On the Dimensionality of Reality

Paul Kotschy

12 August 2016

Compiled on March 10, 2025

Abstract

What is reality?[1] What is physical? What is metaphysical? Does an ineffable supernature exist such that it extends beyond the reach of our empirical and rational faculties? If so, then all is sorted, and we can happily bathe in the warm limpid water of our ignorance. But if not so, then all is *not* sorted, and an icy uncertainty impels us to observe and contemplate reality more seriously, without recourse to putative spirit-world material-world dualisms.

This work, then, derives from three interrelated convictions:

1. that dualism is, well, false.
2. that the physical world we experience day to day in a mundane sort of way is not the full picture of reality, although it is an important picture.
3. that our naturalistic gazes are sufficient for a much deeper insight into reality, provided we look carefully, perhaps more carefully than is comfortable.

In this work I mull the abovementioned questions. I begin by contemplating a most basic notion of reality, namely, the notion of dimension. I then appeal to an empirical and rational mindset to motivate for the existence of some such reality much richer than what we may easily intuit. But no less real.

By exploring the notion of dimension in this manner, I offer an epistemologically rigorous pathway to help discover such possible metaphysical realities. I argue that these realities have as much right to existence as our own physical reality even though they are in principle orthogonal to our own. And importantly, it this orthogonality, not duality, which distinguishes our physical reality from any metaphysical ones. It is a distinction without boundary.

Contents

[1] paul.kotschy@gmail.com

1 Introduction

OUR INTUITION[2] is shaped by an experience of a world that is three-dimensional in space, and where time is some apparent universal parameter relative to which we position ourselves in space. This experience is so powerful that we mostly carry out such positioning unconsciously and automatically.

Space and time, together with the associated notion of dimensionality, therefore seem ubiquitous and profoundly prevalent. And yet, they are difficult for me to grasp intuitively. Indeed, if I begin by contemplating dimensionality beyond three, then I fail just as I begin.

Fortunately, all is not lost. The first three dimensions (in space) are moderately accessible, with the first two objectively so. It is relatively easy to "look down" onto a flat plane or a set of lines on the plane, and to intuit its existence in relation to our three dimensions.

Of course, our experience of space and time in this way is an approximation of something more subtle, something in which space and time are not separate, and where our time keeping faculty is neither constant nor universal.[1] This work neglects these subleties because it is more of a philosophical study of the notion of dimensionality than it is of space and time themselves.

Herein I hope to argue, qualitatively at least, for the plausability of the existence of a much richer geometrical world than ours, a world in which ours is but a particular case, albeit no less real. And a world which exists right before our very eyes, even though we are in principle utterly unable to see it!

As you join me on this philosophical journey, let's take it slowly! Let's take time to reflect on notions so commonplace that they are ordinarily ignored, but which I believe harbour sublime profundity.

And so we begin with abject simplicity. I imagine a hypothetical one-dimensional world. Such a simple world will help clarify the very notion of dimension, and it will hopefully be a guide in how (and where) we may transition between different notions of reality by an increment in the number of dimensions. As we explore this world, we shall discover that it is embedded in our own three-dimensional space. And importantly, we shall obtain heuristics for contemplating dimensions higher than three in number.

2 Alone in a dark room

I EXIST ALONE in a completely dark room. I hear nothing, see nothing, feel nothing. Indeed, I do not feel alone because I am unable to feel. I am, if you will, suspended in a state of weightlessness, free of any intertia, and unable to move.

To be sure, it is really not possible for us to experience a world such as this. Firstly, we are ostensibly three-dimensional spatial beings, and any musing of the nature of an existence outside three dimensions is speculative at best. And secondly, since our metabolic processes take place over time, it is not possible to remove time itself from our imagination of the dark room. Nevertheless, it is helpful for now to try imagine a world having a minimum of variability and stimulus.

The room is empty, save for a calibrated lever of sorts, and a simple calibrated dial, as shown in Figure 1. Why *must* there be such a lever and a dial? Because without them, there would be no world at all. Without them there would be no cause for there to be any effect.

[2] This work was inspired in part by the curious observation that the vector cross-product at a point on a surface embedded in $\mathbb{R}^3$ is identical to the gradient at a corresponding point in a volume embedded in $\mathbb{R}^4$, and for which volume the original surface is a contour surface. The vector cross-product is a useful tool for manipulating vectors in $\mathbb{R}^3$. This is so, firstly, because of its inherent circular character, and secondly, because of the ease with which it creates orthogonal vectors. Although strictly, in a tensorial sense, the vectors it creates are not true vectors. They do not transform as vectors under coordinate transformations. But aside from its mere utilitarian value, the cross-product has more to offer. Its correspondence with the gradient suggests how or where we might look for "hidden" dimensionality in our world.

Surprisingly, I am aware of the lever and dial. I can see—or at least, perceive—the needle on the dial, and I can read off its corresponding value. I can push and pull on the lever at will. This is my one-dimensional world. All I can do is while away my time, manipulating the lever and watching the dial. Although strictly speaking, if the lever is not to represent time itself, then not even time exists.

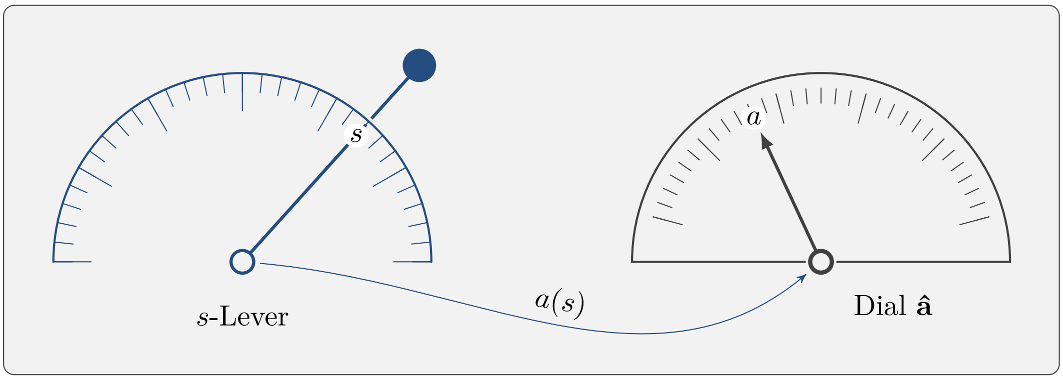

Figure 1: A "dark room" world comprised of nothing more than a lever and a dial. Observation of the world reveals that the lever and the dial are predictably and reliably connected. The lever position determines the dial reading. This connectedness is expressed as the function $a(s)$.

As I pull slightly on the lever, fixing it at some position, I notice the dial's needle shift and come to rest at some value. As I pull further on the lever, fixing it at some other position, the dial's needle comes to rest at some other value. This is of course of little interest. Except, after manipulating the lever repeatedly, I begin to notice I can predict the value of the dial. In fact, I discover that the lever and dial must be connected in some way. The lever position determines the value of the dial. Predictably and reliably.

And that would be that. My ability to control the lever offers control over my reality, albeit a simple one (the dial), and there would be no reason—indeed, no ability—to contemplate any other reality. Of course, there is the nagging question of *how* the lever controls the dial; that is, of how they are connected. But I might incline to relegate that nag to the realm of the metaphysical. Such relegation is reasonable because my experience of reality offers no additional clues.

To describe (or record) my reality, I could, in my mind's eye of course, write something like[3]

$$\mathbf{x}(s) = a\hat{\mathbf{a}} = a(s)\hat{\mathbf{a}} \tag{1}$$

where s is a measure of the lever position, and a is the value I read (or perceive) on the dial, as shown in Figure 1. The notation $a(s)$ captures my observed connection between lever position s and dial reading a. In effect, a is a function of s. The notation $\hat{\mathbf{a}}$ shows that I have complete freedom to fix the dial's value by manipulating the lever. And $\mathbf{x}(s)$ stands for the entire state of my reality—a one-dimensional reality consisting of a dial $\hat{\mathbf{a}}$, the dial's value a, and a connection $a(s)$ between lever s and dial reading a. My world is closed, controlled, and utterly boring.

Now suppose that, for whatever reason, I become aware of a second dial. Pushing and pulling on the lever seems also to have an affect on this second dial's needle reading. And as before, after manipulating the lever repeatedly, I discover a predictable connection between the lever and the second dial, as shown in Figure 2.

Based on these observations, I could now describe (or record) the state of my reality with

$$\mathbf{x}(s) = (a, b)\hat{\mathbf{a}} = (a(s), b(s))\hat{\mathbf{a}} = (a, b)(s)\hat{\mathbf{a}} \tag{2}$$

[3]I have deliberately tried to delay the use of symbolic math language. But the few early mathematical expressions are worth mulling over because I think they encapsulate concisely the notion of dimension and the developing state of my reality.

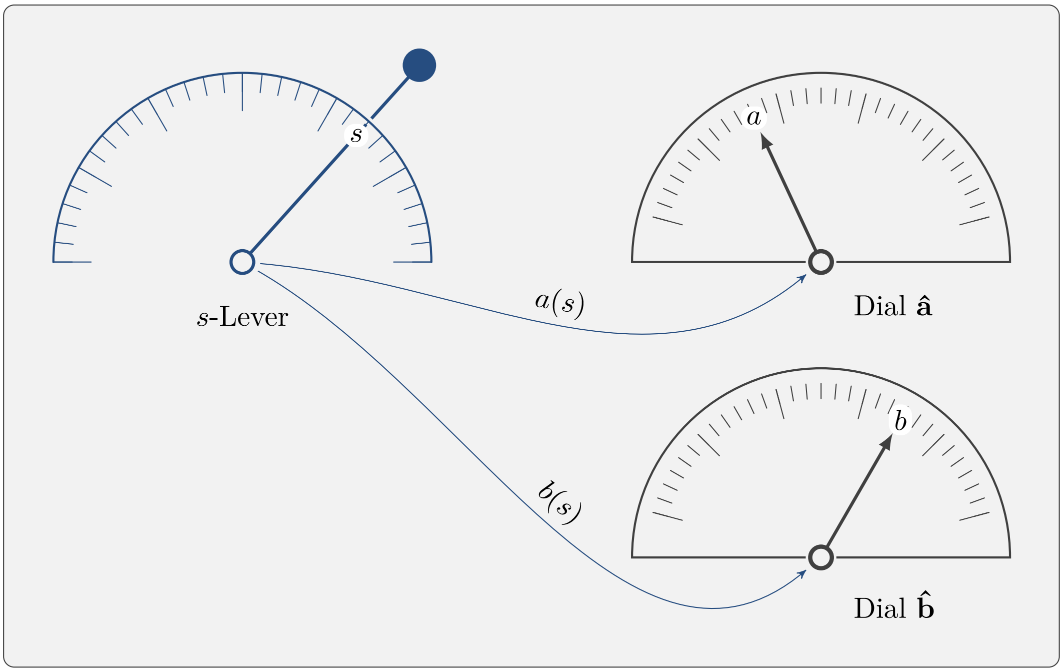

Figure 2: A "dark room" world comprised of a lever and two dials. Observation of the world reveals that the lever and the two dials are predictably and reliably connected. The lever position determines the dial readings. The connectedness is expressed as the two functions $a(s)$ for dial $\hat{\mathbf{a}}$ and $b(s)$ for dial $\hat{\mathbf{b}}$.

where b is the reading on the second dial, and $b(s)$ captures my observed connection between lever s and second dial's value b. The pair $(a(s), b(s))$ emphasises the fact that there are now two such connections, one for each dial. And $(a, b)(s)$ shows that although there are now two dials, the predictability of the two connections between lever and dial value means that the pair can be thought of singularly. Indeed, I am unable to influence one dial alone.

Once again, my world is closed and controlled. But now it's slightly less boring. Why would the alleged metaphysical realm include a second dial, identical to the first in appearance and hence indistinguishable from the first, but connected differently than the first to the lever? And if there are now two dials, could there be more dials to discover? And might there be another lever? And of course, the very existence of these connections between lever and dial is becoming a niggle. Why are they there? There is nothing in my world which motivates for them.

Without any additional immediate insights or artifacts with which to clarify or embelish my experience of reality, I could simply capitulate, convincing myself that that is simply the way things are, predetermined by some opaque metaphysical realm, worked by the Divine Watchmaker, God in control. And I would confess ignorance on the origins of the lever, the two dials, and their connections. I might try gain some comfort from this blissful ignorance. It never is and it never was my role nor right to impugn the metaphysical anyway.

Or, I could decide to contemplate my reality more deeply. Is there a way to make some sense of it? Can I comprehend my context without naïvely appealing to some super-nature? I am inevitably drawn to reflect again on the implicit connection between the two dials (via the lever). Perhaps it hints at something deeper, something richer. Could it be that, in fact, the two dial's *are in general not connected, but that my experience of them being connected is just something unusual*? Perhaps my own sense of reality, although no less real, is just one thread woven amongst many in an invisible tapestry.

If this is so, then instead of writing (2), I am free to recast a description of the state of my reality as

$$\mathbf{x}(s) = a\hat{\mathbf{a}} + b\hat{\mathbf{b}} = a(s)\hat{\mathbf{a}} + b(s)\hat{\mathbf{b}} \tag{3}$$

This now leaves me startled. My experience of my reality has not changed at all! I still have two dials, $\hat{\mathbf{a}}$ and $\hat{\mathbf{b}}$, which are mysteriously connected via the lever s. But now, surreptitiously, I am beginning to imagine them as independent entities. By simply recording the state of my reality using (3), I have allowed for the possibility for dial $\hat{\mathbf{a}}$ and dial $\hat{\mathbf{b}}$ to be independent entities *in general*, but to be connected in my case *in particular.* To record my own experience, I may happily use either (2) or (3) because in my world, in a practical sense, both mean the same thing.

It seems that the narrow walkway of my world is beginning to widen!

The profundity of this sparkle of insight merits further consideration. *A mysterious and intimate connection between two qualitatively indistinguishable observed quantities hints at a richer, higher-dimensional reality, in which mine is embedded as a particular case.* And indeed I may reasonably postulate the existence of this reality without ever being able to perceive it directly. This strikes me as very important philosophical insight.

Whereas previously, I was impelled to ask how or why the two dials are connected, I now ask why not? But this rather nonchalant retort comes with having now to contemplate a much richer two-dimensional world in which my one-dimensional world is somehow embedded as a particular case alongside many. In the general two-dimensional world, the two dials vary independently. But in my particular case, they do not.

So if my reality is labelled as the $\bar{t}$-th one, say, then I should write for the state of my reality:

$$\mathbf{x}(s, \bar{t}) = a\hat{\mathbf{a}} + b\hat{\mathbf{b}} = a(s, \bar{t})\hat{\mathbf{a}} + b(s, \bar{t})\hat{\mathbf{b}} \tag{4}$$

where the notation $(s, \bar{t})$ serves to acknowledge that mine—the $\bar{t}$-th one—is one amongst many.

Henceforth, I shall use the terms *path* and *reality* interchangeably. To be sure, my $\mathbf{x}(s, \bar{t})$ path differs from, say, the $\mathbf{x}(s, t')$ path. And since $\bar{t}$ is merely a label, the value of $\bar{t}$ must remain unchanged throughout my path. That is, as I experience my reality, I can never expect to observe a change in the value of $\bar{t}$. And so for me, $\mathbf{x}(s)$ in (2) and (3) and $\mathbf{x}(s, \bar{t})$ in (4) all mean the same thing. The experience of my own world has not changed.

A putative two-dimensional world is represented in Figure 3. The fact that the individual "t"-labelled paths are circular segments in the diagram is not important here. The figure hints at something crucial. There are numerous (in fact, infinitely many) unique paths, each with their own "t" label, over which respective s-lever positions may vary. I imagine numerous corresponding dark rooms, each with one s-lever and two dials.

But the two-dimensional world in which mine is embedded offers no qualitative character distinction between lever values s and t, nor between dial readings a and b. So I am obliged to augment my reality with an additional lever—a t-lever which is identical to the first, but which remains fixed and unchangeable at the position $\bar{t}$, as shown in Figure 4.

Now I am struck with a tantalising insight. If the two levers and dials are mutually indistinguishable, there must exist another family of paths in the two-dimensional world for which the s-lever is fixed and unchangeable and the t-lever is allowed to vary. If I could just unlock my t-lever and pull and push on it while keeping my s-lever fixed at some position, I would be able to experience one such path. And by manipulating my t-lever now, I would likely discover some new connection between it and my two dials, just as I had done earlier with my s-lever.

If it so happened that in my dark room world, the s-lever was fixed and unchangeable at position s^*, say, with the t-lever able to be manipulated, then everything in my reality would be identical to my present one except for the different connections between the t-lever's position and the two dials. Furthermore, the state of that alternate reality would coincide with my present state whenever $s = s^*$ and $t = \bar{t}$.

In that alternate reality, following (3), I would be obliged to cast my state as

$$\mathbf{x}(t) = a\hat{\mathbf{a}} + b\hat{\mathbf{b}} = a(t)\hat{\mathbf{a}} + b(t)\hat{\mathbf{b}}$$

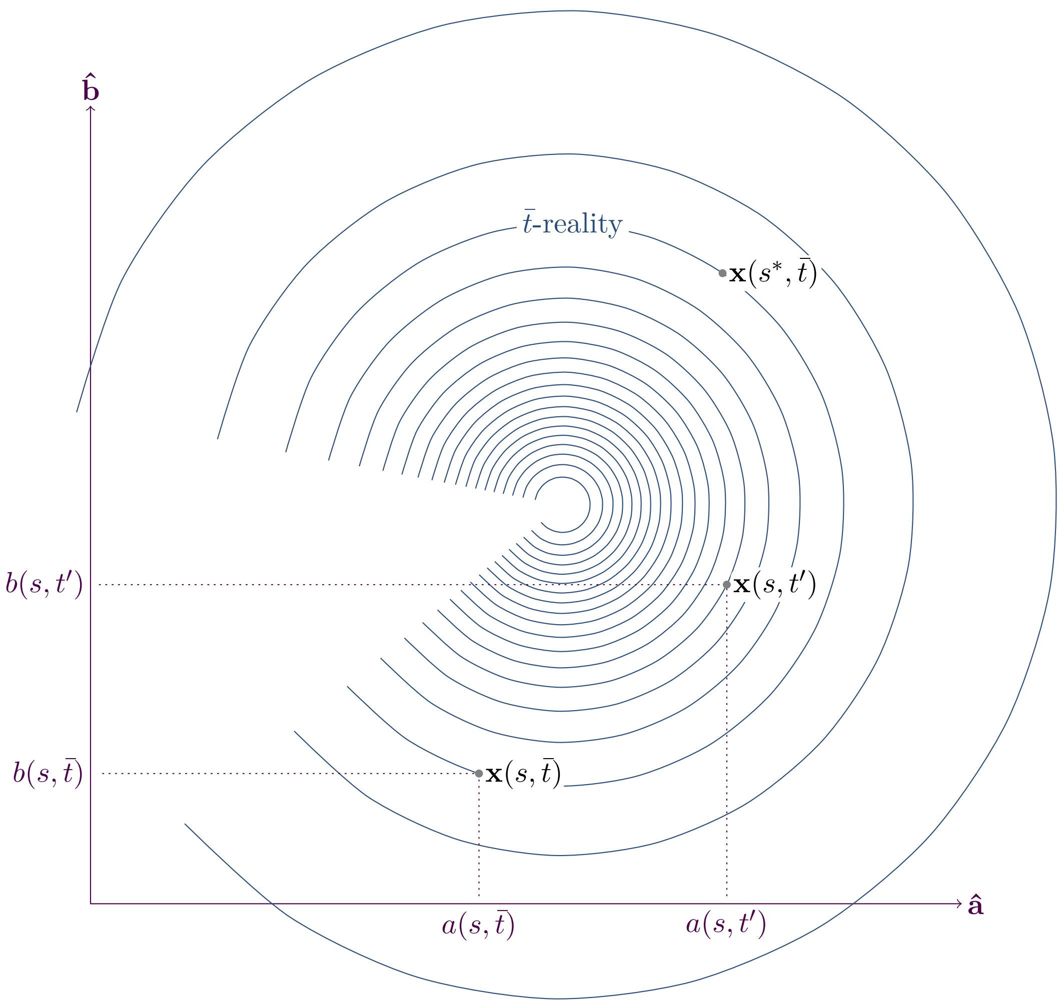

Figure 3: Putative two-dimensional $\hat{\mathbf{a}}\hat{\mathbf{b}}$ world represented as a family of one-dimensional t-realities, of which my $\bar{t}$-reality is but one amongst many "sibling" realities. The present state of my own reality, which is represented by the position $\mathbf{x}(s, \bar{t})$ for some lever value s, differs from the state of some other reality represented by the position $\mathbf{x}(s, t')$. Throughout my reality, $\bar{t}$ remains unchanged.

