( const PKMATHREAL x, const PKMATHREAL y )
31  {
32     return( sqrt( sphereRadius*sphereRadius - x*x - y*y ) );
33  }
```

The `_allocSphereContourPosn0()` private function allocates and initialises a position vector on the z-contour path parametrised locally with t, starting at the specified position $p\hat{\mathbf{i}} + q\hat{\mathbf{2}} + z\hat{\mathbf{3}} = \mathbf{c}(p; r, z)$ (Eq. (3)). Obviously, the z-contour will pass through that position. This function implements (15).

On success, return a pointer to the allocated and initialised `PKREALVECTOR` representing the position vector. Otherwise return `(PKREALVECTOR *)NULL`.

The function must be accompanied by a call to `_freeSphereContourPosn0()`.

```
34  static PKREALVECTOR *_allocSphereContourPosn0( const char *name,
35                                                 const PKREALVECTORREAL t,
36                                                 const PKREALVECTORREAL p,
37                                                 const PKREALVECTORREAL q,
38                                                 const PKREALVECTORREAL z )
39  {
40     return( pkRealVectorAlloc1(
41                   name,
42                   3,
43                   ( 1.0 - 2.0 * t ) * p + 2.0 * sqrt( t * ( 1.0 - t ) ) * q,
44                   ( 1.0 - 2.0 * t ) * q - 2.0 * sqrt( t * ( 1.0 - t ) ) * p,
45                   z ) );
46  }
```

The `_freeSphereContourPosn0()` private function is the complement to `_allocSphereContourPosn0()`.

```
47  static void _freeSphereContourPosn0( PKREALVECTOR *d )
48  {
49     if (d)
```

```
50         pkRealVectorFree1(d);
51      return;
52   }
```

If the specified `p` is not `NULL`, then the `_allocSphereContourPosn()` private function simply returns with the result of the call:

```
    _allocSphereContourPosn0( name,
                              t,
                              pkRealVectorGetComponent(p)[0],
                              pkRealVectorGetComponent(p)[1],
                              pkRealVectorGetComponent(p)[2] )
```

Otherwise return `(PKREALVECTOR *)NULL`. The function must be accompanied by a call to `_freeSphereContourPosn()`.

```
53   static PKREALVECTOR *_allocSphereContourPosn( const char *name,
54                                                 const PKREALVECTORREAL t,
55                                                 const PKREALVECTOR *p )
56   {
57      if (!p)
58         return( (PKREALVECTOR *)NULL );
59      return( _allocSphereContourPosn0( name,
60                                        t,
61                                        pkRealVectorGetComponent(p)[0],
62                                        pkRealVectorGetComponent(p)[1],
63                                        pkRealVectorGetComponent(p)[2] ) );
64   }
```

The `_freeSphereContourPosn()` private function is the complement to `_allocSphereContourPosn()`.

```
65   static void _freeSphereContourPosn( PKREALVECTOR *d )
66   {
67      _freeSphereContourPosn0(d);
68      return;
69   }
```

The `_allocSphereContourPosnM()` private function is identical to `_allocSphereContourPosn()` except that `-pkRealVectorGetComponent(p)[1]` is used instead of `+pkRealVectorGetComponent(p)[1]`.

The function must be accompanied by a call to `_freeSphereContourPosnM()`.

```
70   #if 0
71   static PKREALVECTOR *_allocSphereContourPosnM( const char *name,
72                                                  const PKREALVECTORREAL t,
73                                                  const PKREALVECTOR *p )
74   {
75      if (!p)
76         return( (PKREALVECTOR *)NULL );
77      return( _allocSphereContourPosn0( name,
78                                        t,
79                                        pkRealVectorGetComponent(p)[0],
80                                        pkRealVectorGetComponent(p)[1],
81                                        pkRealVectorGetComponent(p)[2] ) );
82   }
```

The `_freeSphereContourPosnM()` private function is the complement to `_allocSphereContourPosnM()`.

```
83  static void _freeSphereContourPosnM( PKREALVECTOR *d )
84  {
85     _freeSphereContourPosn0(d);
86     return;
87  }
88  #endif
```

The `_allocSphereContourPosns()` private function allocates and initialises an array of z-contour positions beginning at the specified $\mathbf{p}$ position. So obviously, the z-contour will pass through $\mathbf{p}$. This function implements a subset of $\mathcal{D}(\mathbf{p}, \Delta t)$ (Eq. (17)).

On success, return a pointer to the allocated and initialised array of `positions` (`PKREALVECTOR *`)s. Otherwise return `(PKREALVECTOR **)NULL`.

The function must be accompanied by a call to `_freeSphereContourPosns()`.

```
89  static PKREALVECTOR **_allocSphereContourPosns( const PKREALVECTOR *p,
90                                                  const int positions,
91                                                  const PKMATHREAL deltat )
92  {
93     PKREALVECTOR **d;
94
95     if ( positions < 3 )
96        return( (PKREALVECTOR **)NULL );
97
98     d = (PKREALVECTOR **)calloc( positions + 1, sizeof(PKREALVECTOR *) );
99     if (d) {
100
101       char *name;
102       int i;
103
104       d[0] = pkRealVectorAlloc1( "\\vecd^0", 3,
105                                  pkRealVectorGetComponent(p)[0],
106                                  pkRealVectorGetComponent(p)[1],
107                                  pkRealVectorGetComponent(p)[2] );
108       for ( i = 1;  i < positions;  i++ ) {
109          name = strAllocPrintf( "\\vecd^{%d}", i );
110          d[i] = _allocSphereContourPosn( name, deltat, d[i-1] );
111          strFreePrintf(name);
112       }
113
114    }
115
116    return(d);
117 }
```

The `_freeSphereContourPosns()` private function is the complement to `_allocSphereContourPosns()`.

```
118 static void _freeSphereContourPosns( PKREALVECTOR **d, const int positions )
119 {
120    if (d) {
121       int i;
122       for ( i = 0;  i < positions;  i++ )
123          _freeSphereContourPosn(d[i]);
124       free(d);
125    }
126    return;
127 }
```

The `_diagram()` private function below specifies the diagram's three-dimensional landscape. It does this primarily using the `PKREALVECTOR` object class. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the landscape in TEX.

But before this function can do so, it must transform the landscape in such a way that what Ti*k*Z typesets is a two-dimensional projection of the three-dimensional landscape. The function rotationally transforms the landscape onto the space spanned by the $\{\hat{\mathbf{1}}', \hat{\mathbf{2}}', \hat{\mathbf{3}}'\}$ orthonormal vector basis set, where the $\hat{\mathbf{1}}'$ and $\hat{\mathbf{2}}'$ basis vectors lie in the plane of the page and $\hat{\mathbf{3}}'$ is perpendicular to the page, i.e., lying parallel to the reader's line of sight.

In the function, the `xLos`, `yLos` and `zLos` are required for the rotational transformations. They are the three coordinates of the "line-of-sight" vector under the $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}, \hat{\mathbf{3}}\}$ vector basis. The angle θ is the tilt angle between $\hat{\mathbf{2}}$ and the $\hat{\mathbf{2}}'\hat{\mathbf{3}}'$ plane. The actual transformation is affected via calls resembling

```
pkRealVectorsUnderLineOfSightBasis1( xLos, yLos, zLos, theta,
                                     e1, e2, e3,
                                     ...,
                                     NULL )
```

For further details, refer to [1] and [7].

