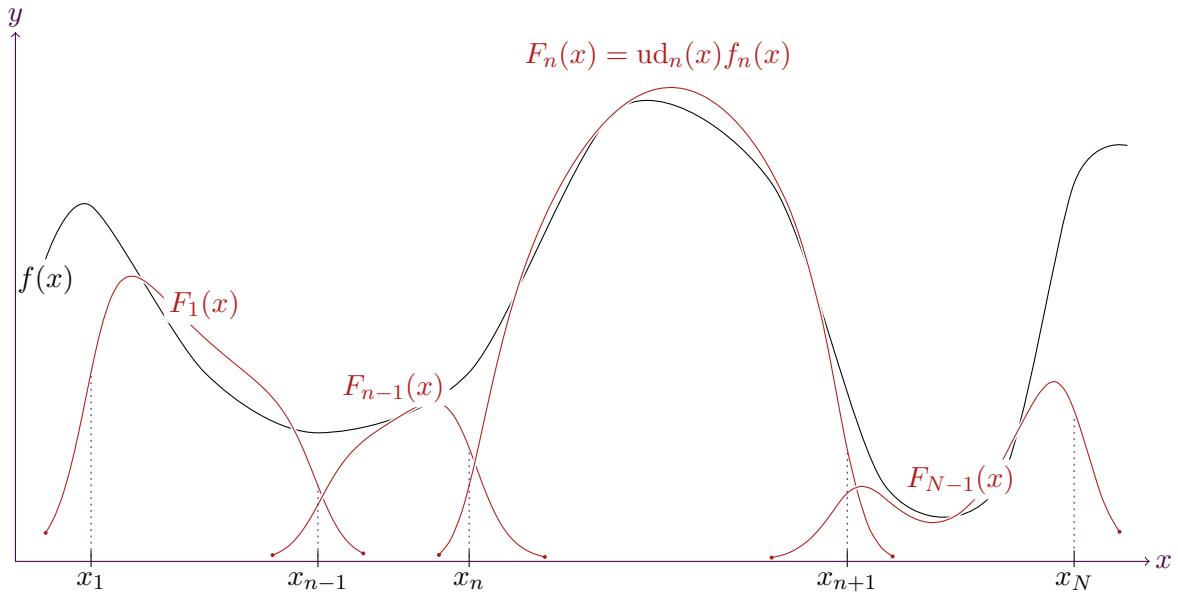
Interpolating with "Plateau Functions"

Paul Kotschy

9 May 2017

Compiled on March 20, 2025



Abstract

In this work[1] a new form of functional interpolation is introduced, dubbed *plateau function interpolation*. As per usual, an approximating function is synthesised as a linear combination of simple well-known functions. But here, the simple functions need not be taken from a single class of functions.

The interpolation scheme enables known qualitative behaviour of the function of interest to be exploited easily and efficiently. For example, the trigonometrics, exponentials and rationals may easily be combined in a single interpolation to capture periodic, secular and asymptotic behaviour, respectively, over different parts of the domain interval.

The interpolation scheme is amenable to analytical work because it enables a single closed-form expression of the synthesised function to be calculated. And the expression is usually C^∞–continuous over the entire domain interval, not piecewise continuous over contiguous sub-intervals.

[1] paul.kotschy@gmail.com

Contents

1 Introduction

Functional interpolation is an established branch in numerical mathematical analysis.[1, 2] The need for functional interpolation arises when: function values are known only for a finite set of domain points, the so-called base points; an explicit expression for the function is not known; or, working with the function is computationally expensive or analytically difficult.

A typical interpolation scheme involves synthesising another function which approximates the first over a domain interval of interest. The synthesised one will not only be relatively easy to work with, but it will also suitably mimic certain behaviour of the first. It is obvious that the more that is known about the first, the better the mimicry may be.

A widely used synthesis involves a linear combination of simple functions taken from a single class of functions. Typical such classes include the trigonometric functions, the exponentials, the rationals, and the polynomials. The cubic spline polynomials, for example, offer an interpolating function which agrees with the first function at the base points, and is smooth in its first derivative and continuous in its second derivative throughout the domain interval.[1, 3] But the interpolating function is essentially a composite of separate cubic splines, one for each domain sub-interval. Evaluation of the function at an arbitrary point in the domain interval therefore requires the sub-interval containing the point to first be identified.

A new interpolation scheme is presented here. It offers the following benefits:

1. A single expression for the synthesised function is obtained. The expression is applicable over the entire domain interval, not just over one of the sub-intervals.
2. The synthesised function is usually C^∞–continuous over the entire domain interval, not just over one of the sub-intervals.
3. The abovementioned simpler functions need not be taken from a single class.

The interpolation scheme enables known behaviour of the first function to be exploited with relative ease and efficiency. For example, if it is known that the function exhibits secular decay behaviour over one part of the domain, and periodic behaviour over the remaining part, then exponentials may best be used to mimic the function in the first part, followed by trigonometrics in the remaining part. The interpolation scheme introduced here will ensure C^∞-continuity not only over the two parts, but also at the boundary between them.

By reading this document, you witness the interpolation scheme in action. The document contains a series of figures to supplement the text. Some of these figures were calculated using the interpolation scheme described here. The calculations were carried out using the C programming language, and were then "drawn" using the Ti*k*Z graphics sub-system[4] of the TEX typesetting system.[5, 6, 7]

In Section 2, the interpolation scheme is presented in detail. Some illustrative yet simple examples are given in Section 3. A specific implementation is given in Section 4. As just mentioned, the implementation was used to help typeset the figures in this document.

2 Analysis

The scheme is essentially an interpolation using a sub-class of rational-type functions. I shall dub them "plateau functions." To begin, suppose we require to work with a function f,

$$f : x \in [x_1, x_N] \to f(x) \in \mathbb{R}$$

with $x_1, x_N \in \mathbb{R}$, for some $N \in \mathbb{N}$. Suppose, too, that function values are available only at N base points $x_1, x_2, \ldots, x_N$. This implies that an exact expression for the function is not known

over its domain interval $[x_1, x_N]$. To work with the function away from the base points, we therefore require an approximation of it which adequately captures necessary elements of its behaviour over $[x_1, x_N]$.

Simple functions. Partition the interval into $N-1$ sub-intervals $[x_n, x_{n+1}]$, $n = 1, \ldots, N-1$. This partitioning must be carried out such that f is adequately approximated over $[x_n, x_{n+1}]$ with some simple well-known function f_n. The set $\{f_n | n = 1, \ldots, N-1\}$ need not be drawn from a single class of functions. The set would typically comprise simple functions, such as simple polynomials, rationals, and exponentials.

An example of such a set is shown schematically in Figure 1. In the figure, f_1 and f_{n-1} happen to be exponentials, and f_n and f_{N-1} happen to be quadratics. They are constructed to pass through two arbitrary positions $\mathbf{p} = p_1\hat{\mathbf{1}} + p_2\hat{\mathbf{2}}$ and $\mathbf{q} = q_1\hat{\mathbf{1}} + q_2\hat{\mathbf{2}}$ in the $\hat{\mathbf{1}}\hat{\mathbf{2}}$ plane:

$$f(x; \mathbf{p}, \mathbf{q}, s) = \begin{cases} \left(\dfrac{x - p_1}{q_1 - p_1}\right)(q_2 - p_2) + p_2, & \text{for linear behaviour.} \\ s(x - p_1)(x - q_1) + \left(\dfrac{x - p_1}{q_1 - p_1}\right)(q_2 - p_2) + p_2, & \text{for quadratic behaviour.} \\ \left(\dfrac{e^{-sx} - e^{-sp_1}}{e^{-sq_1} - e^{-sp_1}}\right)(q_2 - p_2) + p_2, & \text{for exponential behaviour,} \end{cases} \tag{1}$$

for some intensity parameter $s \in \mathbb{R}$.

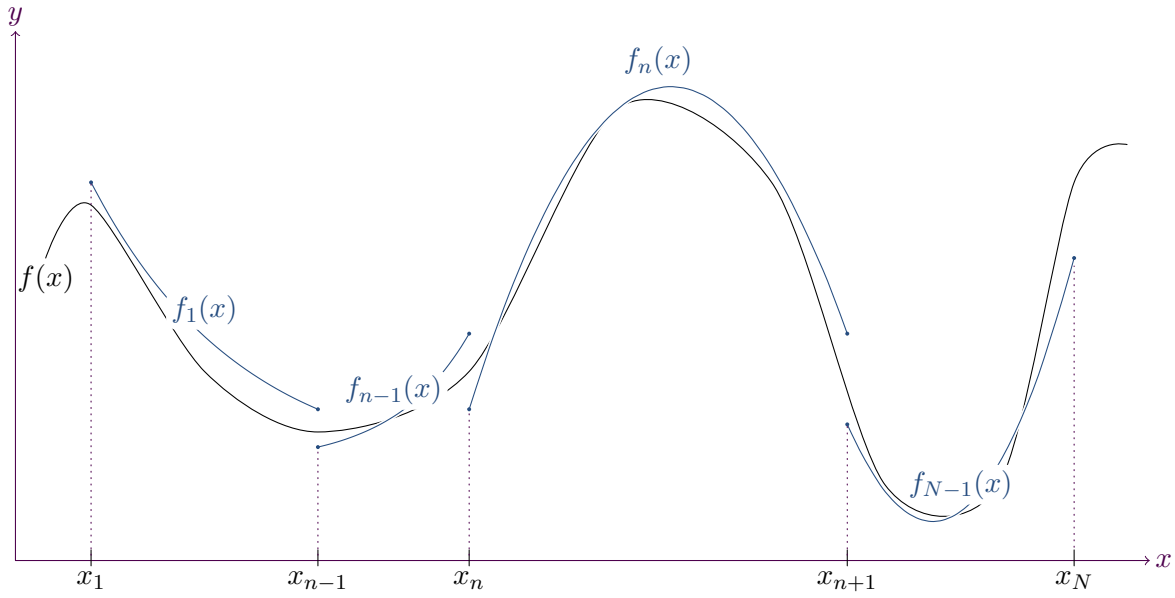


Figure 1: Set $\{f_n(x) \mid n = 1, \ldots, N-1\}$ of simple well-known functions used to approximate the function of interest, f, over the domain interval $[x_1, x_N]$. In the figure, f_1 and f_{n-1} happen to be exponentials, and f_n and f_{N-1} happen to be quadratics of the form (1).

Plateau approximation. If we can find some function $\mathrm{ud}_n(x)$ which is C^∞-continuous over $[x_1, x_N]$, which is mostly equal to 1 over (x_n, x_{n+1}), and which tends to zero sufficiently rapidly outside $[x_n, x_{n+1}]$, then clearly, the function

$$\boxed{F_n(x) = \mathrm{ud}_n(x) f_n(x) \quad \text{for } x \in (x_n, x_{n+1})} \tag{2}$$

could be a significant contributor to an approximation of f over (x_n, x_{n+1}). Other contributors could be F_{n-1} and F_{n+1}, although less so than F_n. And since ud_n is C^∞-continuous

and mostly equal to 1 over (x_n, x_{n+1}), F_n inherits the continuity character of f_n over much of (x_n, x_{n+1}). I shall dub F_n the n-th "plateau function." The set $\{F_n | n = 1, \ldots, N-1\}$ is shown schematically in Figure 2.

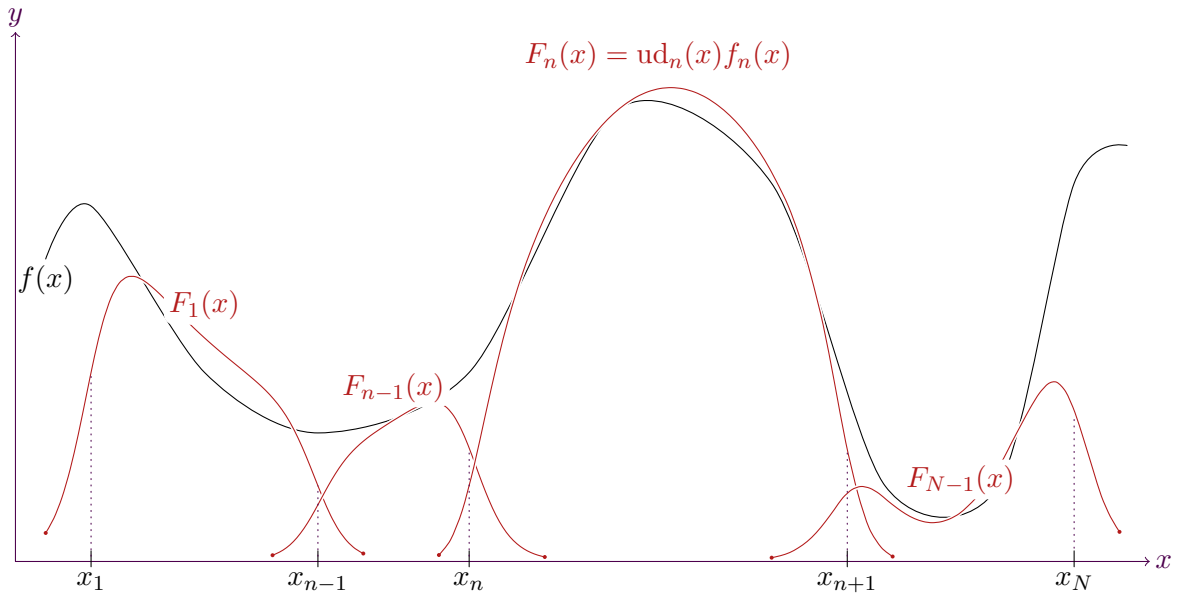


Figure 2: Set $\{F_n(x) \mid n = 1, \ldots, N-1\}$ of "plateau" functions used to approximate the function of interest, f, over the domain interval $[x_1, x_N]$.

A reasonable interpolation function for f is then

$$\boxed{f(x) \approx F(x) = \sum_{n=1}^{N-1} \alpha_n F_n(x) = \sum_{n=1}^{N-1} \alpha_n \mathrm{ud}_n(x) f_n(x), \quad x \in [x_1, x_N]} \tag{3}$$

for some set $\{\alpha_n | n = 1, \ldots, N-1\}$, as yet unspecified. I shall discuss the weighting parameters α_n later, focussing for now on the "up/down" functions, ud_n.

"Up/down" functions. Consider the following "rising" and "falling" rational-type functions:

$$\begin{aligned} \mathrm{up}(x; \beta) &= \frac{1}{1 + \beta^{-x}}, \\ \mathrm{do}(x; \beta) &= \frac{1}{1 + \beta^{x}} = \mathrm{up}(-x; \beta) \end{aligned} \tag{4}$$

for some base $\beta > 1$. A plot of typical $\mathrm{up}(x; \beta)$ and $\mathrm{do}(x; \beta)$ functions is shown in Figure 3.

It is then easy to construct an "up/down" function over $[x_n, x_{n+1}]$ as

$$\mathrm{ud}_n(x) = \mathrm{up}(x - x_n; \beta_n)\mathrm{do}(x - x_{n+1}; \beta_n) \tag{5}$$

A typical $\mathrm{ud}_n(x)$ function is shown in Figure 4.

As the factor $\mathrm{up}(x - x_n; \beta_n)$ works to gradually raise $\mathrm{do}(x - x_{n+1}; \beta_n)$ from 0 to 1 at the point x_n, the factor $\mathrm{do}(x - x_{n+1}; \beta_n)$ works to gradually lower $\mathrm{up}(x - x_n; \beta_n)$ from 1 to 0 at the point x_{n+1}. And so, as the product, $\mathrm{up}(x - x_n; \beta_n)\mathrm{do}(x - x_{n+1}; \beta_n)$, $\mathrm{ud}_n(x; \beta_n)$ works to restrict any influence of f_n to the $[x_n, x_{n+1}]$ sub-interval. But it does its work gradually without any discontinuity. It is then obvious that in the linear combination (3), a contribution by F_n will likely dominate over $[x_n, x_{n+1}]$.

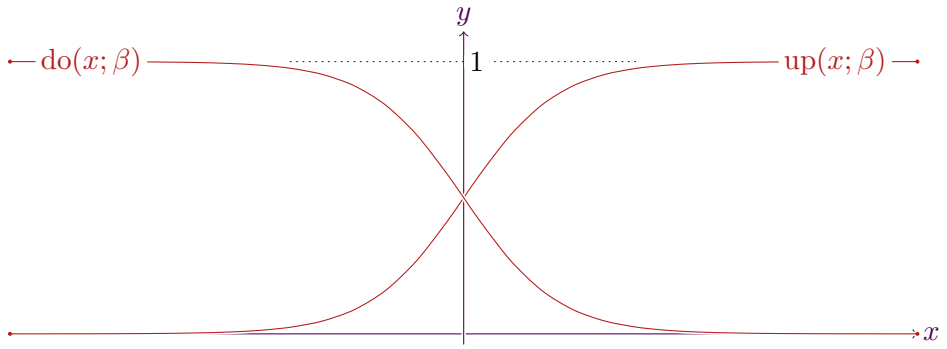


Figure 3: The $\mathrm{up}(x;\beta)$ and $\mathrm{do}(x;\beta)$ functions, with $\beta = 5$. (Equation (4)).

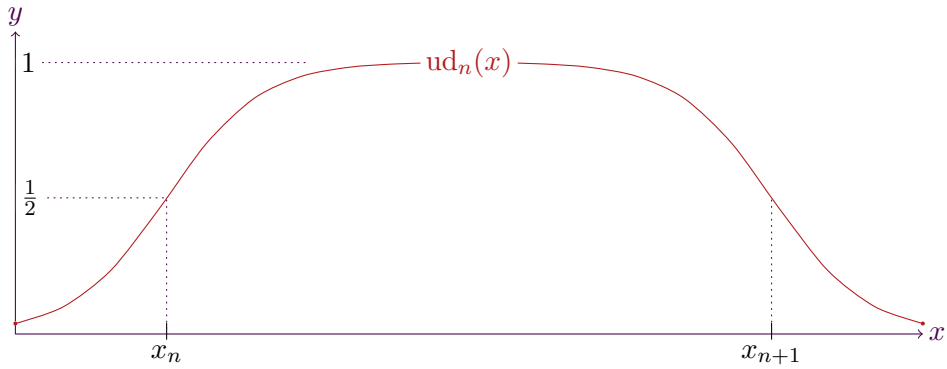


Figure 4: The "up/down" function $\mathrm{ud}_n(x)$, with $\beta_n = 5$. (Equation (5)).