But whereas my own present reality is labelled as $\bar{t}$ with s varying, this alternate reality must carry the label s^* with t varying. So actually, the state of that reality would perhaps be better represented with

$$\mathbf{x}(s^*, t) = a\hat{\mathbf{a}} + b\hat{\mathbf{b}} = a(s^*, t)\hat{\mathbf{a}} + b(s^*, t)\hat{\mathbf{b}} \tag{5}$$

And in that reality $\mathbf{x}(t)$ and $\mathbf{x}(s^*, t)$ would mean the same thing. Comparing (4) with (5), it is obvious that the two realities intersect whenever $s = s^*$ and $t = \bar{t}$, and the coinciding state position is $\mathbf{x}(s^*, \bar{t})$. This particular state position is shown in both Figures 3 and 5.

In that alternate s^*-labelled reality, I should have arrived at similar conclusions, that the presence of the s-lever fixed at s^*, my ability to manipulate my t-lever, and the observed uncanny connections between the two dial readings and t-lever position, all hint at my s^*-labelled reality as being but one amongst many. And I would be forced to acknowledge the existence of a putative two-dimensional world as represented in Figure 5.

But that is not my reality. And no matter how compelling the representations in Figures 3 and 5 may be, I am unable to experience the $\hat{\mathbf{a}}\hat{\mathbf{b}}$ world in full. The best I can do is to observe a connection between the two dials via my manipulable s-lever.

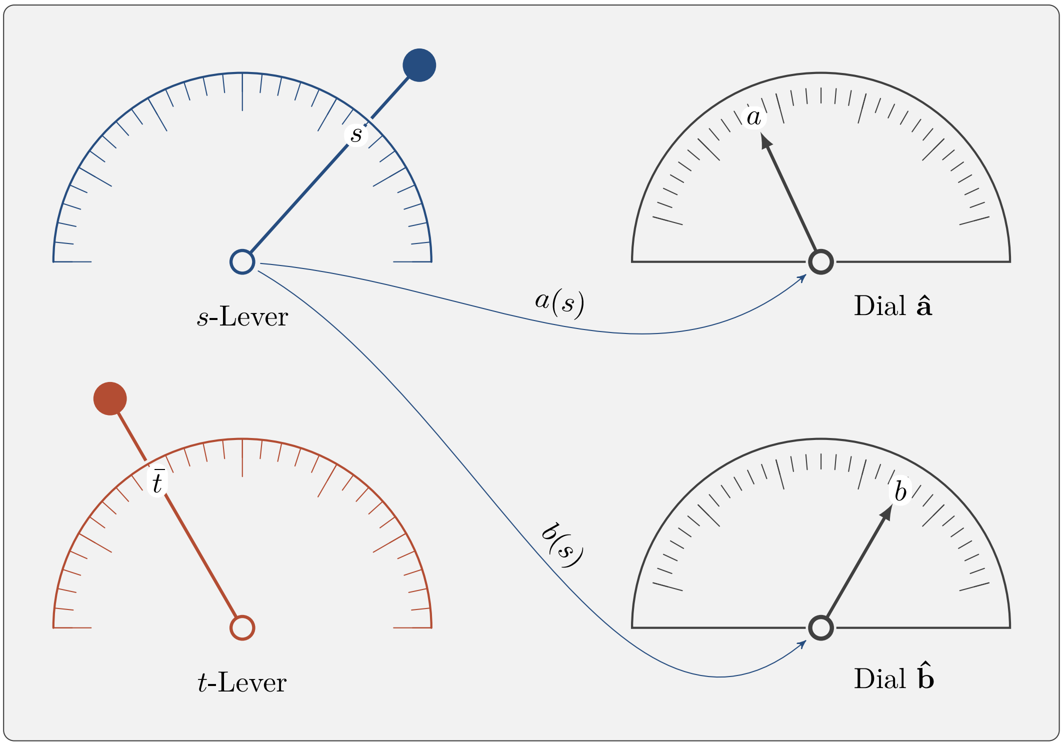

Figure 4: A "dark room" world comprised of two levers and two dials. Observation of the world reveals that the s-lever and the two dials are predictably and reliably connected, and that the t-lever remains fixed at position $\bar{t}$. The s-lever position determines the dial readings. The connectedness is expressed with the two functions $a(s)$ for dial $\hat{\mathbf{a}}$ and $b(s)$ for dial $\hat{\mathbf{b}}$.

3 Pointer off my world

To recap, my existence is profoundly dark, silent and formless, save for two identical levers and two identical dials. I can manipulate only one of the levers. The other appears fixed at position $\bar{t}$. Manipulating my first lever apparently affects both dials differently. But the effect is predictable and reliable. This is uncanny and unexpected. To resist the temptation for fulsome deference to the metaphysical, I must humbly accept that my own reality is a prosaic case of some richer world in which the two dials are *not coupled nor connected*, and in which the two levers can be manipulated independently.

So I place my s-lever at position s^*, say. And as I move it slightly away from s^*, I reflect on what it would be like to move from this position by instead moving my t-lever away from position $\bar{t}$. Recall that the abovementioned s^*-labelled reality intersects mine when its t-lever is set at $\bar{t}$ (Figure 5). I conclude that if that s^*-labelled reality path were to be able to interact with my own $\bar{t}$-labelled reality path, then my experience of the interaction would coincide with an experience in the s^*-labelled reality when my s-lever was set at s^* and its t-lever was set at $\bar{t}$.

But what would constitute such an interaction? Since in my simple world I control my s-lever, an interaction can only mean that my t-lever mysteriously moves. And from the perspective of the other world, an interaction must mean that the other world's s-lever mysteriously moves. And because my t-lever has moved, I should expect the subsequent character of the connection between my s-lever and my two dial readings to change in a surprising manner. Likewise, in the s^*-labelled reality, the character of the connection between its t-lever and its two dial readings would also change in a surprising manner.

So in my reality, as I move my s-lever from the position s^* to position $s^* + \Delta s$, say, I observe that

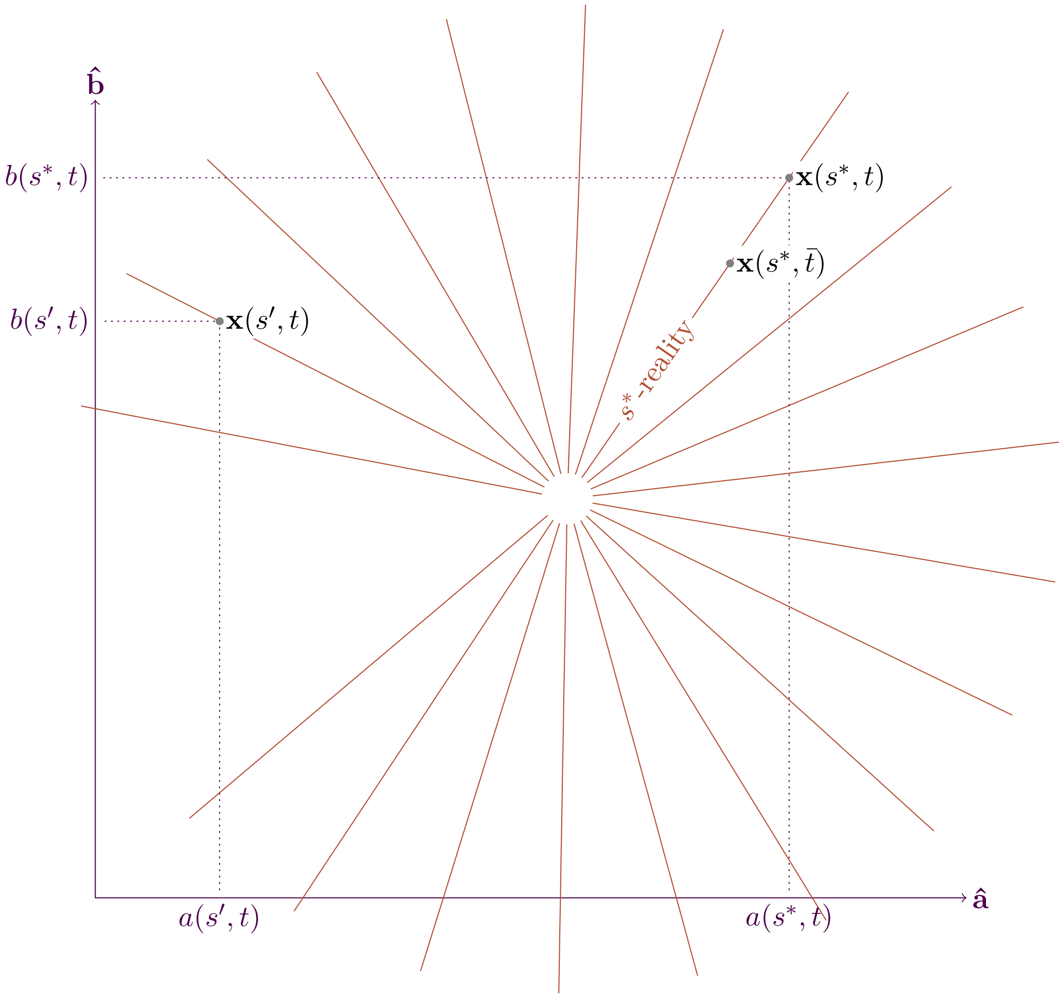

Figure 5: The same putative two-dimensional $\hat{\mathbf{a}}\hat{\mathbf{b}}$ world of Figure 3, but now represented as a family of one-dimensional s-realities, of which the s^*-reality is but one amongst many "sibling" realities. The present state of reality, as represented by the position $\mathbf{x}(s^*, t)$ for some lever value t, differs from the state of some other reality represented by the position $\mathbf{x}(s', t)$. Throughout the s^*-reality, s^* remains fixed. It intersects my own $\bar{t}$-reality (Figure 3) whenever its t-lever is set at $\bar{t}$, as shown by the position $\mathbf{x}(s^*, \bar{t})$.

my respective dial readings change from a to $a + \Delta\bar{a}$, and from b to $b + \Delta\bar{b}$. The change in state of my reality is therefore, using (4),

$$\begin{aligned} \Delta\mathbf{x}(s^*, \bar{t}) &= \mathbf{x}(s^* + \Delta s, \bar{t}) - \mathbf{x}(s^*, \bar{t}) \\ &= \big(a(s^* + \Delta s, \bar{t})\hat{\mathbf{a}} + b(s^* + \Delta s, \bar{t})\hat{\mathbf{b}}\big) \;-\; \big(a(s^*, \bar{t})\hat{\mathbf{a}} + b(s^*, \bar{t})\hat{\mathbf{b}}\big) \\ &= \big(a(s^* + \Delta s, \bar{t}) - a(s^*, \bar{t})\big)\hat{\mathbf{a}} \;+\; \big(b(s^* + \Delta s, \bar{t}) - b(s^*, \bar{t})\big)\hat{\mathbf{b}} \\ &= \Delta\bar{a}\hat{\mathbf{a}} + \Delta\bar{b}\hat{\mathbf{b}} \end{aligned} \tag{6}$$

Similarly, were I to be living in the intersecting reality, I would move my t-lever from position $\bar{t}$ to position $\bar{t} + \Delta t$, say, and observe the dial readings change from a to $a + \Delta a^*$, and from b to $b + \Delta b^*$. And the change in state of reality would then be, using (5),

$$\begin{aligned} &\mathbf{x}(s^*, \bar{t} + \Delta t) - \mathbf{x}(s^*, \bar{t}) \\ &= \big(a(s^*, \bar{t} + \Delta t)\hat{\mathbf{a}} + b(s^*, \bar{t} + \Delta t)\hat{\mathbf{b}}\big) \;-\; \big(a(s^*, \bar{t})\hat{\mathbf{a}} + b(s^*, \bar{t})\hat{\mathbf{b}}\big) \\ &= \big(a(s^*, \bar{t} + \Delta t) - a(s^*, \bar{t})\big)\hat{\mathbf{a}} \;+\; \big(b(s^*, \bar{t} + \Delta t) - b(s^*, \bar{t})\big)\hat{\mathbf{b}} \\ &= \Delta a^*\hat{\mathbf{a}} + \Delta b^*\hat{\mathbf{b}} \end{aligned} \tag{7}$$

As I write this (in my mind, of course), I happen to notice that in my reality, the quantity

$$\Delta a^* \Delta \bar{a} \; + \; \Delta b^* \Delta \bar{b} \tag{8}$$

involves the changes in both my dials and the dials of the intersecting s^*-reality. And importantly, if I was living in the intersecting s^*-reality, the corresponding quantity that I would have written is

$$\Delta \bar{a} \Delta a^* \; + \; \Delta \bar{b} \Delta b^* \tag{9}$$

The two sums are identical, provided that $\Delta a^* \Delta \bar{a} = \Delta \bar{a} \Delta a^*$ and $\Delta b^* \Delta \bar{b} = \Delta \bar{b} \Delta b^*$. So (8) (or (9)) offers a simple and sensible measure of the intensity of any interaction when my s-lever is set at s^* with my t-lever at $\bar{t}$. And satisfyingly, the measure is the same in both realities.

To be sure, as I traverse my path of reality using my s-lever with my t-lever fixed at $\bar{t}$, any contemplation of interactions and interaction intensity is predicated both on the observed and predictable effect that my s-lever has on both dials, and on the insightful conjecture that there exists a richer world in which both levers may be manipulated independently.

But where is this richer world—an imperceptible world, "doubly" as expressive as mine? A world hidden from view, yet all around, ubiquitous, immanent. It must be there because I have two identical dials $\hat{\mathbf{a}}$ and $\hat{\mathbf{b}}$ whose readings depend on my s-lever in two different yet predictable ways.

Can I somehow point to this richer world? With the interaction intensity measure (8) in mind, I notice that

$$(\Delta \bar{b}) \Delta \bar{a} \; + \; (-\Delta \bar{a}) \Delta \bar{b} \; = \; 0 \tag{10}$$

provided of course that $\Delta \bar{b} \Delta \bar{a} = \Delta \bar{a} \Delta \bar{b}$. Comparing (10) with (8), the bracketed $(\Delta \bar{b})$ and $(-\Delta \bar{a})$ in (10) occupy, respectively, the same positions as Δa^* and Δb^* in (8). So with (7) in mind, this suggests that with my s-lever set at s^*, if there was another s^*-labelled reality intersecting mine whose change in state with its t-lever set at $\bar{t}$ was

$$(\Delta \bar{b}) \hat{\mathbf{a}} + (-\Delta \bar{a}) \hat{\mathbf{b}} \tag{11}$$

then my interaction intensity with it would be 0. That is, I would not experience any interaction at all even though the intersecting reality is as viable as mine. Furthermore, since the change of state (11) in the intersecting reality triggers an immeasurable interaction (10) in my reality, then so does this change of state:

$$(\lambda \Delta \bar{b}) \hat{\mathbf{a}} + (-\lambda \Delta \bar{a}) \hat{\mathbf{b}} \tag{12}$$

for any multiple λ. This is precisely because

$$(\lambda \Delta \bar{b}) \Delta \bar{a} \; + \; (-\lambda \Delta \bar{a}) \Delta \bar{b} \; = \; \lambda \left((\Delta \bar{b}) \Delta \bar{a} \; + \; (-\Delta \bar{a}) \Delta \bar{b} \right) \; = \; 0$$

There exists thus an entire family of realities which are likely to be every bit as real as my own, but with which I am unable to interact in a measurable way. And this must surely be "where" the extra dimension is "located!"

Roughly speaking then, to "be released" from the grip of my own one-dimensional reality, I must place myself at some point of interest s^*, say, on my path of reality. In my simple world, that means I must set my s-lever at position s^*. I must then traverse a short distance Δs along my path by moving my s-lever from position s^* to $s^* + \Delta s$. The traversal will trigger dial reading changes $\Delta \bar{a}$ and $\Delta \bar{b}$ on my dials $\hat{\mathbf{a}}$ and $\hat{\mathbf{b}}$. And from these two changes, I construct the new two-dimensional object $(\Delta \bar{b}) \hat{\mathbf{a}} + (-\Delta \bar{a}) \hat{\mathbf{b}}$—a pointing device, if you wish—which "points" into the extra dimension off my own reality path. Such a geometric construction is shown schematically in Figure 6. But remember that in my reality, I cannot actually perceive such a construction.

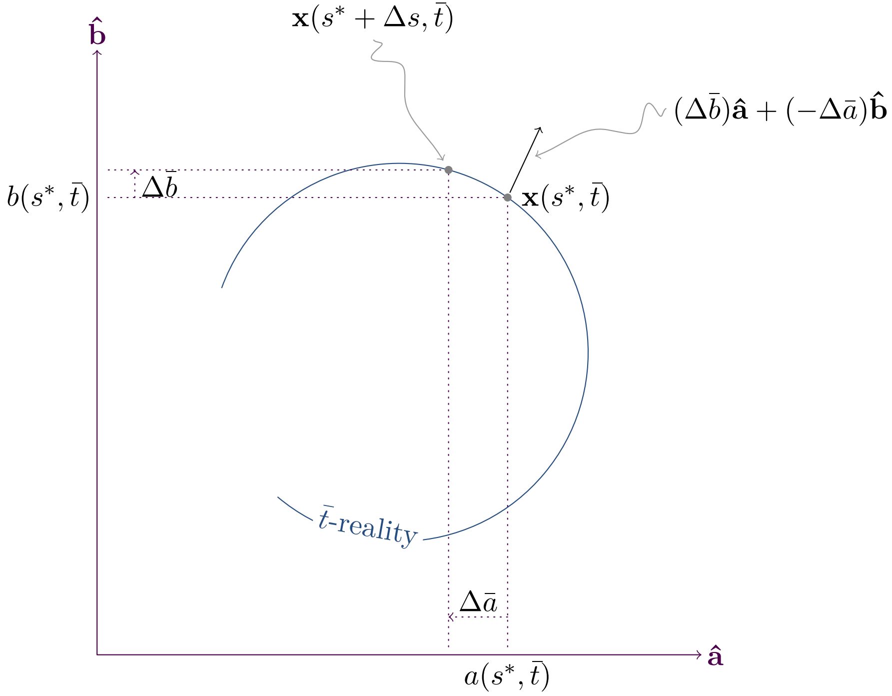

Figure 6: "Pointing" off my $\bar{t}$-reality path at the position s^* into the extra dimension. On my $\bar{t}$-labelled reality path I place myself at position s^* by setting my s-lever at s^*. The state of my one-dimensional reality is therefore $\mathbf{x}(s^*, \bar{t}) = a(s^*, \bar{t})\hat{\mathbf{a}} + b(s^*, \bar{t})\hat{\mathbf{b}}$ (Equation (4)). I move nearby to position $s^* + \Delta s$ and record the change in dial readings as $\Delta \bar{a}$ and $\Delta \bar{b}$. To "point" into the extra dimension off my $\bar{t}$-reality path, I construct the pointing device $(\Delta \bar{b})\hat{\mathbf{a}} + (-\Delta \bar{a})\hat{\mathbf{b}}$.