```
128  static void _diagram(void)
129  {
130     const int Nd = 21;
131     PKMATHREAL xMin, xMax,
132                u, v;
133     PKREALVECTOR *e1,
134                  *e2,
135                  *e3,
136                  *a1,    /* Apex of the sphere along 1. */
137                  *a2,
138                  *a3,
139                  *p,     /* Global position for a local origin on the sphere. */
140                  *p1,    /* Component vector of 'p' along 1. */
141                  *p2,    /* Component vector of 'p' along 2. */
142                  *p3,    /* Component vector of 'p' along 3. */
143                  *p12,   /* Component vector of 'p' in the 1-2 plane. */
144                  *tHat,  /* Local basis vector.  Actually, position at 'p+that(p)'. */
145                  *nHat,  /* Local basis vector.  Actually, position at 'p+nhat(p)'. */
146                  **d,    /* A dynamic array of positions on the contour path */
147                          /* passing thru 'p'. */
148                  *d3[Nd+1],      /* The 3-component position of 'd[i]'. */
149                  *d12[Nd+1];     /* The 12-component position of 'd[i]'. */
150     int i,
151         j;
152
```

Here we specify the three-dimensional landscape. Begin with the $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}, \hat{\mathbf{3}}\}$ vector basis.

```
153     e1 = pkRealVectorAlloc1( "\\one",   3, basisVecLen, 0.0, 0.0 );
154     e2 = pkRealVectorAlloc1( "\\two",   3, 0.0, basisVecLen, 0.0 );
155     e3 = pkRealVectorAlloc1( "\\three", 3, 0.0, 0.0, basisVecLen );
156     pkRealVectorScale(e1,1.2);
157     pkRealVectorScale(e2,0.9);
158     pkRealVectorScale(e3,0.85);
```

The apex positions $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$ on the respective basis vector lines piercing $\mathcal{O}$.

```
159      a1 = pkRealVectorAlloc1( "\\veca_1", 3, sphereRadius, 0.0, 0.0 );
160      a2 = pkRealVectorAlloc1( "\\veca_2", 3, 0.0, sphereRadius, 0.0 );
161      a3 = pkRealVectorAlloc1( "\\veca_3", 3, 0.0, 0.0, sphereRadius );
```

The local origin $\mathbf{p}$.

```
162      u = 0.4 * sphereRadius;
163      v = 0.7 * sphereRadius;
164      p = pkRealVectorAlloc1( "\\vecp",  3, u, v, _sphere(u,v) );
165      //_printVector(p); exit(0);
166      p1 = pkRealVectorAlloc1( "p",      3, u, 0.0, 0.0 );
167      p2 = pkRealVectorAlloc1( "q",      3, 0.0, v, 0.0 );
168      p3 = pkRealVectorAlloc1( "\\barz", 3, 0.0, 0.0, pkRealVectorGetComponent(p)[2] );
169      p12 = pkRealVectorAlloc1( "p12",   3, u, v, 0.0 );
```

The $\hat{\mathbf{t}}(\mathbf{p})$ and $\hat{\mathbf{n}}(\mathbf{p})$ (Eqs. (9) and (10)) basis vectors.

```
170      tHat = pkRealVectorAlloc1( "\\that(\\vecp)", 3,
171                                 pkRealVectorGetComponent(p)[1],
172                                 -pkRealVectorGetComponent(p)[0],
173                                 0.0 );
174      pkRealVectorScale(
175            tHat,
176            1.0 / sqrt( sphereRadius * sphereRadius -
177                        pkRealVectorGetComponent(p)[2] * pkRealVectorGetComponent(p)[2] ) );
178      pkRealVectorScale( tHat, 0.5 * basisVecLen );
179      pkRealVectorIncrease(tHat,p);
180      nHat = pkRealVectorAlloc1( "\\nhat(\\vecp)", 3,
181                                 pkRealVectorGetComponent(p)[0],
182                                 pkRealVectorGetComponent(p)[1],
183                                 0.0 );
184      pkRealVectorScale(
185            nHat,
186            1.0/ sqrt( sphereRadius * sphereRadius -
187                       pkRealVectorGetComponent(p)[2] * pkRealVectorGetComponent(p)[2] ) );
188      pkRealVectorScale( nHat, 0.5 * basisVecLen );
189      pkRealVectorIncrease(nHat,p);
```

Prepare an array of N_d sample positions, $\mathbf{d}^i$, for $\mathcal{O}$'s z-contour path passing through $\mathbf{p}$. Refer to the $\mathcal{D}(\mathbf{p}, \Delta t)$ set in Eq. (17).

```
190      d = _allocSphereContourPosns( p, Nd, 0.021 );
191      for ( i = 0;  i < Nd;  i++ ) {
192         d3[i] = pkRealVectorAlloc1( "",
193                                     3,
194                                     0.0,
195                                     0.0,
196                                     pkRealVectorGetComponent(d[i])[2] );
197         d12[i] = pkRealVectorAlloc1( "",
198                                      3,
199                                      pkRealVectorGetComponent(d[i])[0],
200                                      pkRealVectorGetComponent(d[i])[1],
201                                      0.0 );
202      }
```

Rotationally transform the landscape onto the space spanned by the "line-of-sight" basis. That is, transform all vectors into "shadow" vectors which lie in the $\hat{\mathbf{i}}'\hat{\mathbf{2}}'$ plane lying flat on the page.

```
203     pkRealVectorsUnderLineOfSightBasisV1( xLos, yLos, zLos, theta,
204                                           d, Nd );
205     pkRealVectorsUnderLineOfSightBasisV1( xLos, yLos, zLos, theta,
206                                           d3, Nd );
207     pkRealVectorsUnderLineOfSightBasisV1( xLos, yLos, zLos, theta,
208                                           d12, Nd );
209     if ( 0 == pkRealVectorsUnderLineOfSightBasis1( xLos, yLos, zLos, theta,
210                                                    e1, e2, e3,
211                                                    a1, a2, a3,
212                                                    p, p1, p2, p3, p12,
213                                                    tHat, nHat,
214                                                    NULL ) ) {
215
```

Prepare the Ti*k*Z commands for typesetting the projection of the three-dimensional landscape of $\mathcal{O}$.

```
216         puts(   "\\begin{PkTikzpicture}[scale=1.0]");
217         puts(   "   %");
218         puts(   "   %\\draw[help lines] (-0.2,-0.2) grid (7.1,5.1);");
219         puts(   "   %");
220         puts(   "   %\\def\\setPoint{\\pktikzSetLabelledPoint}");
221         puts(   "   \\def\\setPoint{\\pktikzSetUncircledPoint}");
222         puts(   "   %");
223         puts(   "   % Some coordinates.");
224         puts(   "   %");
225         printf( "   \\setPoint{(0,0)}{origin};\n"
226                 "   \\setPoint{(%g,%g)}{e1};\n"
227                 "   \\setPoint{(%g,%g)}{e2};\n"
228                 "   \\setPoint{(%g,%g)}{e3};\n"
229                 "   \\setPoint{(%g,%g)}{a1};\n"
230                 "   \\setPoint{(%g,%g)}{a2};\n"
231                 "   \\setPoint{(%g,%g)}{a3};\n"
232                 "   \\setPoint{(%g,%g)}{p};\n",
233                 pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
234                 pkRealVectorGetComponent(e2)[0], pkRealVectorGetComponent(e2)[1],
235                 pkRealVectorGetComponent(e3)[0], pkRealVectorGetComponent(e3)[1],
236                 pkRealVectorGetComponent(a1)[0], pkRealVectorGetComponent(a1)[1],
237                 pkRealVectorGetComponent(a2)[0], pkRealVectorGetComponent(a2)[1],
238                 pkRealVectorGetComponent(a3)[0], pkRealVectorGetComponent(a3)[1],
239                 pkRealVectorGetComponent(p)[0],  pkRealVectorGetComponent(p)[1] );
240
241         puts(   "   %");
242         puts(   "   % Basisvectors.");
243         puts(   "   %");
244         puts(   "   \\draw[pktikzbasisvector,dashed,-] (a1) -- (origin) -- (a2) (origin) -- (a3);
245
246         puts(   "   %");
247         puts(   "   % 'z'-contour path 'd'.");
248         puts(   "   %");
249         //for ( i = 0; i < Nd; i++ ) {
250         //   printf( "   \\draw[pktikzdimension] (origin) -- (%g,%g) -- (%g,%g) -- (%g,%g);\n",
251         //           pkRealVectorGetComponent(d12[i])[0],
252         //           pkRealVectorGetComponent(d12[i])[1],
253         //           pkRealVectorGetComponent(d[i])[0],
254         //           pkRealVectorGetComponent(d[i])[1],
255         //           pkRealVectorGetComponent(d3[i])[0],
256         //           pkRealVectorGetComponent(d3[i])[1] );
257         //}
258         //puts(   "   \\path[pktikzsurfacepath]" );
259         //for ( i = 0; i < Nd; i++ ) {
260         //   printf( "      (%g,%g) coordinate[pktikzpoint,label=below:$%s$]%s\n",
```