Weighting parameters. But what of the weighting parameters α_n in (3)? We are in fact free to choose any strategy we please to specify them. One strategy could be to insist that over any sub-interval $[x_n, x_{n+1}]$, the area under the interpolation function must agree with the area under f_n. That is,

$$\int_{x_n}^{x_{n+1}} F(x)\mathrm{d}x = \sum_{m=1}^{N-1} \int_{x_n}^{x_{n+1}} \mathrm{ud}_m(x) f_m(x)\mathrm{d}x\, \alpha_m = \int_{x_n}^{x_{n+1}} f_n(x)\mathrm{d}x, \quad n = 1, \ldots, N-1$$

Another strategy is simply to insist that the interpolating function take on sensible values at the N base points $x_1, \ldots, x_N$. Inspection of Figures 1 and 2 suggests

$$\begin{aligned} F(x_1) + F(x_N) &= \tfrac{1}{2}(f_1(x_1) + f_{N-1}(x_N)) \\ F(x_m) &= \tfrac{1}{2}(f_{m-1}(x_m) + f_m(x_m)), \quad m = 2, \ldots, N-1 \end{aligned}$$

which, using (3), results in the system of equations for the $N-1$ unknowns, α_n

$$\begin{aligned} \sum_{n=1}^{N-1} (F_n(x_1) + F_n(x_N))\alpha_n &= \tfrac{1}{2}(f_1(x_1) + f_{N-1}(x_N)) \\ \sum_{n=1}^{N-1} F_n(x_m)\alpha_n &= \tfrac{1}{2}(f_{m-1}(x_m) + f_m(x_m)), \quad m = 2, \ldots, N-1 \end{aligned} \tag{6}$$

In matrix form, (6) may be written as

$$\begin{bmatrix} F_1(x_1)+F_1(x_N) & F_2(x_1)+F_2(x_N) & \cdots & F_{N-1}(x_1)+F_{N-1}(x_N) \\ F_1(x_2) & F_2(x_2) & \cdots & F_{N-1}(x_2) \\ \vdots & \vdots & & \vdots \\ F_1(x_{N-1}) & F_2(x_{N-1}) & \cdots & F_{N-1}(x_{N-1}) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{N-1} \end{bmatrix}$$
$$= \frac{1}{2} \begin{bmatrix} f_1(x_1)+f_{N-1}(x_N) \\ f_1(x_2)+f_2(x_2) \\ \vdots \\ f_{N-2}(x_{N-1})+f_{N-1}(x_{N-1}) \end{bmatrix} \tag{7}$$

Diagonal dominance. Inspection of Figure 2 suggests that (7) is diagonally dominant. Were it not so, solving (6) (or (7)) would be of cost $O((N-1)^2)$. But the figure is misleading. Diagonal dominance is no *a priori* guarantee. For any chosen sub-interval $[x_n, x_{n+1}]$, the presence of diagonal dominance depends on how rapidly F_n and its neighbouring functions vary over $[x_n, x_{n+1}]$. So to ensure that (7) approaches diagonal dominance, we must regulate the behaviour of the F_n functions by insisting that

$$\begin{aligned} &F_1(x) \approx 0 \quad \forall\, x \geq x_3 \\ &F_n(x) \approx 0 \quad \forall\, x \leq x_{n-1} \text{ and } x \geq x_{n+2},\ n = 2, \ldots, N-2 \\ &F_{N-1}(x) \approx 0 \quad \forall\, x \leq x_{N-2} \end{aligned} \tag{8}$$

These conditions are satisfied provided that for any chosen small numbers $0 < \epsilon_n \ll 1$, $n = 1, \ldots, N-1$

$$\begin{aligned} &|F_1(x_3)| < \epsilon_1 \, |F_1(x_2)| \,, \\ &|F_n(x_{n-1})| < \epsilon_n \, |F_n(x_n)| \text{ and } |F_n(x_{n+2})| < \epsilon_n \, |F_n(x_{n+1})| \,, \quad n = 2, \ldots, N-2, \\ &|F_{N-1}(x_{N-2})| < \epsilon_{N-1} \, |F_{N-1}(x_{N-1})| \end{aligned} \tag{9}$$

Consider the middle requirement in (8) and (9) for the moment. From (2), the requirement becomes

$$\begin{aligned} &\mathrm{ud}_n(x_{n-1}) \, |f_n(x_{n-1})| < \epsilon_n \mathrm{ud}_n(x_n) \, |f_n(x_n)| \,, \quad \text{and} \\ &\mathrm{ud}_n(x_{n+2}) \, |f_n(x_{n+2})| < \epsilon_n \mathrm{ud}_n(x_{n+1})) \, |f_n(x_{n+1})| \end{aligned}$$

Now since

$$\begin{aligned} \mathrm{ud}_n(x_{n-1}) &= \mathrm{up}(x_{n-1} - x_n)\mathrm{do}(x_{n-1} - x_{n+1}) = \mathrm{do}(x_n - x_{n-1})\mathrm{up}(x_{n+1} - x_{n-1}) \\ \mathrm{ud}_n(x_n) &= \mathrm{up}(x_n - x_n)\mathrm{do}(x_n - x_{n+1}) = \tfrac{1}{2}\,\mathrm{up}(x_{n+1} - x_n) \\ \mathrm{ud}_n(x_{n+2}) &= \mathrm{up}(x_{n+2} - x_n)\mathrm{do}(x_{n+2} - x_{n+1}) \\ \mathrm{ud}_n(x_{n+1}) &= \mathrm{up}(x_{n+1} - x_n)\mathrm{do}(x_{n+1} - x_{n+1}) = \tfrac{1}{2}\,\mathrm{up}(x_{n+1} - x_n) \end{aligned}$$

the middle requirement in (8) is satisfied provided that

$$\begin{aligned} &\mathrm{do}(x_n - x_{n-1})\mathrm{up}(x_{n+1} - x_{n-1}) < \mathrm{up}(x_{n+1} - x_n) \, \frac{\epsilon_n}{2} \left| \frac{f_n(x_n)}{f_n(x_{n-1})} \right| , \quad \text{and} \\ &\mathrm{up}(x_{n+2} - x_n)\mathrm{do}(x_{n+2} - x_{n+1}) < \mathrm{up}(x_{n+1} - x_n) \, \frac{\epsilon_n}{2} \left| \frac{f_n(x_{n+1})}{f_n(x_{n+2})} \right| \end{aligned}$$

And using (4)

$$(1+\beta_n^{x_n-x_n}n-1)(1+\beta_n^{-(x_n-x_{n+1})}n-1) > (1+\beta_n^{-(x_{n+1}-x_n)})\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n-1})}{f_n(x_n)}\right|, \quad \text{and}$$

$$(1+\beta_n^{-(x_{n+2}-x_n)})(1+\beta_n^{x_{n+2}-x_{n+1}}) > (1+\beta_n^{-(x_{n+1}-x_n)})\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n+2})}{f_n(x_{n+1})}\right|$$

Now since $1+\beta_n^{-x} > 1$ for any x, it follows that these two conditions are satisfied provided that

$$\begin{aligned} 1+\beta_n^{x_n-x_{n-1}} &> (1+\beta_n^{-(x_{n+1}-x_n)})\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n-1})}{f_n(x_n)}\right|, \quad \text{and} \\ 1+\beta_n^{x_{n+2}-x_{n+1}} &> (1+\beta_n^{-(x_{n+1}-x_n)})\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n+2})}{f_n(x_{n+1})}\right| \end{aligned} \tag{10}$$

Consider for now the first condition in (10). It is satisfied provided that

$$\begin{aligned} & 1 > \beta_n^{-(x_{n+1}-x_n)}\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n-1})}{f_n(x_n)}\right| \quad \text{and} \quad \beta_n^{x_n-x_{n-1}} > \frac{2}{\epsilon_n}\left|\frac{f_n(x_{n-1})}{f_n(x_n)}\right| \\ \Leftarrow \quad & \beta_n > \left(\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n-1})}{f_n(x_n)}\right|\right)^{1/(x_{n+1}-x_n)} \quad \text{and} \quad \beta_n > \left(\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n-1})}{f_n(x_n)}\right|\right)^{1/(x_n-x_{n-1})} \\ \Leftarrow \quad & \beta_n > \left(\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n-1})}{f_n(x_n)}\right|\right)^{1/\min\{x_{n+1}-x_n,\, x_n-x_{n-1}\}} \end{aligned} \tag{11}$$

Equation (11) therefore provides part of a prescription for β_n in the middle condition in (8) in order that the demand be met that (6) approach diagonal dominance.

Consider now the second condition in (10). It is similarly satisfied provided that

$$\beta_n > \left(\frac{2}{\epsilon_n}\left|\frac{f_n(x_{n+2})}{f_n(x_{n+1})}\right|\right)^{1/\min\{x_{n+2}-x_{n+1},\, x_{n+1}-x_n\}} \tag{12}$$

Conditions (11) and (12) may be combined to give a single result for the middle condition in (8). Furthermore, a similar analysis may be carried out on the first and third conditions in (8). A combined condition for the diagonal dominance of (6) (and (7)) is therefore

$$\boxed{\begin{aligned} \beta_1 &> \left(\frac{2}{\epsilon_1}\left|\frac{f_1(x_3)}{f_1(x_2)}\right|\right)^{1/\min\{x_3-x_2,\, x_2-x_1\}} \\ \beta_n &> \left(\frac{2}{\epsilon_n}\max\left\{\left|\frac{f_n(x_{n-1})}{f_n(x_n)}\right|, \left|\frac{f_n(x_{n+2})}{f_n(x_{n+1})}\right|\right\}\right)^{1/\min\{x_{n+2}-x_{n+1},\, x_{n+1}-x_n,\, x_n-x_{n-1}\}}, \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } n = 2,\ldots,N-2 \\ \beta_{N-1} &> \left(\frac{2}{\epsilon_{N-1}}\left|\frac{f_{N-1}(x_{N-2})}{f_{N-1}(x_{N-1})}\right|\right)^{1/\min\{x_N-x_{N-1},\, x_{N-1}-x_{N-2}\}} \end{aligned}} \tag{13}$$

for $0 < \epsilon_1, \ldots, \epsilon_{N-1} \ll 1$.

Weighting parameters revisited. With diagonal dominance now guaranteed to a reasonable approximation, the system (6) for the unkown set $\{\alpha_n | n = 1, \ldots, N-1\}$ becomes,

$$\boxed{\begin{aligned} F_1(x_1)\alpha_1 + F_{N-1}(x_N)\alpha_{N-1} &= \tfrac{1}{2}(f_1(x_1) + f_{N-1}(x_N)) \\ F_{n-1}(x_n)\alpha_{n-1} + F_n(x_n)\alpha_n &= \tfrac{1}{2}(f_{n-1}(x_n) + f_n(x_n)), \quad n = 2,\ldots,N-1 \end{aligned}} \tag{14}$$

By considering the cases $N = 2, 3, 4$ and 5, it is relatively easy to obtain solutions for the α_{N-1} weighting parameter for each case. Inspection of these specific solutions offered a general solution for α_{N-1}, and hence a recursive solution for all the α_n as

$$\boxed{\alpha_1 = \tfrac{1}{2}\,\frac{f_1(x_1) + f_1(x_2)}{F_1(x_1) + F_1(x_2)}, \qquad \text{for } N = 2}$$

and

$$\boxed{\begin{aligned}\alpha_1 = \tfrac{1}{2}\,\frac{1}{F_1(x_1)}\,\frac{1}{1 + (-)^N \prod_{n=1}^{N-1} \frac{F_n(x_{n+1})}{F_n(x_n)}}\,\Bigg[\, & f_1(x_1) + f_{N-1}(x_N) \\ & - \sum_{m=1}^{N-2} (-)^{N+m} \big(f_m(x_{m+1}) + f_{m+1}(x_{m+1})\big) \prod_{o=m+1}^{N-1} \frac{F_o(x_{o+1})}{F_o(x_o)} \,\Bigg], \qquad \text{for } N > 2 \\ \alpha_n = \frac{1}{F_n(x_n)} \Big[\, & \tfrac{1}{2}\big(f_{n-1}(x_n) + f_n(x_n)\big) \;-\; F_{n-1}(x_n)\alpha_{n-1} \,\Big], \qquad \text{for } n = 2, \ldots, N-1\end{aligned}} \tag{15}$$

3 Examples

A FEW EXAMPLES of plateau function interpolations are presented. The first example is of a simple constant value. The second example is of a simple linear rise and fall. The third is of a simple linear fall and rise. The fourth is of a linear rise, fall, and then rise again. The fifth example is of a "lobsided parabola".

3.1 Constant function.

This example is simple albeit not very practical. It is an interpolation of a constant value over the inverval $[x_1, x_3] = [-1, 1]$.

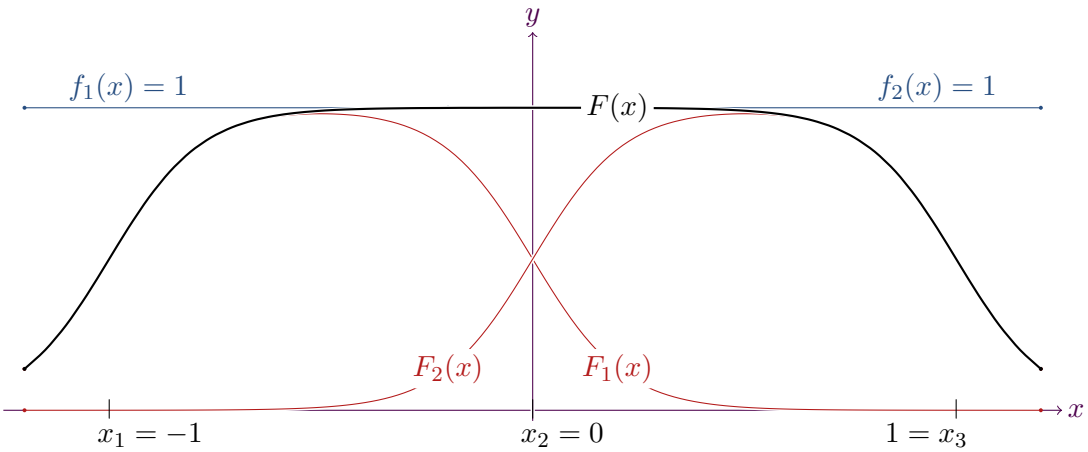


Figure 5: A plateau function interpolation (F in (17)) of a constant value. The constant value is represented by two simple constant functions f_1 and f_2. The domain interval of interest is $[x_1, x_3] = [-1, 1]$. With an accuracy parameter of $\epsilon = 0.0001$, the plateau functions F_1 and F_2 were calculated to be those in (16).

As shown, the constant value is represented by the simple linear functions $f_1(x) = 1$ and $f_2(x) = 1$, respectively. An obvious plateau function interpolation is then

$$F(x) = \alpha_1 F_1(x) + \alpha_2 F_2(x)$$

For an accuracy measure of $\epsilon_1 = \epsilon_2 = \epsilon$, the plateau function bases (Equations (4) and (13)) were calculated to be $\beta_1 = \beta_2 = 4/\epsilon$, so that the two plateau functions are

$$\begin{aligned} F_1(x) = \mathrm{ud}_1(x) f_1(x) &= \frac{1}{[1 + (\epsilon/4)^{x+1}]\,[1 + (\epsilon/4)^{-x}]} \\ F_2(x) = \mathrm{ud}_2(x) f_2(x) &= \frac{1}{[1 + (\epsilon/4)^{x-1}]\,[1 + (\epsilon/4)^{-x}]} \end{aligned} \tag{16}$$

To satisfy (14), the two weighting parameters were calculated using (15) to be

$$\begin{aligned} \alpha_1 &= \frac{F_2(x_2) - F_2(x_3)}{F_1(x_1)F_2(x_2) - F_1(x_2)F_2(x_3)} \\ \alpha_2 &= \frac{F_1(x_1) - F_1(x_2)}{F_1(x_1)F_2(x_2) - F_1(x_2)F_2(x_3)} \end{aligned}$$

Both the numerator and denominator in these two expressions vanish, leaving α_1 and α_2 indeterminate. So for now, we simply "cheat" by setting $\alpha_1 = \alpha_2 = 1$.

3.2 Symmetrical linear rise and fall.

This next example is also simple. It is an interpolation of a symmetrical linear rise and fall about the $\hat{\mathbf{2}}$-axis, the graph of which is shown in Figure 6.

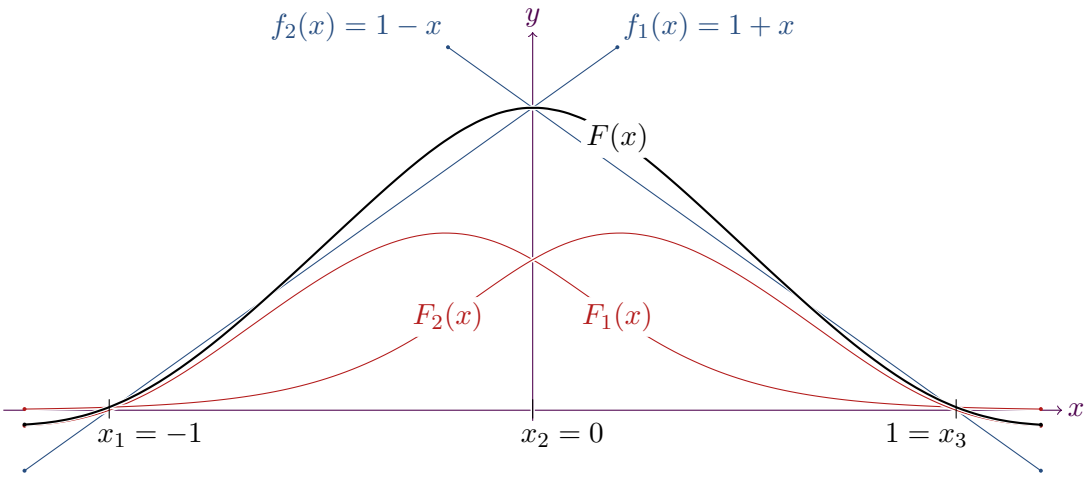


Figure 6: A plateau function interpolation (F in (17)) of a symmetrical linear rise and fall about the $\hat{\mathbf{2}}$-axis. The rise and fall are represented by two simple linear functions f_1 and f_2. The domain interval of interest is $[x_1, x_3] = [-1, 1]$. With an accuracy parameter of $\epsilon = 0.01$, the plateau functions F_1 and F_2 were calculated to be those in (18).

As shown, the rise and fall are represented by the simple linear functions $f_1(x) = 1 + x$ and $f_2(x) = 1 - x$, respectively. Again, an obvious plateau function interpolation is then

$$F(x) = \alpha_1 F_1(x) + \alpha_2 F_2(x) \tag{17}$$

For an accuracy measure of $\epsilon_1 = \epsilon_2 = \epsilon$, the plateau function bases (Equations (4) and (13)) were calculated to be $\beta_1 = \beta_2 = 4/\epsilon$, so that the two plateau functions are

$$\begin{aligned} F_1(x) = \mathrm{ud}_1(x) f_1(x) &= \frac{1 + x}{[1 + (\epsilon/4)^{x+1}]\,[1 + (\epsilon/4)^{-x}]} \\ F_2(x) = \mathrm{ud}_2(x) f_2(x) &= \frac{1 - x}{[1 + (\epsilon/4)^{x-1}]\,[1 + (\epsilon/4)^{-x}]} \end{aligned} \tag{18}$$

To satisfy (14), the two weighting parameters were calculated using (15) to be

$$\alpha_1 = \frac{-F_2(x_3)}{F_1(x_1)F_2(x_2) - F_1(x_2)F_2(x_3)}$$

$$\alpha_2 = \frac{F_1(x_1)}{F_1(x_1)F_2(x_2) - F_1(x_2)F_2(x_3)}$$

Once again, both the numerator and denominator in these two expressions vanish, leaving α_1 and α_2 indeterminate. So again, we "cheat" by setting $\alpha_1 = \alpha_2 = 1$.

3.3 Asymmetrical linear rise and fall.

This example is identical to the first except that the rise and fall is no longer symmetrical about the $\hat{\mathbf{2}}$-axis. The asymetry is implemented by replacing the second linear function with $f_2(x) = \frac{5}{8}(1 - x)$. The three chosen base points remain as is, namely, x_1, x_2 and x_3, as shown in Figures 6 and 7. Clearly, the two graphs of the simple functions f_1 and f_2 no longer intersect at x_2, and it is the task of the weighting parameter specification (Equation (15)) to ensure a reasonable fit of the interpolation function F at x_2, as shown in Figure 7.

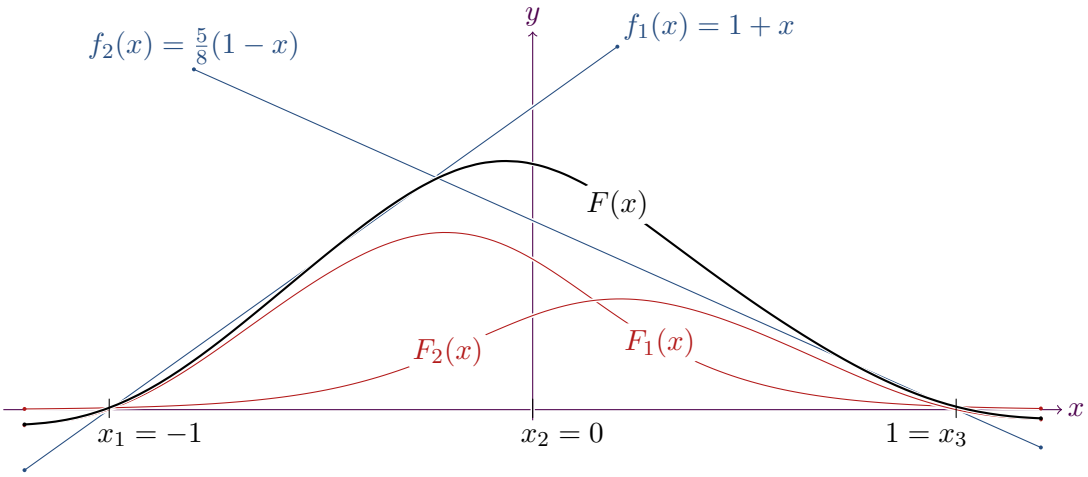


Figure 7: A plateau function interpolation of an asymetrical linear rise and fall about the $\hat{\mathbf{2}}$-axis. The graph of the two simple functions f_1 and f_2 do not intersect at the second base point x_2. The interpolation function F "tries its best" without sacrificing its smoothness. The accuracy parameter is $\epsilon = 0.01$.

3.4 Linear rise and fall and rise.

A rise and fall and rise again is represented by three simple linear functions $f_1(x) = 1 + x$, $f_2(x) = 1 - x$, and $f_3(x) = x - 1$, respectively. The obvious plateau function interpolation is then

$$F(x) = \alpha_1 F_1(x) + \alpha_2 F_2(x) + \alpha_3 F_3(x) \tag{19}$$

The graphs of the three linear functions, the corresponding plateau functions, and resultant interpolation function are shown in Figure 8.

3.5 Linear rise and fall and rise with higher accuracy.

This example is identical to the previous one except that the accuracy parameter ϵ has been reduced, resulting in the graph of the interpolation in Figure 9.

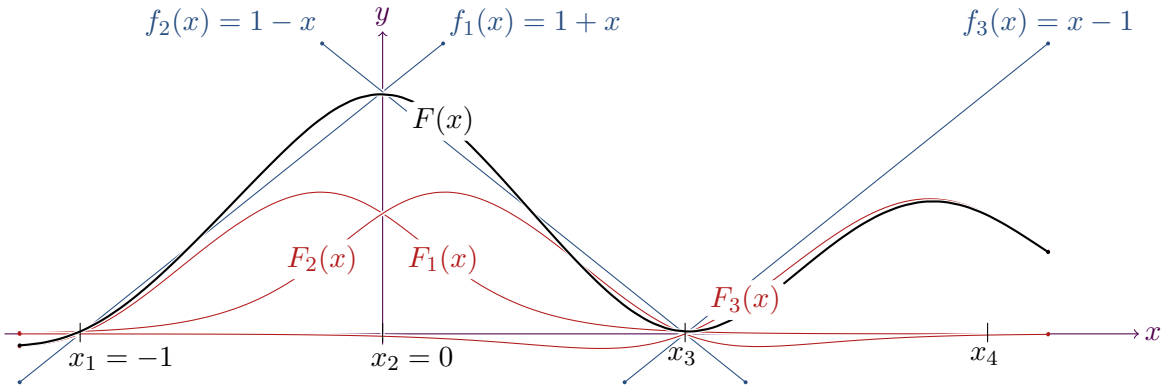


Figure 8: A plateau function interpolation (F in (19)) of a linear rise, fall, and rise again. The domain interval of interest is $[x_1, x_4] = [-1, 2]$. The accuracy parameter is $\epsilon = 0.01$. In the figure, the graph of F_3 is somewhat obscured by that of F.