4 Landscape of worlds

Our own experience of the real world is obviously much richer than that of a lonely dark room containing nothing more than one lever (or two) and two dials (Figure 2). Nevertheless, the dark room is both conceivable and plausible. The room could be the (silent) engine room of a train on a track. The s-lever could be a method for specifying a desired number of wheel rotations. The first dial, $\hat{\mathbf{a}}$, displays the breadth distance covered by the engine room relative to some starting position (the "origin"), a train station, say. And the second dial, $\hat{\mathbf{b}}$, displays the length distance covered.

If it may be assumed that the train responds instantaneously to changes in the s-lever's setting, then the state of reality in Equation (2) fully captures the train driver's one-dimensional experience in the engine room. But (2) fails to capture the real broader two-dimensional landscape. However, where (2) fails, (4) and (5) succeed, even though from the driver's perspective, *all three mean the same thing*. Indeed, the driver is free to choose from any of the three statements of his or her reality. But (4) and (5) suggest something which the driver can never intuit, namely that the driver's one-dimensional reality is *embedded* in a two-dimensional world.

Equipped with the intuitive experience of our own three-dimensional spatial world, it takes little effort for us to merge (4) and (5) into a single statement of reality in order to capture the full extent of the landscape. For (4), we simply loosen the lock on the driver's t-lever. And for (5), we loosen the lock on the s-lever. By doing so we have tacitly released the train from its track! The merged statement of reality is thus

$$\mathbf{x}(s,t) = a\hat{\mathbf{a}} + b\hat{\mathbf{b}} = a(s,t)\hat{\mathbf{a}} + b(s,t)\hat{\mathbf{b}} \tag{13}$$

The simple dark room world with its t-lever loosened is shown in Figure 7. The full extent of the resulting landscape is the set of all such positional states $\mathbf{x}(s,t)$, where each position in the landscape is uniquely identifiable by the (s,t) numerical pair.

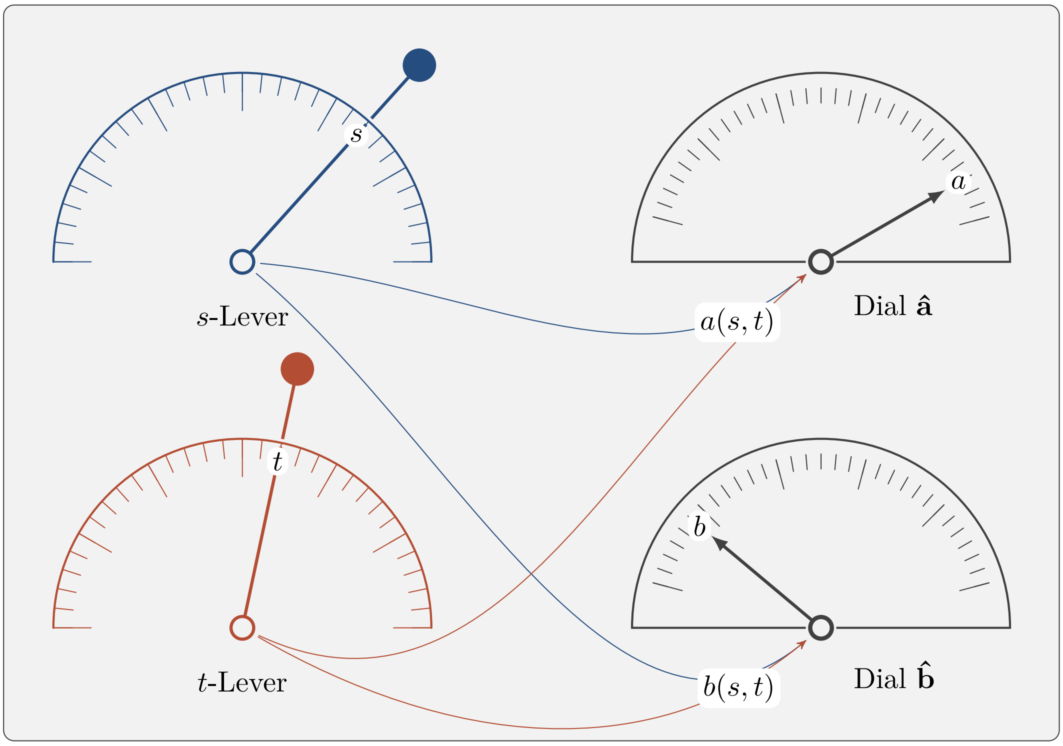

Figure 7: A "dark room" world comprised of two levers and two dials. The lock on the t-lever has now been loosened so that it may be manipulated, just like the s-lever. Both lever positions determine the reading on both dials, collectively. The connectedness between levers and dials is expressed with the functions $a(s, t)$ for dial $\hat{\mathbf{a}}$ and $b(s, t)$ for dial $\hat{\mathbf{b}}$.

For us, the merging of (4) and (5) was easy and perhaps obvious. But there is in principle no reason why the train driver could not have done likewise, even though he or she occupied a lower-dimensional reality. All that the driver needed to do was to observe and ponder the uncanny connection between his or her s-lever and the two dials, and arrive at (6) and (12), leading to the sensible generalised conjecture (13).

5 Worldly epistemology

If you are mathematically inclined, then you might be familiar with objects such as (13), as well as the notions of vectors, vector spaces, parametrised curves, parametrised surfaces, orthonormal vector bases, degrees of freedom, tangent vectors, gradient one-forms, distance metrics, invariance, and so on. And you might even be annoyed that recourse to such objects has not yet been made.

The purpose thus far was to try conceptualise an increment in the number of dimensions of reality. If this can be done for the increment from one dimension to two, and then from two to three, then I might be better equipped to identify and intuit higher increments. And, too early a facile reliance on established mathematical formalism hinders attainment of intuitive insights, because it is just too easy to write objects such as (13) and then to manipulate them mechanically with little regard for any deeper meaning.

However, notwithstanding, recourse to increased mathematical rigour at this point is inevitable and necessary.

The abovementioned two-dimensional landscape is assumed to be the Euclidean set

$$\mathcal{E}^2 = \{(a,b) \mid a,b \in \mathbb{R}\}$$

If $(a_1,b_1) \in \mathcal{E}^2$ and $(a_2,b_2) \in \mathcal{E}^2$, then a distance $d \in \mathbb{R}$ between the two elements is defined by

$$d = \sqrt{(a_2 - a_1)^2 + (b_2 - b_1)^2}$$

The set $\mathcal{E}^2$ admits the vector space over the real numbers, spanned by the orthonormal vector basis $\{\hat{\mathbf{a}}, \hat{\mathbf{b}}\}$, and parametrised with s and t, say, as

$$E^2 = \{\mathbf{x}(s,t) = a(s,t)\hat{\mathbf{a}} + b(s,t)\hat{\mathbf{b}} \mid s,t \in \mathbb{R},\ \hat{\mathbf{a}} \text{ and } \hat{\mathbf{b}} \text{ orthonomal.}\} \tag{14}$$

such that if $\mathbf{x}, \mathbf{y} \in E^2$ then $(\mathbf{y} - \mathbf{x}) \cdot (\mathbf{y} - \mathbf{x}) \in \mathbb{R}$. The one-dimensional world described above is therefore simply the subset of $\mathcal{E}^2$

$$\{(a,b) \mid a = a(s,\bar{t}),\ b = b(s,\bar{t}), \quad \text{and } s \in \mathbb{R}, \quad \text{some } \bar{t}\}$$

And it admits a vector subspace which can can be viewed as an embedded path over E^2

$$\{\mathbf{x}(s,\bar{t}) = a(s,\bar{t})\hat{\mathbf{a}} + b(s,\bar{t})\hat{\mathbf{b}} \mid s \in \mathbb{R}, \quad \text{some } \bar{t}\} \tag{15}$$

Since my “dark-room” one-dimensional $\bar{t}$-reality is nothing more than the set of all positional states $\mathbf{x}(s,\bar{t})$ in (4), the subspace (15) fully represents my reality.

Now instead of moving my s-lever by a small but finite amount from the s^* position to arrive at (6), I now move it by an infinitesimal amount from s^*, and record the rate of change of the state of my reality. Doing so provides a tangent vector at s^* as being the exact analogue of (6):

$$\mathbf{t}(s^*,\bar{t}) = \left.\frac{\partial \mathbf{x}(s,t)}{\partial s}\right|_{(s^*,\bar{t})} = \left.\left(\frac{\partial a(s,t)}{\partial s}\hat{\mathbf{a}} + \frac{\partial b(s,t)}{\partial s}\hat{\mathbf{b}}\right)\right|_{(s^*,\bar{t})} \tag{16}$$

In keeping with the argument leading to (10) and (11), an exact device which points into the extra dimension off my reality path at the s^* position must correspondingly be the normal vector

$$\mathbf{n}(s^*,\bar{t}) = \left.\left(\frac{\partial b(s,t)}{\partial s}\hat{\mathbf{a}} - \frac{\partial a(s,t)}{\partial s}\hat{\mathbf{b}}\right)\right|_{(s^*,\bar{t})} \tag{17}$$

This normal vector is a pointing device of interest because, analogous with (10), the corresponding interaction intensity measure must vanish:

$$\mathbf{n}(s^*,\bar{t}) \cdot \mathbf{t}(s^*,\bar{t}) = \left.\left(\frac{\partial b(s,t)}{\partial s}\frac{\partial a(s,t)}{\partial s}\hat{\mathbf{a}} \cdot \hat{\mathbf{a}} - \frac{\partial a(s,t)}{\partial s}\frac{\partial b(s,t)}{\partial s}\hat{\mathbf{b}} \cdot \hat{\mathbf{b}}\right)\right|_{(s^*,\bar{t})} = 0 \tag{18}$$

That is, as I move along my path of reality by manipulating my s-lever around some position s^*, I am able to discover an alternate reality path which intersects mine when my $s = s^*$ and its $t = \bar{t}$. That reality has just as much right to exist as mine, even though I cannot interact with it given that the interaction intensity is zero. The alternate state of reality must be $\mathbf{x}(s^*,t)$ as in (5) with its s-lever fixed at s^* and its t-lever free to be manipulated. And my normal vector $\mathbf{n}(s^*,\bar{t})$ in (17) points off my world into it!

Using the one-dimensional world as a starting point, the procedure for helping identify extra dimensionality is therefore summarised as:

1. The world is one-dimensional, comprised of a single variable entity, a, say:

   $$\mathbf{x} = a\hat{\mathbf{a}}, \quad a \in \mathbb{R} \quad (\text{or } \mathbb{C})$$

2. Observe that the world is not arbitrary, that there exists cause and effect, allowing for manipulation:

   $$\mathbf{x} = \mathbf{x}(s) = a(s)\hat{\mathbf{a}}, \quad s \in \mathbb{R}$$

3. Observe another variable entity, b, say:

$$\mathbf{x}(s) = (a(s), b)\,\hat{\mathbf{a}}, \quad b \in \mathbb{R}$$

4. Observe that a and b are mysteriously connected:

$$\begin{aligned} \mathbf{x}(s) &= (a(s), b(s))\,\hat{\mathbf{a}} = (a, b)\,(s)\hat{\mathbf{a}} \\ \Rightarrow \mathbf{x}(a) &= (a, B(a))\hat{\mathbf{a}} \quad \text{for some function } B \end{aligned}$$

5. Unable to account for the mysterious connection, postulate an increment in the number of dimensions, and embed my reality inside a new two-dimensional world:

$$\begin{aligned} \mathbf{x}(s) &= a(s)\hat{\mathbf{a}} + b(s)\hat{\mathbf{b}} \quad \text{or} \\ \mathbf{x}(a) &= a\hat{\mathbf{a}} + B(a)\hat{\mathbf{b}} \quad \text{for some function } B\text{, or} \\ \mathbf{x}(b) &= A(b)\hat{\mathbf{a}} + b\hat{\mathbf{b}} \quad \text{for some function } A \end{aligned}$$

The new world is two-dimensional because it is spanned by the two-membered vector basis $\{\hat{\mathbf{a}}, \hat{\mathbf{b}}\}$. But my embedded reality is still one-dimensional.

6. Recognise that my own reality need not be particularly special. It is just one reality amongst many. Arbitrarily label my reality as the $\bar{t}$-th reality:

$$\begin{aligned} \mathbf{x}(s, \bar{t}) &= a(s, \bar{t})\hat{\mathbf{a}} + b(s, \bar{t})\hat{\mathbf{b}} \quad \text{or} \\ \mathbf{x}(a, \bar{b}) &= a\hat{\mathbf{a}} + B(a)\hat{\mathbf{b}} \quad \text{for some function } B\text{, or} \\ \mathbf{x}(\bar{a}, b) &= A(b)\hat{\mathbf{a}} + b\hat{\mathbf{b}} \quad \text{for some function } A \end{aligned}$$

7. Traverse my $\bar{t}$-reality infinitesimally from position s^*, recording dial reading changes. Compute a tangent vector, and from it a pointing device as the normal vector

$$\mathbf{n}(s^*, \bar{t}) = \left(\frac{\partial b(s,t)}{\partial s}\hat{\mathbf{a}} - \frac{\partial a(s,t)}{\partial s}\hat{\mathbf{b}} \right)\Big|_{(s^*, \bar{t})}$$

8. Sever the connection between my two observed quantities a and b by allowing each to vary independently:

$$\begin{aligned} \mathbf{x}(s, t) &= a(s, t)\hat{\mathbf{a}} + b(s, t)\hat{\mathbf{b}} \quad \text{or} \\ \mathbf{x}(a, b) &= a\hat{\mathbf{a}} + b\hat{\mathbf{b}} \end{aligned}$$

6 Embedded landscaping

In my one-dimensional $\bar{t}$-labelled reality, whatever s position I choose to fix my s-lever at, I am able to calculate a corresponding normal vector $\mathbf{n}(s, \bar{t})$, as per (17). There are in fact infinitely many such normal vectors, one for each s. I cannot help wondering if there is not some relatively simple geometrical entity "out there" off my own world which is able to explain not only the movement of my dials $\hat{\mathbf{a}}$ and $\hat{\mathbf{b}}$ as a function of my s-lever, but also the apparent connection between the dials via the lever. Are there one or more higher dimensional geometrical entities from which my dial behaviours emerge naturally? If so, then the normal vector must surely play a role because it points off my simple world, and it is only off my world where such an entity can be found.

It is well known[2] that if a two-dimensional surface embedded in three-dimensional space $\mathbb{R}^3$ is specified by some function c, say, then the gradient of that function is orthogonal to the level curve of some contour path of the embedded surface. So if we can find some function c such that one of the level curves uniquely matches the description of my one-dimensional world, then we would have found such a higher dimensional goemetrical entity.

And importantly, because the description of my simple world exactly matches that of the level curve, there is no reason not to consider the existence of the higher dimensional geometry. *Of course, any specification of the geometrical entity must not depend in any way on the specifics of my own world. Otherwise my world would indirectly be attributed some special status amongst many, and that special status would demand an explanation.*

To affirm these ideas, let us consider two concrete examples. Suppose that my hypothetical specific one-dimensional reality, which I have happened to label with $\bar{t}$, is a segment of a circle, as shown in Figures 3 and 6. To be sure, from the perspective in my $\bar{t}$-reality, I don't know—indeed, cannot know—about such objects as circles. All I can do is observe and record the state of my $\bar{t}$ reality, which happens to be (c.f. (4), (3) and (2))

$$\mathbf{x}(s,\bar{t}) \;=\; a(s,\bar{t})\hat{\mathbf{a}} + b(s,\bar{t})\hat{\mathbf{b}} \;=\; (A + \bar{t}\cos s)\,\hat{\mathbf{a}} + (B + \bar{t}\sin s)\,\hat{\mathbf{b}} \tag{19}$$

for some constants A, B and $\bar{t}$. That is, as I "meander" through my world using my s-lever, I observe my dial $\hat{\mathbf{a}}$'s reading varying with s as $a(s,\bar{t}) = A + \bar{t}\cos s$, with A and $\bar{t}$ remaining constant. And similarly for my dial $\hat{\mathbf{b}}$. And because the same constant $\bar{t}$ is needed to record both dial readings, as indicated in (19), I choose to label my reality using that same number. Of course, nothing prevents me from labelling the state of my reality as $\mathbf{x}(s, A, B)$ or $\mathbf{x}(s, \bar{t}, A, B)$, except that $\mathbf{x}(s,\bar{t})$ is simpler.

Furthermore, the presence of $\bar{t}$ in the record of the response of both dials supports the assertion that my reality is not special and may well be one amongst many. This makes applying Step 6 on page 14 easier.

The state of reality (19) admits a vector subspace as an embedded path over $\mathcal{E}^2$ (c.f. (15)) as

$$\left\{\mathbf{x}(s,\bar{t}) = (A + \bar{t}\cos s)\,\hat{\mathbf{a}} + (B + \bar{t}\sin s)\,\hat{\mathbf{b}} \mid s \in \mathbb{R},\ \ \text{some } \bar{t}\right\} \tag{20}$$

for some observed constants A and B. A tangent vector calculated at the lever position s is (c.f. (16))

$$\mathbf{t}(s,\bar{t}) = \frac{\partial \mathbf{x}(s,\bar{t})}{\partial s} = -\bar{t}\sin s\,\hat{\mathbf{a}} + \bar{t}\cos s\,\hat{\mathbf{b}}$$

so that a normal vector at the same position is (c.f. (17))

$$\mathbf{n}(s,\bar{t}) = \bar{t}\cos s\,\hat{\mathbf{a}} + \bar{t}\sin s\,\hat{\mathbf{b}}$$

This normal vector was obtained simply be requiring that $\mathbf{n}\cdot\mathbf{t}$ vanish at the same dial position s. This is Step 7 complete. Next, in fulfilment of Step 8, I sever the connection between my two dials by allowing each to vary independently. I do this by contemplating a world in which $\bar{t}$ may take on a range of values, just like s (c.f. (14)):

$$\left\{\mathbf{x}(s,t) = (A + t\cos s)\,\hat{\mathbf{a}} + (B + t\sin s)\,\hat{\mathbf{b}} \mid s,t \in \mathbb{R}\right\}$$

My own world (20) is obviously just a special case of this. To think of my world as a level curve of some contour path of some embedded surface, I write (19) as

$$\mathbf{x}(s,\bar{t}) \;=\; a(s,\bar{t})\hat{\mathbf{a}} + b(s,\bar{t})\hat{\mathbf{b}} \;=\; (A + \bar{t}\cos s)\,\hat{\mathbf{a}} + (B + \bar{t}\sin s)\,\hat{\mathbf{b}} + 0\hat{\mathbf{c}} \tag{21}$$

which is the level curve of the c-contour path

$$\mathbf{x}(s,\bar{t}) \;=\; a(s,\bar{t})\hat{\mathbf{a}} + b(s,\bar{t})\hat{\mathbf{b}} \;=\; (A + \bar{t}\cos s)\,\hat{\mathbf{a}} + (B + \bar{t}\sin s)\,\hat{\mathbf{b}} + c\hat{\mathbf{c}} \tag{22}$$

for some constant value c. We therefore seek a two-dimensional surface embedded in three-dimensional space for which the surface's c-contour path is (22). We shall now consider two such surfaces: the *embedded sphere* and the *embedded hump*.