```
261         //           pkRealVectorGetComponent(d[i])[0],
262         //           pkRealVectorGetComponent(d[i])[1],
263         //           pkRealVectorGetName(d[i]),
264         //           ( i < Nd - 1 ) ? " --" : ";" );
265         //}
266         puts(   "   \\draw[pktikzsurfacepath] plot[mark=*,mark size=1pt] coordinates {" );
267         for ( i = 0;  i < Nd;  i++ ) {
268            printf( "      (%g,%g)%s\n",
269                    pkRealVectorGetComponent(d[i])[0],
270                    pkRealVectorGetComponent(d[i])[1],
271                    ( i < Nd - 1 ) ? "" : " };" );
272         }
273         //puts(   "   \\path" );
274         //for ( i = 0;  i < Nd;  i++ ) {
275         //   printf( "   (%g,%g) node[below]{$%s$}%s\n",
276         //           pkRealVectorGetComponent(d[i])[0],
277         //           pkRealVectorGetComponent(d[i])[1],
278         //           pkRealVectorGetName(d[i]),
279         //           ( i < Nd - 1 ) ? " --" : ";" );
280         //}
281         printf( "   \\path (%g,%g) node[below,pktikzsurfacepathcolor]{$\\vecd^i$};\n",
282                 pkRealVectorGetComponent(d[9*Nd/16])[0],
283                 pkRealVectorGetComponent(d[9*Nd/16])[1] );
284         printf( "   \\path (%g,%g)\n"
285                 "      node[below left=-2pt,pktikzsurfacepathcolor]\n"
286                 "          {$\\DcurveSet(\\vecp,\\Delta t)$};\n",
287                 pkRealVectorGetComponent(d[Nd/5])[0],
288                 pkRealVectorGetComponent(d[Nd/5])[1] );
289
290         puts(   "   %");
291         puts(   "   % The sphere.");
292         puts(   "   %");
293         printf( "   \\draw[pktikzsurfacedrawcolor]\n"
294                 "      (0,0) circle[radius=%g]\n"
295                 "      (60:%g) node[above right]{$\\sphereSet(r)$};\n",
296                 sphereRadius,
297                 sphereRadius );
298
299         puts(   "   %");
300         puts(   "   % Position 'p'.");
301         puts(   "   %");
302         printf( "   \\draw[pktikzdimension]\n"
303                 "      (origin) --\n"
304                 "      (%g,%g) --\n"
305                 "      (%g,%g) --\n"
306                 "      (%g,%g) node[left]{$%s$}\n"
307                 "      (%g,%g) node[above left=-2pt]{$%s$} --\n"
308                 "      (%g,%g) --\n"
309                 "      (%g,%g) node[above right=-2pt]{$%s$};\n",
310                 pkRealVectorGetComponent(p12)[0],
311                 pkRealVectorGetComponent(p12)[1],
312                 pkRealVectorGetComponent(p)[0],
313                 pkRealVectorGetComponent(p)[1],
314                 pkRealVectorGetComponent(p3)[0],
315                 pkRealVectorGetComponent(p3)[1],
316                 pkRealVectorGetName(p3),
317                 pkRealVectorGetComponent(p1)[0],
318                 pkRealVectorGetComponent(p1)[1],
319                 pkRealVectorGetName(p1),
320                 pkRealVectorGetComponent(p12)[0],
321                 pkRealVectorGetComponent(p12)[1],
322                 pkRealVectorGetComponent(p2)[0],
```

```
                pkRealVectorGetComponent(p2)[1],
                pkRealVectorGetName(p2) );

        puts(   "   %");
        puts(   "   % More on basisvectors.");
        puts(   "   %");
        printf( "   \\draw[pktikzbasisvector,->]\n"
                "      (a1) -- (e1) node[below left]{$%s$};\n",
                pkRealVectorGetName(e1) );
        printf( "   \\draw[pktikzbasisvector,->]\n"
                "      (a2) -- (e2) node[right]{$%s$};\n",
                pkRealVectorGetName(e2) );
        printf( "   \\draw[pktikzbasisvector,->]\n"
                "      (a3) -- (e3) node[above]{$%s$};\n",
                pkRealVectorGetName(e3) );

        puts(   "   %");
        puts(   "   % Local basis vectors at 'p'.");
        puts(   "   %");
        printf( "   \\draw[pktikzbasisvectorp,<->]\n"
                "      (%g,%g) node[below]{$%s$} --\n"
                "      (p) --\n"
                "      (%g,%g) node[below right=-3pt]{$%s$};\n",
                pkRealVectorGetComponent(tHat)[0],
                pkRealVectorGetComponent(tHat)[1],
                pkRealVectorGetName(tHat),
                pkRealVectorGetComponent(nHat)[0],
                pkRealVectorGetComponent(nHat)[1],
                pkRealVectorGetName(nHat) );

        puts(   "   %");
        puts(   "   % More on position 'p'.");
        puts(   "   %");
        printf( "   \\path (%g,%g) coordinate[pktikzpoint] node[above right]{$%s$};\n",
                pkRealVectorGetComponent(p)[0],
                pkRealVectorGetComponent(p)[1],
                pkRealVectorGetName(p) );
        puts(   "\\end{PkTikzpicture}");

    } else {

        puts("ERROR: 'pkRealVectorsUnderLineOfSightBasis1()' failed.");

    }
```

Finally, clean up.

```
    pkRealVectorFree1(e1);
    pkRealVectorFree1(e2);
    pkRealVectorFree1(e3);
    pkRealVectorFree1(a1);
    pkRealVectorFree1(a2);
    pkRealVectorFree1(a3);
    pkRealVectorFree1(p);
    pkRealVectorFree1(p1);
    pkRealVectorFree1(p2);
    pkRealVectorFree1(p3);
    pkRealVectorFree1(p12);
    pkRealVectorFree1(tHat);
    pkRealVectorFree1(nHat);
    _freeSphereContourPosns(d,Nd);
```

```
381      for ( i = 0;  i < Nd;  i++ ) {
382         pkRealVectorFree1(d3[i]);
383         pkRealVectorFree1(d12[i]);
384      }
385
386      return;
387   }
388
389   /*-------------------------------------------------------------*/
390
391   int main( const int argc, const char *argv[] )
392   {
393      _diagram();
394      exit(0);
395   }
```

6.2 Embedded hump

The content of the `humpfigured.c` C source file was used to create the TikZ code in `humpfigure.tex` for typesetting Figure 5. A listing of `humpfigure.c` follows:

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "globals.h"
20
21  const char *LOGFNAME = "/tmp/diagram.log";
22
23  static void _printVector( const PKREALVECTOR *v )
24  {
25     printf( "Vector %s = ( ", pkRealVectorGetName(v) );
26     pkRealVectorPrintf( v, "__COMPONENTVALUE__", ", " );
27     puts(" )");
28     return;
29  }


30  static PKREALVECTORREAL _min( const PKREALVECTORREAL a, const PKREALVECTORREAL b )
31  {
32     return( ( a < b ) ? a : b );
33  }
```

The `_hump()` private function below simply returns the value $z(x, y)$ of the constitutive equation for the hump $\mathcal{H}$ (Eq. (18)).

```
34  static PKMATHREAL _hump( const PKMATHREAL x, const PKMATHREAL y )
35  {
36     return( humpHeight / ( ( x - humpX ) * ( x - humpX ) +
37                            ( y - humpY ) * ( y - humpY ) +
38                            1 ) );
39  }
```

The `_allocHumpPosnArr()` private function allocates and initialises a two-dimensional array of position vectors corresponding to points on $\mathcal{H}$. The function composes `pkRealVectorAlloc1()` and `_hump()`, amongst others. On success, return a `(PKREALVECTOR ***)` pointer to the allocated and initialised array of position vectors. Otherwise return `(PKREALVECTOR ***)NULL`. The function must be accompanied by a call to `_freeHumpPosnArr()`.