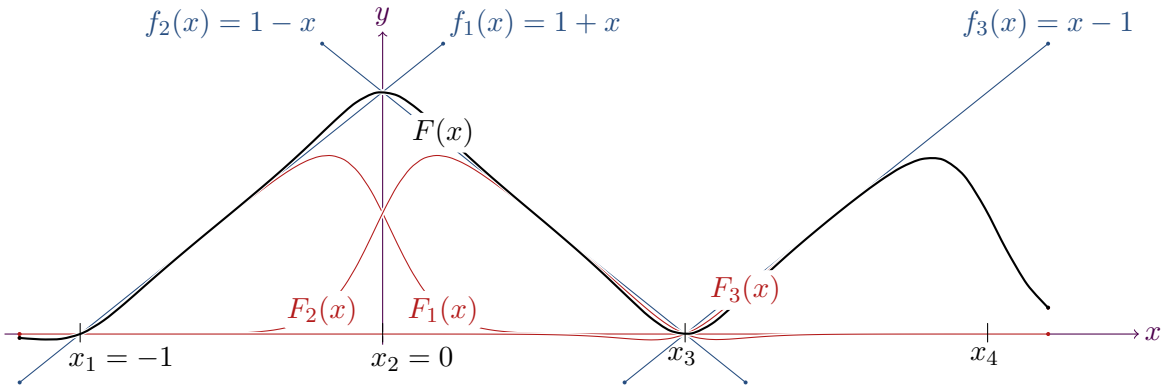


Figure 9: A plateau function interpolation (F in (19)) of a linear rise, fall, and rise again. The accuracy parameter is $\epsilon = 0.000010$.

3.6 A lobsided parabola.

The "lobsided parabola" was behind an initial motivation for formulating plateau function interpolation. A functional shape was sought which was a blending of two parabolas, and I was not content with simply "stitching" the two together. I wanted a single functional expression which encapsulated a blending of the two, and which was smooth over the entire domain interval.

A blending of two parabolas, f_1 and f_2, in a neighbourhood of a point x_2 may easily be achieved by identifying x_2 as one of three base points x_1, x_2 and x_3. If we know that the parabola f_1 must pass through the positions $\mathbf{a} = a_1\hat{\mathbf{1}} + a_2\hat{\mathbf{2}}$ and $\mathbf{b} = b_1\hat{\mathbf{1}} + b_2\hat{\mathbf{2}}$, and that the parabola f_2 must pass through $\mathbf{c} = c_1\hat{\mathbf{1}} + c_2\hat{\mathbf{2}}$ and $\mathbf{d} = d_1\hat{\mathbf{1}} + d_2\hat{\mathbf{2}}$, then the functional form for f_1 and f_2 is obtained at once from (1) as $f_1(x) = f(x; \mathbf{a}, \mathbf{b}, s)$ and $f_2(x) = f(x; \mathbf{c}, \mathbf{d}, t)$ for some intensity parameters $s, t \in \mathbb{R}$. And the plateau function interpolation is then

$$F(x) = \alpha_1 F_1(x) + \alpha_2 F_2(x) \tag{20}$$

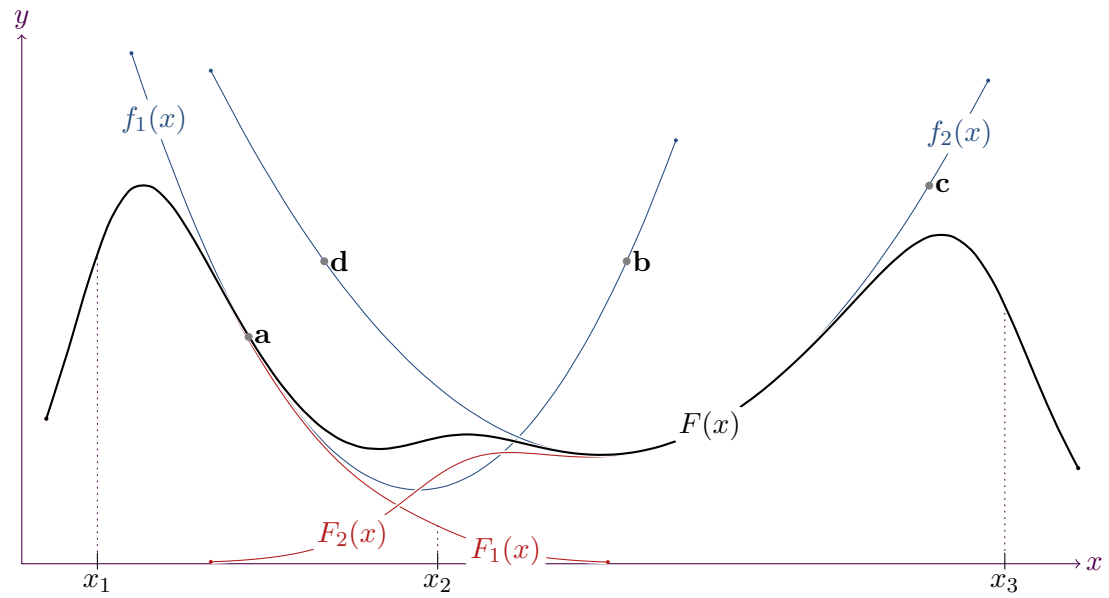


Figure 10: An unsatisfactory plateau function interpolation (F in (20)) of a "lobsided parabola". Two parabolas f_1 and f_2 have been blended in a neighbourhood of the point x_2. The plateau functions F_1 and F_2 were calculated using (1), (2) and (5) to be those in (21). Values for the bases β_1 and β_2 in (21) were computed using an accuracy parameter $\epsilon = 0.001$. The intensities in (21) were set at $s = 0.4$ and $t = 0.19$.

with

$$\begin{aligned} F_1(x) &= \mathrm{ud}_1(x) f(x; \mathbf{a}, \mathbf{b}, s) \\ &= \frac{\left(b_2 - a_2 + s\left(b_1 - a_1\right)\left(x - b_1\right)\right)\left(\dfrac{x - a_1}{b_1 - a_1}\right) + a_2}{\left(1 + \beta_1^{-(x - x_1)}\right)\left(1 + \beta_1^{x - x_2}\right)} \\ F_2(x) &= \mathrm{ud}_2(x) f(x; \mathbf{c}, \mathbf{d}, t) \\ &= \frac{\left(d_2 - c_2 + t\left(d_1 - c_1\right)\left(x - d_1\right)\right)\left(\dfrac{x - c_1}{d_1 - c_1}\right) + c_2}{\left(1 + \beta_2^{-(x - x_2)}\right)\left(1 + \beta_2^{x - x_3}\right)} \end{aligned} \tag{21}$$

To satisfy (14), the two weighting parameters α_1 and α_2 in (20) were calculated using (15).

It is clear that the plateau function interpolation shown in Figure 10 does not quite resemble a lobsided parabola. But the interpolation does the best it can given the two parabolas f_1 and f_2. To improve the interpolation, choose position $\mathbf{b} = x_2\hat{\mathbf{1}} + b_2\hat{\mathbf{2}}$ for some specified b_2, and specify that $\dfrac{\mathrm{d}f_1(x_2)}{\mathrm{d}x} = \dfrac{\mathrm{d}f_2(x_2)}{\mathrm{d}x} = 0$. Doing so gives the parabolic intensities $s = -\dfrac{b_2 - a_2}{(x_2 - a_1)^2}$ and $t = -\dfrac{b_2 - c_2}{(x_2 - c_1)^2}$, so that

$$\begin{aligned} f_1(x) &= \left(1 + \frac{x_2 - x}{x_2 - a_1}\right)\left(\frac{x - a_1}{x_2 - a_1}\right)(b_2 - a_2) + a_2 \\ f_2(x) &= \left(1 + \frac{x_2 - x}{x_2 - c_1}\right)\left(\frac{x - c_1}{x_2 - c_1}\right)(b_2 - c_2) + c_2 \end{aligned} \tag{22}$$

The resulting interpolation is shown in Figure 11.

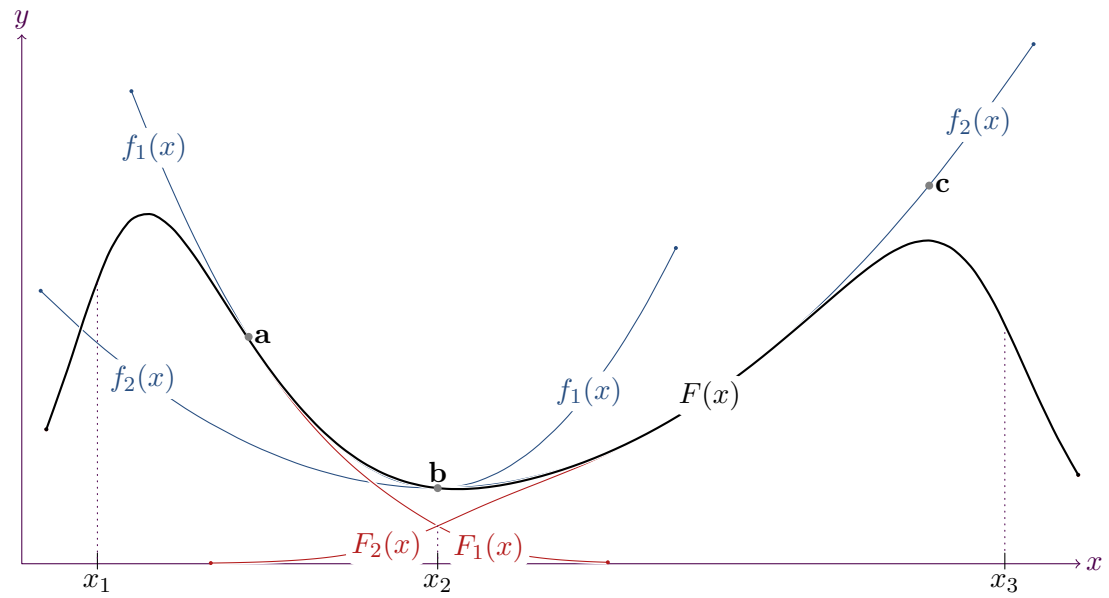


Figure 11: A satisfactory plateau function interpolation (F in (20)) of a "lobsided parabola". The two parabolas f_1 and f_2 in (22) have been blended in a neighbourhood of the point x_2. The accuracy parameter is $\epsilon = 0.001$.

4 Implementation—Computed drawing with TEX, Ti*k*Z and some C

PERHAPS NOT suprisingly, the text in the document was typeset with TEX. The figures were typeset with Ti*k*Z. Ti*k*Z is software capability for typesetting graphical content directly in TEX. Much of the specification and calculation of the diagram was 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 $\mathbb{R}^n$.

To typeset a figure, the TEX source file for this document `\input{}`'s another external TEX source file. In the case of Figure 1, the file was named `simplefuncs.tex`. The file contains the Ti*k*Z source code instructions to typeset the figure. The file was generated dynamically as the output of the execution of the `simplefuncs.run` program, which in turn was created by compiling the C code located in the `simplefuncs.c` file. See below for a listing of the `simplefuncs.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.

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.[8] PKTECHDOC makes it possible to closely juxtapose TEX code and non-TEX code both for typesetting and for compilation outside of TEX.

4.1 Typesetting the figures with TEX and Ti*k*Z

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

```
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{pktechdoc}
\usepackage{pkdocument}
```

```
%\usepackage{pkshell}
%\usepackage[color]{showkeys}
\usepackage[margin=5ex]{caption}
\usepackage{tikz}
\usepackage[obeyspaces]{url}

\usetikzlibrary{calc}
\usetikzlibrary{arrows,positioning,intersections,backgrounds,fit}

\input{defs.tex}
```

4.1.1 The `defs.tex` file

TEX macro definitions applicable over the whole document were placed in the `defs.tex` file. An annotated listing of the file follows:

```
1  \newcommand*\ud{\text d}
2  \newcommand*\deriv[2]{\frac{\ud #1}{\ud #2}}
3  \newcommand*\myHlf{{\textstyle{\frac{1}{2}}}}
4  \newcommand*\myFiveEighth{{\textstyle{\frac{5}{8}}}}
5  \newcommand*\abs[1]{\protect\left|#1\protect\right|}
```

`\undr` For reasons which escape me, the use of the '_' character when in math mode in files which are processed with pkTechDoc ends up being treated as a '^'. The `\undr` macro addresses that problem.

```
6  \newcommand*\undr{_} % Needed in the ".c" source files.
7
8  \newcommand*\realSet{\mathbb{R}}
9  \newcommand*\naturalSet{\mathbb{N}}
10
11 \makeatletter
12 \newcommand*\@interval[4]{#1#2,#3#4}
13 \newcommand*\closedInterval[2]{\@interval{[}{#1}{#2}{]}}
14 \newcommand*\openInterval[2]{\@interval{(}{#1}{#2}{)}}
15 \makeatother
16 \newcommand*\domainInterval{\closedInterval{x\undr{1}}{x\undr{N}}}
17 \newcommand*\subInterval[1]{\closedInterval{x\undr{#1}}{x\undr{#1+1}}}
18 \newcommand*\openSubInterval[1]{\openInterval{x\undr{#1}}{x\undr{#1+1}}}
19
20 \newcommand*\one{\pktikzBasisVector{1}}
21 \newcommand*\two{\pktikzBasisVector{2}}
22 \newcommand*\vecp{\pktikzVector{p}}
23 \newcommand*\vecq{\pktikzVector{q}}
24 \newcommand*\veca{\pktikzVector{a}}
25 \newcommand*\vecb{\pktikzVector{b}}
26 \newcommand*\vecc{\pktikzVector{c}}
27 \newcommand*\vecd{\pktikzVector{d}}
28
29 \newcommand*\upFunc{\text{up}}
30 \newcommand*\doFunc{\text{do}}
31 \newcommand*\udFunc{\text{ud}} % Not to be confused with '\ud'.
32
33 \makeatletter
34 \def\@iwpfEmpty{}
```

Some specific shorthands begin.

`\@funcAtX` The `\@funcAtX` private macro is an encapsulated way of typesetting a function to be evaluated at one of the base points in the $[x_1, x_N]$ domain interval. So the private call `$\@funcAtX{f}{m}$` would expand to `$f(x_m)$` which would be typeset as $f(x_m)$.

```
\newcommand*\@funcAtX[2]{%
   \begingroup%
      \def\@funcName{#1}%
      \def\@xIndex{#2}%
      \ifx\@funcName\@iwpfEmpty\relax%
         \errmessage{\string\@funcAtX: ERROR: A function name has not been supplied}%
      \else%
         \ifx\@xIndex\@iwpfEmpty%
            \@funcName%
         \else%
            \@funcName(x\undr{\@xIndex})%
         \fi\relax%
      \fi\relax%
   \endgroup}
```

`\indexedFuncAtX` The `\indexedFuncAtX` public macro is an encapsulated way of typesetting an indexed function to be evaluated at one of the base points in the $[x_1, x_N]$ domain interval. So the call `$\indexedFuncAtX{\udFunc}{n}{m}$` would expand to `$\udFunc_n(x_m)$` which, given the meaning of `\udFunc`, would be typeset as $\mathrm{ud}_n(x_m)$.

```
\newcommand*\indexedFuncAtX[3]{%
   \begingroup%
      \def\@funcName{#1}%
      \def\@funcIndex{#2}%
      \def\@xIndex{#3}%
      \ifx\@funcName\@iwpfEmpty\relax%
         \errmessage{\string\indexedFuncAtX: ERROR: A function name has not been supplied
      \fi%
      \ifx\@funcIndex\@iwpfEmpty\relax%
         \errmessage{\string\indexedFuncAtX: ERROR: A function index has not been supplie
      \fi%
      \@funcAtX{#1\undr{#2}}{#3}%
   \endgroup}
\makeatother

\newcommand*\simpleAt[2]{\indexedFuncAtX{f}{#1}{#2}}
\newcommand*\plateauAt[2]{\indexedFuncAtX{F}{#1}{#2}}
\newcommand*\udAt[2]{\indexedFuncAtX{\udFunc}{#1}{#2}}
\newcommand*\basePntDiff[2]{{x\undr{#1}-x\undr{#2}}}
\newcommand*\upAtDiff[2]{\upFunc(\basePntDiff{#1}{#2})}
\newcommand*\doAtDiff[2]{\doFunc(\basePntDiff{#1}{#2})}
\newcommand*\simpleRatioAt[3]{\frac{\simpleAt{#1}{#2}}{\simpleAt{#1}{#3}}}
\newcommand*\simpleRatioBAt[3]{\frac{\simpleAt{#1}{#3}}{\simpleAt{#2}{#3}}}
\newcommand*\absSimpleRatioAt[3]{\abs{\simpleRatioAt{#1}{#2}{#3}}}
\newcommand*\udRatioBAt[3]{\frac{\udAt{#1}{#3}}{\udAt{#2}{#3}}}

\newcommand*\myBeta{\beta}
\newcommand*\betaSubSup[2]{\myBeta_{#1}^{#2}}
\newcommand*\betanSubSupDiff[3]{\betaSubSup{#1}{\basePntDiff{#2}{#3}}}
\newcommand*\betanSubSupNegDiff[3]{\betaSubSup{#1}{-(\basePntDiff{#2}{#3})}}

\newcommand*\betan{\myBeta_{n}}
\newcommand*\betanSup[1]{\betaSubSup{n}{#1}}
\newcommand*\betanSupDiff[2]{\betaSubSup{n}{\basePntDiff{#1}{#2}}}
\newcommand*\betanSupNegDiff[2]{\betaSubSup{n}{-(\basePntDiff{#1}{#2})}}

```

```
\newcommand*\twoOverEps[1]{\frac{2}{\epsilon\undr{#1}}}
\newcommand*\twoOverEpsn{\twoOverEps{n}}
```

Some specific shorthands end.

\tikzset Stuff for Ti*k*Z begins.

```
\definecolor{basiscolor}{rgb}{0.3,0.0,0.3}
\definecolor{simplefunctioncolor}{rgb}{0.15,0.3,0.5}
\definecolor{updownfunctioncolor}{rgb}{0.7,0.1,0.1}
\definecolor{plateaufunctioncolor}{rgb}{0.7,0.1,0.1}
\definecolor{plateauapproxcolor}{rgb}{0.0,0.0,0.0}
\definecolor{positionvectorcolor}{rgb}{0.0,0.0,0.0}

\tikzset{%line cap=round,
         %line join=bevel,
         %inner sep=2pt,
         opaquelabel/.style={fill=white,rounded corners},
         %
         function/.style={pktikzshadowed},
         functionpoint/.style={pktikzpoint,minimum size=1.5pt},
         simplefunction/.style={function,color=simplefunctioncolor},
         simplefunctionpoint/.style={functionpoint,color=simplefunctioncolor},
         updownfunction/.style={function,color=updownfunctioncolor},
         updownfunctionpoint/.style={functionpoint,color=updownfunctioncolor},
         plateaufunction/.style={function,color=plateaufunctioncolor},
         plateaufunctionpoint/.style={functionpoint,color=plateaufunctioncolor},
         plateauapprox/.style={function,thick,color=plateauapproxcolor},
         plateauapproxpoint/.style={functionpoint,color=plateauapproxcolor}}
```

Stuff for Ti*k*Z ends.