Embedded sphere. Using notation consistent with this text, a sphere embedded in $\mathbb{R}^3$ of radius r and centred at the position $A\hat{\mathbf{a}} + B\hat{\mathbf{b}} + 0\hat{\mathbf{c}}$ is the set

$$\mathcal{O}(r) = \{(a,b,c) \mid (a-A)^2 + (b-B)^2 + c^2 = r^2, \quad \text{some } A, B \in \mathbb{R}\}$$

A geometric representation of $\mathcal{O}$ is the set of positions

$$O(r) = \{\mathbf{x}(a,b) = a\hat{\mathbf{a}} + b\hat{\mathbf{b}} + c(a,b)\hat{\mathbf{c}} \mid c^2 = r^2 - (a-A)^2 - (b-B)^2, \quad \text{some } A, B \in \mathbb{R}\}$$

Here $\mathbf{x}(a,b)$ is some position located on $\mathcal{O}$. $\mathcal{O}$ is two-dimensional because there are only two degrees of freedom, a and b. It is embedded in $\mathbb{R}^3$ because O is spanned by the orthonormal vector basis $\{\hat{\mathbf{a}}, \hat{\mathbf{b}}, \hat{\mathbf{c}}\}$.

To determine if (22) is some contour path of O, we must evaluate

$$\begin{aligned} \mathbf{x}(a,b) &= \mathbf{x}(a(s,\bar{t}), b(s,\bar{t})) \\ &= \mathbf{x}(A + \bar{t}\cos s, B + \bar{t}\sin s) \\ &= (A + \bar{t}\cos s)\hat{\mathbf{a}} \,+\, (B + \bar{t}\sin s)\hat{\mathbf{b}} \,\pm\, \sqrt{r^2 - \bar{t}^2}\hat{\mathbf{c}} \end{aligned} \tag{23}$$

Since $\sqrt{r^2 - \bar{t}^2}$ is constant with respect to variation in s, (23) is of the form (22). So my one-dimensional $\bar{t}$-reality (19) is exactly the level curve of the $\sqrt{r^2 - \bar{t}^2}$-contour path of $\mathcal{O}$. And the value of $\bar{t}$ which I happened to discover through simple observation turns out to be a selector for the particular contour path with which my $\bar{t}$-reality coincides, namely, the $\sqrt{r^2 - \bar{t}^2}$-th one!

Next, the gradient of the $c(a,b)$ function in (6), which specified $\mathcal{O}$, is

$$\nabla_{(a,b)} c = \frac{\partial c}{\partial a}\hat{\mathbf{a}} + \frac{\partial c}{\partial b}\hat{\mathbf{b}} = \frac{1}{c}\Big((a-A)\hat{\mathbf{a}} + (b-B)\hat{\mathbf{b}}\Big)$$

And evaluated at my position (22),

$$\begin{aligned} \nabla_{(a,b)} c\Big|_{(A+\bar{t}\cos s, B+\bar{t}\sin s)} &= \pm\frac{1}{\sqrt{r^2 - \bar{t}^2}}\Big(\bar{t}\cos s\,\hat{\mathbf{a}} + \bar{t}\sin s\,\hat{\mathbf{b}}\Big) \\ &= \pm\frac{1}{\sqrt{r^2 - \bar{t}^2}}\,\mathbf{n}(s,\bar{t}) \end{aligned}$$

which aligns with my normal vector $\mathbf{n}$.

We may therefore happily assert the possibility that my world is nothing more than the level curve of the $\sqrt{r^2 - \bar{t}^2}$-th contour path on the sphere embedded in a three-dimensional universe. The existence of a relatively simple geometrical object in $\mathbb{R}^3$, namely the $\mathcal{O}$ sphere, is a plausible explanation for the surprising and unexpected observed character of my one-dimensional $\bar{t}$-reality.

Embedded hump. Again, using notation consistent with this text, a hump embedded in $\mathbb{R}^3$ of height h and centred at the position $A\hat{\mathbf{a}} + B\hat{\mathbf{b}} + 0\hat{\mathbf{c}}$ is the set

$$\mathcal{H}(h) = \{(a,b,c) \mid c = \frac{h}{(A-a)^2 + (B-b)^2 + 1}, \quad \text{some } A, B \in \mathbb{R}\} \tag{24}$$

A geometric representation of $\mathcal{H}$ is

$$H(h) = \{\mathbf{x}(a,b) = a\hat{\mathbf{a}} + b\hat{\mathbf{b}} + c(a,b)\hat{\mathbf{c}} \mid c = \frac{h}{(A-a)^2 + (B-b)^2 + 1}, \quad \text{some } A, B \in \mathbb{R}\}$$

My $\bar{t}$-reality is a contour path of $H(h)$ because

$$\begin{aligned} \mathbf{x}(a(s,\bar{t}), b(s,\bar{t})) &= \mathbf{x}(A + \bar{t}\cos s, B + \bar{t}\sin s) \\ &= (A + \bar{t}\cos s)\hat{\mathbf{a}} \,+\, (B + \bar{t}\sin s)\hat{\mathbf{b}} \,+\, \frac{h}{\bar{t}^2 + 1}\hat{\mathbf{c}} \end{aligned} \tag{25}$$

And since $h/(\bar{t}^2+1)$ is constant with respect to variation in s, (25) is of the form (22). So in the case of the hump $\mathcal{H}$, my $\bar{t}$-reality (19) is exactly the level curve of the $h/(\bar{t}^2+1)$-contour path of $H(h)$. And again, my choice of $\bar{t}$ as a label for my reality turns out to select a specific contour path of $\mathcal{H}$.

The gradient of $c(a,b)$ for $\mathcal{H}$ is

$$\nabla_{(a,b)}c = \frac{2c^2}{h}\left((A-a)\hat{\mathbf{a}} + (B-b)\hat{\mathbf{b}}\right)$$

Evaluated at my position (22)

$$\begin{aligned}\nabla_{(a,b)}c\Big|_{(A+\bar{t}\cos s, B+\bar{t}\sin s)} &= -\frac{2h}{(\bar{t}^2+1)^2}\left(\bar{t}\cos s\,\hat{\mathbf{a}} + \bar{t}\sin s\,\hat{\mathbf{b}}\right)\\ &= -\frac{2h}{(\bar{t}^2+1)^2}\,\mathbf{n}(s,\bar{t})\end{aligned}$$

which obviously aligns with $\mathbf{n}$.

So my one-dimensional $\bar{t}$-reality is simply the level curve of the $h/(\bar{t}^2+1)$-contour path on the hump embedded in a three-dimensional universe. And the surprising and unexpected observed character of my reality is attributable to the existence of $\mathcal{H}$ embedded in $\mathbb{R}^3$.

The geometric representation of $\mathcal{H}$ is shown in Figure 8. In fact, the hypothetical family of t-realities shown in Figure 3, and the family of s-realities in Figure 5, were all drawn computationally using $\mathcal{H}$ as the assumed geometrical object.

If I can find an embedded surface for which my one-dimensional world is a contour path, or more specifically, a level curve, then I can calculate a normal direction which "points into" the two-dimensional space off my one-dimensional world. There are in fact infinitely many such surfaces, and I shall call them *embedded difference hypersurfaces*.

Placing myself once again into the one-dimensional dark room consisting of two dials whose readings are mysteriously connected, I apply Step 5 above as $\mathbf{x}(a) = a\hat{\mathbf{a}} + B(a)\hat{\mathbf{b}}$. To recap, the surprising functional dependance of the second dial's reading B on a is what I have observed about my reality. From my one-dimensional perspective, the set

$$\boxed{\mathcal{D}^{1+2}(B,M) = \{(a,b,c) \mid c = c(a,b) = M(a,b)(b - B(a))\}} \tag{26}$$

for some function $M(a,b)$, is a two-dimensional hypersurface embedded in $\mathbb{R}^3$. Any point (a,b,c) in $\mathcal{D}^{1+2}$ may be represented geometrically by the position

$$\mathbf{d}(a,b) \;=\; a\hat{\mathbf{a}} + b\hat{\mathbf{b}} + c(a,b)\hat{\mathbf{c}} \;=\; a\hat{\mathbf{a}} + b\hat{\mathbf{b}} + M(a,b)(b-B(a))\hat{\mathbf{c}}$$

And since

$$\begin{aligned}\mathbf{d}(a,B(a)) &= a\hat{\mathbf{a}} + B(a)\hat{\mathbf{b}} + M(a,b)\left(B(a)-B(a)\right)\hat{\mathbf{c}}\\ &= a\hat{\mathbf{a}} + B(a)\hat{\mathbf{b}} + 0\hat{\mathbf{c}} \quad (\text{0-level curve of } \mathcal{D}^{1+2})\\ &= \mathbf{x}(a)\end{aligned}$$

the $\mathcal{D}^{1+2}$ set is an embedded difference hypersurface for which my one-dimensional reality path $\mathbf{x}(a)$ is identically the 0-level curve of the $\mathcal{D}^{1+2}$ surface. I may therefore use the surface's defining function in (26) to construct a "pointing device" $\mathbf{n}(a)$ as being the function's gradient at the position $\mathbf{d}(a,B(a))$:

$$\begin{aligned}\mathbf{n}(a) &= \nabla_{(a,b)}c(a,B(a))\\ &= \frac{\partial c(a,B(a))}{\partial a}\hat{\mathbf{a}} + \frac{\partial c(a,B(a))}{\partial b}\hat{\mathbf{b}}\end{aligned}$$

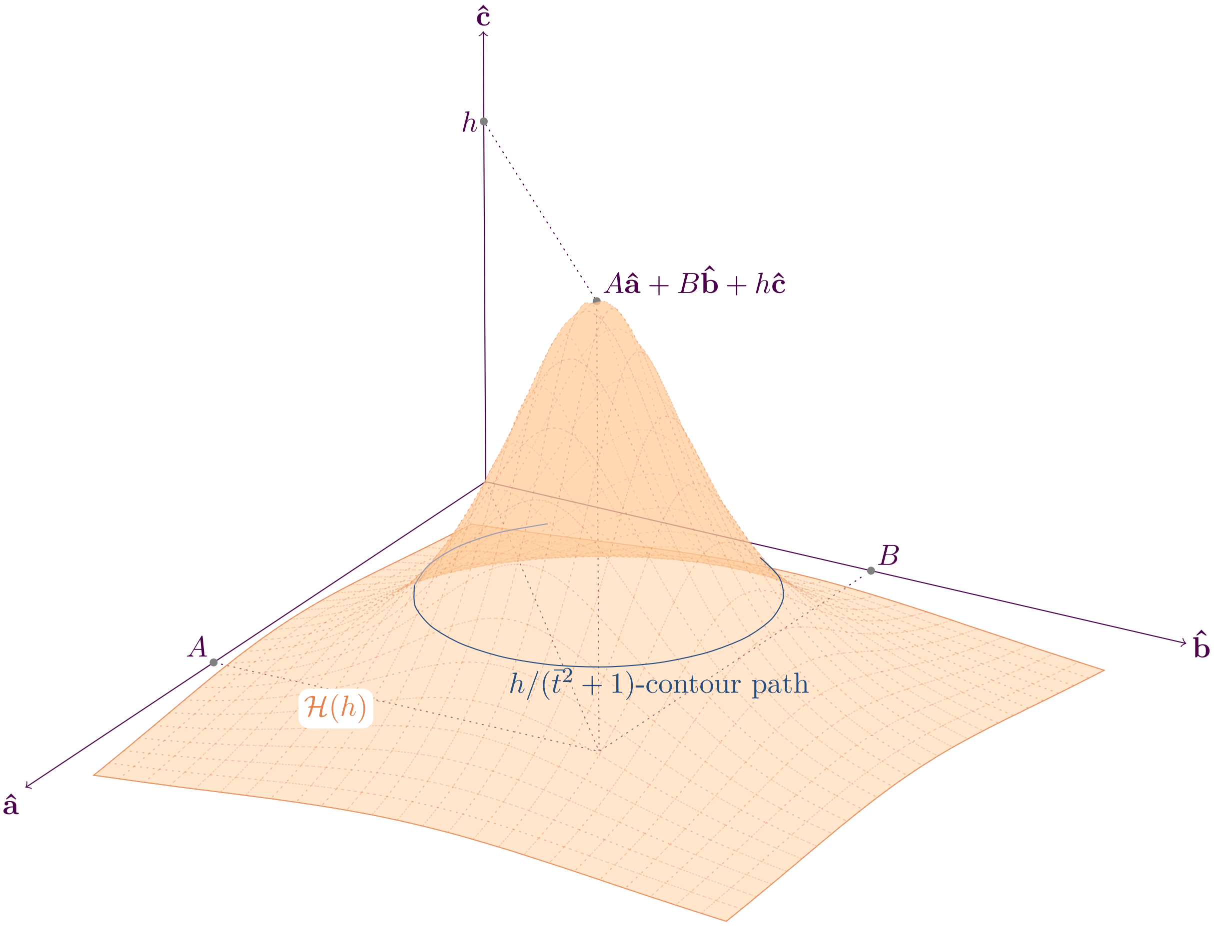

Figure 8: A geometrical representation of the embedded hump $\mathcal{H}(h)$ specified by (24). Also shown is the $h/(\bar{t}^2 + 1)$-contour path of $H(h)$. The level curve corresponding to that contour path identifies with my observed $\bar{t}$-reality as recorded in (21) and labelled as such in Figure 3. The embedded hump and the contour path were calculated and drawn computationally. (Refer to the annotated listing of the C source file, `humpfigure.c`, in the appendix on page 35.)

Now

$$\begin{aligned}
\frac{\partial c(a, B(a))}{\partial a} &= -\frac{\mathrm{d}B(a)}{\mathrm{d}a} M(a, B(a)) \,+\, (B(a) - B(a))\, \frac{\partial M(a, B(a))}{\partial a} \\
&= -\frac{\mathrm{d}B(a)}{\mathrm{d}a} M(a, B(a))
\end{aligned}$$

And

$$\begin{aligned}
\frac{\partial c(a, B(a))}{\partial b} &= M(a, B(a)) \,+\, (B(a) - B(a))\, \frac{\partial M(a, B(a))}{\partial b} \\
&= M(a, B(a))
\end{aligned}$$

A pointing device in my one-dimensional world is therefore the gradient vector

$$\boxed{\mathbf{n}(a) = M(a, B(a)) \left[-\frac{\mathrm{d}B(a)}{\mathrm{d}a} \hat{\mathbf{a}} + \hat{\mathbf{b}} \right]} \tag{27}$$

The device points off my world because it is orthogonal to it. That is, the gradient vector $\mathbf{n}$ evaluated at my position a is orthogonal to any tangent vector $\mathrm{d}\mathbf{x}(a)/\mathrm{d}a$ evaluated at the same point. Using (5)

$$
\begin{aligned}
\mathbf{n}(a) \cdot \frac{\mathrm{d}\mathbf{x}(a)}{\mathrm{d}a} &= M(a, B(a)) \left[-\frac{\mathrm{d}B(a)}{\mathrm{d}a}\hat{\mathbf{a}} + \hat{\mathbf{b}} \right] \cdot \left[\hat{\mathbf{a}} + \frac{\mathrm{d}B(a)}{\mathrm{d}a}\hat{\mathbf{b}} \right] \\
&= M(a, B(a)) \left[-\frac{\mathrm{d}B(a)}{\mathrm{d}a}\hat{\mathbf{a}} \cdot \hat{\mathbf{a}} + \frac{\mathrm{d}B(a)}{\mathrm{d}a}\hat{\mathbf{b}} \cdot \hat{\mathbf{b}} \right] \\
&= 0
\end{aligned} \tag{28}
$$

7 Acknowledgments

As always Mels, thanks for being such a close friend and supportive partner, and for showing an interest in this work. I'm so glad that you reside in my embedded 0-contour volume!

8 Appendix—Computed drawing with LaTeX, TikZ, pkTikZ and PKREALVECTOR

In this section I demonstrate the combined use of LaTeX, TikZ, my pkTikZ LaTeX package[?], and my C object class called `PKREALVECTOR`[3] 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 TikZ. TikZ 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[4] includes calls to perform the rotational coordinate transformations needed to render on paper a two-dimensional projection of a three-dimensional landscape.

To typeset a figure, the LaTeX source file for this document `\input{}`'s another external LaTeX source file. In the case of Figure 8 on page 18, the file was named `humpfigure.tex`. The file contains the TikZ source code instructions to typeset the figure. The file was generated dynamically as the output of the execution of the `humpfigure.run` program, which in turn was created by compiling the C code located in the `humpfigure.c` file. See below for a listing of the `humpfigure.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.