The subset $[x_{\min}, x_{\max}][y_{\min}, y_{\max}]$ of the $\hat{\mathbf{1}}\hat{\mathbf{2}}$-plane is required in this function. To compute the subset, set $r^2 = (x - a)^2 + (y - b)^2$, and choose r such that $z(r) = \alpha z(0) = \alpha h$ for some specified

α. This gives

$$\begin{aligned} x_{\min} &= a - \sqrt{(1-\alpha)/\alpha}, \quad x_{\max} = a + \sqrt{(1-\alpha)/\alpha} \\ y_{\min} &= b - \sqrt{(1-\alpha)/\alpha}, \quad y_{\max} = b + \sqrt{(1-\alpha)/\alpha} \end{aligned} \tag{26}$$

```
40  static PKREALVECTOR ***_allocHumpPosnArr( const int xPosns,
41                                            const int yPosns,
42                                            const PKMATHREAL alpha )
43  {
44     PKREALVECTOR ***posn;
45
46     posn = (PKREALVECTOR ***)calloc( xPosns + 1, sizeof(PKREALVECTOR **) );
47     if (posn) {
48        const PKMATHREAL xMin = humpX - sqrt( ( 1 - alpha ) / alpha ),
49                         xMax = humpX + sqrt( ( 1 - alpha ) / alpha ),
50                         yMin = humpY - sqrt( ( 1 - alpha ) / alpha ),
51                         yMax = humpY + sqrt( ( 1 - alpha ) / alpha );
52        PKMATHREAL p,
53                   q;
54        char *name;
55        int i,
56            j;
57        for ( i = 0; i < xPosns; i++ ) {
58           posn[i] = (PKREALVECTOR **)calloc( yPosns + 1, sizeof(PKREALVECTOR *) );
59           p = xMin + (double)i / (double)(xPosns-1) * ( xMax - xMin );
60           for ( j = 0; j < yPosns; j++ ) {
61              name = strAllocPrintf( "\\pktikzVector{r}_{%d%d}", i, j );
62              q = yMin + (double)j / (double)(yPosns-1) * ( yMax - yMin );
63              posn[i][j] = pkRealVectorAlloc1( name, 3, p, q, _hump(p,q) );
64              strFreePrintf(name);
65           }
66        }
67     }
68
69     return(posn);
70  }
```

The `_freeHumpPosnArr()` private function is the complement to `_allocHumpPosnArr()`.

```
71  static void _freeHumpPosnArr( PKREALVECTOR ***posn,
72                                const int xPosns,
73                                const int yPosns )
74  {
75     if (posn) {
76        int i,
77            j;
78        for ( i = 0;  i < xPosns;  i++ ) {
79           for ( j = 0;  j < yPosns;  j++ ) {
80              pkRealVectorFree1( posn[i][j] );
81              posn[i][j] = (PKREALVECTOR *)NULL;
82           }
83           free( posn[i] );
84           posn[i] = (PKREALVECTOR **)NULL;
85        }
86        free(posn);
87     }
88
89     return;
90  }
```

The `_allocHumpContourPosn0()` private function below allocates and initialises a position vector on the z-contour path parametrised locally with t, starting at the specified position $p\hat{\mathbf{1}} + q\hat{\mathbf{2}} + z\hat{\mathbf{3}} = \mathbf{c}(p; h, a, b, z)$ (Eq. (19)). Obviously, the z-contour will pass through that position. This function implements (24). On success, return a pointer to the allocated and initialised `PKREALVECTOR` representing the position vector. Otherwise return `(PKREALVECTOR *)NULL`. The function must be accompanied by a call to `_freeHumpContourPosn0()`.

```
91  static PKREALVECTOR *_allocHumpContourPosn0( const char *name,
92                                               const PKREALVECTORREAL t,
93                                               const PKREALVECTORREAL p,
94                                               const PKREALVECTORREAL q,
95                                               const PKREALVECTORREAL z )
96  {
97     return( pkRealVectorAlloc1( name, 3,
98                                 ( 1.0 - 2.0 * t ) * ( p - humpX ) +
99                                 2.0 * sqrt( t * ( 1.0 - t ) ) * ( q - humpY ) +
100                                humpX,
101                                ( 1.0 - 2.0 * t ) * ( q - humpY ) -
102                                2.0 * sqrt( t * ( 1.0 - t ) ) * ( p - humpX ) +
103                                humpY,
104                                z ) );
105 }
```

The `_freeHumpContourPosn0()` private function is the complement to `_allocHumpContourPosn0()`.

```
106 static void _freeHumpContourPosn0( PKREALVECTOR *d )
107 {
108    if (d)
109       pkRealVectorFree1(d);
110    return;
111 }
```

If the specified `p` is not `NULL`, then the `_allocHumpContourPosn()` private function below simply returns with the result of the call:

```
_allocHumpContourPosn0( name,
                        t,
                        pkRealVectorGetComponent(p)[0],
                        pkRealVectorGetComponent(p)[1],
                        pkRealVectorGetComponent(p)[2] )
```

Otherwise the function returns `(PKREALVECTOR *)NULL`. The function must be accompanied by a call to `_freeHumpContourPosn()`.

```
112 static PKREALVECTOR *_allocHumpContourPosn( const char *name,
113                                             const PKREALVECTORREAL t,
114                                             const PKREALVECTOR *p )
115 {
116    if (!p)
117       return( (PKREALVECTOR *)NULL );
118    return( _allocHumpContourPosn0( name,
119                                    t,
120                                    pkRealVectorGetComponent(p)[0],
121                                    pkRealVectorGetComponent(p)[1],
122                                    pkRealVectorGetComponent(p)[2] ) );
123 }
```

The `_freeHumpContourPosn()` private function is the complement to `_allocHumpContourPosn()`.

```
124  static void _freeHumpContourPosn( PKREALVECTOR *d )
125  {
126     _freeHumpContourPosn0(d);
127     return;
128  }
```