4.2 Source code listings and the PKREALVECTOR C object class

The `simplefuncs.tex` file contains Ti*k*Z code for Figure 1. The `simplefuncs.tex` file was incorporated into the body of the text with an `\input{}` TEX command, as follows:

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

4.2.1 The simplefuncs.c file

A listing of the `simplefuncs.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 "common.h"
#include "simplefunc.h"
```

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 (15.1,7.1);");
   puts(   "   %");
   puts(   "   % Some coordinates.");
   puts(   "   %");
   puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
   return;
}
```

The `_drawSomeLabelling()` private function simply `printf()`s to standard output a body of Ti*k*Z source code which may be used to typeset some labelling in the figure.

```
static void _drawSomeLabelling( const SIMPLEFUNC *f1,
                                const SIMPLEFUNC *f2,
                                const SIMPLEFUNC *f3,
                                const SIMPLEFUNC *f4 )
{
   const int points = 5;
   const PKMATHREAL x1 = pkRealVectorGetComponent(simpleFuncGetP(f1))[0],
                    x2 = pkRealVectorGetComponent(simpleFuncGetP(f2))[0],
                    x3 = pkRealVectorGetComponent(simpleFuncGetP(f3))[0],
                    x4 = pkRealVectorGetComponent(simpleFuncGetP(f4))[0],
                    x5 = pkRealVectorGetComponent(simpleFuncGetQ(f4))[0];
   /*
    * The 'p[points] was changed to 'p[]' following a compiler
    * warning. PJ Kotschy. 31Jan25.
    */
   //const struct {
   //   const PKMATHREAL x;
   //   const PKMATHREAL y;
   //   const char *axisLabel;
   //} p[points] = { { x1, simpleFuncAt(f1,x1), "x_1" },
   //                { x2, simpleFuncAt(f2,x2), "x_{n-1}" },
   //                { x3, simpleFuncAt(f3,x3), "x_n" },
   //                { x4, simpleFuncAt(f4,x4), "x_{n+1}" },
   //                { x5, simpleFuncAt(f4,x5), "x_N" } };
   const struct {
      const PKMATHREAL x;
      const PKMATHREAL y;
      const char *axisLabel;
   } p[] = { { x1, simpleFuncAt(f1,x1), "x_1" },
             { x2, simpleFuncAt(f2,x2), "x_{n-1}" },
             { x3, simpleFuncAt(f3,x3), "x_n" },
```

```
61              { x4, simpleFuncAt(f4,x4), "x_{n+1}" },
62              { x5, simpleFuncAt(f4,x5), "x_N" } };
63     /*
64      * The '*f[points-1]' was changed to '*f[]' following
65      * a compiler warning. PJ Kotschy. 31Jan25.
66      */
67     //const SIMPLEFUNC *f[points-1] = { f1, f2, f3, f4 };
68     const SIMPLEFUNC *f[] = { f1, f2, f3, f4 };
69     int i;
70
71     puts( "   %");
72     puts( "   % Some labelling begins.");
73     puts( "   %");
74     puts( "   % Labelling on the x-axis.");
75     puts( "   %");
76     puts( "   \\draw");
77     for ( i = 0; i < points; i++ )
78        printf( "      (" FLTFMT ",0) +(0,-2pt) node[below]{$%s$} -- +(0,4pt)%s\n",
79                p[i].x,
80                p[i].axisLabel,
81                ( i == points - 1 ) ? ";" : "" );
82     puts( "   \\draw[pktikzdimension]");
83     for ( i = 0; i < points; i++ )
84        printf( "      (" FLTFMT ",0) +(0,2pt) -- (" FLTFMT "," FLTFMT ")%s\n",
85                p[i].x,
86                p[i].x,
87                p[i].y,
88                ( i == points - 1 ) ? ";" : "" );
89
90     puts( "   %");
91     puts( "   % The functions.");
92     puts( "   %");
93     puts( "   \\path");
94     for ( i = 0; i < points - 1; i++ ) {
95        printf( "      (" FLTFMT "," FLTFMT ")"
96                " node[simplefunctioncolor,opaquelabel,above=0.5ex]{$%s$}%s\n",
97                0.5 * ( p[i].x + p[i+1].x ),
98                simpleFuncAt( f[i], 0.5 * ( p[i].x + p[i+1].x ) ),
99                simpleFuncGetName(f[i]),
100               ( i == points - 1 - 1 ) ? ";" : "" );
101    }
102    puts( "   %");
103    puts( "   % Some labelling ends.");
104    puts( "   %");
105
106    return;
107 }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
108 static void _diagram(void)
109 {
110    const PKMATHREAL x1 = 1.0,
111                     xnM1 = 4.0,
112                     xn = 6.0,
113                     xnP1 = 11.0,
114                     xN = 14.0;
115    PKREALVECTOR *e1,
```

```
116                   *e2;
117     SIMPLEFUNC *f1,
118                *fnM1,
119                *fn,
120                *fNM1;
121
```

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

```
122     e1 = pkRealVectorAlloc1( "x", 2, 15.0, 0.0 );
123     e2 = pkRealVectorAlloc1( "y", 2, 0.0,  7.0 );
124     //pkRealVectorScale(e1,1.2);
125     //pkRealVectorScale(e2,1.2);
126
127     f1 = simpleFuncAlloc1( "f_1(x)",
128                            exponential,
129                            1.0, 5.0,
130                            4.0, 2.0,
131                            0.5 );
132     fnM1 = simpleFuncAlloc1( "f_{n-1}(x)",
133                              exponential,
134                              4.0, 1.5,
135                              6.0, 3.0,
136                              -1.0 );
137     fn = simpleFuncAlloc1( "f_n(x)",
138                            quadratic,
139                            6.0,  2.0,
140                            11.0, 3.0,
141                            -0.6 );
142     fNM1 = simpleFuncAlloc1( "f_{N-1}(x)",
143                              quadratic,
144                              11.0, 1.8,
145                              14.0, 4.0,
146                              1.0 );
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
147     puts(   "\\begin{PkTikzpicture}[scale=1.0]");
148     _drawCoordinates();
149     drawBasisVectors(e1,e2);
150     drawInterestFunction();
151     simpleFuncDraw( f1,   x1,   xnM1, 20 );
152     simpleFuncDraw( fnM1, xnM1, xn,   20 );
153     simpleFuncDraw( fn,   xn,   xnP1, 20 );
154     simpleFuncDraw( fNM1, xnP1, xN,   20 );
155     _drawSomeLabelling( f1, fnM1, fn, fNM1 );
156     puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
157     pkRealVectorFree1(e1);
158     pkRealVectorFree1(e2);
159     simpleFuncFree1(f1);
160     simpleFuncFree1(fnM1);
161     simpleFuncFree1(fn);
162     simpleFuncFree1(fNM1);
163
164     return;
165 }
```

```
166  int main( const int argc, const char *argv[] )
167  {
168     _diagram();
169     //memPrintf();
170     exit(0);
171  }
```

4.2.2 The `plateaufuncs.c` file

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

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "common.h"
20  #include "simplefunc.h"
21  #include "plateaufunc.h"
```

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

```
22  static void _drawCoordinates(void)
23  {
24     puts(   "   %\\draw[help lines] (-0.2,-0.2) grid (15.1,7.1);");
25     puts(   "   %");
26     puts(   "   % Some coordinates.");
27     puts(   "   %");
28     puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
29     return;
30  }
```

The `_drawSomeLabelling()` private function simply `printf()`s to standard output a body of Ti*k*Z source code which may be used to typeset some labelling in the figure.

```
31  static void _drawSomeLabelling( const PLATEAUFUNC *F1,
32                                  const PLATEAUFUNC *F2,
33                                  const PLATEAUFUNC *F3,
34                                  const PLATEAUFUNC *F4 )
35  {
36     const int points = 5;
37     const PKMATHREAL x1 = pkRealVectorGetComponent(
38                              simpleFuncGetP(
39                                 plateauFuncGetSimpleFunc(F1)))[0],
40                      x2 = pkRealVectorGetComponent(
41                              simpleFuncGetP(
42                                 plateauFuncGetSimpleFunc(F2)))[0],
43                      x3 = pkRealVectorGetComponent(
44                              simpleFuncGetP(
45                                 plateauFuncGetSimpleFunc(F3)))[0],
46                      x4 = pkRealVectorGetComponent(
47                              simpleFuncGetP(
```

```
                                plateauFuncGetSimpleFunc(F4)))[0],
                    x5 = pkRealVectorGetComponent(
                            simpleFuncGetQ(
                                plateauFuncGetSimpleFunc(F4)))[0];
    /*
     * The 'p[points] was changed to 'p[]' following a compiler
     * warning. PJ Kotschy. 31Jan25.
     */
    //const struct {
    //   const PKMATHREAL x;
    //   const PKMATHREAL y;
    //   const char *axisLabel;
    //} p[points] = { { x1, plateauFuncAt(F1,x1), "x_1" },
    //                { x2, plateauFuncAt(F1,x2), "x_{n-1}" },
    //                { x3, plateauFuncAt(F2,x3), "x_n" },
    //                { x4, plateauFuncAt(F3,x4), "x_{n+1}" },
    //                { x5, plateauFuncAt(F4,x5), "x_N" } };
    const struct {
       const PKMATHREAL x;
       const PKMATHREAL y;
       const char *axisLabel;
    } p[] = { { x1, plateauFuncAt(F1,x1), "x_1" },
              { x2, plateauFuncAt(F1,x2), "x_{n-1}" },
              { x3, plateauFuncAt(F2,x3), "x_n" },
              { x4, plateauFuncAt(F3,x4), "x_{n+1}" },
              { x5, plateauFuncAt(F4,x5), "x_N" } };
    /*
     * The '*f[points-1]' was changed to '*f[]' following
     * a compiler warning. PJ Kotschy. 31Jan25.
     */
    //const PLATEAUFUNC *f[points-1] = { F1, F2, F3, F4 };
    const PLATEAUFUNC *f[] = { F1, F2, F3, F4 };
    int i;

    puts( "   %");
    puts( "   % Some labelling begins.");
    puts( "   %");
    puts( "   % Labelling on the x-axis.");
    puts( "   %");
    puts( "   \\draw");
    for ( i = 0; i < points; i++ )
       printf( "      (" FLTFMT ",0) +(0,-2pt) node[below]{$%s$} -- +(0,4pt)%s\n",
               p[i].x,
               p[i].axisLabel,
               ( i == points - 1 ) ? ";" : "" );
    puts( "   \\draw[pktikzdimension]");
    for ( i = 0; i < points; i++ )
       printf( "      (" FLTFMT ",0) +(0,2pt) -- (" FLTFMT "," FLTFMT ")%s\n",
               p[i].x,
               p[i].x,
               p[i].y,
               ( i == points - 1 ) ? ";" : "" );

    puts( "   %");
    puts( "   % The function labels.");
    puts( "   %");
    puts( "   \\path");
    for ( i = 0; i < points - 1; i++ ) {
       printf( "      (" FLTFMT "," FLTFMT ")"
               " node[plateaufunctioncolor,opaquelabel,above=1ex]{$%s$}%s\n",
               0.5 * ( p[i].x + p[i+1].x ),
               plateauFuncAt( f[i], 0.5 * ( p[i].x + p[i+1].x ) ),
```

```
110                plateauFuncGetName(f[i]),
111                ( i == points - 1 - 1 ) ? ";" : "" );
112     }
113     puts( "   %");
114     puts( "   % Some labelling ends.");
115     puts( "   %");
116
117     return;
118   }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
119   static void _diagram(void)
120   {
121     const PKMATHREAL x1 = 1.0,
122                      xnM1 = 4.0,
123                      xn = 6.0,
124                      xnP1 = 11.0,
125                      xN = 14.0;
126     PKREALVECTOR *e1,
127                  *e2;
128     SIMPLEFUNC *f;
129     PLATEAUFUNC *F1,
130                 *FnM1,
131                 *Fn,
132                 *FNM1;
133
```

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

```
134     e1 = pkRealVectorAlloc1( "x", 2, 15.0, 0.0 );
135     e2 = pkRealVectorAlloc1( "y", 2, 0.0,  7.0 );
136     //pkRealVectorScale(e1,1.2);
137     //pkRealVectorScale(e2,1.2);
138
139     f = simpleFuncAlloc1( "f_1(x)",
140                           exponential,
141                           x1, 5.0,
142                           xnM1, 2.0,
143                           0.5 );
144     F1 = plateauFuncAlloc( "F_1(x)", f, 100.0, x1, xnM1 );
145     simpleFuncFree1(f);
146
147     f = simpleFuncAlloc1( "f_{n-1}(x)",
148                           exponential,
149                           xnM1, 1.5,
150                           xn, 3.0,
151                           -1.0 );
152     FnM1 = plateauFuncAlloc( "F_{n-1}(x)", f, 100.0, xnM1, xn );
153     simpleFuncFree1(f);
154
155     f = simpleFuncAlloc1( "f_n(x)",
156                           quadratic,
157                           xn,  2.0,
158                           xnP1, 3.0,
159                           -0.6 );
```

```
160     Fn = plateauFuncAlloc( "F_n(x)=\\udFunc_n(x)f_n(x)", f, 100.0, xn, xnP1 );
161     simpleFuncFree1(f);
162
163     f = simpleFuncAlloc1( "f_{N-1}(x)",
164                           quadratic,
165                           xnP1, 1.8,
166                           xN, 4.0,
167                           1.0 );
168     FNM1 = plateauFuncAlloc( "F_{N-1}(x)", f, 100.0, xnP1, xN );
169     simpleFuncFree1(f);
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
170     puts(   "\\begin{PkTikzpicture}[scale=1.0]");
171     _drawCoordinates();
172     drawBasisVectors(e1,e2);
173     drawInterestFunction();
174     plateauFuncDraw( F1,   bitLess( x1, xnM1, 0.2 ), bitMore( xnM1, x1, 0.2 ), 40 );
175     plateauFuncDraw( FnM1, bitLess( xnM1, x1, 0.2 ), bitMore( xn, xnP1, 0.2 ), 40 );
176     plateauFuncDraw( Fn,   bitLess( xn, xnM1, 0.2 ), bitMore( xnP1, xN, 0.2 ), 40 );
177     plateauFuncDraw( FNM1, bitLess( xnP1, xn, 0.2 ), bitMore( xN, xnP1, 0.2 ), 40 );
178     _drawSomeLabelling( F1, FnM1, Fn, FNM1 );
179     puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
180     pkRealVectorFree1(e1);
181     pkRealVectorFree1(e2);
182     plateauFuncFree(F1);
183     plateauFuncFree(FnM1);
184     plateauFuncFree(Fn);
185     plateauFuncFree(FNM1);
186
187     return;
188 }


189 int main( const int argc, const char *argv[] )
190 {
191     _diagram();
192     //memPrintf();
193     exit(0);
194 }
```

4.2.3 The `simplefunc.h` and `simplefunc.c` files

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

```
1   #ifndef _SIMPLEFUNC
2   #define _SIMPLEFUNC
```

Inclusions.

```
3   #include <pkfeatures.h>
4
5   #include <stdlib.h>
6   #include <stdarg.h>
7   #include <stdio.h>
8   #include <string.h>
9   #include <sys/stat.h>
10  #include <sys/types.h>
11  #include <unistd.h>
12  #include <math.h>
13  #include <float.h>
14
15  #include <pkmemdebug.h>
16  #include <pkerror.h>
17  #include <pktypes.h>
18  #include <pkmath.h>
19  #include <pkrealvector.h>
```

Type definitions.

```
20  typedef PKMATHREAL (*SIMPLEFUNCTION)( const PKMATHREAL x,
21                                        const PKMATHREAL p1,
22                                        const PKMATHREAL p2,
23                                        const PKMATHREAL q1,
24                                        const PKMATHREAL q2,
25                                        const PKMATHREAL strength );
26  typedef struct SimpleFuncs {
27     char *name;
28     SIMPLEFUNCTION func;
29     PKREALVECTOR *p,
30                  *q;
31     PKMATHREAL strength;
32  } SIMPLEFUNC;
```

Function declarations.

```
33  extern SIMPLEFUNC *simpleFuncAlloc0( const char *name,
34                                       const SIMPLEFUNCTION func,
35                                       const PKREALVECTOR *p,
36                                       const PKREALVECTOR *q,
37                                       const PKMATHREAL strength );
38  extern void simpleFuncFree0( SIMPLEFUNC *sf );
39  extern SIMPLEFUNC *simpleFuncAlloc1( const char *name,
40                                       const SIMPLEFUNCTION func,
41                                       const PKMATHREAL p1,
42                                       const PKMATHREAL p2,
43                                       const PKMATHREAL q1,
44                                       const PKMATHREAL q2,
45                                       const PKMATHREAL strength );
```

```
46  extern void simpleFuncFree1( SIMPLEFUNC *sf );
47  extern SIMPLEFUNC *simpleFuncAllocCopy( const SIMPLEFUNC *sf );
48  extern void simpleFuncFreeCopy( SIMPLEFUNC *sf );
49  extern char *simpleFuncGetName( const SIMPLEFUNC *sf );
50  extern SIMPLEFUNCTION simpleFuncGetFunc( const SIMPLEFUNC *sf );
51  extern PKREALVECTOR *simpleFuncGetP( const SIMPLEFUNC *sf );
52  extern PKREALVECTOR *simpleFuncGetQ( const SIMPLEFUNC *sf );
53  extern PKMATHREAL simpleFuncGetStrength( const SIMPLEFUNC *sf );
54  extern PKMATHREAL simpleFuncAt( const SIMPLEFUNC *sf, const PKMATHREAL x );
55  extern void simpleFuncDraw( const SIMPLEFUNC *sf,
56                              const PKMATHREAL x1,
57                              const PKMATHREAL x2,
58                              const int samples );

59  #endif
```

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

```
1   #include "simplefunc.h"
2
3   #include <pkfeatures.h>
4
5   #include <sys/types.h>
6   #include <stdlib.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <unistd.h>
10  #include <math.h>
11  #include <float.h>
12
13  #include <pkmemdebug.h>
14  #include <pkerror.h>
15  #include <pktypes.h>
16  #include <pkstring.h>
17  #include <pkmath.h>
18  #include <pkrealvector.h>
19
20  #include "common.h"
```

The `simpleFuncAlloc0()` allocates and initialises a `SIMPLEFUNC` struct to encapsulate the details of a "simple" function. In the specified argument list, `func`, `p`, `q` and `stength` correspond to f, $\mathbf{p}$, $\mathbf{q}$ and s in Equation (1).

On success return a pointer to the allocated and initialised `SIMPLEFUNC`. Otherwise return `(SIMPLEFUNC *)NULL`. The conditions `!func`, `!p` and `!q` are considered input errors.

NB NOTE: Must be accompanied by a call to `simpleFuncFree0()`.

```
21  SIMPLEFUNC *simpleFuncAlloc0( const char *name,
22                                const SIMPLEFUNCTION func,
23                                const PKREALVECTOR *p,
24                                const PKREALVECTOR *q,
25                                const PKMATHREAL strength )
26  {
27     SIMPLEFUNC *sf;
28     PKREALVECTOR *pb,    /* Local clone of 'p'. */
29                  *qb;    /* Local clone of 'q'. */
30
31     if ( !func || !p || !q )
32        return( (SIMPLEFUNC *)NULL );
33
34     pb = pkRealVectorAlloc0( pkRealVectorGetName(p),
35                              pkRealVectorGetComponents(p),
36                              pkRealVectorGetComponent(p) );
37     if ( !pb )
38        return( (SIMPLEFUNC *)NULL );
39     qb = pkRealVectorAlloc0( pkRealVectorGetName(q),
40                              pkRealVectorGetComponents(q),
41                              pkRealVectorGetComponent(q) );
42     if ( !qb ) {
43        pkRealVectorFree0(pb);
44        return( (SIMPLEFUNC *)NULL );
45     }
46
47     sf = (SIMPLEFUNC *)calloc( 1, sizeof(SIMPLEFUNC) );
48     if (sf) {
49        sf->name = strAllocInit(name);
```

```
50          sf->func = func;
51          sf->p = pb;
52          sf->q = qb;
53          sf->strength = strength;
54       } else {
55          pkRealVectorFree0(pb);
56          pkRealVectorFree0(qb);
57       }
58
59       return(sf);
60    }
```

`simpleFuncFree0()` is the complement to `simpleFuncAlloc0()`.

```
61    void simpleFuncFree0( SIMPLEFUNC *sf )
62    {
63       if (sf) {
64          strFreeInit(sf->name);
65          sf->func = (SIMPLEFUNCTION)NULL;  /* For good measure. */
66          pkRealVectorFree0(sf->p);
67          sf->p = (PKREALVECTOR *)NULL;     /* For good measure. */
68          pkRealVectorFree0(sf->q);
69          sf->q = (PKREALVECTOR *)NULL;     /* For good measure. */
70          sf->strength = 0.0;               /* For good measure. */
71          free(sf);
72       }
73       return;
74    }
```

The `simpleFuncAlloc1()` allocates and initialises a `SIMPLEFUNC` struct to encapsulate the details of a "simple" function. The function is identical to `simpleFuncAlloc0()` except that the two components for the **p** and **q** positions are given explicitly.

On success return a pointer to the allocated and initialised `SIMPLEFUNC`. Otherwise return `(SIMPLEFUNC *)NULL`.

NB NOTE: Must be accompanied by a call to `simpleFuncFree1()`.

```
75    SIMPLEFUNC *simpleFuncAlloc1( const char *name,
76                                  const SIMPLEFUNCTION func,
77                                  const PKMATHREAL p1,
78                                  const PKMATHREAL p2,
79                                  const PKMATHREAL q1,
80                                  const PKMATHREAL q2,
81                                  const PKMATHREAL strength )
82    {
83       SIMPLEFUNC *sf;
84       PKREALVECTOR *p;
85
86       if ( !func )
87          return( (SIMPLEFUNC *)NULL );
88
89       /*
90        * Error by default.
91        */
92       sf = (SIMPLEFUNC *)NULL;
93
94       p = pkRealVectorAlloc1( "p", 2, p1, p2 );
95       if (p) {
96          PKREALVECTOR *q = pkRealVectorAlloc1( "q", 2, q1, q2 );
```

```
97          if (q) {
98             sf = simpleFuncAlloc0( name, func, p, q, strength );
99             pkRealVectorFree1(q);
100         }
101         pkRealVectorFree1(p);
102      }
103
104      return(sf);
105   }
```

`simpleFuncFree1()` is the complement to `simpleFuncAlloc1()`.

```
106   void simpleFuncFree1( SIMPLEFUNC *sf )
107   {
108      if (sf)
109         simpleFuncFree0(sf);
110      return;
111   }
```

The `simpleFuncAllocCopy()` function allocates and initialises a `SIMPLEFUNC` struct to encapsulate the details of a "simple" function. It does this by creating a copy of the `SIMPLEFUNC` pointed to by the specified `sf`. And it does that by returning the result of an appropriately crafted call to `simpleFuncAlloc0()`.

On success return a pointer to the allocated and initialised `SIMPLEFUNC`. Otherwise return `(SIMPLEFUNC *)NULL`. The condition `sf` is considered an input error.

NB NOTE: Must be accompanied by a call to `simpleFuncFreeCopy()`.

```
112   SIMPLEFUNC *simpleFuncAllocCopy( const SIMPLEFUNC *sf )
113   {
114      if ( !sf )
115         return( (SIMPLEFUNC *)NULL );
116      return( simpleFuncAlloc0( sf->name,
117                                sf->func,
118                                sf->p,
119                                sf->q,
120                                sf->strength ) );
121   }
```

`simpleFuncFreeCopy()` is the complement to `simpleFuncAllocCopy()`.