8.1 Typesetting the figures with LaTeX, TikZ and pkTikZ

To incorporate TikZ'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}
```

TikZ was customised "globally" for all figures in the document using the following lines of LaTeX code:

```
\newcommand*\myWordMeaning[2]{\mbox{\emph{#1}---}#2}

\newcommand*\undr[1]{_{#1}}
\newcommand*\ud{\text d}
\newcommand*\deriv[2]{\frac{\ud #1}{\ud #2}}
\newcommand*\derivB[2]{\ud #1/\ud #2}
\newcommand*\parDeriv[2]{\frac{\partial #1}{\partial #2}}
\newcommand*\evalAt[2]{\left.#1\right|_{#2}}%
%\newcommand*\evalFromTo[3]{\left.#1\right|_{#2}^{#3}}%

\newcommand*\one{\pktikzBasisVector{1}}
\newcommand*\two{\pktikzBasisVector{2}}
\newcommand*\three{\pktikzBasisVector{3}}

\newcommand*\vecx{\pktikzVector{x}}
\newcommand*\vecy{\pktikzVector{y}}
\newcommand*\vecc{\pktikzVector{c}}
```

[4] Application Programming Interface

```
\newcommand*\vecd{\pktikzVector{d}}
\newcommand*\vecg{\pktikzVector{g}}
\newcommand*\vecp{\pktikzVector{p}}
\newcommand*\vecn{\pktikzVector{n}}
\newcommand*\vect{\pktikzVector{t}}

\newcommand*\ahat{\pktikzUnitVector{a}}
\newcommand*\bhat{\pktikzUnitVector{b}}
\newcommand*\chat{\pktikzUnitVector{c}}

\newcommand*\tbar{\bar{t}}
\newcommand*\abar{\bar{a}}
\newcommand*\bbar{\bar{b}}
\newcommand*\sstar{s^*}
\newcommand*\astar{a^*}
\newcommand*\bstar{b^*}

\newcommand*\stbar{(s,\tbar)}

\newcommand*\sphereSet{\mathcal{O}}
\newcommand*\humpSet{\mathcal{H}}
\newcommand*\DcurveSet{\mathcal{D}}
\newcommand*\euclSet{\mathcal{E}}
\newcommand*\Etwo{\euclSet^2}
\newcommand*\diffSet{\mathcal{D}}
\newcommand*\diffSetOneTwo{\diffSet^{1+2}}

\newcommand*\vecComp[2]{\pktikzVector{#1}\cdot\pktikzBasisVector{#2}}

\newcommand*\realSet{\mathbb{R}}
\newcommand*\Rthree{\realSet^3}
\newcommand*\complexSet{\mathbb{C}}

%
% Definitions needed for the lever/dial diagrams begin.
%
\newcommand*\leverRadius{2.5}
\newcommand*\leverEndAngle{180}%{120}
\newcommand*\leverBigTick{0.5}
\newcommand*\leverSmallTick{0.25}
\newcommand*\leverHandleSize{0.4cm}
\newcommand*\dialRadius{\leverRadius}
\newcommand*\dialBigTick{0.4}
\newcommand*\dialSmallTick{0.2}
%
\newcommand*\typesetLever[5]{%
   %
   % Arguments:
   %
   %    1 : Lever internal name
   %    2 : Lever positioning clause(s)
   %    3 : Lever handle angle multiple
   %    4 : Lever position label
   %    5 : Lever name for label
   %
   % Lever arm.
   %
   \node(#1Hub)[#2,#1arm,circle,minimum size=\leverHubSize]{};
   \path (#1Hub)
         +(6*#3:\leverRadius+2*\leverBigTick) node(#1Nob)
                                                   [#1arm,fill,circle,
                                                    minimum size=\leverHandleSize]{};
```

```
   \draw[#1arm] (#1Hub) -- (#1Nob);
   %
   % Lever meter.
   %
   \path (#1Hub) +(0:\leverRadius) node(#1Right){};
   \draw[#1cover,myshadowed]
   %\draw[#1cover,pktikzshadowed]
      (#1Right)
      arc[start angle=0,end angle=\leverEndAngle,radius=\leverRadius] node(#1End){};
   \draw[#1meter]
      foreach \angle in { 0, 6, ..., \leverEndAngle } {
            (#1Hub)
            +(\angle:\leverRadius-\leverSmallTick)
         -- +(\angle:\leverRadius) };
   \draw[#1meter]
      foreach \angle in { 0, 30, ..., \leverEndAngle } {
            (#1Hub)
            +(\angle:\leverRadius-\leverBigTick)
         -- +(\angle:\leverRadius) };
   %
   % This path is merely to improve the appearance of the above
   % "myshadowed" paths.
   %
   \draw[#1meter]
      (#1Right) arc[start angle=0,end angle=\leverEndAngle,radius=\leverRadius];
   \path (#1Hub) +(90:\leverRadius) node(#1Top){};
   \path (#1Hub)
         +(6*#3:\leverRadius-\leverSmallTick) node[fill=white,rounded corners]{$#4$};
   %
   % Lever label.
   %
   \node(#1Label)[below=\leverBigTick] at (#1Hub) {$#5$-Lever};
   %
   % Full lever.
   %
   \node(#1)
        [fit=(#1Right) (#1Nob) (#1Hub) (#1Top) (#1End) (#1Label)]
        {};}
\newcommand*\typesetSlever{\typesetLever{slever}}
\newcommand*\typesetTlever{\typesetLever{tlever}}
%
\newcommand*\typesetDial[5]{%
   %
   % Arguments:
   %
   %    1 : Dial internal name
   %    2 : Dial positioning clause(s)
   %    3 : Dial needle angle multiple
   %    4 : Dial position label
   %    5 : Dial name for label
   %
   \node(#1Left)[#2]{};
   \path (#1Left) ++(0:\dialRadius) coordinate(#1Hub);
   \draw[dialcover]
         (#1Hub)
      -- ++(0:\dialRadius)
      arc[start angle=0,end angle=180,radius=\dialRadius]
      -- cycle;
   \draw[dialmeter]
      (#1Hub)
      foreach \angle in { 15, 45, ..., 165 } {
            +(\angle:\dialRadius-0.2-\dialBigTick)
```

```
      -- +(\angle:\dialRadius-0.2) };
  \draw[dialmeter]
    (#1Hub)
    foreach \angle in { 15, 20, ..., 165 } {
        +(\angle:\dialRadius-0.2-\dialSmallTick)
      -- +(\angle:\dialRadius-0.2) };
  %
  % The '#1Right' node is simply to capture a rightmost
  % location for use below.
  %
  \path (#1Hub) +(0:\dialRadius) node(#1Right){};
  %
  % Dial needle.
  %
  \draw[dialneedle,myshadowed] (#1Hub) circle[radius=\dialHubSize];
  \draw[dialneedle] (#1Hub) -- +(15+5*#3:\dialRadius-0.2-\dialBigTick);
  \draw[fill,roomcolor] (#1Hub) circle[radius=\dialHubSize-\dialNeedleThick];
  \path (#1Hub)
      +(15+5*#3:\dialRadius-\dialBigTick) node[fill=white,rounded corners]{$#4$};
  %
  % Dial label.
  %
  \node(#1Label)[below right=\leverBigTick] at (#1Hub) {Dial $#5$};
  %
  % Full dial.
  %
  \node(#1)[fit=(#1Left) (#1Right) (#1Label)]{};}
\newcommand*\typesetAdial{\typesetDial{adial}}
\newcommand*\typesetBdial{\typesetDial{bdial}}
%
\newcommand*\typesetLeverDialConnection[4]{%
  \draw[leverdialconnection,#1color]
      (#1Hub)
    to[out=#4,in=220] node[black,sloped,above]{$#3$}
      (#2Hub);}
\newcommand*\typesetDarkRoom[1]{%
  \begin{scope}[on background layer]
    \node(room)[darkroom,fit=#1]{};
  \end{scope}}
%
% Definitions needed for the lever/dial diagrams end.
%

\definecolor{contourpathcolor}{rgb}{0.15,0.3,0.5}
\definecolor{gradientpathcolor}{rgb}{0.7,0.3,0.2}
%
\colorlet{roomwallcolor}{darkgray}
%\colorlet{roomcolor}{lightgray}
\colorlet{roomcolor}{black!5}
\colorlet{slevercolor}{contourpathcolor}
\colorlet{tlevercolor}{gradientpathcolor}
\colorlet{dialcolor}{darkgray}

\newcommand*\leverArmThick{1.2pt}
\newcommand*\leverHubSize{0.3cm}
\newcommand*\dialNeedleThick{\leverArmThick}
\newcommand*\dialHubSize{0.5*\leverHubSize}

\tikzset{%inner sep=2pt,
      %labelpointer/.style={->,
      %              pktikzdimension,
      %              text=black,
```

```
        %                       pktikzshadowed},
        %basisaxis/.style={thin,basiscolor},
        %occludedsurfacepath/.style={surfacepath,occludedsurfacepathcolor},
        contourpath/.style={smooth,contourpathcolor},
        gradientpath/.style={smooth,gradientpathcolor},
        %
        % Styles for the lever and dial begin.
        %
        darkroom/.style={draw=roomwallcolor,
                         fill=roomcolor,
                         rounded corners,
                         inner sep=3ex},
        sleverarm/.style={slevercolor,
                          draw,
                          line width=\leverArmThick},
        slevermeter/.style={slevercolor},
        slevercover/.style={slevermeter,thick},
        tleverarm/.style={sleverarm,tlevercolor},
        tlevermeter/.style={slevermeter,tlevercolor},
        tlevercover/.style={slevercover,tlevercolor},
        dialneedle/.style={->,
                           >=latex,
                           dialcolor,
                           draw,
                           fill,
                           line width=\dialNeedleThick},
        %dialmeter/.style={dialcolor,thick},
        dialmeter/.style={dialcolor},
        dialcover/.style={dialmeter,thick},
        leverdialconnection/.style={->,
                                    >=stealth',
                                    shorten <=2pt,
                                    shorten >=\dialHubSize+3pt}
        %
        % Styles for the lever and dial end.
        %
}
```

The file `humpfigure.tex`, for example, contains Ti*k*Z code for Figure 8 on page 18. The `humpfigure.tex` file was incorporated into the body of the text with an `\input{}` LaTeX command, as follows:

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

8.2 Source code listings and the PKREALVECTOR C object class

The content of the various C source code files, which were used to create the Ti*k*Z code in the corresponding "`.tex`" file, have all been primed to be typeset using the PKTECHDOC "literate programming" LaTeX package.[4] PKTECHDOC makes it possible to closely juxtapose LaTeX code and non-LaTeX code both for typesetting and for compilation outside of LaTeX.

8.2.1 The `t-realities.c` file

A listing of the `t-realities.c` file follows:

```
#include <pkfeatures.h>

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <float.h>

#include <pkmemdebug.h>
#include <pkerror.h>
#include <pktypes.h>
#include <pkstring.h>
#include <pkmath.h>
#include <pkrealvector.h>

#include "sundry.h"
#include "hump.h"
```

```
const char *LOGFNAME = "/tmp/diagram.log";
```

```
static void _drawPath( const PKREALVECTOR **path, const int M )
{
   int j;

   if ( !path  ||  M < 1 )
      return;
   //printf("%d positions.",M); return;

   puts(   "   \\draw[emphvectorcolor] plot[smooth] coordinates {" );
   for ( j = 0;  j < M;  j++ ) {
      printf( "      (" FLTFMT "," FLTFMT ")%s\n",
              pkRealVectorGetComponent(path[j])[0],
              pkRealVectorGetComponent(path[j])[1],
              ( j < M - 1 ) ? "" : " };" );
   }

   return;
}
```

The `_drawCoordinates()` private function below simply `printf`s a few Ti*k*Z commands for coordinates.

```
static void _drawCoordinates(void)
{
   puts(   "   %\\draw[help lines] (-0.2,-0.2) grid (7.1,5.1);");
   puts(   "   %");
   puts(   "   % Some coordinates.");
   puts(   "   %");
   puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
   return;
}
```

```
static void _drawBasisVectors( const PKREALVECTOR *e1,
                               const PKREALVECTOR *e2 )
{
   if ( !e1  ||  !e2  )
      return;
```

```

   puts(   "   %");
   puts(   "   % Basisvectors.");
   puts(   "   %");
   printf( "   \\draw[pktikzbasisvector,<->]\n"
           "      (" FLTFMT "," FLTFMT ") node[right] {$%s$} --\n"
           "      (origin) -- (" FLTFMT "," FLTFMT ") node[above] {$%s$};\n",
           pkRealVectorGetComponent(e1)[0],
           pkRealVectorGetComponent(e1)[1],
           pkRealVectorGetName(e1),
           pkRealVectorGetComponent(e2)[0],
           pkRealVectorGetComponent(e2)[1],
           pkRealVectorGetName(e2) );

   return;
}
```

```
static void _printfPosition( const char *prefix,
                             const PKREALVECTOR *posn,
                             const char *suffix )
{
   if ( !posn )
      return;

   printf( "%s(" FLTFMT "," FLTFMT ")%s\n",
           strIsNull(prefix) ? "" : prefix,
           pkRealVectorGetComponent(posn)[0],
           pkRealVectorGetComponent(posn)[1],
           strIsNull(suffix) ? "" : suffix );

   return;
}
```

The `_drawContourPath()` private function prints to standard output a set of Ti*k*Z `\draw` commands for drawing the z-contour path represented by the specified `d` array assumed to contain `M` entries.

```
static void _drawContourPath( PKREALVECTOR **d, const int M )
{
   int j;

   if ( !d  ||  M < 1 )
      return;

   puts(   "   %");
   printf( "   %% " FLTFMT "-contour path.\n", pkRealVectorGetComponent(d[0])[2] );
   puts(   "   %");
   //puts(   "   \\draw[contourpath] plot[mark=*,mark size=0.7pt] coordinates {" );
   puts(   "   \\draw[contourpath] plot[] coordinates {" );
   for ( j = 0;  j < M;  j++ )
      _printfPosition( "      ", d[j], ( j < M - 1 ) ? "" : " };" );

   return;
}
```

The `_drawHumpContourPaths()` private function simply calls `_drawContourPath()` for each set of z-contour path positions represented by the array member `dArr[i]`, $i = 0, 1, 2, \ldots, N-1$.

```
static void _drawHumpContourPaths( PKREALVECTOR ***dArr,
                                   const int N,
```

```
                                    const int M )
{
   int i;

   if ( !dArr  ||  N < 1  ||  M < 1 )
      return;

   puts(   "   %");
   printf( "   %% %d contour paths with %d positions on each.\n", N, M );
   for ( i = 0; i < N; i++ )
      _drawContourPath( dArr[i], M );

   return;
}

static void _drawXposition( const PKREALVECTOR *x,
                            const char *name,
                            const char *xName,
                            const char *yName )
{
   if ( !x  ||  strIsNull(name)  ||  strIsNull(xName)  ||  strIsNull(yName) )
      return;

   puts(   "   %");
   puts(   "   % A position and its components.");
   puts(   "   %");
   printf( "   \\draw[pktikzdimension]\n"
           "      (" FLTFMT ",0.0) node[below]{$%s$} --\n"
           //"      (" FLTFMT "," FLTFMT ") node[right,black,fill=white,rounded corners]{$%s$}"
           "      (" FLTFMT "," FLTFMT ") node[pktikzlabel,right]{$%s$}"
                                   " coordinate[pktikzpoint] --\n"
           "      (0.0," FLTFMT ") node[left]{$%s$};\n",
           pkRealVectorGetComponent(x)[0],
           xName,
           pkRealVectorGetComponent(x)[0], pkRealVectorGetComponent(x)[1],
           name,
           pkRealVectorGetComponent(x)[1],
           yName );

   return;
}

static void _drawXpositions( PKREALVECTOR ***dArr,
                             const int N,
                             const int M )
{
   const int tbarReality = 2,
             tprimeReality = 5;
   PKREALVECTOR *commonx,
                *p;     /* Some arbitrary position on the 'tbarReality' contour path. */

   if ( !dArr  ||  N < 1  ||  M < 1 )
      return;

   _drawXposition( dArr[tbarReality][M/8],     "\\vecx\\stbar", "a\\stbar", "b\\stbar" );
   _drawXposition( dArr[tprimeReality][3*M/8], "\\vecx(s,t')",  "a(s,t')",  "b(s,t')" );

   commonx = dArr[tbarReality][5*M/8];
   printf( "   \\draw[pktikzdimension]\n"
           //"      (" FLTFMT "," FLTFMT ") node[right,black,fill=white,rounded corners]{$%s$}"
           "      (" FLTFMT "," FLTFMT ") node[pktikzlabel,right]{$%s$}"
```

```
                                              " coordinate[pktikzpoint];\n",
             pkRealVectorGetComponent(commonx)[0], pkRealVectorGetComponent(commonx)[1],
             "\\vecx(\\sstar,\\tbar)" );

    p = dArr[tbarReality][23*M/32];
    printf( "   \\path (" FLTFMT "," FLTFMT ")\n"
             "      node[contourpathcolor,fill=white]{$\\tbar$-reality};\n",
             pkRealVectorGetComponent(p)[0], pkRealVectorGetComponent(p)[1] );

    return;
}
```

Initialise the diagram's primary landscape parameters, and then draw the landscape.

The `_diagram()` private function below specifies the diagram's landscape. It does this 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.

```
static void _diagram(void)
{
    const int N = 20,    /* Number of contour paths. */
              M = 20;    /* Number of positions on each contour path. */
    const PKMATHREAL deltat = 0.02;      /* Parametrisation parameter value */
                                         /* for next position on path. */
    PKREALVECTOR *e1,
                 *e2,
                 ***dArr;        /* Dynamic array of 'N' dynamic arrays of */
                                 /* 'M' contour path positions.  That is, */
                                 /* an array of arrays of PKREALVECTORs. */

```

Prepare the objects in the landscape. Here we specify the two-dimensional landscape. Begin with the $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}\}$ vector basis.

```
    e1 = pkRealVectorAlloc1( "\\ahat", 3, basisVecLen, 0.0, 0.0 );
    e2 = pkRealVectorAlloc1( "\\bhat", 3, 0.0, basisVecLen, 0.0 );
    pkRealVectorScale(e1,1.2);
    pkRealVectorScale(e2,1.1);
```

Allocate and initialise as a dynamic array the set of N contour paths over $\mathcal{H}$ with each contour path having M positions.

```
    dArr = allocHumpContourArr( N, M, deltat );
```

Prepare the Ti*k*Z commands for typesetting the set of contour paths.