The `_allocHumpContourPosnArr()` private function allocates and initialises an array of z-contour positions beginning at the specified $\mathbf{p}$ position. So obviously, the z-contour will pass through $\mathbf{p}$. This function implements a subset of $\mathcal{D}(\mathbf{p}, \Delta t)$ (Eq. (25)). On success, return a pointer to the allocated and initialised array of `positions` (`PKREALVECTOR *`)s. Otherwise return `(PKREALVECTOR **)NULL`. The function must be accompanied by a call to `_freeHumpContourPosnArr()`.

```
129  static PKREALVECTOR **_allocHumpContourPosnArr( const PKREALVECTOR *p,
130                                                  const int positions,
131                                                  const PKMATHREAL deltat )
132  {
133     PKREALVECTOR **d;
134
135     if ( positions < 3 )
136        return( (PKREALVECTOR **)NULL );
137
138     d = (PKREALVECTOR **)calloc( positions + 1, sizeof(PKREALVECTOR *) );
139     if (d) {
140
141        char *name;
142        int i;
143
144        d[0] = pkRealVectorAlloc1( "\\vecd^0", 3,
145                                   pkRealVectorGetComponent(p)[0],
146                                   pkRealVectorGetComponent(p)[1],
147                                   pkRealVectorGetComponent(p)[2] );
148        for ( i = 1;  i < positions;  i++ ) {
149           name = strAllocPrintf( "\\vecd^{%d}", i );
150           d[i] = _allocHumpContourPosn( name, deltat, d[i-1] );
151           strFreePrintf(name);
152        }
153
154     }
155
156     return(d);
157  }
```

The `_freeHumpContourPosnArr()` private function is the complement to `_allocHumpContourPosnArr()`.

```
158  static void _freeHumpContourPosnArr( PKREALVECTOR **d, const int positions )
159  {
160     if (d) {
161        int i;
162        for ( i = 0;  i < positions;  i++ )
163           _freeHumpContourPosn(d[i]);
164        free(d);
165     }
166     return;
167  }
```

The `_drawCoordinates()` private function below simply `printf`s a few TikZ commands for coordinates.

```
168  static void _drawCoordinates(void)
169  {
170     puts( "   %\\draw[help lines] (-0.2,-0.2) grid (7.1,5.1);");
171     puts( "   %");
172     puts( "   % Some coordinates.");
173     puts( "   %");
174     puts( "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
175     return;
176  }
```

The `_drawBasisVectors()` private function `printf`s Ti*k*Z commands for typesetting the specified global vector basis $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}, \hat{\mathbf{3}}\}$.

```
177  static void _drawBasisVectors( const PKREALVECTOR *e1,
178                                 const PKREALVECTOR *e2,
179                                 const PKREALVECTOR *e3 )
180  {
181     if ( e1  &&  e2  &&  e3 ) {
182        puts(   "   %");
183        puts(   "   % Basisvectors.");
184        puts(   "   %");
185        printf( "   \\draw[pktikzbasisvector,<->]\n"
186                "      (" FLTFMT "," FLTFMT ") node[below left] {$%s$} --\n"
187                "      (origin) -- (" FLTFMT "," FLTFMT ") node[right] {$%s$};\n",
188                pkRealVectorGetComponent(e1)[0],
189                pkRealVectorGetComponent(e1)[1],
190                pkRealVectorGetName(e1),
191                pkRealVectorGetComponent(e2)[0],
192                pkRealVectorGetComponent(e2)[1],
193                pkRealVectorGetName(e2) );
194        printf( "   \\draw[pktikzbasisvector]\n"
195                "      (origin) -- (" FLTFMT "," FLTFMT ") node[above] {$%s$};\n",
196                pkRealVectorGetComponent(e3)[0],
197                pkRealVectorGetComponent(e3)[1],
198                pkRealVectorGetName(e3) );
199     }
200
201     return;
202  }
```

The `_drawApex()` private function `printf`s Ti*k*Z commands for typesetting various coordinates associated with $\mathcal{H}$'s apex position $a\hat{\mathbf{1}} + b\hat{\mathbf{2}} + z(a,b)\hat{\mathbf{3}} = a\hat{\mathbf{1}} + b\hat{\mathbf{2}} + h\hat{\mathbf{3}}$.

```
203  static void _drawApex( const PKREALVECTOR *a,
204                         const PKREALVECTOR *a1,
205                         const PKREALVECTOR *a2,
206                         const PKREALVECTOR *a3,
207                         const PKREALVECTOR *a12 )
208  {
209     if ( a  &&  a1  &&  a2  &&  a3  &&  a12 ) {
210        puts(   "   %");
211        puts(   "   % Position 'apex'.");
212        puts(   "   %");
213        printf( "   \\draw[pktikzdimension]\n"
214                "      (origin) --\n"
215                "      (" FLTFMT "," FLTFMT ") --\n"
216                "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[above right]{$%s$} --
217                "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[left]{$%s$}\n"
218                "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[above left]{$%s$} --\
219                "      (" FLTFMT "," FLTFMT ") --\n"
```

```
220                "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[above right]{$%s$};\r
221                pkRealVectorGetComponent(a12)[0],
222                pkRealVectorGetComponent(a12)[1],
223                pkRealVectorGetComponent(a)[0],
224                pkRealVectorGetComponent(a)[1],
225                pkRealVectorGetName(a),
226                pkRealVectorGetComponent(a3)[0],
227                pkRealVectorGetComponent(a3)[1],
228                pkRealVectorGetName(a3),
229                pkRealVectorGetComponent(a1)[0],
230                pkRealVectorGetComponent(a1)[1],
231                pkRealVectorGetName(a1),
232                pkRealVectorGetComponent(a12)[0],
233                pkRealVectorGetComponent(a12)[1],
234                pkRealVectorGetComponent(a2)[0],
235                pkRealVectorGetComponent(a2)[1],
236                pkRealVectorGetName(a2) );
237     }
238
239     return;
240  }
```

```
241  static void _printfPosition( const char *prefix,
242                               const PKREALVECTOR *posn,
243                               const char *suffix )
244  {
245     if (posn)
246        printf( "%s(" FLTFMT "," FLTFMT ")%s\n",
247                strIsNull(prefix) ? "" : prefix,
248                pkRealVectorGetComponent(posn)[0],
249                pkRealVectorGetComponent(posn)[1],
250                strIsNull(suffix) ? "" : suffix );
251     return;
252  }
```

The `_drawFacet()` private function `printf`s Ti*k*Z commands for typesetting a quadrilateral surface (or "facet") specified by the four `(PKREALVECTOR *)` position vectors.

```
253  static void _drawFacet( const PKREALVECTOR *posn1,
254                          const PKREALVECTOR *posn2,
255                          const PKREALVECTOR *posn3,
256                          const PKREALVECTOR *posn4,
257                          const char *tikzStyle )
258  {
259     if ( posn1  &&  posn2  &&  posn3  &&  posn4 ) {
260        printf( "   \\path[%s]\n", strIsNull(tikzStyle) ? "draw" : tikzStyle );
261        printf( "      (" FLTFMT "," FLTFMT ") -- "
262                      "(" FLTFMT "," FLTFMT ") -- "
263                      "(" FLTFMT "," FLTFMT ") -- "
264                      "(" FLTFMT "," FLTFMT ") -- cycle;\n",
265                pkRealVectorGetComponent(posn1)[0], pkRealVectorGetComponent(posn1)[1],
266                pkRealVectorGetComponent(posn2)[0], pkRealVectorGetComponent(posn2)[1],
267                pkRealVectorGetComponent(posn3)[0], pkRealVectorGetComponent(posn3)[1],
268                pkRealVectorGetComponent(posn4)[0], pkRealVectorGetComponent(posn4)[1] );
269     }
270
271     return;
272  }
```