```
122   void simpleFuncFreeCopy( SIMPLEFUNC *sf )
123   {
124      if (sf)
125         simpleFuncFree0(sf);
126      return;
127   }
```

On success the `simpleFuncGetName()` function simply returns `sf->name`. Otherwise it returns `(char *)NULL`. The condition `!sf` is considered an input error.

```
128   char *simpleFuncGetName( const SIMPLEFUNC *sf )
129   {
130      if ( !sf )
131         return( (char *)NULL );
132      return( sf->name );
133   }
```

On success the `simpleFuncGetFunc()` function simply returns `sf->func`. Otherwise it returns `(SIMPLEFUNCTION)NULL`. The condition `!sf` is considered an input error.

```
134  SIMPLEFUNCTION simpleFuncGetFunc( const SIMPLEFUNC *sf )
135  {
136     if ( !sf )
137        return( (SIMPLEFUNCTION)NULL );
138     return( sf->func );
139  }
```

On success the `simpleFuncGetP()` function returns the `sf->p (PKREALVECTOR *)`. Otherwise it returns `(PKREALVECTOR *)NULL`. The condition `!sf` is considered an input error.

```
140  PKREALVECTOR *simpleFuncGetP( const SIMPLEFUNC *sf )
141  {
142     if ( !sf )
143        return( (PKREALVECTOR *)NULL );
144     return( sf->p );
145  }
```

On success the `simpleFuncGetQ()` function returns the `sf->q (PKREALVECTOR *)`. Otherwise it returns `(PKREALVECTOR *)NULL`. The condition `!sf` is considered an input error.

```
146  PKREALVECTOR *simpleFuncGetQ( const SIMPLEFUNC *sf )
147  {
148     if ( !sf )
149        return( (PKREALVECTOR *)NULL );
150     return( sf->q );
151  }
```

On success the `simpleFuncGetStrength()` function returns the `sf->strength` field. Otherwise it returns 0.0. The condition `!sf` is considered an input error.

```
152  PKMATHREAL simpleFuncGetStrength( const SIMPLEFUNC *sf )
153  {
154     if ( !sf )
155        return(0.0);
156     return( sf->strength );
157  }
```

The `simpleFuncAt()` function calls the "simple function" associated with the `SIMPLEFUNC` pointed to by the specified `sf`. The simple function is evaluated at the specified x value. What this function actually does is to place the call

```
(*(simpleFuncGetFunc(sf)))( x,
                            pkRealVectorGetComponent(simpleFuncGetP(sf))[0],
                            pkRealVectorGetComponent(simpleFuncGetP(sf))[1],
                            pkRealVectorGetComponent(simpleFuncGetQ(sf))[0],
                            pkRealVectorGetComponent(simpleFuncGetQ(sf))[1],
                            simpleFuncGetStrength(sf) ) );
```

To better understand this call, refer to Equation (1).

On success the function returns with the result of the abovementioned call. Otherwise it returns 0.0. The condition `!sf` is considered an input error.

```
158  PKMATHREAL simpleFuncAt( const SIMPLEFUNC *sf, const PKMATHREAL x )
159  {
160     if ( !sf )
161        return(0.0);
162     return( (*(simpleFuncGetFunc(sf)))( x,
163                                         pkRealVectorGetComponent(simpleFuncGetP(sf))[0],
164                                         pkRealVectorGetComponent(simpleFuncGetP(sf))[1],
165                                         pkRealVectorGetComponent(simpleFuncGetQ(sf))[0],
166                                         pkRealVectorGetComponent(simpleFuncGetQ(sf))[1],
167                                         simpleFuncGetStrength(sf) ) );
168  }
```

The `simpleFuncDraw()` function simply `printf()`s to standard output a body of TikZ source code which may be used to typeset the function referred to by the specified `SIMPLEFUNC`, `sf`, and pointed to by `simpleFuncGetFunc(sf)`. The function is evaluated indirectly at some position `x` via a `simpleFuncAt(sf,x)` call. The function's domain interval of interest is the specified interval $[x_1, x_2]$.

```
169  void simpleFuncDraw( const SIMPLEFUNC *sf,
170                       const PKMATHREAL x1,
171                       const PKMATHREAL x2,
172                       const int samples )
173  {
174     PKMATHREAL x;
175     int i;
176
177     if ( !sf  ||  x1 >= x2  ||  samples < 2 )
178        return;
179
180     puts(   "   %");
181     printf( "   %% A simple function '%s' over the "
182             "[" FLTFMT "," FLTFMT "] domain interval.\n",
183             simpleFuncGetName(sf),
184             x1, x2 );
185     puts(   "   %");
186     printf( "   \\draw[simplefunction]\n"
187             "      (" FLTFMT "," FLTFMT ") coordinate[simplefunctionpoint]\n",
188             x1, simpleFuncAt(sf,x1) );
189     /*puts(   "      plot[smooth,mark=*,mark size=1pt] coordinates {");*/
190     puts(   "      plot[smooth] coordinates {");
191     for ( i = 1; i <= samples; i++ ) {
192        x = x1 + ( (double)(i-1.0) / (double)(samples-1) ) * ( x2 - x1 );
193        printf( "         (" FLTFMT "," FLTFMT ")\n",
194                x, simpleFuncAt(sf,x) );
195     }
196     puts(   "      } coordinate[simplefunctionpoint];");
197
198     return;
199  }
```

4.2.4 The plateaufunc.h and plateaufunc.c files

A listing of the plateaufunc.h file follows:

```
1   #ifndef _PLATEAUFUNC
2   #define _PLATEAUFUNC
```

Inclusions.

```
3   #include <pkfeatures.h>
4
5   #include <stdlib.h>
6   #include <stdarg.h>
7   #include <stdio.h>
8   #include <string.h>
9   #include <sys/stat.h>
10  #include <sys/types.h>
11  #include <unistd.h>
12  #include <math.h>
13  #include <float.h>
14
15  #include <pkmemdebug.h>
16  #include <pkerror.h>
17  #include <pktypes.h>
18  #include <pkmath.h>
19
20  #include "simplefunc.h"
```

Type definitions.

```
21  typedef struct PlateauFuncs {
22     char *name;
23     SIMPLEFUNC *sf;
24     PKMATHREAL base,     /* The base value used in the 'up()' and */
25                          /* 'do()' functions. */
26                 xUp,     /* Domain point of ascent of the plateau */
27                          /* function. Note that 'xUp < xDown'. */
28                 xDown;   /* Domain point of descent of the plateau function. */
29  } PLATEAUFUNC;
```

Function declarations.

```
30  extern PLATEAUFUNC *plateauFuncAlloc( const char *name,
31                                        const SIMPLEFUNC *sf,
32                                        const PKMATHREAL base,
33                                        const PKMATHREAL xUp,
34                                        const PKMATHREAL xDown );
35  extern void plateauFuncFree( PLATEAUFUNC *pf );
36  extern PLATEAUFUNC *plateauFuncAllocCopy( const PLATEAUFUNC *pf );
37  extern void plateauFuncFreeCopy( PLATEAUFUNC *pf );
38
39  extern char *plateauFuncGetName( const PLATEAUFUNC *pf );
40  extern SIMPLEFUNC *plateauFuncGetSimpleFunc( const PLATEAUFUNC *pf );
41  extern PKMATHREAL plateauFuncGetBase( const PLATEAUFUNC *pf );
42  extern int plateauFuncSetBase( PLATEAUFUNC *pf, const PKMATHREAL base );
43  extern PKMATHREAL plateauFuncGetXUp( const PLATEAUFUNC *pf );
44  extern PKMATHREAL plateauFuncGetXDown( const PLATEAUFUNC *pf );
45  extern PKMATHREAL plateauFuncAt( const PLATEAUFUNC *pf, const PKMATHREAL x );
```

```
46  extern void plateauFuncDraw( const PLATEAUFUNC *pf,
47                               const PKMATHREAL x1,
48                               const PKMATHREAL x2,
49                               const int samples );

50  #endif
```

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

```
1   #include "plateaufunc.h"
2
3   #include <pkfeatures.h>
4
5   #include <sys/types.h>
6   #include <stdlib.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <unistd.h>
10  #include <math.h>
11  #include <float.h>
12
13  #include <pkmemdebug.h>
14  #include <pkerror.h>
15  #include <pktypes.h>
16  #include <pkstring.h>
17  #include <pkmath.h>
18  #include <pkrealvector.h>
19
20  #include "common.h"
21  #include "simplefunc.h"
```

The `plateauFuncAlloc()` function allocates and initialises a `PLATEAUFUNC` struct to encapsulate the details of a "plateau" function. In the argument list `sf`, `base`, `xUp` and `xDown` correspond respectively to f_n, b, x_n and x_{n+1} in Equations (2), (4) and (5).

On success return a pointer to the allocated and initialised `PLATEAUFUNC`. Otherwise return `(PLATEAUFUNC *)NULL`. The conditions `!sf`, `base` $<= 1.0$ and `xUp` $>=$ `xDown` are all considered input errors.

NB NOTE: Must be accompanied by a call to `plateauFuncFree()`.

```
22  PLATEAUFUNC *plateauFuncAlloc( const char *name,
23                                 const SIMPLEFUNC *sf,
24                                 const PKMATHREAL base,
25                                 const PKMATHREAL xUp,
26                                 const PKMATHREAL xDown )
27  {
28     PLATEAUFUNC *pf;
29
30     if ( !sf  ||  base <= 1.0  ||  xUp >= xDown )
31        return( (PLATEAUFUNC *)NULL );
32
33     pf = (PLATEAUFUNC *)calloc( 1, sizeof(PLATEAUFUNC) );
34     if (pf) {
35        pf->name = strAllocInit(name);
36        pf->sf = simpleFuncAllocCopy(sf);
37        pf->base = base;
38        pf->xUp = xUp;
39        pf->xDown = xDown;
40     }
41
42     return(pf);
43  }
```

`plateauFuncFree()` is the complement to `plateauFuncAlloc()`.

```
44  void plateauFuncFree( PLATEAUFUNC *pf )
```

```
45  {
46     if (pf) {
47        strFreeInit(pf->name);
48        pf->name = (char *)NULL;          /* For good measure. */
49        simpleFuncFreeCopy(pf->sf);
50        pf->sf = (SIMPLEFUNC *)NULL;      /* For good measure. */
51        free(pf);
52     }
53     return;
54  }
```

The `plateauFuncAllocCopy()` function allocates and initialises a `PLATEAUFUNC` struct to encapsulate the details of a "plateau" function. It does this by creating a copy of the `PLATEAUFUNC` pointed to by the specified `pf`. On success return a pointer to the allocated and initialised `PLATEAUFUNC`. Otherwise return `(PLATEAUFUNC *)NULL`. The condition `!pf` is considered an input error.

NB NOTE: Must be accompanied by a call to `plateauFuncFreeCopy()`.

```
55  PLATEAUFUNC *plateauFuncAllocCopy( const PLATEAUFUNC *pf )
56  {
57     if ( !pf )
58        return( (PLATEAUFUNC *)NULL );
59     return( plateauFuncAlloc( pf->name,
60                               pf->sf,
61                               pf->base,
62                               pf->xUp,
63                               pf->xDown ) );
64  }
```

`plateauFuncFreeCopy()` is the complement to `plateauFuncAllocCopy()`.

```
65  void plateauFuncFreeCopy( PLATEAUFUNC *pf )
66  {
67     if (pf)
68        plateauFuncFree(pf);
69     return;
70  }
```

On success the `plateauFuncGetName()` function returns `pf->name`. Otherwise on error, it returns `(char *)NULL`. The condition `!pf` is considered an input error.

```
71  char *plateauFuncGetName( const PLATEAUFUNC *pf )
72  {
73     if ( !pf )
74        return( (char *)NULL );
75     return( pf->name );
76  }
```

On success the `plateauFuncGetSimpleFunc()` function returns `pf->sf` (Equation (2)). (Equations (2), (4) and (5)). Otherwise on error, it returns `(SIMPLEFUNC *)NULL`. The condition `!pf` is considered an input error.

```
77  SIMPLEFUNC *plateauFuncGetSimpleFunc( const PLATEAUFUNC *pf )
78  {
79     if ( !pf )
80        return( (SIMPLEFUNC *)NULL );
81     return( pf->sf );
82  }
```

On success the `plateauFuncGetBase()` function returns `pf->base` (Equations (2), (4) and (5)). Otherwise on error, it returns 0.0. The condition `!pf` is considered an input error.

```
83  PKMATHREAL plateauFuncGetBase( const PLATEAUFUNC *pf )
84  {
85     if ( !pf )
86        return(0.0);
87     return( pf->base );
88  }
```

The `plateauFuncSetBase()` function sets `pf-base` to equal the specified `base`. Equations (2), (4) and (5) refer. On success return 1. Otherwise return 0. The conditions `!pf` and `base` ≤ 1 are considered input errors.

```
89  int plateauFuncSetBase( PLATEAUFUNC *pf, const PKMATHREAL base )
90  {
91     if ( !pf  ||  base <= 1.0 )
92        return(0);
93     pf->base = base;
94     return(1);
95  }
```

On success the `plateauFuncGetXUp()` function returns `pf->xUp` (Equations (2), (4) and (5)). Otherwise on error, it returns 0.0. The condition `!pf` is considered an input error.

```
96  PKMATHREAL plateauFuncGetXUp( const PLATEAUFUNC *pf )
97  {
98     if ( !pf )
99        return(0.0);
100    return( pf->xUp );
101 }
```

The `plateauFuncGetXDown()` function is identical to `plateauFuncGetXUp()` except that on success it returns `pf->xDown`.

```
102 PKMATHREAL plateauFuncGetXDown( const PLATEAUFUNC *pf )
103 {
104    if ( !pf )
105       return(0.0);
106    return( pf->xDown );
107 }
```

On success the `plateauFuncAt()` function returns the result of a call resembling

```
udFunc( x,
        plateauFuncGetXUp(pf),
        plateauFuncGetXDown(pf),
        plateauFuncGetBase(pf) )
* simpleFuncAt( plateauFuncGetSimpleFunc(pf), x )
```

Equations (2) and (5) refer. Otherwise the function returns 0.0. The condition `!pf` is considered an input error.

```
PKMATHREAL plateauFuncAt( const PLATEAUFUNC *pf, const PKMATHREAL x )
{
   SIMPLEFUNC *f;

   if ( !pf )
      return(0.0);

   f = plateauFuncGetSimpleFunc(pf);
   if ( !f )
      return(0.0);

   return( udFunc( x,
                   plateauFuncGetXUp(pf),
                   plateauFuncGetXDown(pf),
                   plateauFuncGetBase(pf) ) * simpleFuncAt(f,x) );
}
```

The `plateauFuncDraw()` function is similar to `simpleFuncDraw()` except that the former may be used to typeset the plateau function associated with the specified `pf`. The plateau function is evaluated indirectly at some position `x` via a `plateauFuncAt(pf,x)` call.

```
void plateauFuncDraw( const PLATEAUFUNC *pf,
                      const PKMATHREAL xLo,
                      const PKMATHREAL xHi,
                      const int samples )
{
   PKMATHREAL x;
   int i;

   if ( !pf  ||  xLo >= xHi  ||  samples < 2
        ||  !plateauFuncGetSimpleFunc(pf) )
      return;

   puts(   "   %");
   printf( "   %% A plateau function '%s' over the "
           "[" FLTFMT "," FLTFMT "] domain interval.\n",
           plateauFuncGetName(pf),
           xLo, xHi );
   puts(   "   %");
   printf( "   \\draw[plateaufunction]\n"
           "      (" FLTFMT "," FLTFMT ") coordinate[plateaufunctionpoint]\n",
           xLo, plateauFuncAt(pf,xLo) );
   /*puts(   "      plot[smooth,mark=*,mark size=1pt] coordinates {");*/
   puts(   "      plot[smooth] coordinates {");
   for ( i = 1; i <= samples; i++ ) {
      x = xLo + ( (double)(i-1.0) / (double)(samples-1) ) * ( xHi - xLo );
      printf( "         (" FLTFMT "," FLTFMT ")\n",
              x, plateauFuncAt(pf,x) );
   }
   puts(   "      } coordinate[plateaufunctionpoint];");

   return;
}
```

4.2.5 The plateauapprox.h and plateauapprox.c files

A listing of the plateauapprox.h file follows:

```
1  #ifndef _PLATEAUAPPROX
2  #define _PLATEAUAPPROX
```

Inclusions.

```
3  #include <pkfeatures.h>
4
5  #include <stdlib.h>
6  #include <stdarg.h>
7  #include <stdio.h>
8  #include <string.h>
9  #include <sys/stat.h>
10 #include <sys/types.h>
11 #include <unistd.h>
12 #include <math.h>
13 #include <float.h>
14
15 #include <pkmemdebug.h>
16 #include <pkerror.h>
17 #include <pktypes.h>
18 #include <pkmath.h>
19
20 #include "simplefunc.h"
21 #include "plateaufunc.h"
```

Type definitions.

```
22 typedef struct PlateauApproxs {
23    char *name;
24    unsigned int basepoints;     /* Number of "base points" in the interpolation. */
25    PKMATHREAL *basepoint;       /* Dynamic array of 'basepoints' base points. */
26    PLATEAUFUNC **pf;    /* Dynamic array of 'basepoints-1' (PLATEAUFUNC *)s. */
27    PKMATHREAL accuracy; /* Accuracy ratio.  Used to fix 'PLATEAUFUNC.base' */
28                         /*    in the 'pf[n]'. */
29    PKMATHREAL *weight;  /* Derived dynamic array of 'basepoints-1' weight */
30                         /* parameters for the 'pf[n]'. */
31 } PLATEAUAPPROX;
```

Function declarations.

```
32 extern PLATEAUAPPROX *plateauApproxAlloc0( const char *name,
33                                             const unsigned int basepoints,
34                                             const PKMATHREAL *basepoint,
35                                             const PLATEAUFUNC **pf,
36                                             const PKMATHREAL accuracy );
37 extern void plateauApproxFree0( PLATEAUAPPROX *pa );
38 extern PLATEAUAPPROX *plateauApproxAlloc1( const char *name,
39                                             const unsigned int basepoints,
40                                             const PKMATHREAL *basepoint,
41                                             const SIMPLEFUNC **sf,
42                                             const PKMATHREAL accuracy );
43 extern void plateauApproxFree1( PLATEAUAPPROX *pa );
44
45 extern char *plateauApproxGetName( const PLATEAUAPPROX *pa );
```

```
46  extern unsigned int plateauApproxGetBasepoints( const PLATEAUAPPROX *pa );
47  extern PKMATHREAL *plateauApproxGetBasepoint( const PLATEAUAPPROX *pa );
48  extern PLATEAUFUNC **plateauApproxGetPf( const PLATEAUAPPROX *pa );
49  extern PKMATHREAL plateauApproxGetAccuracy( const PLATEAUAPPROX *pa );
50  extern PKMATHREAL *plateauApproxGetWeight( const PLATEAUAPPROX *pa );
51
52  extern PKMATHREAL plateauApproxAt( const PLATEAUAPPROX *pa, const PKMATHREAL x );
53  extern void plateauApproxPrint( const PLATEAUAPPROX *pa );
54  extern void plateauApproxDraw( const PLATEAUAPPROX *pa,
55                                 const PKMATHREAL xLo,
56                                 const PKMATHREAL xHi,
57                                 const int samples );

58  #endif
```

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

```
1   #include "plateauapprox.h"
2
3   #include <pkfeatures.h>
4
5   #include <sys/types.h>
6   #include <stdlib.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <unistd.h>
10  #include <stdarg.h>
11  #include <math.h>
12  #include <float.h>
13
14  #include <pkmemdebug.h>
15  #include <pkerror.h>
16  #include <pktypes.h>
17  #include <pkstring.h>
18  #include <pkmath.h>
19
20  #include "common.h"
21  #include "simplefunc.h"
22  #include "plateaufunc.h"
```

The `_plateauApproxProdQuots()` private function simply computes and returns the product of quotients

$$\prod_{n=A+1}^{B+1} \frac{F_n(x_{n+1})}{F_n(x_n)}$$

for some specified integers A and B. The specified `pf` is a pointer to an array of `PLATEAUFUNC` plateau functions as defined in Equation (2). And since in C, the first entry in an array is indexed with 0, we have the correspondence $F_n =$ `pf[n-1]`, $n = 0, 1, \ldots$ And by virtue of the definition of `PLATEAUFUNC`, we have

$$F_n(x_n) = \texttt{plateauFuncAt(pf[n-1],plateauFuncGetXUp(pf[n-1])}$$
$$F_n(x_{n+1}) = \texttt{plateauFuncAt(pf[n-1],plateauFuncGetXDown(pf[n-1])}$$

```
23  static PKMATHREAL _plateauApproxProdQuots( const PLATEAUFUNC **pf,
24                                             const unsigned int A,
25                                             const unsigned int B )
26  {
27     PKMATHREAL prod,
28                numer,
29                denom;
30     PLATEAUFUNC **F;
31     int n;
32
33     if ( !pf  ||  B < A )
34        return(1.0);
35
36     prod = 1.0;
37     for ( n = A, F = (PLATEAUFUNC **)pf + A;  n <= B;  n++, F++ ) {
38        denom = plateauFuncAt( *F, plateauFuncGetXUp(*F) );
39        if ( fabs(denom) > FLT_EPSILON )
40           prod *= plateauFuncAt( *F, plateauFuncGetXDown(*F) ) / denom;
41     }
42
43     return(prod);
44  }
```

The `_plateauApproxPowMinusOne()` private function simply returns $(-1)^n$, for some specified integer n.