```
    puts(   "\\begin{PkTikzpicture}[scale=1.6]");
    _drawCoordinates();
    _drawBasisVectors(e1,e2);
    _drawHumpContourPaths( dArr, N, M );
    _drawXpositions( dArr, N, M );
    puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
    pkRealVectorFree1(e1);
    pkRealVectorFree1(e2);
    freeHumpContourArr( dArr, N, M );

    return;
}

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

8.2.2 The `s-realities.c` file

A listing of the `s-realities.c` file follows:

```
#include <pkfeatures.h>

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <float.h>

#include <pkmemdebug.h>
#include <pkerror.h>
#include <pktypes.h>
#include <pkstring.h>
#include <pkmath.h>
#include <pkrealvector.h>

#include "sundry.h"
#include "hump.h"
```

```
const char *LOGFNAME = "/tmp/diagram.log";
```

```
static void _drawPath( const PKREALVECTOR **path, const int M )
{
   int j;

   if ( !path  ||  M < 1 )
      return;
   //printf("%d positions.",M); return;

   puts(   "   \\draw[pktikzemphvectorcolor] plot[smooth] coordinates {" );
   for ( j = 0;  j < M;  j++ ) {
      printf( "      (" FLTFMT "," FLTFMT ")%s\n",
              pkRealVectorGetComponent(path[j])[0],
              pkRealVectorGetComponent(path[j])[1],
              ( j < M - 1 ) ? "" : " };" );
   }

   return;
}
```

The `_drawCoordinates()` private function below simply `printf`s a few Ti*k*Z commands for coordinates.

```
static void _drawCoordinates(void)
{
   puts(   "   %\\draw[help lines] (-0.2,-0.2) grid (7.1,5.1);");
   puts(   "   %");
   puts(   "   % Some coordinates.");
   puts(   "   %");
   puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
   return;
}
```

```
static void _drawBasisVectors( const PKREALVECTOR *e1,
                               const PKREALVECTOR *e2 )
{
   if ( !e1  ||  !e2  )
      return;

   puts(   "   %");
   puts(   "   % Basisvectors.");
   puts(   "   %");
   printf( "   \\draw[pktikzbasisvector,<->]\n"
           "      (" FLTFMT "," FLTFMT ") node[right] {$%s$} --\n"
           "      (origin) -- (" FLTFMT "," FLTFMT ") node[above] {$%s$};\n",
           pkRealVectorGetComponent(e1)[0],
           pkRealVectorGetComponent(e1)[1],
           pkRealVectorGetName(e1),
           pkRealVectorGetComponent(e2)[0],
           pkRealVectorGetComponent(e2)[1],
           pkRealVectorGetName(e2) );

   return;
}
```

```
static void _printfPosition( const char *prefix,
                             const PKREALVECTOR *posn,
                             const char *suffix )
{
   if ( !posn )
      return;

   printf( "%s(" FLTFMT "," FLTFMT ")%s\n",
           strIsNull(prefix) ? "" : prefix,
           pkRealVectorGetComponent(posn)[0],
           pkRealVectorGetComponent(posn)[1],
           strIsNull(suffix) ? "" : suffix );

   return;
}
```

The _drawGradientPath() private function prints to standard output a set of Ti*k*Z \draw commands for drawing the gradient path represented by the specified d array assumed to contain M entries.

```
static void _drawGradientPath( PKREALVECTOR **d, const int M )
{
   int j;

   if ( !d  ||  M < 1 )
      return;

   puts(   "   %");
   printf( "   %% (" FLTFMT "," FLTFMT ")-gradient path.\n",
           pkRealVectorGetComponent(d[0])[0],
           pkRealVectorGetComponent(d[0])[1] );
   puts(   "   %");
   //puts(   "   \\draw[gradientpath] plot[mark=*,mark size=0.7pt] coordinates {" );
   puts(   "   \\draw[gradientpath] plot[] coordinates {" );
   for ( j = 0;  j < M - 1;  j++ )
      _printfPosition( "      ", d[j], ( j < M - 2 ) ? "" : " };" );

   return;
}
```

The _drawHumpGradientPaths() private function simply calls _drawGradientPath() for each set of gradient path positions represented by the array member dArr[i], $i = 0, 1, 2, \ldots, N - 1$.

```
static void _drawHumpGradientPaths( PKREALVECTOR ***dArr,
                                    const int N,
                                    const int M )
{
   int i;

   if ( !dArr  ||  N < 1  ||  M < 1 )
      return;

   puts(   "   %");
   printf( "   %% %d gradient paths with %d positions on each.\n", N, M );
   for ( i = 0; i < N; i++ )
      _drawGradientPath( dArr[i], M );

   return;
}
```

```
static void _drawXposition( const PKREALVECTOR *x,
                            const char *name,
                            const char *xName,
                            const char *yName )
{
   if ( !x  ||  strIsNull(name)  ||  strIsNull(xName)  ||  strIsNull(yName) )
      return;

   puts(   "   %");
   puts(   "   % A position and its components.");
   puts(   "   %");
   printf( "   \\draw[pktikzdimension]\n"
           "      (" FLTFMT ",0.0) node[below]{$%s$} --\n"
           "      (" FLTFMT "," FLTFMT ") node[pktikzlabel,right]{$%s$}"
                                      " coordinate[pktikzpoint] --\n"
           "      (0.0," FLTFMT ") node[left]{$%s$};\n",
           pkRealVectorGetComponent(x)[0],
           xName,
           pkRealVectorGetComponent(x)[0], pkRealVectorGetComponent(x)[1],
           name,
           pkRealVectorGetComponent(x)[1],
           yName );

   return;
}
```

```
static void _drawXpositions( PKREALVECTOR ***dArr,
                             const int N,
                             const int M )
{
   const int sstarReality = 12,
             sprimeReality = 18;
   PKREALVECTOR *commonx,
                *p;     /* An arbitrary position on the ’sstarReality’ gradient path. */

   if ( !dArr  ||  N < 1  ||  M < 1 )
      return;

   _drawXposition( dArr[sstarReality][M/5],
                   "\\vecx(\\sstar,t)",
                   "a(\\sstar,t)",
```

```
                   "b(\\sstar,t)" );
    _drawXposition( dArr[sprimeReality][M/5],
                   "\\vecx(s',t)",
                   "a(s',t)",
                   "b(s',t)" );

    commonx = dArr[sstarReality][5*M/12];
    printf( "   \\draw[pktikzdimension]\n"
            "      (" FLTFMT "," FLTFMT ") node[pktikzlabel,right]{$%s$}"
                                       " coordinate[pktikzpoint];\n",
            pkRealVectorGetComponent(commonx)[0], pkRealVectorGetComponent(commonx)[1],
            "\\vecx(\\sstar,\\tbar)" );

    //p = dArr[sstarReality][0];
    //printf( "   \\path (" FLTFMT "," FLTFMT ")\n"
    //        "      node[above right,gradientpathcolor]{$\\sstar$-reality};\n",
    //        pkRealVectorGetComponent(p)[0], pkRealVectorGetComponent(p)[1] );
    p = dArr[sstarReality][M-2];
    printf( "   \\path (" FLTFMT "," FLTFMT ")\n"
            "       -- node[gradientpathcolor,fill=white,sloped]{$\\sstar$-reality}\n"
            "          (" FLTFMT "," FLTFMT ");\n",
            pkRealVectorGetComponent(p)[0], pkRealVectorGetComponent(p)[1],
            pkRealVectorGetComponent(commonx)[0], pkRealVectorGetComponent(commonx)[1] );

    return;
}
```

Initialise the diagram's primary landscape parameters, and then draw the landscape.

The `_diagram()` private function below specifies the diagram's landscape. It does this 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.

```
static void _diagram(void)
{
    const int N = 20,    /* Number of gradient paths. */
              M = 20;    /* Number of positions on each gradient path. */
    const PKMATHREAL deltaGamma = 0.2;   /* Parametrisation parameter value */
                                         /* for next position on path. */
    PKREALVECTOR *e1,
                 *e2,
                 ***gArr;        /* Dynamic array of 'N' dynamic arrays of */
                                 /* 'M' gradient path positions.  That is, */
                                 /* an array of arrays of PKREALVECTORs. */

```

Prepare the objects in the landscape. Here we specify the two-dimensional landscape. Begin with the $\{\hat{\mathbf{1}}, \hat{\mathbf{2}}\}$ vector basis.

```
    e1 = pkRealVectorAlloc1( "\\ahat", 3, basisVecLen, 0.0, 0.0 );
    e2 = pkRealVectorAlloc1( "\\bhat", 3, 0.0, basisVecLen, 0.0 );
    pkRealVectorScale(e1,1.2);
    pkRealVectorScale(e2,1.1);
```

Allocate and initialise as a dynamic array the set of N gradient paths over $\mathcal{H}$ with each gradient path having M positions.

```
    gArr = allocHumpGradientArr( N, M, humpX / 5.0, humpY / 5.0 );
```

Prepare the Ti*k*Z commands for typesetting the set of gradient paths.

```
    puts(   "\\begin{PkTikzpicture}[scale=1.6]");
    _drawCoordinates();
    _drawBasisVectors(e1,e2);
    _drawHumpGradientPaths( gArr, N, M );
    _drawXpositions( gArr, N, M );
    puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
    pkRealVectorFree1(e1);
    pkRealVectorFree1(e2);
    freeHumpGradientArr( gArr, N, M );

    return;
}
```

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

8.2.3 The `humpfigure.c` file

A listing of the `humpfigure.c` file follows:

```
#include <pkfeatures.h>

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <float.h>

#include <pkmemdebug.h>
#include <pkerror.h>
#include <pktypes.h>
#include <pkstring.h>
#include <pkmath.h>
#include <pkrealvector.h>

#include "sundry.h"
#include "hump.h"
```

```
const char *LOGFNAME = "/tmp/diagram.log";
```

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 $p\hat{\mathbf{1}} + q\hat{\mathbf{2}} + z\hat{\mathbf{3}}$. Obviously, the z-contour will pass through that position. A rationale for the local parametrisation may be found in [3].

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()`.

```
static PKREALVECTOR *_allocHumpContourPosn0( const char *name,
                                             const PKREALVECTORREAL t,
                                             const PKREALVECTORREAL p,
                                             const PKREALVECTORREAL q,
                                             const PKREALVECTORREAL z )
{
   return( pkRealVectorAlloc1( name, 3,
                               ( 1.0 - 2.0 * t ) * ( p - humpX ) +
                               2.0 * sqrt( t * ( 1.0 - t ) ) * ( q - humpY ) +
                               humpX,
                               ( 1.0 - 2.0 * t ) * ( q - humpY ) -
                               2.0 * sqrt( t * ( 1.0 - t ) ) * ( p - humpX ) +
                               humpY,
                               z ) );
}
```

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

```
static void _freeHumpContourPosn0( PKREALVECTOR *d )
{
   if (d)
      pkRealVectorFree1(d);
   return;
}
```

If the specified `p` is not `NULL`, then the `_allocHumpContourPosn()` private function below simply returns with the result of the call to `_allocHumpContourPosn0()`. Otherwise the function returns `(PKREALVECTOR *)NULL`. The function must be accompanied by a call to `_freeHumpContourPosn()`.

```
static PKREALVECTOR *_allocHumpContourPosn( const char *name,
                                            const PKREALVECTORREAL t,
                                            const PKREALVECTOR *p )
{
   if (!p)
      return( (PKREALVECTOR *)NULL );
   return( _allocHumpContourPosn0( name,
                                   t,
                                   pkRealVectorGetComponent(p)[0],
                                   pkRealVectorGetComponent(p)[1],
                                   pkRealVectorGetComponent(p)[2] ) );
}
```

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

```
static void _freeHumpContourPosn( PKREALVECTOR *d )
{
   _freeHumpContourPosn0(d);
   return;
}
```

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 $\mathcal{D}(\mathbf{p}, \Delta t)$ which is discussed in detail in [5]. The function composes `pkRealVectorAlloc1()`, `_allocHumpContourPosn()`, amongst others.

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()`.

```
static PKREALVECTOR **_allocHumpContourPosnArr( const PKREALVECTOR *p,
                                                const int positions,
                                                const PKMATHREAL deltat )
{
   PKREALVECTOR **d;

   if ( positions < 3 )
      return( (PKREALVECTOR **)NULL );

   d = (PKREALVECTOR **)calloc( positions + 1, sizeof(PKREALVECTOR *) );
   if (d) {

      char *name;
      int i;

      d[0] = pkRealVectorAlloc1( "\\vecd^0", 3,
                                 pkRealVectorGetComponent(p)[0],
                                 pkRealVectorGetComponent(p)[1],
                                 pkRealVectorGetComponent(p)[2] );
      for ( i = 1;  i < positions;  i++ ) {
         name = strAllocPrintf( "\\vecd^{%d}", i );
         d[i] = _allocHumpContourPosn( name, deltat, d[i-1] );
         strFreePrintf(name);
      }

```

```
    }

    return(d);
}
```

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

```
static void _freeHumpContourPosnArr( PKREALVECTOR **d, const int positions )
{
    if (d) {
        int i;
        for ( i = 0;  i < positions;  i++ )
            _freeHumpContourPosn(d[i]);
        free(d);
    }
    return;
}
```

The `_drawCoordinates()` private function below simply `printf()`s a few Ti*k*Z commands for coordinates.

```
static void _drawCoordinates(void)
{
    puts(   "   %\\draw[help lines] (-0.2,-0.2) grid (7.1,5.1);");
    puts(   "   %");
    puts(   "   % Some coordinates.");
    puts(   "   %");
    puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
    return;
}
```

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

```
static void _drawBasisVectors( const PKREALVECTOR *e1,
                               const PKREALVECTOR *e2,
                               const PKREALVECTOR *e3 )
{
    if ( e1  &&  e2  &&  e3 ) {
        puts(   "   %");
        puts(   "   % Basisvectors.");
        puts(   "   %");
        printf( "   \\draw[pktikzbasisvector,<->]\n"
                "       (" FLTFMT "," FLTFMT ") node[below left] {$%s$} --\n"
                "       (origin) -- (" FLTFMT "," FLTFMT ") node[right] {$%s$};\n",
                pkRealVectorGetComponent(e1)[0],
                pkRealVectorGetComponent(e1)[1],
                pkRealVectorGetName(e1),
                pkRealVectorGetComponent(e2)[0],
                pkRealVectorGetComponent(e2)[1],
                pkRealVectorGetName(e2) );
        printf( "   \\draw[pktikzbasisvector,pktikzshadowed,->]\n"
                "       (origin) -- (" FLTFMT "," FLTFMT ") node[above] {$%s$};\n",
                pkRealVectorGetComponent(e3)[0],
                pkRealVectorGetComponent(e3)[1],
                pkRealVectorGetName(e3) );
    }

    return;
}
```

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

```
static void _drawApex( const PKREALVECTOR *a,
                       const PKREALVECTOR *a1,
                       const PKREALVECTOR *a2,
                       const PKREALVECTOR *a3,
                       const PKREALVECTOR *a12 )
{
   if ( a  &&  a1  &&  a2  &&  a3  &&  a12 ) {
      puts(   "   %");
      puts(   "   % Position 'apex'.");
      puts(   "   %");
      printf( "   \\draw[pktikzdimension]\n"
              "      (origin) --\n"
              "      (" FLTFMT "," FLTFMT ") --\n"
              "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[above right]{$%s$} --
              "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[left]{$%s$}\n"
              "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[above left]{$%s$} --\
              "      (" FLTFMT "," FLTFMT ") --\n"
              "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[above right]{$%s$};\n
              pkRealVectorGetComponent(a12)[0],
              pkRealVectorGetComponent(a12)[1],
              pkRealVectorGetComponent(a)[0],
              pkRealVectorGetComponent(a)[1],
              pkRealVectorGetName(a),
              pkRealVectorGetComponent(a3)[0],
              pkRealVectorGetComponent(a3)[1],
              pkRealVectorGetName(a3),
              pkRealVectorGetComponent(a1)[0],
              pkRealVectorGetComponent(a1)[1],
              pkRealVectorGetName(a1),
              pkRealVectorGetComponent(a12)[0],
              pkRealVectorGetComponent(a12)[1],
              pkRealVectorGetComponent(a2)[0],
              pkRealVectorGetComponent(a2)[1],
              pkRealVectorGetName(a2) );
   }

   return;
}
```

```
static void _printfPosition( const char *prefix,
                             const PKREALVECTOR *posn,
                             const char *suffix )
{
   if (posn)
      printf( "%s(" FLTFMT "," FLTFMT ")%s\n",
              strIsNull(prefix) ? "" : prefix,
              pkRealVectorGetComponent(posn)[0],
              pkRealVectorGetComponent(posn)[1],
              strIsNull(suffix) ? "" : suffix );
   return;
}
```

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

```
static void _drawFacet( const PKREALVECTOR *posn1,
                        const PKREALVECTOR *posn2,
```

```
                        const PKREALVECTOR *posn3,
                        const PKREALVECTOR *posn4,
                        const char *tikzStyle )
{
   if ( posn1  &&  posn2  &&  posn3  &&  posn4 ) {
      printf( "   \\path[%s]\n", strIsNull(tikzStyle) ? "draw" : tikzStyle );
      printf( "      (" FLTFMT "," FLTFMT ") -- "
                     "(" FLTFMT "," FLTFMT ") -- "
                     "(" FLTFMT "," FLTFMT ") -- "
                     "(" FLTFMT "," FLTFMT ") -- cycle;\n",
              pkRealVectorGetComponent(posn1)[0], pkRealVectorGetComponent(posn1)[1],
              pkRealVectorGetComponent(posn2)[0], pkRealVectorGetComponent(posn2)[1],
              pkRealVectorGetComponent(posn3)[0], pkRealVectorGetComponent(posn3)[1],
              pkRealVectorGetComponent(posn4)[0], pkRealVectorGetComponent(posn4)[1] );
   }

   return;
}
```

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.

```
static void _drawSurfaceElement ( const PKREALVECTOR *corner1,
                                  const PKREALVECTOR *mid1,
                                  const PKREALVECTOR *corner2,
                                  const PKREALVECTOR *mid2,
                                  const PKREALVECTOR *corner3,
                                  const PKREALVECTOR *mid3,
                                  const PKREALVECTOR *corner4,
                                  const PKREALVECTOR *mid4,
                                  const char *tikzStyle )
{
   if ( corner1  &&  corner2  &&  corner3  &&  corner4  &&
        mid1     &&  mid2     &&  mid3     &&  mid4 ) {
      printf( "   \\path[%s]\n", strIsNull(tikzStyle) ? "draw" : tikzStyle );
      printf( "      plot[smooth] coordinates { (" FLTFMT "," FLTFMT ")"
                                               "(" FLTFMT "," FLTFMT ")"
                                               "(" FLTFMT "," FLTFMT ") } --\n",
              pkRealVectorGetComponent(corner1)[0],
              pkRealVectorGetComponent(corner1)[1],
              pkRealVectorGetComponent(mid1)[0],
              pkRealVectorGetComponent(mid1)[1],
              pkRealVectorGetComponent(corner2)[0],
              pkRealVectorGetComponent(corner2)[1] );
      printf( "      plot[smooth] coordinates { (" FLTFMT "," FLTFMT ")"
                                               "(" FLTFMT "," FLTFMT ")"
                                               "(" FLTFMT "," FLTFMT ") } --\n",
              pkRealVectorGetComponent(corner2)[0],
              pkRealVectorGetComponent(corner2)[1],
              pkRealVectorGetComponent(mid2)[0],
              pkRealVectorGetComponent(mid2)[1],
              pkRealVectorGetComponent(corner3)[0],
              pkRealVectorGetComponent(corner3)[1] );
      printf( "      plot[smooth] coordinates { (" FLTFMT "," FLTFMT ")"
                                               "(" FLTFMT "," FLTFMT ")"
                                               "(" FLTFMT "," FLTFMT ") } --\n",
              pkRealVectorGetComponent(corner3)[0],
              pkRealVectorGetComponent(corner3)[1],
              pkRealVectorGetComponent(mid3)[0],
              pkRealVectorGetComponent(mid3)[1],
```

```
               pkRealVectorGetComponent(corner4)[0],
               pkRealVectorGetComponent(corner4)[1] );
      printf( "      plot[smooth] coordinates { (" FLTFMT "," FLTFMT ")"
                                            "(" FLTFMT "," FLTFMT ")"
                                            "(" FLTFMT "," FLTFMT ") } -- cycle;\n",
               pkRealVectorGetComponent(corner4)[0],
               pkRealVectorGetComponent(corner4)[1],
               pkRealVectorGetComponent(mid4)[0],
               pkRealVectorGetComponent(mid4)[1],
               pkRealVectorGetComponent(corner1)[0],
               pkRealVectorGetComponent(corner1)[1] );
   }

   return;
}
```