The `_drawSurfaceElement()` private function `printf`s Ti*k*Z commands for typesetting a surface

specified by the four `corner`n `(PKREALVECTOR *)` corner positions. The remaining specified `mid`n position vectors are assumed to lie on the desired path between two corner positions.

```
273  static void _drawSurfaceElement ( const PKREALVECTOR *corner1,
274                                    const PKREALVECTOR *mid1,
275                                    const PKREALVECTOR *corner2,
276                                    const PKREALVECTOR *mid2,
277                                    const PKREALVECTOR *corner3,
278                                    const PKREALVECTOR *mid3,
279                                    const PKREALVECTOR *corner4,
280                                    const PKREALVECTOR *mid4,
281                                    const char *tikzStyle )
282  {
283     if ( corner1  &&  corner2  &&  corner3  &&  corner4  &&
284          mid1     &&  mid2     &&  mid3     &&  mid4 ) {
285        printf( "   \\path[%s]\n", strIsNull(tikzStyle) ? "draw" : tikzStyle );
286        printf( "      plot[smooth] coordinates { (" FLTFMT "," FLTFMT ")"
287                                                  "(" FLTFMT "," FLTFMT ")"
288                                                  "(" FLTFMT "," FLTFMT ") } --\n",
289                pkRealVectorGetComponent(corner1)[0],
290                pkRealVectorGetComponent(corner1)[1],
291                pkRealVectorGetComponent(mid1)[0],
292                pkRealVectorGetComponent(mid1)[1],
293                pkRealVectorGetComponent(corner2)[0],
294                pkRealVectorGetComponent(corner2)[1] );
295        printf( "      plot[smooth] coordinates { (" FLTFMT "," FLTFMT ")"
296                                                  "(" FLTFMT "," FLTFMT ")"
297                                                  "(" FLTFMT "," FLTFMT ") } --\n",
298                pkRealVectorGetComponent(corner2)[0],
299                pkRealVectorGetComponent(corner2)[1],
300                pkRealVectorGetComponent(mid2)[0],
301                pkRealVectorGetComponent(mid2)[1],
302                pkRealVectorGetComponent(corner3)[0],
303                pkRealVectorGetComponent(corner3)[1] );
304        printf( "      plot[smooth] coordinates { (" FLTFMT "," FLTFMT ")"
305                                                  "(" FLTFMT "," FLTFMT ")"
306                                                  "(" FLTFMT "," FLTFMT ") } --\n",
307                pkRealVectorGetComponent(corner3)[0],
308                pkRealVectorGetComponent(corner3)[1],
309                pkRealVectorGetComponent(mid3)[0],
310                pkRealVectorGetComponent(mid3)[1],
311                pkRealVectorGetComponent(corner4)[0],
312                pkRealVectorGetComponent(corner4)[1] );
313        printf( "      plot[smooth] coordinates { (" FLTFMT "," FLTFMT ")"
314                                                  "(" FLTFMT "," FLTFMT ")"
315                                                  "(" FLTFMT "," FLTFMT ") } -- cycle;\n",
316                pkRealVectorGetComponent(corner4)[0],
317                pkRealVectorGetComponent(corner4)[1],
318                pkRealVectorGetComponent(mid4)[0],
319                pkRealVectorGetComponent(mid4)[1],
320                pkRealVectorGetComponent(corner1)[0],
321                pkRealVectorGetComponent(corner1)[1] );
322     }
323
324     return;
325  }
```

The `_drawHump()` private function `printf`s TikZ commands for typesetting the hump surface $\mathcal{H}$.

```
326  static void _drawHump( PKREALVECTOR ***posn,
327                         const int Nx,
```

```
328                         const int Ny )
329  {
330     int i,
331         j;
332
333     if ( !posn )
334        return;
335
336     puts(   "   %");
337     puts(   "   % The hump's cut-off edge.");
338     puts(   "   %");
339     puts( "   \\draw[draw=pktikzsurfacedrawcolor]" );
340     for ( j = 0; j < Ny; j++ )
341        _printfPosition( "      ", posn[0][j], " --" );
342     for ( i = 0; i < Nx; i++ )
343        _printfPosition( "      ", posn[i][Ny-1], " --" );
344     for ( j = Ny-1; j >= 0; j-- )
345        _printfPosition( "      ", posn[Nx-1][j], " --" );
346     for ( i = Nx-1; i >= 0; i-- )
347        _printfPosition( "      ", posn[i][0], ( i > 0 ) ? " --" : ";" );
348
349     //puts(   "   %");
350     //puts(   "   % The hump.");
351     //puts(   "   %");
352     //for ( i = 1;  i < Nx - 1;  i++ ) {
353     //   puts( "   \\draw[surface]" );
354     //   for ( j = 0;  j < Ny;  j++ )
355     //      _printfPosition( "      ", posn[i][j], ( j < Ny - 1 ) ? "--" : ";" );
356     //}
357     //puts( "   %");
358     //for ( j = 1;  j < Ny - 1;  j++ ) {
359     //   puts( "   \\draw[surface]" );
360     //   for ( i = 0;  i < Nx;  i++ )
361     //      _printfPosition( "      ", posn[i][j], ( i < Nx - 1 ) ? "--" : ";" );
362     //}
363
364     //puts(   "   %");
365     //puts(   "   % The hump.");
366     //puts(   "   %");
367     //for ( i = 1;  i < Nx - 1;  i++ ) {
368     //   //puts( "   \\draw[surface] plot[smooth,mark=*,mark size=1pt] coordinates {" );
369     //   puts( "   \\draw[surface] plot[smooth] coordinates {" );
370     //   for ( j = 0;  j < Ny;  j++ )
371     //      _printfPosition( "      ", posn[i][j], ( j < Ny - 1 ) ? "" : "};" );
372     //}
373     //puts( "   %");
374     //for ( j = 1;  j < Ny - 1;  j++ ) {
375     //   puts( "   \\draw[surface] plot[smooth] coordinates {" );
376     //   for ( i = 0;  i < Nx;  i++ )
377     //      _printfPosition( "      ", posn[i][j], ( i < Nx - 1 ) ? "" : "};" );
378     //}
379
380     //puts(   "   %");
381     //puts(   "   % The hump.");
382     //puts(   "   %");
383     //for ( i = 0;  i < Nx - 1;  i++ ) {
384     //   for ( j = 0;  j < Ny - 1;  j++ )
385     //      _drawFacet( posn[i][j], posn[i][j+1], posn[i+1][j+1], posn[i+1][j],
386     //                  "surface" );
387     //}
388
389     puts(   "   %");
```

```
390      puts(   "   % The hump.");
391      puts(   "   %");
392      for ( i = 0;  i < Nx - 2;  i += 2 ) {
393         for ( j = 0;  j < Ny - 2;  j += 2 ) {
394            _drawSurfaceElement( posn[i][j],     posn[i][j+1],
395                                 posn[i][j+2],   posn[i+1][j+2],
396                                 posn[i+2][j+2], posn[i+2][j+1],
397                                 posn[i+2][j],   posn[i+1][j],
398                                 "hump" );
399         }
400      }
401      _printfPosition( "   \\path ",
402                       posn[3*Nx/4][Ny/4],
403                       " node[pktikzsurfacedrawcolor,below,fill=white,rounded corners]"
404                            "{$\\humpSet(h,a,b)$};" );
405
406      return;
407   }
```

The `_drawContourPath()` private function `printf`s Ti*k*Z commands for typesetting a subset of the $\mathcal{D}(\mathbf{p}, \Delta t)$ set in Eq. (25). The specified array `d` of `Nd` entries are assumed to be the required $\mathbf{d}^i$ position vectors.

```
408   static void _drawContourPath( PKREALVECTOR **d, const int Nd )
409   {
410      const int Md = _min(Nd,11);
411      int i;
412
413      if (!d)
414         return;
415
416      puts(   "   %");
417      puts(   "   % 'z'-contour path 'd'.");
418      puts(   "   %");
419      puts(   "   \\draw[pktikzsurfacepath] plot[mark=*,mark size=0.7pt] coordinates {" );
420      for ( i = 0;  i < Md;  i++ )
421         _printfPosition( "      ", d[i], ( i < Md - 1 ) ? "" : " };" );
422      if ( Md < Nd ) {
423         puts(   "   \\draw[occludedpath] plot[mark=*,mark size=0.6pt] coordinates {" );
424         for ( i = Md - 1;  i < Nd;  i++ )
425            _printfPosition( "      ", d[i], ( i < Nd - 1 ) ? "" : " };" );
426      }
427
428      printf( "   \\path (" FLTFMT "," FLTFMT ") node[below left,pktikzsurfacepathcolor]{$\\vecd^i
429              pkRealVectorGetComponent(d[11*Nd/24])[0],
430              pkRealVectorGetComponent(d[11*Nd/24])[1] );
431      printf( "   \\path (" FLTFMT "," FLTFMT ")\n"
432              "      node[below=2pt,pktikzsurfacepathcolor]{$\\DcurveSet(\\vecp,\\Delta t)$};\n",
433              pkRealVectorGetComponent(d[7*Nd/24])[0],
434              pkRealVectorGetComponent(d[7*Nd/24])[1] );
435
436      return;
437   }
```