```
45  static PKMATHREAL _plateauApproxPowMinusOne( const unsigned int n )
46  {
47     if ( n % 2 == 0 )
48        return(1.0);
49     return(-1.0);
50  }
```

The `_plateauApproxAllocWeights()` private function allocates and initialises a dynamic array of plateau function interpolation weighting parameters, α_n, $n = 1, \ldots,$ `pfs`, as per Equation (15).

On success return a pointer the allocated and initialised array. Otherwise return `(PKMATHREAL *)NULL`. The conditions `!pf`, `!pf[n]` for each n, and `pfs` < 1 are considered input errors.

NB NOTE: Must be accompanied by a call to `_plateauApproxFreeWeights()`.

```
51  static PKMATHREAL *_plateauApproxAllocWeights( const PLATEAUFUNC **pf,
52                                                 const unsigned int pfs )
53  {
54     PKMATHREAL *alpha;
55     PLATEAUFUNC **F;     /* Running pointer along 'pf'. */
56     PKMATHREAL numer,
57                denom;
58     int n;
59
60     if ( !pf  ||  pfs < 1 )
61        return( (PKMATHREAL *)NULL );
62     for ( n = 0; n < pfs; n++ ) {
63        if ( !pf[n] )
64           return( (PKMATHREAL *)NULL );
65     }
66
67     alpha = (PKMATHREAL *)calloc( pfs + 1, sizeof(PKMATHREAL) );
68     if ( !alpha )
69        return( (PKMATHREAL *)NULL );
70
71     //// Delete this now.
72     //for ( n = 0; n <= pfs - 1; n++ )
73     //   alpha[n] = 1.0;
74     //return(alpha);
```

Deal with the special case of there being only one plateau function. I.e., `pfs`=N-1=1.

```
75     if ( pfs == 1 ) {
76        denom = plateauFuncAt( *pf, plateauFuncGetXUp(*pf) )
77              + plateauFuncAt( *pf, plateauFuncGetXDown(*pf) );
78        numer = simpleFuncAt( plateauFuncGetSimpleFunc(*pf), plateauFuncGetXUp(*pf) )
79              + simpleFuncAt( plateauFuncGetSimpleFunc(*pf), plateauFuncGetXDown(*pf) );
80        if ( fabs(denom) < FLT_EPSILON ) {
81           if ( fabs(numer) < FLT_EPSILON ) {
82              /*
83               * '0/0'.  So simply cheat.
84               */
85              alpha[0] = 1.0;
86           } else {
87              /*
88               * '1/0'.  Oops!
```

```
89                */
90               printf( "ERROR: '1.0/0.0' problem. Unable to calculate the weighting parameter
91                       plateauFuncGetName(*pf) );
92               exit(1);
93            }
94         } else {
95            alpha[0] = 0.5 * numer / denom;
96         }
97         return(alpha);
98      }
```

First compute α_1 as per Equation (15).

Note that since `pfs` $= N-1$ and since arrays are indexed from 0, we have $\alpha_1 =$ `alpha`[0] and $\alpha_{N-1} = \alpha_{\texttt{pfs}} =$ `alpha`[`pfs-1`].

```
99      denom = plateauFuncAt( pf[0], plateauFuncGetXUp(pf[0]) )
100            * ( 1.0 + _plateauApproxPowMinusOne( pfs + 1 ) * _plateauApproxProdQuots( pf,
101     if ( fabs(denom) > FLT_EPSILON ) {
102
103        PKMATHREAL sum;
104        int m;
105
106        sum = 0.0;
107        for ( m = 0, F = (PLATEAUFUNC **)pf;  m <= pfs - 2;  m++, F++ ) {
108           sum += _plateauApproxPowMinusOne( pfs + 1 + m + 1 )
109                * ( simpleFuncAt( plateauFuncGetSimpleFunc(*F), plateauFuncGetXDown(*F) )
110                  + simpleFuncAt( plateauFuncGetSimpleFunc(*(F+1)), plateauFuncGetXUp(*(F+1
111                * _plateauApproxProdQuots( pf, m + 1, pfs - 1 );
112        }
113        numer = simpleFuncAt( plateauFuncGetSimpleFunc(pf[0]), plateauFuncGetXUp(pf[0]) )
114              + simpleFuncAt( plateauFuncGetSimpleFunc(pf[pfs-1]), plateauFuncGetXDown(pf[p
115              - sum;
116        alpha[0] = 0.5 * numer / denom;
117
118     } else {
119
120        /*
121         * Division by zero.  So cheat.
122         */
123        alpha[0] = 1.0;
124
125     }
```

Next the $\alpha_n, n = 2, \ldots, N-1$ as per Equation (15).

```
126     for ( n = 1;  n <= pfs - 1;  n++ ) {
127        denom = plateauFuncAt( pf[n], plateauFuncGetXUp( pf[n] ) );
128        if ( fabs(denom) > FLT_EPSILON ) {
129           numer = 0.5 * ( simpleFuncAt( plateauFuncGetSimpleFunc(pf[n-1]), plateauFuncGetX
130                         + simpleFuncAt( plateauFuncGetSimpleFunc(pf[n]),   plateauFuncGetX
131                 - plateauFuncAt( pf[n-1], plateauFuncGetXDown(pf[n-1]) ) * alpha[n-1];
132           alpha[n] = numer / denom;
133        } else {
134           alpha[n] = 1.0;
135        }
136     }
137
138     //logMsg( "%d plateau functions:", pfs );
139     //for ( n = 0; n <= pfs - 1; n++ ) {
```

```
140      //   logMsg( "   %s:", plateauFuncGetName(pf[n]) );
141      //   logMsg( "      base = %f", plateauFuncGetBase(pf[n]) );
142      //   logMsg( "      [xUp,xDown] = [%f,%f]", plateauFuncGetXUp(pf[n]), plateauFuncGetXD
143      //   logMsg( "      [f(xUp),f(xDown)] = [%f,%f]",
144      //           simpleFuncAt( plateauFuncGetSimpleFunc(pf[n]), plateauFuncGetXUp(pf[n]) )
145      //           simpleFuncAt( plateauFuncGetSimpleFunc(pf[n]), plateauFuncGetXDown(pf[n])
146      //   logMsg( "      [F(xUp),F(xDown)] = [%f,%f]",
147      //           plateauFuncAt( pf[n], plateauFuncGetXUp(pf[n]) ),
148      //           plateauFuncAt( pf[n], plateauFuncGetXDown(pf[n]) ) );
149      //   logMsg( "      alpha = %f", alpha[n] );
150      //}
151
152      return(alpha);
153   }
154   static PKMATHREAL *_plateauApproxAllocWeightsAllUnity( const PLATEAUFUNC **pf,
155                                                          const unsigned int pfs )
156   {
157      PKMATHREAL *alpha;
158      int n;
159
160      if ( !pf  ||  pfs < 1 )
161         return( (PKMATHREAL *)NULL );
162      for ( n = 0; n < pfs; n++ ) {
163         if ( !pf[n] )
164            return( (PKMATHREAL *)NULL );
165      }
166
167      alpha = (PKMATHREAL *)calloc( pfs + 1, sizeof(PKMATHREAL) );
168      if ( !alpha )
169         return( (PKMATHREAL *)NULL );
170
171      for ( n = 0; n < pfs; n++ )
172         alpha[n] = 1.0;
173
174      return(alpha);
175   }
```

The `_plateauApproxFreeWeights()` private function is the complement to `_plateauApproxAllocWeights()`.

```
176   static void _plateauApproxFreeWeights( PKMATHREAL *alpha )
177   {
178      if (alpha)
179         free(alpha);
180      return;
181   }
```

The `_plateaApproxCalcBase()` private function calculates and returns

$$\left\lceil \frac{|r|}{\epsilon} \right\rceil^{1/d}$$

where $\epsilon =$ `epsilon`, $r =$ `simpleFuncRatio` and $d =$ `basepointDiff`. But it deals with the possibilities of small ϵ, small d, or both.

```
182   static PKMATHREAL _plateaApproxCalcBase( const PKMATHREAL epsilon,
183                                            const PKMATHREAL simpleFuncRatio,
184                                            const PKMATHREAL basepointDiff )
185   {
186      PKMATHREAL base,
```

```
187                eps,
188                diff;
189     eps = fmax( fabs(epsilon), FLT_EPSILON );
190     diff = fmax( fabs(basepointDiff), FLT_EPSILON );
191     base = pow( fabs(simpleFuncRatio) / eps, 1.0 / diff );
192     //logMsg( "_plateaApproxCalcBase():    'base' = %g.", base );
193     return(base);
194  }
```

The `_plateauApproxSetBases()` private function uses the specified small number $\epsilon =$ `accuracy` to fix the base b_n for the `pfs` `PLATEAUFUNC`s in the specified `pf` array. This is done as per Equation (13).

```
195  static void _plateauApproxSetBases( PLATEAUFUNC **pf,
196                                      const unsigned int pfs,
197                                      const PKMATHREAL *basepoint,
198                                      const PKMATHREAL accuracy )
199  {
200     SIMPLEFUNC *f;
201     int n;
202
203     //logMsg("_plateauApproxSetBases(): %d PLATEAUFUNCs.",pfs);
204     if ( !pf  ||  !basepoint  ||  pfs < 1 ||  fabs(accuracy) < FLT_EPSILON )
205        return;
206
207     if ( pfs == 1 ) {
208
```

Then there are only two basepoints in the `basepoint` array. So we must only deal with the base b_1 for the F_1 plateau function, i.e., for `pf[0]`.

```
209        //logMsg( "_plateauApproxSetBases(): Computing 'base' for '%s'.", plateauFuncGetNam
210        plateauFuncSetBase( *pf,
211                            _plateaApproxCalcBase( accuracy,
212                                                   1.0,
213                                                   fabs( basepoint[1] - basepoint[0] ) ) );
214
215     } else {
216
217        PKMATHREAL fRatio,
218                   fRatio1,
219                   fRatio2,
220                   basepointDiff,
221                   denom;
222        PLATEAUFUNC **F;
223
```

Then there are three or more basepoints in the `basepoint` array. Deal first with the base b_1 for the F_1 plateau function, i.e., for `pf[1]`.

```
224        //logMsg( "_plateauApproxSetBases(): Computing 'base' for '%s'.", plateauFuncGetNam
225        f = plateauFuncGetSimpleFunc(*pf);
226        if ( *pf  &&  f ) {
227           basepointDiff = fmin( fabs( basepoint[2] - basepoint[1] ),
228                                 fabs( basepoint[1] - basepoint[0] ) );
229           denom = simpleFuncAt( f, basepoint[1] );
230           if ( fabs(denom) > FLT_EPSILON )
231             fRatio = fabs( simpleFuncAt( f, basepoint[2] ) / denom );
```

```
232            else
233              fRatio = 1.0;
234            plateauFuncSetBase( *pf,
235                                _plateaApproxCalcBase( accuracy,
236                                                       fRatio,
237                                                       basepointDiff ) );
238         }
239
```

Now deal with the bases $b_n, n = 2, \ldots, N-2$, i.e., for `pf[n]`, `n=1,...,pfs-2`.

```
240         for ( n = 1, F = (PLATEAUFUNC **)pf + 1;  n <= pfs - 2;  n++, F++ ) {
241            //logMsg( "_plateauApproxSetBases(): Computing 'base' for '%s'.",
242            //        plateauFuncGetName(*F) );
243            f = plateauFuncGetSimpleFunc(*F);
244            if ( *F  &&  f ) {
245               basepointDiff = fmin( fmin( fabs( basepoint[n+2] - basepoint[n+1] ),
246                                           fabs( basepoint[n+1] - basepoint[n] ) ),
247                                     fabs( basepoint[n] - basepoint[n-1] ) );
248               denom = simpleFuncAt( f, basepoint[n] );
249               if ( fabs(denom) > FLT_EPSILON )
250                  fRatio1 = fabs( simpleFuncAt( f, basepoint[n-1] ) / denom );
251               else
252                  fRatio1 = 1.0;
253               denom = simpleFuncAt( f, basepoint[n+1] );
254               if ( fabs(denom) > FLT_EPSILON )
255                  fRatio2 = fabs( simpleFuncAt( f, basepoint[n+2] ) / denom );
256               else
257                  fRatio2 = 1.0;
258               plateauFuncSetBase( *F,
259                                   _plateaApproxCalcBase( accuracy,
260                                                          fmax( fRatio1, fRatio2 ),
261                                                          basepointDiff ) );
262            }
263         }
264
```

Finally, deal with base $b_{N-1} = b_{\texttt{pfs}}$, i.e., for `pf[pfs-1]`.

```
265         F = (PLATEAUFUNC **)pf + pfs - 1;
266         //logMsg( "_plateauApproxSetBases(): Computing 'base' for '%s'.",
267         //        plateauFuncGetName(*F) );
268         f = plateauFuncGetSimpleFunc(*F);
269         if ( *F && f ) {
270            basepointDiff = fmin( fabs( basepoint[pfs] - basepoint[pfs-1] ),
271                                  fabs( basepoint[pfs-1] - basepoint[pfs-2] ) );
272            denom = simpleFuncAt( f, basepoint[pfs-1] );
273            if ( fabs(denom) > FLT_EPSILON )
274               fRatio = fabs( simpleFuncAt( f, basepoint[pfs-2] ) / denom );
275            else
276               fRatio = 1.0;
277            plateauFuncSetBase( *F,
278                                _plateaApproxCalcBase( accuracy,
279                                                       fRatio,
280                                                       basepointDiff ) );
281         }
282
283      }
284
285      return;
286   }
```

The `plateauApproxAlloc0()` function allocates and initialises a `PLATEAUAPPROX` struct to encapsulate the details of a "plateau approximation" (Equation (3)).

The specified `basepoint` must point to an array of `basepoints` "base points". The specified `pf` must point to an array of `basepoints` $-$ 1 pointers to allocated and initialised `PLATEAUFUNC`s. So referring to Equation (3), the `basepoint` array must correspond to the set $\{x_n | n = 1, \ldots, N\}$, where N = `basepoints`. The `pf` array must correspond to the set $\{F_n | n = 1, \ldots, N-1\}$. The `accuracy` value must be specified as a small number `accuracy` $= \epsilon$ satisfying $\epsilon \leq \min\{\epsilon_n | n = 1, \ldots, N-1\}$. Equation (13) refers. The value for ϵ will be used to fix the base b_n for each `PLATEAUFUNC` in the `pf` array, as per Equation (13). That is, ϵ is a measure of the accuracy of the interpolation over the $[x_1, x_N]$ domain interval as a whole.

Note that to promote "environmental robustness," the function creates its own copy of the `basepoint` and `pf` arrays and of the `name` character string. So `basepoint`, `pf` and `name` may be allowed to vanish (i.e., `free()`-ed) once this function is called in the calling environment. And of course, provided the environment calls this function's sibling, `plateauApproxFree0()`, the copies will also be `free()`-ed.

On success return a pointer to the allocated and initialised `PLATEAUAPPROX` struct. Otherwise return `(PLATEAUAPPROX *)NULL`. The conditions `basepoints` < 2, `!basepoint`, `!pf` and |`accuracy`| $<$ `FLT_EPSILON` are considered input errors. The condition `!pf[n]` for any $0 \leq$ `n` $<$ `basepoint` $- 1$ is also considered an input error.

NB NOTE: Must be accompanied by a call to `plateauApproxFree0()`.

```
287  PLATEAUAPPROX *plateauApproxAlloc0( const char *name,
288                                      const unsigned int basepoints,
289                                      const PKMATHREAL *basepoint,
290                                      const PLATEAUFUNC **pf,
291                                      const PKMATHREAL accuracy )
292  {
293     PLATEAUAPPROX *pa;
294     PKMATHREAL *weight,  /* Dynamic array of 'basepoints-1' PLATEAUFUNC */
295                          /* weighting parameters. */
296                 *basepointb;
297     PLATEAUFUNC **pfb;
298     int n;
299
```

Validate the inputs.

```
300     if ( basepoints < 2  ||  !basepoint  ||  !pf
301           ||  fabs(accuracy) < FLT_EPSILON )
302        return( (PLATEAUAPPROX *)NULL );
303     for ( n = 0; n < basepoints - 1; n++ ) {
304        if ( !pf[n] )
305           return( (PLATEAUAPPROX *)NULL );
306        if ( fabs( basepoint[n+1] - basepoint[n] ) < FLT_EPSILON )
307           return( (PLATEAUAPPROX *)NULL );
308     }
309
```

Allocate and initialise an array `pfb` of pointers to `PLATEAUFUNC`s.

```
310     basepointb = (PKMATHREAL *)calloc( basepoints + 1, sizeof(PKMATHREAL) );
311     if ( !basepointb )
312        return( (PLATEAUAPPROX *)NULL );
313     for ( n = 0; n < basepoints; n++ )
314        basepointb[n] = basepoint[n];
```

```
315      pfb = (PLATEAUFUNC **)calloc( basepoints - 1 + 1, sizeof(PLATEAUFUNC *) );
316      if ( !pfb ) {
317         free(basepointb);
318         return( (PLATEAUAPPROX *)NULL );
319      }
320      for ( n = 0; n < basepoints - 1; n++ )
321         pfb[n] = plateauFuncAllocCopy( pf[n] );
```

Use |accuracy| to set the PLATEAUFUNC.base values in the pfb array.

```
322      _plateauApproxSetBases( pfb,
323                              basepoints - 1,
324                              basepoint,
325                              accuracy );
```

Allocate and initialise the array weight of weighting parameters.

```
326      weight = _plateauApproxAllocWeights( (const PLATEAUFUNC **)pfb, basepoints - 1 );
327      if ( !weight ) {
328         free(basepointb);
329         for ( n = 0; n < basepoints - 1; n++ )
330            plateauFuncFreeCopy( pfb[n] );
331         free(pfb);
332         return( (PLATEAUAPPROX *)NULL );
333      }
```

Allocate and initialise a PLATEAUAPPROX struct.

```
334      pa = (PLATEAUAPPROX *)calloc( 1, sizeof(PLATEAUAPPROX) );
335      if (pa) {
336         pa->name = strAllocInit(name);
337         pa->basepoints = basepoints;
338         pa->basepoint = basepointb;
339         pa->pf = pfb;
340         pa->weight = weight;
341         pa->accuracy = fabs(accuracy);
342      } else {
343         /*
344          * Roll back.
345          */
346         free(basepointb);
347         for ( n = 0; n < basepoints - 1; n++ )
348            plateauFuncFreeCopy(pfb[n]);
349         free(pfb);
350         _plateauApproxFreeWeights(weight);
351      }
352
353      return(pa);
354   }
```

plateauApproxFree0() is the complement to plateauApproxAlloc0().

```
355   void plateauApproxFree0( PLATEAUAPPROX *pa )
356   {
357      if (pa) {
358         strFreeInit(pa->name);
359         pa->name = (char *)NULL;  /* For good measure. */
360         if ( pa->basepoint ) {
```

```
361            free(pa->basepoint);
362            pa->basepoint = (PKMATHREAL *)NULL;    /* For good measure. */
363         }
364         if ( pa->pf ) {
365            int n;
366            for ( n = 0; n < pa->basepoints - 1; n++ )
367               plateauFuncFreeCopy( pa->pf[n] );
368            free(pa->pf);
369            pa->pf = (PLATEAUFUNC **)NULL;         /* For good measure. */
370         }
371         if ( pa->weight ) {
372            _plateauApproxFreeWeights( pa->weight );
373            pa->weight = (PKMATHREAL *)NULL;       /* For good measure. */
374         }
375         free(pa);
376      }
377      return;
378   }
```

The `plateauApproxAlloc1()` function allocates and initialises a `PLATEAUAPPROX` struct to encapsulate the details of a "plateau approximation" (Equation (3)). The function is very similar to `plateauApproxAlloc1()` except that the `PLATEAUAPPROX` is allocated and initialised using the specified array of `(SIMPLEFUNC *)`s instead of an array of `(PLATEAUFUNC *)`s.

The specified `sf` array must point to an array of `basepoints` $-$ 1 pointers to allocated and initialised `SIMPLEFUNC`s. So referring to Equation (3), the `sf` array must correspond to the set $\{f_n | n = 1, \ldots, N-1\}$ of "simple functions", where $N =$ `basepoints`. For a discussion on the specified `accuracy`, refer to the annotation of the `plateauApproxAlloc0()` function.

Note that the comment in the annotation of `plateauApproxAlloc0()` about "environmental robustness" applies here too.

On success return a pointer to the allocated and initialised `PLATEAUAPPROX` struct. Otherwise return `(PLATEAUAPPROX *)NULL`.

NB NOTE: Must be accompanied by a call to `plateauApproxFree1()`.

```
379   PLATEAUAPPROX *plateauApproxAlloc1( const char *name,
380                                       const unsigned int basepoints,
381                                       const PKMATHREAL *basepoint,
382                                       const SIMPLEFUNC **sf,
383                                       const PKMATHREAL accuracy )
384   {
385      PLATEAUAPPROX *pa;
386      PLATEAUFUNC **pf;
387      char *buf;
388      int n;
389
```

Validate the inputs.

```
390      if ( basepoints < 2  ||  !basepoint  ||  !sf )
391         return( (PLATEAUAPPROX *)NULL );
392      for ( n = 0; n < basepoints - 1; n++ ) {
393         if ( !sf[n] )
394            return( (PLATEAUAPPROX *)NULL );
395      }
```

Allocate and initialise an array `pf` of pointers to `PLATEAUFUNC`s. Use the specified `sf` array of `(SIMPLEFUNC *)` pointers.