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

```
static void _drawHump( PKREALVECTOR ***posn,
                       const int Nx,
                       const int Ny )
{
   int i,
       j;

   if ( !posn )
      return;

   puts(   "   %");
   puts(   "   % The hump's cut-off edge.");
   puts(   "   %");
   puts( "   \\draw[draw=pktikzsurfacedrawcolor]" );
   for ( j = 0; j < Ny; j++ )
      _printfPosition( "      ", posn[0][j], " --" );
   for ( i = 0; i < Nx; i++ )
      _printfPosition( "      ", posn[i][Ny-1], " --" );
   for ( j = Ny-1; j >= 0; j-- )
      _printfPosition( "      ", posn[Nx-1][j], " --" );
   for ( i = Nx-1; i >= 0; i-- )
      _printfPosition( "      ", posn[i][0], ( i > 0 ) ? " --" : ";" );

   //puts(   "   %");
   //puts(   "   % The hump.");
   //puts(   "   %");
   //for ( i = 1;  i < Nx - 1;  i++ ) {
   //   puts( "   \\draw[surface]" );
   //   for ( j = 0;  j < Ny;  j++ )
   //      _printfPosition( "      ", posn[i][j], ( j < Ny - 1 ) ? "--" : ";" );
   //}
   //puts( "   %");
   //for ( j = 1;  j < Ny - 1;  j++ ) {
   //   puts( "   \\draw[surface]" );
   //   for ( i = 0;  i < Nx;  i++ )
   //      _printfPosition( "      ", posn[i][j], ( i < Nx - 1 ) ? "--" : ";" );
   //}

   //puts(   "   %");
   //puts(   "   % The hump.");
   //puts(   "   %");
   //for ( i = 1;  i < Nx - 1;  i++ ) {
   //   //puts( "   \\draw[surface] plot[smooth,mark=*,mark size=1pt] coordinates {" );
```

```
   //   puts( "   \\draw[surface] plot[smooth] coordinates {" );
   //   for ( j = 0;  j < Ny;  j++ )
   //      _printfPosition( "      ", posn[i][j], ( j < Ny - 1 ) ? "" : "};" );
   //}
   //puts( "   %");
   //for ( j = 1;  j < Ny - 1;  j++ ) {
   //   puts( "   \\draw[surface] plot[smooth] coordinates {" );
   //   for ( i = 0;  i < Nx;  i++ )
   //      _printfPosition( "      ", posn[i][j], ( i < Nx - 1 ) ? "" : "};" );
   //}

   //puts(   "   %");
   //puts(   "   % The hump.");
   //puts(   "   %");
   //for ( i = 0;  i < Nx - 1;  i++ ) {
   //   for ( j = 0;  j < Ny - 1;  j++ )
   //      _drawFacet( posn[i][j], posn[i][j+1], posn[i+1][j+1], posn[i+1][j],
   //                  "surface" );
   //}

   puts(   "   %");
   puts(   "   % The hump.");
   puts(   "   %");
   for ( i = 0;  i < Nx - 2;  i += 2 ) {
      for ( j = 0;  j < Ny - 2;  j += 2 ) {
         _drawSurfaceElement( posn[i][j],     posn[i][j+1],
                              posn[i][j+2],   posn[i+1][j+2],
                              posn[i+2][j+2], posn[i+2][j+1],
                              posn[i+2][j],   posn[i+1][j],
                              "hump" );
      }
   }
   _printfPosition( "   \\path ",
                    posn[3*Nx/4][Ny/4],
                    " node[pktikzsurfacedrawcolor,below,fill=white,rounded corners]"
                        "{$\\humpSet(h)$};" );

   return;
}
```

The _drawContourPath() private function printf()s Ti*k*Z commands for typesetting a subset of the $\mathcal{D}(\mathbf{p}, \Delta t)$ set, which set is described in detail in [5]. The specified array d of Nd entries are assumed to be the required $\mathbf{d}^i$ position vectors.[5]

```
static void _drawContourPath( PKREALVECTOR **d, const int Nd )
{
   const int Md = realMin(Nd,14);
   int i;

   if (!d)
      return;

   puts(   "   %");
   puts(   "   % 'z'-contour path 'd'.");
   puts(   "   %");
   //puts(   "   \\draw[pktikzsurfacepath] plot[mark=*,mark size=0.7pt] coordinates {" );
   puts(   "   \\draw[pktikzsurfacepath] plot coordinates {" );
   for ( i = 0;  i < Md;  i++ )
      _printfPosition( "      ", d[i], ( i < Md - 1 ) ? "" : " };" );
   if ( Md < Nd ) {
      //puts(   "   \\draw[occludedsurfacepath] plot[mark=*,mark size=0.6pt] coordinates {" );
```

```
      puts(   "   \\draw[occludedpath] plot coordinates {" );
      for ( i = Md - 1;  i < Nd;  i++ )
         _printfPosition( "       ", d[i], ( i < Nd - 1 ) ? "" : " };" );
   }

   _printfPosition( "   \\path (",
                    d[8*Nd/24],
                    "       node[contourpathcolor,\n"
                    "            below]\n"
                    "          {$h/(\\tbar^2+1)$-contour path};\n" );

   return;
}
```

The _drawContourPathStart() private function printf()s Ti*k*Z commands for typesetting the specified position **p**, as described in detail in [5]. This is the "local origin" position.

```
static void _drawContourPathStart( PKREALVECTOR *p )
{
   if (!p)
      return;

   puts(   "   %");
   puts(   "   % Contour start position.");
   puts(   "   %");
   printf( "   \\path (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[below=2pt]{$%s$};\n"
           pkRealVectorGetComponent(p)[0],
           pkRealVectorGetComponent(p)[1],
           pkRealVectorGetName(p) );

   return;
}
```

The _diagram() private function below specifies the diagram's three-dimensional landscape. It does this primarily using the PKREALVECTOR object class.[3] The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the landscape in LaTeX.

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., aligned with 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 [5] and [6].

```
static void _diagram(void)
{
   const int Nx = /*11*/ /*31*/ 51,
             Ny = Nx,
```

```
                Nd = 18;
    const PKMATHREAL xLos = 2.0, /* Line-of-sight vector components. */
                     yLos = 1.2,
                     zLos = 1.0,
                     theta = -103.0 / 180.0 * M_PI; /* Angle for Line-of-sight transformations.
                     //theta = -0.0 / 180.0 * M_PI; /* Angle for Line-of-sight transformations.
    PKMATHREAL u, v;
    PKREALVECTOR *e1,
                 *e2,
                 *e3,
                 *apex,          /* Apex of the hump. */
                 *apex1,         /* Apex of the hump along 1. */
                 *apex2,
                 *apex3,
                 *apex12,        /* Apex of the hump in the '1-2'-plane. */
                 *p,     /* Global position for a local origin on the hump. */
                 ***r,   /* Dynamic array of positions on the hump. */
                 **d;    /* Dynamic array of positions on the contour path */
                         /* passing thru 'p'. */
    int i,
        j;
```

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.

```
    e1 = pkRealVectorAlloc1( "\\ahat", 3, basisVecLen, 0.0, 0.0 );
    e2 = pkRealVectorAlloc1( "\\bhat", 3, 0.0, basisVecLen, 0.0 );
    e3 = pkRealVectorAlloc1( "\\chat", 3, 0.0, 0.0, basisVecLen );
    pkRealVectorScale(e1,1.1);
    //pkRealVectorScale(e2,0.8);
    pkRealVectorScale(e3,0.6);
```

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

```
    r = allocHumpPosnArr( Nx, Ny, 0.12 );
```

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

```
    u = humpX;
    v = humpY;
    apex =   pkRealVectorAlloc1( "A\\ahat+B\\bhat+h\\chat", 3, u, v, hump(u,v) );
    apex1 =  pkRealVectorAlloc1( "A", 3, u,   0.0, 0.0 );
    apex2 =  pkRealVectorAlloc1( "B", 3, 0.0, v,   0.0 );
    apex3 =  pkRealVectorAlloc1( "h", 3, 0.0, 0.0, 0.8 * hump(u,v) );
    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

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

```
    u = humpX + sqrt( ( 1.0 - 0.35 ) / 0.35 ) * cos( 5.0 * M_PI / 6.0 );
    v = humpY + sqrt( ( 1.0 - 0.35 ) / 0.35 ) * sin( 5.0 * M_PI / 6.0 );
    p = pkRealVectorAlloc1( "\\vecp",  3, u, v, hump(u,v) );
```

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 which is described in detail in [5].

```
    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 $\hat{\mathbf{1}}'\hat{\mathbf{2}}'$ plane lying flat on the page.

```
    for ( i = 0;  i < Nx;  i++ ) {
       for ( j = 0;  j < Ny;  j++ ) {
          pkRealVectorUnderLineOfSightBasis1( r[i][j],
                                              xLos, yLos, zLos, theta );
       }
    }
    pkRealVectorsUnderLineOfSightBasisV1( xLos, yLos, zLos, theta,
                                          d, Nd );
    if ( 0 == pkRealVectorsUnderLineOfSightBasis1( xLos, yLos, zLos, theta,
                                                   e1, e2, e3,
                                                   apex, apex1, apex2, apex3, apex12,
                                                   p,
                                                   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 );
       //_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);
   _freeHumpContourPosnArr(d,Nd);

   return;
}

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

8.2.4 The `hump.h` and `hump.c` files

The `hump.h`, `hump.c`, `sundry.h` and `sundry.c` files C source files contain common code definitions. A listing of the `hump.h` file follows:

```
#ifndef _HUMP
#define _HUMP
```

Inclusions.

```
#include <math.h> /* For 'M_PI'. */
#include <pkmath.h>
#include <pkrealvector.h>
```

Function declarations.

```
extern PKMATHREAL hump( const PKMATHREAL x, const PKMATHREAL y );
extern PKREALVECTOR ***allocHumpPosnArr( const int xPosns,
                                         const int yPosns,
                                         const PKMATHREAL alpha );
extern void freeHumpPosnArr( PKREALVECTOR ***posn,
                             const int xPosns,
                             const int yPosns );
extern PKREALVECTOR *allocHumpContourPosn0( const char *name,
                                            const PKREALVECTORREAL t,
                                            const PKREALVECTORREAL p,
                                            const PKREALVECTORREAL q,
                                            const PKREALVECTORREAL z );
extern void freeHumpContourPosn0( PKREALVECTOR *d );
extern PKREALVECTOR *allocHumpContourPosn( const char *name,
                                           const PKREALVECTORREAL t,
                                           const PKREALVECTOR *p );
extern void freeHumpContourPosn( PKREALVECTOR *d );
extern PKREALVECTOR **allocHumpContourPosnArr( const PKREALVECTOR *p,
                                               const int M,
                                               const PKMATHREAL deltat );
extern void freeHumpContourPosnArr( PKREALVECTOR **d, const int M );
extern PKREALVECTOR ***allocHumpContourArr( const int N,
                                            const int M,
                                            const PKMATHREAL deltat );
extern void freeHumpContourArr( PKREALVECTOR ***contourArr,
                                const int N,
                                const int M );
extern PKREALVECTOR *allocHumpGradientPosn0( const char *name,
                                             const PKREALVECTORREAL gamma,
                                             const PKREALVECTORREAL p,
                                             const PKREALVECTORREAL q );
extern void freeHumpGradientPosn0( PKREALVECTOR *d );
extern PKREALVECTOR *allocHumpGradientPosn( const char *name,
                                            const PKREALVECTORREAL gamma,
                                            const PKREALVECTOR *p );
extern void freeHumpGradientPosn( PKREALVECTOR *d );
extern PKREALVECTOR **allocHumpGradientPosnArr( const PKREALVECTOR *p,
                                                const int M );
extern void freeHumpGradientPosnArr( PKREALVECTOR **d, const int M );
extern PKREALVECTOR ***allocHumpGradientArr( const int N,
                                             const int M,
                                             const PKREALVECTORREAL px,
                                             const PKREALVECTORREAL py );
```

```
extern void freeHumpGradientArr( PKREALVECTOR ***gradientArr,
                                 const int N,
                                 const int M );
```

Global variable definitions.

```
extern const PKMATHREAL basisVecLen;
extern const PKMATHREAL humpHeight,
                        humpX,
                        humpY;
#endif
```

A listing of the hump.c file follows:

```c
#include "hump.h"

#include <pkfeatures.h>

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <float.h>

#include <pkmemdebug.h>
#include <pkerror.h>
#include <pktypes.h>
#include <pkstring.h>
#include <pkmath.h>
#include <pkrealvector.h>
```

Non-normalised length of the vectors $\hat{\mathbf{1}}$, $\hat{\mathbf{2}}$ and $\hat{\mathbf{3}}$.

```c
const PKMATHREAL basisVecLen = 6.0;
```

Some defining parameters for the hump $\mathcal{H}$.

```c
const PKMATHREAL humpHeight = 0.6 * basisVecLen,
                 humpX = 0.65 * basisVecLen, /*humpX = 0.6 * basisVecLen,*/
                 humpY = 0.55 * basisVecLen; /*humpY = 0.5 * basisVecLen;*/
```

The hump() function below simply returns the value $z(x, y)$ of the constitutive equation for the hump $\mathcal{H}$ centred at the position $a\hat{\mathbf{1}} + b\hat{\mathbf{2}}$:

$$\mathcal{H}(h, a, b) = \{(x, y, z) \mid z = \frac{h}{(x-a)^2 + (y-b)^2 + 1}\} \tag{29}$$

```c
PKMATHREAL hump( const PKMATHREAL x, const PKMATHREAL y )
{
   return( humpHeight / ( ( x - humpX ) * ( x - humpX ) +
                          ( y - humpY ) * ( y - humpY ) +
                          1.0 ) );
}
```

The allocHumpPosnArr() 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, the function returns a (PKREALVECTOR ***) pointer to the allocated and initialised array of position vectors. Otherwise it returns (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, we 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

$$\begin{aligned} x_{\min} &= a - \sqrt{(1-\alpha)/\alpha}, \quad & x_{\max} &= a + \sqrt{(1-\alpha)/\alpha} \\ y_{\min} &= b - \sqrt{(1-\alpha)/\alpha}, \quad & y_{\max} &= b + \sqrt{(1-\alpha)/\alpha} \end{aligned} \tag{30}$$

```
PKREALVECTOR ***allocHumpPosnArr( const int xPosns,
                                  const int yPosns,
                                  const PKMATHREAL alpha )
{
   PKREALVECTOR ***posn;

   posn = (PKREALVECTOR ***)calloc( xPosns + 1, sizeof(PKREALVECTOR **) );
   if (posn) {
      const PKMATHREAL xMin = humpX - sqrt( ( 1.0 - alpha ) / alpha ),
                       xMax = humpX + sqrt( ( 1.0 - alpha ) / alpha ),
                       yMin = humpY - sqrt( ( 1.0 - alpha ) / alpha ),
                       yMax = humpY + sqrt( ( 1.0 - alpha ) / alpha );
      PKMATHREAL p,
                 q;
      char *name;
      int i,
          j;
      for ( i = 0; i < xPosns; i++ ) {
         posn[i] = (PKREALVECTOR **)calloc( yPosns + 1, sizeof(PKREALVECTOR *) );
         p = xMin + (double)i / (double)(xPosns-1) * ( xMax - xMin );
         for ( j = 0; j < yPosns; j++ ) {
            name = strAllocPrintf( "\\pktikzVector{r}_{%d%d}", i, j );
            q = yMin + (double)j / (double)(yPosns-1) * ( yMax - yMin );
            posn[i][j] = pkRealVectorAlloc1( name, 3, p, q, hump(p,q) );
            strFreePrintf(name);
         }
      }
   }

   return(posn);
}
```

The `freeHumpPosnArr()` function is the complement to `allocHumpPosnArr()`.

```
void freeHumpPosnArr( PKREALVECTOR ***posn,
                      const int xPosns,
                      const int yPosns )
{
   if (posn) {
      int i,
          j;
      for ( i = 0;  i < xPosns;  i++ ) {
         for ( j = 0;  j < yPosns;  j++ ) {
            pkRealVectorFree1( posn[i][j] );
            posn[i][j] = (PKREALVECTOR *)NULL;
         }
         free( posn[i] );
         posn[i] = (PKREALVECTOR **)NULL;
      }
      free(posn);
   }

   return;
}
```

The `allocHumpContourPosn0()` function below allocates and initialises a position vector on the z-contour path parametrised with t using a locally-centric "regular" parametrisation, starting at the specified position $\mathbf{p} = p\hat{\mathbf{1}} + q\hat{\mathbf{2}} + z\hat{\mathbf{3}}$. Obviously, the z-contour will pass through $\mathbf{p}$. This function implements:

$$
\begin{aligned}
\mathbf{d}(t;\mathbf{p}) = {} & \Big((1-2t)(\mathbf{p}\cdot\hat{\mathbf{1}}-a) + 2\sqrt{t(1-t)}(\mathbf{p}\cdot\hat{\mathbf{2}}-b) + a\Big)\,\hat{\mathbf{1}} \\
& + \Big((1-2t)(\mathbf{p}\cdot\hat{\mathbf{2}}-b) - 2\sqrt{t(1-t)}(\mathbf{p}\cdot\hat{\mathbf{1}}-a) + b\Big)\,\hat{\mathbf{2}} \\
& + (\mathbf{p}\cdot\hat{\mathbf{3}})\,\hat{\mathbf{3}}, \quad 0 \le t \le 1
\end{aligned}
\tag{31}
$$

On success, the function returns a pointer to the allocated and initialised `PKREALVECTOR` representing the position vector. Otherwise it returns `(PKREALVECTOR *)NULL`. The function must be accompanied by a call to `freeHumpContourPosn0()`.

```
PKREALVECTOR *allocHumpContourPosn0( const char *name,
                                     const PKREALVECTORREAL t,
                                     const PKREALVECTORREAL p,
                                     const PKREALVECTORREAL q,
                                     const PKREALVECTORREAL z )
{
   return( pkRealVectorAlloc1(
              name,
              3,
              ( 1.0 - 2.0 * t ) * ( p - humpX )
                 - 2.0 * sqrt( t * ( 1.0 - t ) ) * ( q - humpY ) + humpX,
              ( 1.0 - 2.0 * t ) * ( q - humpY )
                 + 2.0 * sqrt( t * ( 1.0 - t ) ) * ( p - humpX ) + humpY,
              z ) );
}
```

The `freeHumpContourPosn0()` function is the complement to `allocHumpContourPosn0()`.

```
void freeHumpContourPosn0( PKREALVECTOR *d )
{
   if (d)
      pkRealVectorFree1(d);
   return;
}
```

If the specified `(PKREALVECTOR *)` pointer `p` is not `NULL`, then the `allocHumpContourPosn()` 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()`.

```
PKREALVECTOR *allocHumpContourPosn( const char *name,
                                    const PKREALVECTORREAL t,
                                    const PKREALVECTOR *p )
{
   if (!p)
      return( (PKREALVECTOR *)NULL );
   return( allocHumpContourPosn0( name,
                                  t,
                                  pkRealVectorGetComponent(p)[0],
                                  pkRealVectorGetComponent(p)[1],
                                  pkRealVectorGetComponent(p)[2] ) );
}
```

The `freeHumpContourPosn()` function is the complement to `allocHumpContourPosn()`.

```
void freeHumpContourPosn( PKREALVECTOR *d )
{
   if (d)
      freeHumpContourPosn0(d);
   return;
}
```

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

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

On success, the function return a pointer to the allocated and initialised array of M (`PKREALVECTOR *`)s. Otherwise it returns `(PKREALVECTOR **)NULL`. The function must be accompanied by a call to `freeHumpContourPosnArr()`.