The `_drawContourPathStart()` private function `printf`s Ti*k*Z commands for typesetting the specified position $\mathbf{p}$ (Eq. (4)). This is the "local origin" position.

```
438   static void _drawContourPathStart( PKREALVECTOR *p )
439   {
440      if (!p)
```

```
441         return;
442
443      puts(   "   %");
444      puts(   "   % Contour start position.");
445      puts(   "   %");
446      printf( "   \\path (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[below=2pt]{$%s$};\n"
447              pkRealVectorGetComponent(p)[0],
448              pkRealVectorGetComponent(p)[1],
449              pkRealVectorGetName(p) );
450
451      return;
452   }
```

The `_drawLocalBasisVectors()` private function `printf`s Ti*k*Z commands for typesetting the specified local basis vectors $\hat{\mathbf{t}}$ and $\hat{\mathbf{n}}$ at the specified position $\mathbf{p}$ (Eq. (4)).

```
453   static void _drawLocalBasisVectors( const PKREALVECTOR *tHat,
454                                       const PKREALVECTOR *nHat,
455                                       const PKREALVECTOR *p )
456   {
457      if ( tHat  &&  nHat  &&  p ) {
458         puts(   "   %");
459         puts(   "   % Local basis vectors at 'p'.");
460         puts(   "   %");
461         //printf( "   \\draw[pktikzbasisvectorp,<->]\n"
462         //        "      (" FLTFMT "," FLTFMT ") node[below]{$%s$} --\n"
463         //        "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[below right]{$%s$}
464         //        "      (" FLTFMT "," FLTFMT ") node[below right]{$%s$};\n",
465         //        pkRealVectorGetComponent(tHat)[0],
466         //        pkRealVectorGetComponent(tHat)[1],
467         //        pkRealVectorGetName(tHat),
468         //        pkRealVectorGetComponent(p)[0],
469         //        pkRealVectorGetComponent(p)[1],
470         //        pkRealVectorGetName(p),
471         //        pkRealVectorGetComponent(nHat)[0],
472         //        pkRealVectorGetComponent(nHat)[1],
473         //        pkRealVectorGetName(nHat) );
474         printf( "   \\draw[pktikzbasisvectorp,<->]\n"
475                 //"      (" FLTFMT "," FLTFMT ") node[below right]{$%s$} --\n"
476                 "      (" FLTFMT "," FLTFMT ") node[below right]{$%s$} --\n"
477                 "      (" FLTFMT "," FLTFMT ") --\n"
478                 "      (" FLTFMT "," FLTFMT ") node[below]{$%s$};\n",
479                 pkRealVectorGetComponent(tHat)[0],
480                 pkRealVectorGetComponent(tHat)[1],
481                 pkRealVectorGetName(tHat),
482                 pkRealVectorGetComponent(p)[0],
483                 pkRealVectorGetComponent(p)[1],
484                 pkRealVectorGetComponent(nHat)[0],
485                 pkRealVectorGetComponent(nHat)[1],
486                 pkRealVectorGetName(nHat) );
487      }
488
489      return;
490   }
```

The `_diagram()` private function below specifies the diagram's three-dimensional landscape. It does this primarily using the `PKREALVECTOR` object class. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the landscape in TEX.

But before this function can do so, it must transform the landscape in such a way that what Ti*k*Z typesets is a two-dimensional projection of the three-dimensional landscape. The function

rotationally transforms the landscape onto the space spanned by the $\{\hat{\mathbf{1}}', \hat{\mathbf{2}}', \hat{\mathbf{3}}'\}$ orthonormal vector basis set, where the $\hat{\mathbf{1}}'$ and $\hat{\mathbf{2}}'$ basis vectors lie in the plane of the page and $\hat{\mathbf{3}}'$ is perpendicular to the page, i.e., lying parallel to the reader's line of sight.

In the function, the `xLos`, `yLos` and `zLos` are required for the rotational transformations. They are the three coordinates of the "line-of-sight" vector under the $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}, \hat{\mathbf{3}}\}$ vector basis. The angle θ is the tilt angle between $\hat{\mathbf{2}}$ and the $\hat{\mathbf{2}}'\hat{\mathbf{3}}'$ plane. The actual transformation is affected via calls resembling

```
pkRealVectorsUnderLineOfSightBasis1( xLos, yLos, zLos, theta,
                                     e1, e2, e3,
                                     ...,
                                     NULL )
```

For further details, refer to [1] and [7].

```
491  static void _diagram(void)
492  {
493     const int Nx = /*11*/ /*31*/ 51,
494               Ny = Nx,
495               Nd = 18;
496     PKMATHREAL u, v;
497     PKREALVECTOR *e1,
498                  *e2,
499                  *e3,
500                  *apex,          /* Apex of the hump. */
501                  *apex1,         /* Apex of the hump along 1. */
502                  *apex2,
503                  *apex3,
504                  *apex12,        /* Apex of the hump in the '1-2'-plane. */
505                  *p,     /* Global position for a local origin on the hump. */
506                  *tHat,  /* Local basis vector.  Actually, position at 'p+that(p)'. */
507                  *nHat,  /* Local basis vector.  Actually, position at 'p+nhat(p)'. */
508                  ***r,   /* Dynamic array of positions on the hump. */
509                  **d;    /* Dynamic array of positions on the contour path */
510                          /* passing thru 'p'. */
511     int i,
512         j;
513
```

Prepare the objects in the landscape.

Here we specify the three-dimensional landscape. Begin with the $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}, \hat{\mathbf{3}}\}$ vector basis.

```
514     e1 = pkRealVectorAlloc1( "\\one",   3, basisVecLen, 0.0, 0.0 );
515     e2 = pkRealVectorAlloc1( "\\two",   3, 0.0, basisVecLen, 0.0 );
516     e3 = pkRealVectorAlloc1( "\\three", 3, 0.0, 0.0, basisVecLen );
517     pkRealVectorScale(e2,0.8);
518     pkRealVectorScale(e3,0.6);
```

The array $\mathbf{r}$ represents an $N_x \times N_y$ matrix of positions vectors on $\mathcal{H}$.

```
519     r = _allocHumpPosnArr( Nx, Ny, 0.12 );
```

$\mathcal{H}$'s apex position.

```
520     u = humpX;
521     v = humpY;
522     apex =   pkRealVectorAlloc1( "\\veca", 3, u, v, _hump(u,v) );
```

```
523      apex1 =  pkRealVectorAlloc1( "a", 3, u,   0.0, 0.0 );
524      apex2 =  pkRealVectorAlloc1( "b", 3, 0.0, v,   0.0 );
525      apex3 =  pkRealVectorAlloc1( "h", 3, 0.0, 0.0, 0.8 * _hump(u,v) );
526      apex12 = pkRealVectorAlloc1( "a12",  3, u, v,  0.0 );
```