```
396      pf = (PLATEAUFUNC **)calloc( basepoints - 1 + 1, sizeof(PLATEAUFUNC *) );
397      if ( !pf )
398         return( (PLATEAUAPPROX *)NULL );
399      for ( n = 0; n < basepoints - 1; n++ ) {
400         buf = strAllocPrintf( "F\\undr{%d}(x)", n + 1 );
401         pf[n] = plateauFuncAlloc( buf,
402                                   sf[n],
403                                   1.1,
404                                   basepoint[n],
405                                   basepoint[n+1] );
406         strFreePrintf(buf);
407      }
```

Finally...

```
408      pa = plateauApproxAlloc0( name,
409                                basepoints,
410                                basepoint,
411                                (const PLATEAUFUNC **)pf,
412                                accuracy );
413
```

Clean up.

```
414      for ( n = 0; n < basepoints - 1; n++ )
415         plateauFuncFree( pf[n] );
416      free(pf);
417
418      return(pa);
419   }
```

`plateauApproxFree1()` is the complement to `plateauApproxAlloc1()`.

```
420   void plateauApproxFree1( PLATEAUAPPROX *pa )
421   {
422      if (pa)
423         plateauApproxFree0(pa);
424      return;
425   }
```

On success the `plateauApproxGetName()` function simply returns `pa->name`. Otherwise it returns `(char *)NULL`. The condition `!pa` is considered an input error.

```
426   char *plateauApproxGetName( const PLATEAUAPPROX *pa )
427   {
428      if ( !pa )
429         return( (char *)NULL );
430      return( pa->name );
431   }
```

On success the `plateauApproxGetBasepoints()` function returns `pa->basepoints`. Otherwise it returns 0. The condition `!pa` is considered an input error.

```
432   unsigned int plateauApproxGetBasepoints( const PLATEAUAPPROX *pa )
433   {
434      if ( !pa )
435         return(0);
436      return( pa->basepoints );
437   }
```

On success the `plateauApproxGetBasepoint()` function simply returns `pa->basepoint`. Otherwise it returns `(PKMATHREAL *)NULL`. The condition `!pa` is considered an input error.

```
438  PKMATHREAL *plateauApproxGetBasepoint( const PLATEAUAPPROX *pa )
439  {
440     if ( !pa )
441        return( (PKMATHREAL *)NULL );
442     return( pa->basepoint );
443  }
```

On success the `plateauApproxGetPf()` function simply returns `pa->pf`. Otherwise it returns `(PLATEAUFUNC **)NULL`. The condition `!pa` is considered an input error.

```
444  PLATEAUFUNC **plateauApproxGetPf( const PLATEAUAPPROX *pa )
445  {
446     if ( !pa )
447        return( (PLATEAUFUNC **)NULL );
448     return( pa->pf );
449  }
```

On success the `plateauApproxGetAccuracy()` function returns `pa->accuracy`. Otherwise it returns 1.0. The condition `!pa` is considered an input error.

```
450  PKMATHREAL plateauApproxGetAccuracy( const PLATEAUAPPROX *pa )
451  {
452     if ( !pa )
453        return(1.0);
454     return( pa->accuracy );
455  }
```

On success the `plateauApproxGetWeight()` function simply returns `pa->weight`. Otherwise it returns `(PKMATHREAL *)NULL`. The condition `!pa` is considered an input error.

```
456  PKMATHREAL *plateauApproxGetWeight( const PLATEAUAPPROX *pa )
457  {
458     if ( !pa )
459        return( (PKMATHREAL *)NULL );
460     return( pa->weight );
461  }
```

The `plateauApproxAt()` function computes the approximation function $F(x)$ as a linear interpolation over the set $\{F_n | n = 1, \ldots, N-1\}$ of "plateau functions," evaluated at the specified point x in its domain, and where it is assumed that $N =$ `plateauApproxGetBasepoints(pa)`. Equation (3) refers. Specifically, the function computes a result equivalent to

```
pfs = plateauApproxGetBasepoints(pa) - 1;
weight = plateauApproxGetWeight(pa);
pf = plateauApproxGetPf(pa);

weight[0] * plateauFuncAt( pf[0], x )
   + weight[1] * plateauFuncAt( pf[1], x )
   .
   .
   .
   + weight[pfs-1] * plateauFuncAt( pf[pfs-1], x )
```

On success return the result of the calculation. Otherwise return 0.0. The condition `!pa` is considered an input error.

```
462  PKMATHREAL plateauApproxAt( const PLATEAUAPPROX *pa, const PKMATHREAL x )
463  {
464     PKMATHREAL res;
465     int pfs;
466     PKMATHREAL *weight;
467     PLATEAUFUNC **pf;
468     int n;
469
470     if ( !pa )
471        return(0.0);
472
473     pfs = plateauApproxGetBasepoints(pa) - 1;
474     weight = plateauApproxGetWeight(pa);
475     pf = plateauApproxGetPf(pa);
476     if ( pfs < 1  ||  !weight  ||  !pf )
477        return(0.0);
478
479     res = 0.0;
480     for ( n = 0; n < pfs; n++ )
481        res += weight[n] * plateauFuncAt( pf[n], x );
482
483     return(res);
484  }
```

The `plateauApproxPrint()` function prints to standard output the dataset comprising the `PLATEAUAPPROX` pointed to by the specified `pa`.

```
485  void plateauApproxPrint( const PLATEAUAPPROX *pa )
486  {
487     PLATEAUFUNC **pf;
488     SIMPLEFUNC *sf;
489     PKMATHREAL *weight;
490     unsigned int basepoints;
491     int n;
492
493     if (!pa)
494        return;
495
496     basepoints = plateauApproxGetBasepoints(pa);
497     weight = plateauApproxGetWeight(pa);
498     pf = plateauApproxGetPf(pa);
499
500     printf( "PLATEAUAPPROX %s:\n", plateauApproxGetName(pa) );
501     printf( "   accuracy = %g\n", plateauApproxGetAccuracy(pa) );
502     printf( "   %d basepoints: ", basepoints );
503     for ( n = 0; n < basepoints; n++ )
504        printf( "%g%s",
505                plateauApproxGetBasepoint(pa)[n],
506                ( n == basepoints - 1 ) ? "\n" : ", " );
507     printf( "   %d weights for %d PLATEAUFUNCs:\n", basepoints - 1, basepoints - 1 );
508     for ( n = 0; n < basepoints - 1; n++ ) {
509        sf = plateauFuncGetSimpleFunc(pf[n]);
510        printf( "      %g * %s:\n", weight[n], plateauFuncGetName(pf[n]) );
511        printf( "         [xUp,xDown] = [%g,%g]\n",
512                plateauFuncGetXUp(pf[n]), plateauFuncGetXDown(pf[n]) );
513        printf( "         base = %g\n", plateauFuncGetBase(pf[n]) );
514        printf( "         SIMPLEFUNC %s:\n",
```

```
                simpleFuncGetName(sf) );
        printf( "            Passes through (%g,%g) and (%g,%g).\n",
                pkRealVectorGetComponent( simpleFuncGetP(sf) )[0],
                pkRealVectorGetComponent( simpleFuncGetP(sf) )[1],
                pkRealVectorGetComponent( simpleFuncGetQ(sf) )[0],
                pkRealVectorGetComponent( simpleFuncGetQ(sf) )[1] );
        printf( "            strength = %g\n", simpleFuncGetStrength(sf) );
    }

    return;
}
```

The `plateauApproxDraw()` function is similar to `simpleFuncDraw()` except that the former may be used to typeset the "plateau approximation function" associated with the specified `pa`. The plateau approximation is evaluated indirectly at any position x via a `plateauApproxAt(pa,x)` call.

```
void plateauApproxDraw( const PLATEAUAPPROX *pa,
                        const PKMATHREAL x1,
                        const PKMATHREAL x2,
                        const int samples )
{
    PKMATHREAL x;
    int i;

    if ( !pa  ||  x1 >= x2  ||  samples < 2
         ||  !plateauApproxGetPf(pa) )
       return;

    puts(   "   %");
    printf( "   %% A plateau approximation '%s' over the "
            "[" FLTFMT "," FLTFMT "] domain interval.\n",
            plateauApproxGetName(pa),
            x1, x2 );
    puts(   "   %");
    printf( "   \\draw[plateauapprox]\n"
            "      (" FLTFMT "," FLTFMT ") coordinate[plateauapproxpoint]\n",
            x1, plateauApproxAt(pa,x1) );
    /*puts(   "      plot[smooth,mark=*,mark size=1pt] coordinates {");*/
    puts(   "      plot[smooth] coordinates {");
    for ( i = 1; i <= samples; i++ ) {
       x = x1 + ( (double)(i-1.0) / (double)(samples-1) ) * ( x2 - x1 );
       printf( "         (" FLTFMT "," FLTFMT ")\n",
               x, plateauApproxAt(pa,x) );
    }
    puts(   "      } coordinate[plateauapproxpoint];");

    return;
}
```

4.2.6 The updownfuncs.c file

A listing of the updownfuncs.c file follows:

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "common.h"
```

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

```
20  static void _drawCoordinates( const PKREALVECTOR *e1,
21                                const PKREALVECTOR *e2 )
22  {
23     if ( !e1 || !e2 )
24        return;
25
26     printf( "   %%\\draw[help lines]\n"
27             "      (" FLTFMT "," FLTFMT ")"
28             " grid (" FLTFMT "," FLTFMT ");\n",
29             -pkRealVectorGetComponent(e1)[0] - 0.2, -0.2,
30             pkRealVectorGetComponent(e1)[0] + 0.2, pkRealVectorGetComponent(e2)[1] + 0.2 );
31     puts(   "   %");
32     puts(   "   % Some coordinates.");
33     puts(   "   %");
34     puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
35
36     return;
37  }
```

The _drawBasisVectors private function simply printf() to standard output Ti*k*Z commands to typeset the $\hat{\mathbf{1}}$ and $\hat{\mathbf{2}}$ basis vectors.

```
38  static void _drawBasisVectors( const PKREALVECTOR *e1,
39                                 const PKREALVECTOR *e2 )
40  {
41     if ( !e1  ||  !e2  )
42        return;
43
44     puts(   "   %");
45     puts(   "   % The '1' and '2' basisvectors.");
46     puts(   "   %");
47     printf( "   \\draw[pktikzbasisvector,->]\n"
```

```
48              "      (" FLTFMT "," FLTFMT ") -- (" FLTFMT "," FLTFMT ") node[right]{$%s$};\n
49              -pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
50              pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
51              pkRealVectorGetName(e1) );
52      printf( "   \\draw[pktikzbasisvector,->]\n"
53              "      (origin) +(0,-4pt) -- (" FLTFMT "," FLTFMT ") node[above]{$%s$};\n",
54              pkRealVectorGetComponent(e2)[0], pkRealVectorGetComponent(e2)[1],
55              pkRealVectorGetName(e2) );
56
57      return;
58  }
```

The `_drawUpAndDownFunctions()` private function `printf()`s Ti*k*Z commands for typesetting a typical $\mathrm{up}(x;\beta)$ function (Equation (4)), where β = `base`. The function domain interval of interest is the specified interval $[x_1, x_2]$.

```
59  static void _drawUpAndDownFunctions( const PKMATHREAL x1,
60                                       const PKMATHREAL x2,
61                                       const PKMATHREAL base,
62                                       const PKMATHREAL vertExagg )
63  {
64      const int points = 30;
65      PKMATHREAL x;
66      int i;
67
68      if ( x1 >= x2 || base <= 0.0 || vertExagg <= 0.0 )
69          return;
```

The rising function $\mathrm{up}(x;\beta)$.

```
70      puts(   "   %");
71      printf( "   %% The rising function '" FLTFMT " * up(x;" FLTFMT ")' over the "
72              "[" FLTFMT "," FLTFMT "] domain interval.\n",
73              vertExagg,
74              base,
75              x1, x2 );
76      printf( "   %% '" FLTFMT "' is simply a vertical exaggeration factor.\n",
77              vertExagg );
78      puts(   "   %");
79
80      printf( "   \\draw[updownfunction]\n"
81              "      (" FLTFMT "," FLTFMT ") coordinate[updownfunctionpoint]\n",
82              x1,
83              vertExagg * upFunc(x1,base) );
84      /*puts(   "      plot[smooth,mark=*,mark size=1pt] coordinates {");*/
85      puts(   "      plot[smooth] coordinates {");
86      for ( i = 1; i <= points; i++ ) {
87          x = x1 + ( (double)(i-1.0) / (double)(points-1) ) * ( x2 - x1 );
88          printf( "         (" FLTFMT "," FLTFMT ")\n",
89                  x,
90                  vertExagg * upFunc(x,base) );
91      }
92      puts(   "      } coordinate[updownfunctionpoint]");
93      puts(   "        node[opaquelabel,left=2ex]{$\\upFunc(x;\\myBeta)$};");
```

The falling function $\mathrm{do}(x;\beta)$.

```
94      puts(   "   %");
95      printf( "   %% The falling function '" FLTFMT " * do(x;" FLTFMT ")'.\n",
```

```
96              vertExagg,
97              base );
98      puts(   "   %");
99      printf( "   \\draw[updownfunction]\n"
100             "      (" FLTFMT "," FLTFMT ") coordinate[updownfunctionpoint]\n"
101             "                node[opaquelabel,right=2ex]{$\\doFunc(x;\\myBeta)$}\n",
102             x1,
103             vertExagg * doFunc(x1,base) );
104     /*puts(   "      plot[smooth,mark=*,mark size=1pt] coordinates {");*/
105     puts(   "      plot[smooth] coordinates {");
106     for ( i = 1; i <= points; i++ ) {
107        x = x1 + ( (double)(i-1.0) / (double)(points-1) ) * ( x2 - x1 );
108        printf( "         (" FLTFMT "," FLTFMT ")\n",
109                x,
110                vertExagg * doFunc(x,base) );
111     }
112     puts(   "      } coordinate[updownfunctionpoint];");
```

Some cosmetics.

```
113     puts(   "   %");
114     printf( "   \\draw[pktikzdimension] (" FLTFMT "," FLTFMT ")\n"
115             "      -- (" FLTFMT "," FLTFMT ");\n",
116             -0.2 * ( x2 - x1 ), vertExagg,
117             0.2 * ( x2 - x1 ), vertExagg );
118     printf( "   \\node[opaquelabel,right] at (0.0," FLTFMT ") {$1$};\n", vertExagg );
119
120     return;
121 }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
122 static void _diagram(void)
123 {
124     const PKMATHREAL base = 5.0;
125     PKREALVECTOR *e1,
126                  *e2;
```

Prepare the objects in the landscape.

```
127     e1 = pkRealVectorAlloc1( "x", 2, 6.0, 0.0 );
128     e2 = pkRealVectorAlloc1( "y", 2, 0.0, 4.0 );
129     //pkRealVectorScale(e1,1.2);
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
130     puts(   "\\begin{PkTikzpicture}[scale=1.0]");
131     _drawCoordinates(e1,e2);
132     _drawBasisVectors(e1,e2);
133     _drawUpAndDownFunctions( -pkRealVectorGetComponent(e1)[0],
134                              pkRealVectorGetComponent(e1)[0],
135                              base,
136                              0.9 * pkRealVectorGetComponent(e2)[1] );
137     puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
138      pkRealVectorFree1(e1);
139      pkRealVectorFree1(e2);
140
141      return;
142   }


143   int main( const int argc, const char *argv[] )
144   {
145      _diagram();
146      //memPrintf();
147      exit(0);
148   }
```

4.2.7 The udfunc.c file

A listing of the udfunc.c file follows:

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "common.h"
20  #include "simplefunc.h"
21  #include "plateaufunc.h"
```

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

```
22  static void _drawCoordinates( const PKREALVECTOR *e1,
23                                const PKREALVECTOR *e2 )
24  {
25     if ( !e1 || !e2 )
26        return;
27
28     printf( "   %%\\draw[help lines] (-0.2,-0.2) grid (" FLTFMT "," FLTFMT ");\n",
29             pkRealVectorGetComponent(e1)[0] + 0.2,
30             pkRealVectorGetComponent(e2)[1] + 0.2 );
31     puts(   "   %");
32     puts(   "   % Some coordinates.");
33     puts(   "   %");
34     puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
35
36     return;
37  }
```

The _drawSomeLabelling() private function simply printf()s to standard output a body of Ti*k*Z source code which may be used to typeset some labelling in the figure.

```
38  static void _drawSomeLabelling( const PLATEAUFUNC *F )
39  {
40     const PKMATHREAL
41        xa = pkRealVectorGetComponent(simpleFuncGetP(plateauFuncGetSimpleFunc(F)))[0],
42        xb = pkRealVectorGetComponent(simpleFuncGetQ(plateauFuncGetSimpleFunc(F)))[0];
43
44     puts(   "   %");
45     puts(   "   % Labelling on the x-axis.");
46     puts(   "   %");
47     printf( "   \\draw (" FLTFMT ",0) +(0,-2pt) node[below]{$x_n$}\n"
```

```
48              "      -- +(0,4pt)\n"
49              "      (" FLTFMT ",0) +(0,-2pt) node[below]{$x_{n+1}$}\n"
50              "      -- +(0,4pt);\n",
51              xa,
52              xb );
53      printf( "   \\draw[pktikzdimension] (" FLTFMT ",0) +(0,2pt)\n"
54              "      -- ++(0," FLTFMT ")\n"
55              "      -- ++(-" FLTFMT ",0) node[opaquelabel,right]{$\\frac{1}{2}$}\n"
56              "      (" FLTFMT ",0) +(0,2pt)\n"
57              "      -- ++(0," FLTFMT ")\n"
58              "      (0," FLTFMT ") node[opaquelabel,right]{$1$}\n"
59              "      -- ++(" FLTFMT ",0);\n",
60              xa, plateauFuncAt(F,xa), xa,
61              xb, plateauFuncAt(F,xb),
62              plateauFuncAt( F, 0.5 * ( xa + xb ) ),
63              xa + 0.25 * ( xb -xa ) );
64
65      puts(   "   %");
66      puts(   "   % The function label.");
67      puts(   "   %");
68      puts(   "   \\node[plateaufunctioncolor,opaquelabel]");
69      printf( "      at (" FLTFMT "," FLTFMT ") {$%s$};\n",
70              0.5 * ( xa + xb ),
71              plateauFuncAt( F, 0.5 * ( xa + xb ) ),
72              plateauFuncGetName(F) );
73
74      return;
75  }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
76  static void _diagram(void)
77  {
78      const PKMATHREAL base = 5.0,
79                       xn = 2.0,
80                       xnP1 = xn + 8.0;
81      PKREALVECTOR *e1,
82                   *e2;
83      SIMPLEFUNC *fn;
84      PLATEAUFUNC *Fn;
85
```

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

```
86      e1 = pkRealVectorAlloc1( "x", 2, xnP1 + 2.0, 0.0 );
87      e2 = pkRealVectorAlloc1( "y", 2, 0.0,        4.0 );
88
89      fn = simpleFuncAlloc1( "f_n(x)",
90                             linear,
91                             xn,   0.9 * pkRealVectorGetComponent(e2)[1],
92                             xnP1, 0.9 * pkRealVectorGetComponent(e2)[1],
93                             1.0 );
94      Fn = plateauFuncAlloc( "\\udFunc_n(x)", fn, base, xn, xnP1 );
95      simpleFuncFree1(fn);
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
96      puts(   "\\begin{PkTikzpicture}[scale=1.0]");
97      _drawCoordinates(e1,e2);
98      drawBasisVectors(e1,e2);
99      plateauFuncDraw( Fn, 0.0, pkRealVectorGetComponent(e1)[0], 20 );
100     _drawSomeLabelling(Fn);
101     puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
102     pkRealVectorFree1(e1);
103     pkRealVectorFree1(e2);
104     plateauFuncFree(Fn);
105
106     return;
107  }

108  int main( const int argc, const char *argv[] )
109  {
110     _diagram();
111     //memPrintf();
112     exit(0);
113  }
```

4.2.8 The `example1.c` file

A listing of the `example1.c` file follows:

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "common.h"
20  #include "simplefunc.h"
21  #include "plateaufunc.h"
22  #include "plateauapprox.h"
```

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

```
23  static void _drawCoordinates( const PKREALVECTOR *e1,
24                                const PKREALVECTOR *e2 )
25  {
26     if ( !e1 || !e2 )
27        return;
28
29     printf( "   %%\\draw[help lines]\n"
30             "      (" FLTFMT "," FLTFMT ")"
31             " grid (" FLTFMT "," FLTFMT ");\n",
32             -pkRealVectorGetComponent(e1)[0] - 0.2, -0.2,
33             pkRealVectorGetComponent(e1)[0] + 0.2, pkRealVectorGetComponent(e2)[1] + 0.2 );
34     puts(   "   %");
35     puts(   "   % Some coordinates.");
36     puts(   "   %");
37     puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
38
39     return;
40  }
```