```
PKREALVECTOR **allocHumpContourPosnArr( const PKREALVECTOR *p,
                                        const int M,
                                        const PKMATHREAL deltat )
{
   PKREALVECTOR **d;

   if ( !p  ||  M < 1  ||  deltat < 0.0  ||  deltat > 1.0 )
      return( (PKREALVECTOR **)NULL );

   d = (PKREALVECTOR **)calloc( M + 1, sizeof(PKREALVECTOR *) );
   if (d) {

      char *name;
      int j;

      d[0] = pkRealVectorAlloc1( "\\vecd^0", 3,
                                 pkRealVectorGetComponent(p)[0],
                                 pkRealVectorGetComponent(p)[1],
                                 pkRealVectorGetComponent(p)[2] );
      for ( j = 1;  j < M;  j++ ) {
         name = strAllocPrintf( "\\vecd^{%d}", j );
         d[j] = allocHumpContourPosn( name, deltat, d[j-1] );
         strFreePrintf(name);
      }

   }

   return(d);
}
```

The `freeHumpContourPosnArr()` function is the complement to `allocHumpContourPosnArr()`.

```
void freeHumpContourPosnArr( PKREALVECTOR **d, const int M )
{
```

```
    if (d) {
       int j;
       for ( j = 0;  j < M;  j++ )
          freeHumpContourPosn(d[j]);
       free(d);
    }
    return;
}
```

The `allocHumpContourArr()` function allocates and initialises an array of N pointers to arrays of M z-contour path positions. The starting position of each such array of z-contour path positions is taken to be on a path over $\mathcal{H}$ beginning arbitrarily at $\mathbf{p}_0 = \frac{1}{5}a\hat{\mathbf{1}} + \frac{1}{5}b\hat{\mathbf{2}} + z(\frac{1}{5}a, \frac{1}{5}b)\hat{\mathbf{3}}$ and ending at $\mathbf{p}_{N-1} = a\hat{\mathbf{1}} + b\hat{\mathbf{2}} + z(a,b)\hat{\mathbf{3}}$, and where along that path, $y/x = b/a$. From (29), it is easy to show then that

$$x = x(z) = \left(1 \pm \sqrt{\frac{h/z - 1}{a^2 + b^2}}\right) a$$

$$y = y(x(z)) = \frac{bx(z)}{a}$$

Also, along that path we identify the N z-values

$$z_i = z_0 + \frac{i}{N-1}(z_{N-1} - z_0), \quad i = 0, 1, 2, \ldots, N-1$$

This then provides starting positions for the N z-contour paths as

$$\{\mathbf{p}_i = x(z_i)\hat{\mathbf{1}} + y(x(z_i))\hat{\mathbf{2}} + z_i\hat{\mathbf{3}} \mid z_i = z_0 + \frac{i}{N-1}(z_{N-1} - z_0),\ i = 0, 1, 2, \ldots, N-1\}$$

Once a `PKREALVECTOR` representing $\mathbf{p}_i$ has been allocated and initialised with a call to `pkRealVectorAlloc1()`, the i-th array of M z-contour path positions is allocated and initialised with the call to `allocHumpContourPosnArr(p,M,deltat)`.

On success, the function return a (`PKREALVECTOR ***`) pointer to the allocated and initialised array of N (`PKREALVECTOR **`) pointers to arrays of M (`PKREALVECTOR *`) z-contour path positions. Otherwise it returns `(PKREALVECTOR ***)NULL`. The function must be accompanied by a call to `freeHumpContourArr()`.

```
PKREALVECTOR ***allocHumpContourArr( const int N,
                                     const int M,
                                     const PKMATHREAL deltat )
{
    PKREALVECTOR ***contourArr;

    if ( N < 1  ||  M < 1 )
       return( (PKREALVECTOR ***)NULL );

    contourArr = (PKREALVECTOR ***)calloc( N + 1, sizeof(PKREALVECTOR **) );
    if (contourArr) {

       PKREALVECTORREAL firstz, lastz;   /* z-coordinates of first and last position. */
       int i;

       firstz = hump( humpX / 5.0, humpY / 5.0 );
       lastz  = hump( humpX, humpY );

       for ( i = 0; i < N; i++ ) {

          PKREALVECTORREAL px,
```

```
                        py,
                        pz;
         PKREALVECTOR *p;
         char *name;

         name = strAllocPrintf( "\\vecp_%d", i );
         pz = firstz + (PKREALVECTORREAL)i / (PKREALVECTORREAL)( N - 1 )
                                          * ( lastz - firstz );
         px = humpX * ( 1.0 - sqrt( ( humpHeight / pz - 1.0 )
                                 / ( humpX * humpX + humpY * humpY ) ) );
         py = humpY / humpX * px;
         p = pkRealVectorAlloc1( name, 3, px, py, pz );

         contourArr[i] = allocHumpContourPosnArr( p, M, deltat );

         strFreePrintf(name);
         pkRealVectorFree1(p);

      }

   }

   return(contourArr);
}
```

The `freeHumpContourArr()` function is the complement to `allocHumpContourArr()`.

```
void freeHumpContourArr( PKREALVECTOR ***contourArr,
                         const int N,
                         const int M )
{
   if (contourArr) {
      int i;
      for ( i = 0; i < N; i++ ) {
         if (contourArr[i]) {
            freeHumpContourPosnArr( contourArr[i], M );
            contourArr[i] = (PKREALVECTOR **)NULL;
         }
      }
      free(contourArr);
   }
   return;
}
```

The `allocHumpGradientPosn0()` function below allocates and initialises a position vector on the (p, q)-gradient path parametrised with γ, starting at the specified position $\mathbf{p} = p\hat{\mathbf{1}} + q\hat{\mathbf{2}} + z(p, q)\hat{\mathbf{3}}$. Obviously, the gradient will pass through $\mathbf{p}$. The function implements the specific parametrisation

$$\mathbf{g}(\gamma; p, q) = g_1(\gamma; p, q)\hat{\mathbf{1}} + g_2(\gamma; p, q)\hat{\mathbf{2}} + z(g_1, g_2)\hat{\mathbf{3}} \tag{32}$$

with

$$g_1(\gamma; p, q) = p + \gamma(a - p), \quad g_2(\gamma; p, q) = b + \frac{q - b}{p - a}(g_1(\gamma; p, q) - a) \quad \text{for } p \neq a \tag{33}$$

and

$$g_1(\gamma; p, q) = a, \quad g_2(\gamma; p, q) = q + \gamma(b - q) \quad \text{for } p = a \tag{34}$$

It is easy to verify that $\mathbf{g}(0; p, q) = p\hat{\mathbf{1}} + q\hat{\mathbf{2}} + z(p, q)\hat{\mathbf{3}}$, and that $\mathbf{g}(1; p, q) = a\hat{\mathbf{1}} + b\hat{\mathbf{2}} + h\hat{\mathbf{3}}$, as expected.

Case $p \neq a$. The parametrisation (32) was obtained following the recipe described in my article "A study of surfaces embedded in $\mathbb{R}^3$". From (29), a gradient vector evaluated at the position $p\hat{\mathbf{1}} + q\hat{\mathbf{2}} + z(p,q)\hat{\mathbf{3}}$ is

$$\begin{aligned}
\frac{\mathrm{d}\mathbf{g}(\gamma;p,q)}{\mathrm{d}\gamma} &= \frac{\mathrm{d}g_1(\gamma;p,q)}{\mathrm{d}\gamma}\hat{\mathbf{1}} + \frac{\mathrm{d}g_2(\gamma;p,q)}{\mathrm{d}\gamma}\hat{\mathbf{2}} + \frac{\mathrm{d}z(g_1,g_2)}{\mathrm{d}\gamma}\hat{\mathbf{3}} \\
&= \frac{\partial z(g_1(\gamma;p,q),g_2(\gamma;p,q))}{\partial x}\hat{\mathbf{1}} + \frac{\partial z(g_1,g_2)}{\partial y}\hat{\mathbf{2}} + \left[\nabla_{(x,y)} z(g_1,g_2)\right]^2 \hat{\mathbf{3}} \\
&= -\frac{2(g_1-a)z^2(g_1,g_2)}{h}\hat{\mathbf{1}} - \frac{2(g_2-b)z^2(g_1,g_2)}{h}\hat{\mathbf{2}} + \left[\nabla_{(x,y)} z(g_1,g_2)\right]^2 \hat{\mathbf{3}}
\end{aligned}$$

If we assume a functional dependence $g_2(\gamma) = g_2(g_1(\gamma))$, then by the familiar differential calculus Chain Rule

$$\frac{\mathrm{d}g_2}{\mathrm{d}g_1} = \frac{\mathrm{d}g_2}{\mathrm{d}\gamma}\Big/\frac{\mathrm{d}g_1}{\mathrm{d}\gamma} = \frac{\partial z(g_1,g_2)}{\partial y}\Big/\frac{\partial z(g_1,g_2)}{\partial x} = \frac{g_2-b}{g_1-a}$$

from which

$$\int \frac{\mathrm{d}g_2}{g_2-b} = \int \frac{\mathrm{d}g_1}{g_1-a}$$

giving

$$g_2(\gamma;p,q) = b + C(p,q)(g_1(\gamma;p,q)-a) \qquad \text{for some } C(p,q).$$

So we have

$$\mathbf{g}(\gamma;p,q) = g_1\hat{\mathbf{1}} \;+\; [b + C(g_1-a)]\,\hat{\mathbf{2}} \;+\; z(g_1, b + C(g_1-a))\hat{\mathbf{3}}$$

We are now free to choose a suitable or convenient parametrisation for $\mathbf{g}$. If we wish that $\mathbf{g}(0;p,q) = p\hat{\mathbf{1}} + q\hat{\mathbf{2}} + z(p,q)\hat{\mathbf{3}}$, then we may choose $g_1(\gamma;p,q) = p + \gamma(a-p)$, so that $C(p,q) = (q-b)/(p-a)$. And the final parametrisation is that in (32) and (33). But to be sure, we could also have chosen something like

$$g_1(\gamma;p,q) = \frac{1}{e^{-1}-e}\left[\left(pe^{-1}-a\right)e^{\gamma} - (pe-a)\,e^{-\gamma}\right]$$

as a less convenient parametrisation.

Case $p = a$. Since $g_1(\gamma;p,q) = a$ for any γ, we are free to set the parametrisation as in (34).

On success, this function returns a pointer to the allocated and initialised `PKREALVECTOR` representing the position vector. Otherwise the function returns `(PKREALVECTOR *)NULL`. The function must be accompanied by a call to `freeHumpGradientPosn0()`.

```
PKREALVECTOR *allocHumpGradientPosn0( const char *name,
                                      const PKREALVECTORREAL gamma,
                                      const PKREALVECTORREAL p,
                                      const PKREALVECTORREAL q )
{
   PKREALVECTORREAL g1,
                    g2;

   if ( fabs(p-humpX) <= FLT_EPSILON ) {
      g1 = p;
      g2 = q + gamma * ( humpY - q );
   } else {
      g1 = p + gamma * ( humpX - p );
      g2 = humpY + ( q - humpY ) / ( p - humpX ) * ( g1 - humpX );
   }

   return( pkRealVectorAlloc1( name, 3, g1, g2, hump(g1,g2) ) );
}
```

The `freeHumpGradientPosn0()` function is the complement to `allocHumpGradientPosn0()`.

```
void freeHumpGradientPosn0( PKREALVECTOR *d )
{
   if (d)
      pkRealVectorFree1(d);
   return;
}
```

If the specified `(PKREALVECTOR *)` pointer `p` is not `NULL`, then the `allocHumpGradientPosn()` function below simply returns with the result of the call:

```
allocHumpGradientPosn0( name,
                        t,
                        pkRealVectorGetComponent(p)[0],
                        pkRealVectorGetComponent(p)[1] )
```

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

```
PKREALVECTOR *allocHumpGradientPosn( const char *name,
                                     const PKREALVECTORREAL gamma,
                                     const PKREALVECTOR *p )
{
   if (!p)
      return( (PKREALVECTOR *)NULL );
   return( allocHumpGradientPosn0( name,
                                   gamma,
                                   pkRealVectorGetComponent(p)[0],
                                   pkRealVectorGetComponent(p)[1] ) );
}
```

The `freeHumpGradientPosn()` function is the complement to `allocHumpGradientPosn()`.

```
void freeHumpGradientPosn( PKREALVECTOR *d )
{
   if (d)
      freeHumpGradientPosn0(d);
   return;
}
```

The `allocHumpGradientPosnArr()` function allocates and initialises an array of M `(PKREALVECTOR *)` gradient path positions, beginning at the specified **p** position. Obviously, the gradient will pass through **p**. In fact, the 0-th position in the allocated array is allocated and initialised to represent **p**. The j-th position, $j = 1, \ldots, M-1$ in the allocated array is allocated and initialised with a call to `allocHumpGradientPosn()` by setting the parametrisation parameter `gamma`$= \gamma = \dfrac{j}{M-1}$

On success, the function returns a pointer to the allocated and initialised array of M `(PKREALVECTOR *)`s. Otherwise it returns `(PKREALVECTOR **)NULL`. The function must be accompanied by a call to `freeHumpGradientPosnArr()`.

```
PKREALVECTOR **allocHumpGradientPosnArr( const PKREALVECTOR *p,
                                         const int M )
{
   PKREALVECTOR **d;

```

```
    if ( !p  ||  M < 3  )
       return( (PKREALVECTOR **)NULL );

    d = (PKREALVECTOR **)calloc( M + 1, sizeof(PKREALVECTOR *) );
    if (d) {

       PKREALVECTORREAL gamma;
       char *name;
       int j;

       d[0] = pkRealVectorAlloc1( "\\vecg^0", 3,
                                  pkRealVectorGetComponent(p)[0],
                                  pkRealVectorGetComponent(p)[1],
                                  pkRealVectorGetComponent(p)[2] );
       for ( j = 1;  j < M;  j++ ) {
          name = strAllocPrintf( "\\vecg^{%d}", j );
          gamma = (double)j / (double)( M - 1 );
          d[j] = allocHumpGradientPosn( name, gamma, p );
          strFreePrintf(name);
       }

    }

    return(d);
 }
```

The `freeHumpGradientPosnArr()` function is the complement to `allocHumpGradientPosnArr()`.

```
 void freeHumpGradientPosnArr( PKREALVECTOR **d, const int M )
 {
    if (d) {
       int j;
       pkRealVectorFree1(d[0]);
       for ( j = 1;  j < M;  j++ )
          freeHumpGradientPosn(d[j]);
       free(d);
    }
    return;
 }
```

The `allocHumpGradientArr()` function allocates and initialises an array of N pointers to arrays of M gradient path positions. The starting position of each such array of M gradient path positions is taken to be one of the N positions on the $z(\frac{1}{5}a, \frac{1}{5}b)$-contour path over $\mathcal{H}$.

Once a `PKREALVECTOR` representing the position $\mathbf{p} = \frac{1}{5}a\hat{\mathbf{1}} + \frac{1}{5}b\hat{\mathbf{2}} + z(\frac{1}{5}a, \frac{1}{5}b)\hat{\mathbf{3}}$ has been allocated and initialised with an appropriate call to `pkRealVectorAlloc1()`, a finite subset of the $\mathcal{D}(\mathbf{p}, \Delta t)$ set is implemented with the call to `allocHumpContourPosnArr(p,N,0.02)`. That finite subset of contour positions is then used as the abovementioned N starting positions for the N gradient paths. With the i-th such starting position labelled as $\mathbf{c}_i$ (i.e., as `contour[i]` below), then the i-th array of M gradient path positions is allocated and initialised with the call to `allocHumpGradientPosnArr(contour[i],M)`.

On success, the function returns a (`PKREALVECTOR ***`) pointer to the allocated and initialised array of N (`PKREALVECTOR **`) pointers to arrays of M (`PKREALVECTOR *`) gradient path positions. Otherwise the function returns `(PKREALVECTOR ***)NULL`. The function must be accompanied by a call to `freeHumpGradientArr()`.

```
 PKREALVECTOR ***allocHumpGradientArr( const int N,
                                       const int M,
```

```
                                      const PKREALVECTORREAL px,
                                      const PKREALVECTORREAL py )
{
   PKREALVECTOR ***gradientArr;
   PKREALVECTOR *p;
   int i;

   if ( N < 1  ||  M < 1 )
      return( (PKREALVECTOR ***)NULL );

   /*
    * Error by default.
    */
   gradientArr = (PKREALVECTOR ***)NULL;

   p = pkRealVectorAlloc1( "\\vecp", 3, px, py, hump(px,py) );
   if (p) {
      PKREALVECTOR **contour = allocHumpContourPosnArr( p, N, 0.02 );
      if (contour) {
         gradientArr = (PKREALVECTOR ***)calloc( N + 1, sizeof(PKREALVECTOR **) );
         if (gradientArr) {
            for ( i = 0; i < N; i++ )
               gradientArr[i] = allocHumpGradientPosnArr( contour[i], M );
         }
         freeHumpContourPosnArr(contour,N);
      }
      pkRealVectorFree1(p);
   }

   return(gradientArr);
}
```

The `freeHumpGradientArr()` function is the complement to `allocHumpGradientArr()`.

```
void freeHumpGradientArr( PKREALVECTOR ***gradientArr,
                          const int N,
                          const int M )
{
   if (gradientArr) {
      int i;
      for ( i = 0; i < N; i++ ) {
         if (gradientArr[i]) {
            freeHumpGradientPosnArr( gradientArr[i], M );
            gradientArr[i] = (PKREALVECTOR **)NULL;
         }
      }
      free(gradientArr);
   }
   return;
}
```

8.2.5 The `sundry.h` and `sundry.c` files

A listing of the `sundry.h` file follows:

```
#ifndef _SUNDRY
#define _SUNDRY
```

Inclusions.

```
#include <math.h> /* For 'M_PI'. */
#include <pkmath.h>
#include <pkrealvector.h>
```

Macro definitions.

```
#define FLTFMT "%.4g"
```

Function declarations.

```
extern PKREALVECTORREAL realMin( const PKREALVECTORREAL a, const PKREALVECTORREAL b );
extern void printVector( const PKREALVECTOR *v );
```

```
#endif
```

A listing of the `sundry.c` file follows:

```
#include "sundry.h"

#include <pkfeatures.h>

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <float.h>

#include <pkmemdebug.h>
#include <pkerror.h>
#include <pktypes.h>
#include <pkstring.h>
#include <pkmath.h>
#include <pkrealvector.h>

PKREALVECTORREAL realMin( const PKREALVECTORREAL a, const PKREALVECTORREAL b )
{
   return( ( a < b ) ? a : b );
}

void printVector( const PKREALVECTOR *v )
{
   if (!v)
      return;

   printf( "Vector %s = ( ", pkRealVectorGetName(v) );
   pkRealVectorPrintf( v, "__COMPONENTVALUE__", ", " );
   puts(" )");

   return;
}
```

8.3 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: dimensionality-of-reality.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.

```

dimensionality-of-reality.pdf: spherefigure.tex \
                                Makefile.demo \
                                dimensionality-of-reality.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.[7]

```

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

References

[1] Charles Misner, Kip Thorne, and John Wheeler. *Gravitation*. Number 0-7167-0334-3. W.H. Freeman and Company, 1973.

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

[3] Paul Kotschy. The PKLIB C software library. paul.kotschy@gmail.com.

[4] Paul Kotschy. pkTechDoc: Literate programming for non-TeX programmers. paul.kotschy@gmail.com.

[5] Paul Kotschy. Parametrisation of irregular paths on surfaces embedded in $\mathbb{R}^3$. paul.kotschy@gmail.com, September 2016.

[6] Paul Kotschy. Rotational transformations in three dimensions. paul.kotschy@gmail.com, May 2016.

[7] Paul Kotschy. The pkLaTeXmake package. paul.kotschy@gmail.com.