The local origin $\mathbf{p}$. Here I cheat a bit by making recourse to a global property of $\mathcal{H}$. Let $x = a + r\cos\theta$ and $y = b + r\sin\theta$. Then $z(r) = h/(r^2+1)$. Choose r such that $z(r) = \beta z(0) = \beta h$ for some β. This gives

$$x(\beta,\theta) = a + \sqrt{(1-\beta)/\beta}\cos\theta, \quad y(\beta,\theta) = b + \sqrt{(1-\beta)/\beta}\sin\theta$$

```
527      u = humpX + sqrt( ( 1.0 - 0.35 ) / 0.35 ) * cos( 75.0 / 180.0 * M_PI );
528      v = humpY + sqrt( ( 1.0 - 0.35 ) / 0.35 ) * sin( 75.0 / 180.0 * M_PI );
529      p = pkRealVectorAlloc1( "\\vecp",  3, u, v, _hump(u,v) );
```

The $\hat{\mathbf{t}}(\mathbf{p})$ and $\hat{\mathbf{n}}(\mathbf{p})$ basis vectors (Eqs. (20) and (21)).

```
530      tHat = pkRealVectorAlloc1( "\\that(\\vecp)", 3,
531                                 pkRealVectorGetComponent(p)[1] - humpY,
532                                 -pkRealVectorGetComponent(p)[0] + humpX,
533                                 0.0 );
534      pkRealVectorScale(
535            tHat,
536            sqrt( pkRealVectorGetComponent(p)[2] /
537                  ( humpHeight - pkRealVectorGetComponent(p)[2] ) ) );
538      pkRealVectorScale( tHat, 0.4 * basisVecLen );
539      pkRealVectorIncrease(tHat,p);
540      nHat = pkRealVectorAlloc1( "\\nhat(\\vecp)", 3,
541                                 pkRealVectorGetComponent(p)[0] - humpX,
542                                 pkRealVectorGetComponent(p)[1] - humpY,
543                                 0.0 );
544      pkRealVectorScale(
545            nHat,
546            sqrt( pkRealVectorGetComponent(p)[2] /
547                  ( humpHeight - pkRealVectorGetComponent(p)[2] ) ) );
548      pkRealVectorScale( nHat, 0.3 * basisVecLen );
549      pkRealVectorIncrease(nHat,p);
```

Prepare an array of N_d sample positions, $\mathbf{d}^i$, for $\mathcal{H}$'s z-contour path passing through $\mathbf{p}$. Refer to the $\mathcal{D}(\mathbf{p}, \Delta t)$ set in Eq. (25).

```
550      d = _allocHumpContourPosnArr( p, Nd, 0.021 );
```

Rotationally transform the landscape onto the space spanned by the "line-of-sight" basis. That is, transform all vectors into "shadow" vectors which lie in the $\hat{\mathbf{i}}'\hat{\mathbf{2}}'$ plane lying flat on the page.

```
551      for ( i = 0;  i < Nx;  i++ ) {
552         for ( j = 0;  j < Ny;  j++ ) {
553            pkRealVectorUnderLineOfSightBasis1( r[i][j],
554                                                xLos, yLos, zLos, theta );
555         }
556      }
557      pkRealVectorsUnderLineOfSightBasisV1( xLos, yLos, zLos, theta,
558                                            d, Nd );
559      if ( 0 == pkRealVectorsUnderLineOfSightBasis1( xLos, yLos, zLos, theta,
560                                                     e1, e2, e3,
561                                                     apex, apex1, apex2, apex3, apex12,
```

```
                                                    p,
                                                    tHat, nHat,
                                                    NULL ) ) {

```

Prepare the Ti*k*Z commands for typesetting the projection of the three-dimensional landscape of $\mathcal{H}$.

```
         puts(   "\\begingroup" );
         puts(   "\\definecolor{occludedpathcolor}{rgb}{0.6,0.6,0.7}" );
         puts(   "\\begin{PkTikzpicture}[scale=1.8," );
         puts(   "                       hump/.style={pktikzsurfacelines," );
         puts(   "                                    fill=pktikzsurfacefillcolor," );
         puts(   "                                    opacity=0.5}," );
         puts(   "                       occludedpath/.style={pktikzsurfacepath,occludedpathcolor}]
         _drawCoordinates();
         _drawBasisVectors( e1, e2, e3 );
         _drawApex( apex, apex1, apex2, apex3, apex12 );
         _drawHump( r, Nx, Ny );
         _drawContourPath( d, Nd );
         _drawLocalBasisVectors( tHat, nHat, p );
         _drawContourPathStart(p);
         puts(   "\\end{PkTikzpicture}");
         puts(   "\\endgroup" );

      } else {

         puts("ERROR: 'pkRealVectorsUnderLineOfSightBasis1()' failed.");

      }

```

Finally, clean up.

```
      pkRealVectorFree1(e1);
      pkRealVectorFree1(e2);
      pkRealVectorFree1(e3);
      _freeHumpPosnArr( r, Nx, Ny );
      pkRealVectorFree1(apex);
      pkRealVectorFree1(apex1);
      pkRealVectorFree1(apex2);
      pkRealVectorFree1(apex3);
      pkRealVectorFree1(apex12);
      pkRealVectorFree1(p);
      pkRealVectorFree1(tHat);
      pkRealVectorFree1(nHat);
      _freeHumpContourPosnArr(d,Nd);

      return;
   }

   int main( const int argc, const char *argv[] )
   {
      _diagram();
      exit(0);
   }
```

6.3 Making it all with `make`

This simple UNIX "makefile" captures the necessary file dependencies, and demonstrates how to compile the C files.

Generic Make targets.

```
all: path-parametrisation-in-R3.pdf

clobber: latexclobber
        @rm -f *.o
        @rm -f *.run spherefigure.tex
        @rm -f *.core

backup: clobber
        @PACKDIR=‘basename \‘pwd\‘‘  &&  cd ..  &&  tar -czvf ${TARPATH} $${PACKDIR}

```

File based Make targets.

```

path-parametrisation-in-R3.pdf: spherefigure.tex \
                                Makefile.demo \
                                path-parametrisation-in-R3.bib

```

Implicit rule targets.

```

.SUFFIXES: .c .o .run .tex
.c.o:
        clang -c -DDEBUG=2 -I/usr/local/pklib/include -DFreeBSD -o ${@} ${<}
.o.run:
        clang -DDEBUG=2 -I/usr/local/pklib/include -DFreeBSD -o ${@} ${<} \
              /usr/local/pklib/lib/libpk.a \
              /usr/local/pklib/lib/libpkmath.a \
              -lm
.run.tex:
        ./${<} > ${@}

```

Incorporate PKLATEXMAKE.[8]

```

# Added by 'pklatexmake.mk'.  Do not delete. 26Jul16
.include "/usr/local/pklatexmake/lib/pklatexmake.mk"
```

References

[1] Paul Kotschy. The PKLIB C software library.

[2] Saturnino L. Salas and Einar I. Hille. *Calculus—One and Several Variables.* Number 0-471-86548-6. John Wiley & Sons, Inc., 1982.

[3] Murray R. Spiegel. *Schaum's Outline Series—Mathematical Handbook of Formulas and Tables.* Number 07-060224-7. McGraw–Hill, 1968.

[4] Erwin Kreyszig. *Advanced Engineering Mathematics. Fifth Edition.* Number 0-471-88941-5. John Wiley & Sons, Inc., 1983.

[5] Paul Kotschy. pkTi*k*Z: A simple LATEX package providing Paul Kotschy's local style and command definitions for use with Ti*k*Z. *Still to be published.*

[6] Paul Kotschy. pkTechDoc: Literate programming for non-TEX programmers. *Still to be published.*

[7] Paul Kotschy. Rotational transformations in three dimensions. *Still to be published.*

[8] Paul Kotschy. The pkLATEXmake package. *Still to be published.*