The `_drawBasisVectors` private function simply `printf()` to standard output Ti*k*Z commands to typeset the $\hat{\mathbf{1}}$ and $\hat{\mathbf{2}}$ basis vectors.

```
41  static void _drawBasisVectors( const PKREALVECTOR *e1,
42                                 const PKREALVECTOR *e2 )
43  {
44     if ( !e1  ||  !e2  )
45        return;
46
47     puts(   "   %");
```

```
48      puts(   "   % The '1' and '2' basisvectors.");
49      puts(   "   %");
50      printf( "   \\draw[pktikzbasisvector,->]\n"
51              "      (" FLTFMT "," FLTFMT ") -- (" FLTFMT "," FLTFMT ") node[right]{$%s$};\n
52              -pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
53              pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
54              pkRealVectorGetName(e1) );
55      printf( "   \\draw[pktikzbasisvector,->]\n"
56              "      (origin) +(0,-4pt) -- (" FLTFMT "," FLTFMT ") node[above]{$%s$};\n",
57              pkRealVectorGetComponent(e2)[0], pkRealVectorGetComponent(e2)[1],
58              pkRealVectorGetName(e2) );
59
60      return;
61  }
```

The `_drawSomeLabelling()` private function simply `printf()`s to standard output a body of Ti*k*Z source code which may be used to typeset some labelling in the figure.

```
62  static void _drawSomeLabelling( const PLATEAUAPPROX *pa )
63  {
64      SIMPLEFUNC *f1,
65                 *f2;
66      PLATEAUFUNC **F;
67      PKREALVECTOR *a,
68                   *b,
69                   *c;
70      PKMATHREAL x;
71
72      if ( !pa )
73         return;
74
75      F = plateauApproxGetPf(pa);
76      f1 = plateauFuncGetSimpleFunc(F[0]);
77      f2 = plateauFuncGetSimpleFunc(F[1]);
78      a = simpleFuncGetP(f1);
79      b = simpleFuncGetQ(f1);
80      c = simpleFuncGetQ(f2);
81
82      puts(   "   %");
83      puts(   "   % Labelling on the x-axis.");
84      puts(   "   %");
85      puts( "   \\draw");
86      printf( "      (" FLTFMT ",0) +(0,-3pt)"
87              " node[below right=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
88              pkRealVectorGetComponent(a)[0],
89              pkRealVectorGetName(a) );
90      printf( "      (" FLTFMT ",0) +(0,-3pt)"
91              " node[below right=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
92              pkRealVectorGetComponent(b)[0],
93              pkRealVectorGetName(b) );
94      printf( "      (" FLTFMT ",0) +(0,-3pt)"
95              " node[below left=0pt and -1.4ex]{$%s$} -- +(0,4pt);\n",
96              pkRealVectorGetComponent(c)[0],
97              pkRealVectorGetName(c) );
98
99      puts( "   %");
100     puts( "   % Labelling the simple functions.");
101     puts( "   %");
102     puts( "   \\path");
103     x = bitLess( pkRealVectorGetComponent(c)[0], pkRealVectorGetComponent(b)[0], 0.2 );
104     printf( "      (" FLTFMT "," FLTFMT ")"
```

```
105             " node[simplefunctioncolor,above right]{$%s$}\n",
106             x,
107             simpleFuncAt(f2,x),
108             simpleFuncGetName(f2) );
109     x = bitMore( pkRealVectorGetComponent(a)[0], pkRealVectorGetComponent(b)[0], 0.2 );
110     printf( "      (" FLTFMT "," FLTFMT ")"
111             " node[simplefunctioncolor,above left]{$%s$};\n",
112             x,
113             simpleFuncAt(f1,x),
114             simpleFuncGetName(f1) );
115
116     puts( "   %");
117     puts( "   % Labelling the plateau functions.");
118     puts( "   %");
119     puts( "   \\path");
120     x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
121     printf( "      (" FLTFMT "," FLTFMT ")"
122             " node[plateaufunctioncolor,opaquelabel]{$%s$}\n",
123             x,
124             plateauFuncAt(F[0],x),
125             plateauFuncGetName(F[0]) );
126     x = bitLess( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
127     printf( "      (" FLTFMT "," FLTFMT ")"
128             " node[plateaufunctioncolor,opaquelabel]{$%s$};\n",
129             x,
130             plateauFuncAt(F[1],x),
131             plateauFuncGetName(F[1]) );
132
133     //puts(   "   %");
134     //puts(   "   % Labelling the plateau approximation.");
135     //puts(   "   %");
136     //puts(   "   \\path[->]");
137     //x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.1 );
138     //printf( "      (" FLTFMT "," FLTFMT ") coordinate (a)\n",
139     //        x,
140     //        plateauApproxAt(pa,x) );
141     //x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.5 );
142     //printf( "      (" FLTFMT "," FLTFMT ") node[plateauapproxcolor] (b) {$%s$};\n",
143     //        x,
144     //        plateauApproxAt( pa, pkRealVectorGetComponent(b)[0] ),
145     //        plateauApproxGetName(pa) );
146     //puts(   "   \\draw[pktikzdimension,->] (b) to[out=170,in=40] (a);");
147     puts(   "   %");
148     puts(   "   % Labelling the plateau approximation.");
149     puts(   "   %");
150     puts(   "   \\path[->]");
151     x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
152     printf( "      (" FLTFMT "," FLTFMT ")"
153             " node[plateauapproxcolor,opaquelabel]{$%s$};\n",
154             x,
155             plateauApproxAt(pa,x),
156             plateauApproxGetName(pa) );
157
158     return;
159 }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
160  static void _diagram(void)
161  {
162     const PKMATHREAL accuracy = 0.0001;
163     PKREALVECTOR *e1,
164                  *e2,
165                  *a,
166                  *b,
167                  *c;
168     PKMATHREAL x[3];
169     SIMPLEFUNC *f[2];
170     PLATEAUAPPROX *pa;
171
```

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

```
172     e1 = pkRealVectorAlloc1( "x", 2, 7.0, 0.0 );
173     e2 = pkRealVectorAlloc1( "y", 2, 0.0, 5.0 );
174     //pkRealVectorScale(e1,1.2);
175     //pkRealVectorScale(e2,1.2);
176
177     printf( "\\long\\gdef\\egOneAccuracy{%g}\n", accuracy );
178
179     /*
180      * The base points.
181      */
182     x[0] = -0.8 * pkRealVectorGetComponent(e1)[0];
183     x[1] = 0.0;
184     x[2] = -x[0];
185
186     /*
187      * Important positions in the '1.2' plane.
188      */
189     a = pkRealVectorAlloc1( "x\\undr{1}=-1", 2, x[0], 0.8 * pkRealVectorGetComponent(e2)[1]
190     b = pkRealVectorAlloc1( "x\\undr{2}=0",  2, x[1], 0.8 * pkRealVectorGetComponent(e2)[1]
191     c = pkRealVectorAlloc1( "1=x\\undr{3}",  2, x[2], 0.8 * pkRealVectorGetComponent(e2)[1]
192
193     f[0] = simpleFuncAlloc0( "f\\undr{1}(x)=1", linear, a, b, 1.0 );
194     f[1] = simpleFuncAlloc0( "f\\undr{2}(x)=1", linear, b, c, 1.0 );
195     pa = plateauApproxAlloc1( "F(x)", 3, x, (const SIMPLEFUNC **)f, accuracy );
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
196     puts(   "\\begin{PkTikzpicture}[scale=1.0]");
197     _drawCoordinates(e1,e2);
198     _drawBasisVectors(e1,e2);
199     simpleFuncDraw( f[0],
200                     bitLess( x[0], x[1], 0.2 ),
201                     bitMore( x[1], x[2], 0.2 ),
202                     2 );
203     simpleFuncDraw( f[1],
204                     bitLess( x[1], x[0], 0.2 ),
205                     bitMore( x[2], x[1], 0.2 ),
206                     2 );
207     plateauFuncDraw( plateauApproxGetPf(pa)[0],
208                      bitLess( x[0], x[1], 0.2 ),
209                      bitMore( x[2], x[1], 0.2 ),
210                      80 );
211     plateauFuncDraw( plateauApproxGetPf(pa)[1],
212                      bitLess( x[0], x[1], 0.2 ),
```

```
                    bitMore( x[2], x[1], 0.2 ),
                    80 );
    plateauApproxDraw( pa,
                      bitLess( x[0], x[1], 0.2 ),
                      bitMore( x[2], x[1], 0.2 ),
                      50 );
    _drawSomeLabelling(pa);
    puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
    pkRealVectorFree1(e1);
    pkRealVectorFree1(e2);
    pkRealVectorFree1(a);
    pkRealVectorFree1(b);
    pkRealVectorFree1(c);
    simpleFuncFree0(f[0]);
    simpleFuncFree0(f[1]);
    plateauApproxFree1(pa);

    return;
}

int main( const int argc, const char *argv[] )
{
    //logMsg("-----------------------------| %s |-------------------------------", argv[0]
    _diagram();
    //memPrintf();
    exit(0);
}
```

4.2.9 The `example2.c` file

A listing of the `example2.c` file follows:

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "common.h"
20  #include "simplefunc.h"
21  #include "plateaufunc.h"
22  #include "plateauapprox.h"
```

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

```
23  static void _drawCoordinates( const PKREALVECTOR *e1,
24                                const PKREALVECTOR *e2 )
25  {
26     if ( !e1 || !e2 )
27        return;
28
29     printf( "   %%\\draw[help lines]\n"
30             "      (" FLTFMT "," FLTFMT ")"
31             " grid (" FLTFMT "," FLTFMT ");\n",
32             -pkRealVectorGetComponent(e1)[0] - 0.2, -0.2,
33             pkRealVectorGetComponent(e1)[0] + 0.2, pkRealVectorGetComponent(e2)[1] + 0.2 );
34     puts(   "   %");
35     puts(   "   % Some coordinates.");
36     puts(   "   %");
37     puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
38
39     return;
40  }
```

The `_drawBasisVectors` private function simply `printf()` to standard output Ti*k*Z commands to typeset the $\hat{\mathbf{1}}$ and $\hat{\mathbf{2}}$ basis vectors.

```
41  static void _drawBasisVectors( const PKREALVECTOR *e1,
42                                 const PKREALVECTOR *e2 )
43  {
44     if ( !e1  ||  !e2  )
45        return;
46
47     puts(   "   %");
```

```
48      puts(   "   % The '1' and '2' basisvectors.");
49      puts(   "   %");
50      printf( "   \\draw[pktikzbasisvector,->]\n"
51              "      (" FLTFMT "," FLTFMT ") -- (" FLTFMT "," FLTFMT ") node[right]{$%s$};\n
52              -pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
53              pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
54              pkRealVectorGetName(e1) );
55      printf( "   \\draw[pktikzbasisvector,->]\n"
56              "      (origin) +(0,-4pt) -- (" FLTFMT "," FLTFMT ") node[above]{$%s$};\n",
57              pkRealVectorGetComponent(e2)[0], pkRealVectorGetComponent(e2)[1],
58              pkRealVectorGetName(e2) );
59
60      return;
61  }
```

The `_drawSomeLabelling()` private function simply `printf()`s to standard output a body of Ti*k*Z source code which may be used to typeset some labelling in the figure.

```
62  static void _drawSomeLabelling( const PLATEAUAPPROX *pa )
63  {
64      SIMPLEFUNC *f1,
65                 *f2;
66      PLATEAUFUNC **F;
67      PKREALVECTOR *a,
68                   *b,
69                   *c;
70      PKMATHREAL x;
71
72      if ( !pa )
73         return;
74
75      F = plateauApproxGetPf(pa);
76      f1 = plateauFuncGetSimpleFunc(F[0]);
77      f2 = plateauFuncGetSimpleFunc(F[1]);
78      a = simpleFuncGetP(f1);
79      b = simpleFuncGetQ(f1);
80      c = simpleFuncGetQ(f2);
81
82      puts(   "   %");
83      puts(   "   % Labelling on the x-axis.");
84      puts(   "   %");
85      puts( "   \\draw");
86      printf( "      (" FLTFMT ",0) +(0,-3pt)"
87              " node[below right=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
88              pkRealVectorGetComponent(a)[0],
89              pkRealVectorGetName(a) );
90      printf( "      (" FLTFMT ",0) +(0,-3pt)"
91              " node[below right=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
92              pkRealVectorGetComponent(b)[0],
93              pkRealVectorGetName(b) );
94      printf( "      (" FLTFMT ",0) +(0,-3pt)"
95              " node[below left=0pt and -1.4ex]{$%s$} -- +(0,4pt);\n",
96              pkRealVectorGetComponent(c)[0],
97              pkRealVectorGetName(c) );
98
99      puts( "   %");
100     puts( "   % Labelling the simple functions.");
101     puts( "   %");
102     puts( "   \\path");
103     x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
104     printf( "      (" FLTFMT "," FLTFMT ")"
```

```
105             " node[simplefunctioncolor,above right]{$%s$}\n",
106             x,
107             simpleFuncAt(f1,x),
108             simpleFuncGetName(f1) );
109     x = bitLess( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
110     printf( "      (" FLTFMT "," FLTFMT ")"
111             " node[simplefunctioncolor,above left]{$%s$};\n",
112             x,
113             simpleFuncAt(f2,x),
114             simpleFuncGetName(f2) );
115
116     puts( "   %");
117     puts( "   % Labelling the plateau functions.");
118     puts( "   %");
119     puts( "   \\path");
120     x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
121     printf( "      (" FLTFMT "," FLTFMT ")"
122             " node[plateaufunctioncolor,opaquelabel]{$%s$}\n",
123             x,
124             plateauFuncAt(F[0],x),
125             plateauFuncGetName(F[0]) );
126     x = bitLess( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
127     printf( "      (" FLTFMT "," FLTFMT ")"
128             " node[plateaufunctioncolor,opaquelabel]{$%s$};\n",
129             x,
130             plateauFuncAt(F[1],x),
131             plateauFuncGetName(F[1]) );
132
133     //puts(   "   %");
134     //puts(   "   % Labelling the plateau approximation.");
135     //puts(   "   %");
136     //puts(   "   \\path[->]");
137     //x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.1 );
138     //printf( "      (" FLTFMT "," FLTFMT ") coordinate (a)\n",
139     //        x,
140     //        plateauApproxAt(pa,x) );
141     //x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.5 );
142     //printf( "      (" FLTFMT "," FLTFMT ") node[plateauapproxcolor] (b) {$%s$};\n",
143     //        x,
144     //        plateauApproxAt( pa, pkRealVectorGetComponent(b)[0] ),
145     //        plateauApproxGetName(pa) );
146     //puts(   "   \\draw[pktikzdimension,->] (b) to[out=170,in=40] (a);");
147     puts(   "   %");
148     puts(   "   % Labelling the plateau approximation.");
149     puts(   "   %");
150     puts(   "   \\path[->]");
151     x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
152     printf( "      (" FLTFMT "," FLTFMT ")"
153             " node[plateauapproxcolor,opaquelabel]{$%s$};\n",
154             x,
155             plateauApproxAt(pa,x),
156             plateauApproxGetName(pa) );
157
158     return;
159 }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
160  static void _diagram(void)
161  {
162     const PKMATHREAL accuracy = 0.01;
163     PKREALVECTOR *e1,
164                  *e2,
165                  *a,
166                  *b,
167                  *c;
168     PKMATHREAL x[3];
169     SIMPLEFUNC *f[2];
170     PLATEAUAPPROX *pa;
171
```

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

```
172     e1 = pkRealVectorAlloc1( "x", 2, 7.0, 0.0 );
173     e2 = pkRealVectorAlloc1( "y", 2, 0.0, 5.0 );
174     //pkRealVectorScale(e1,1.2);
175     //pkRealVectorScale(e2,1.2);
176
177     printf( "\\long\\gdef\\egOneAccuracy{%g}\n", accuracy );
178
179     /*
180      * The base points.
181      */
182     x[0] = -0.8 * pkRealVectorGetComponent(e1)[0];
183     x[1] = 0.0;
184     x[2] = -x[0];
185
186     /*
187      * Important positions in the '1.2' plane.
188      */
189     a = pkRealVectorAlloc1( "x\\undr{1}=-1", 2, x[0], 0.0 );
190     b = pkRealVectorAlloc1( "x\\undr{2}=0",  2, x[1], 0.8 * pkRealVectorGetComponent(e2)[1]
191     c = pkRealVectorAlloc1( "1=x\\undr{3}",  2, x[2], 0.0 );
192
193     f[0] = simpleFuncAlloc0( "f\\undr{1}(x)=1+x", linear, a, b, 1.0 );
194     f[1] = simpleFuncAlloc0( "f\\undr{2}(x)=1-x", linear, b, c, 1.0 );
195     pa = plateauApproxAlloc1( "F(x)", 3, x, (const SIMPLEFUNC **)f, accuracy );
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
196     puts(   "\\begin{PkTikzpicture}[scale=1.0]");
197     _drawCoordinates(e1,e2);
198     _drawBasisVectors(e1,e2);
199     simpleFuncDraw( f[0],
200                     bitLess( x[0], x[1], 0.2 ),
201                     bitMore( x[1], x[2], 0.2 ),
202                     2 );
203     simpleFuncDraw( f[1],
204                     bitLess( x[1], x[0], 0.2 ),
205                     bitMore( x[2], x[1], 0.2 ),
206                     2 );
207     plateauFuncDraw( plateauApproxGetPf(pa)[0],
208                      bitLess( x[0], x[1], 0.2 ),
209                      bitMore( x[2], x[1], 0.2 ),
210                      80 );
211     plateauFuncDraw( plateauApproxGetPf(pa)[1],
212                      bitLess( x[0], x[1], 0.2 ),
```

```
                    bitMore( x[2], x[1], 0.2 ),
                    80 );
    plateauApproxDraw( pa,
                      bitLess( x[0], x[1], 0.2 ),
                      bitMore( x[2], x[1], 0.2 ),
                      50 );
    _drawSomeLabelling(pa);
    puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
    pkRealVectorFree1(e1);
    pkRealVectorFree1(e2);
    pkRealVectorFree1(a);
    pkRealVectorFree1(b);
    pkRealVectorFree1(c);
    simpleFuncFree0(f[0]);
    simpleFuncFree0(f[1]);
    plateauApproxFree1(pa);

    return;
}

int main( const int argc, const char *argv[] )
{
    //logMsg("-----------------------------| %s |-------------------------------", argv[0]
    _diagram();
    //memPrintf();
    exit(0);
}
```

4.2.10 The `example3.c` file

A listing of the `example3.c` file follows:

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "common.h"
20  #include "simplefunc.h"
21  #include "plateaufunc.h"
22  #include "plateauapprox.h"
```

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

```
23  static void _drawCoordinates( const PKREALVECTOR *e1,
24                                const PKREALVECTOR *e2 )
25  {
26     if ( !e1 || !e2 )
27        return;
28
29     printf( "   %%\\draw[help lines]\n"
30             "      (" FLTFMT "," FLTFMT ")"
31             " grid (" FLTFMT "," FLTFMT ");\n",
32             -pkRealVectorGetComponent(e1)[0] - 0.2, -0.2,
33             pkRealVectorGetComponent(e1)[0] + 0.2, pkRealVectorGetComponent(e2)[1] + 0.2 );
34     puts(   "   %");
35     puts(   "   % Some coordinates.");
36     puts(   "   %");
37     puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
38
39     return;
40  }
```

The `_drawBasisVectors` private function simply `printf()` to standard output Ti*k*Z commands to typeset the $\hat{\mathbf{1}}$ and $\hat{\mathbf{2}}$ basis vectors.

```
41  static void _drawBasisVectors( const PKREALVECTOR *e1,
42                                 const PKREALVECTOR *e2 )
43  {
44     if ( !e1  ||  !e2  )
45        return;
46
47     puts(   "   %");
```

```
48      puts(   "   % The '1' and '2' basisvectors.");
49      puts(   "   %");
50      printf( "   \\draw[pktikzbasisvector,->]\n"
51              "      (" FLTFMT "," FLTFMT ") -- (" FLTFMT "," FLTFMT ") node[right]{$%s$};\n
52              -pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
53              pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
54              pkRealVectorGetName(e1) );
55      printf( "   \\draw[pktikzbasisvector,->]\n"
56              "      (origin) +(0,-4pt) -- (" FLTFMT "," FLTFMT ") node[above]{$%s$};\n",
57              pkRealVectorGetComponent(e2)[0], pkRealVectorGetComponent(e2)[1],
58              pkRealVectorGetName(e2) );
59
60      return;
61  }
```

The `_drawSomeLabelling()` private function simply `printf()`s to standard output a body of Ti*k*Z source code which may be used to typeset some labelling in the figure.

```
62  static void _drawSomeLabelling( const PLATEAUAPPROX *pa )
63  {
64      SIMPLEFUNC *f1,
65                 *f2;
66      PLATEAUFUNC **F;
67      PKREALVECTOR *a,
68                   *b1,
69                   *c;
70      PKMATHREAL x;
71
72      if ( !pa )
73         return;
74
75      F = plateauApproxGetPf(pa);
76      f1 = plateauFuncGetSimpleFunc(F[0]);
77      f2 = plateauFuncGetSimpleFunc(F[1]);
78      a = simpleFuncGetP(f1);
79      b1 = simpleFuncGetQ(f1);
80      c = simpleFuncGetQ(f2);
81
82      puts(   "   %");
83      puts(   "   % Labelling on the x-axis.");
84      puts(   "   %");
85      puts( "   \\draw");
86      printf( "      (" FLTFMT ",0) +(0,-3pt)"
87              " node[below right=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
88              pkRealVectorGetComponent(a)[0],
89              pkRealVectorGetName(a) );
90      printf( "      (" FLTFMT ",0) +(0,-3pt)"
91              " node[below right=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
92              pkRealVectorGetComponent(b1)[0],
93              pkRealVectorGetName(b1) );
94      printf( "      (" FLTFMT ",0) +(0,-3pt)"
95              " node[below left=0pt and -1.4ex]{$%s$} -- +(0,4pt);\n",
96              pkRealVectorGetComponent(c)[0],
97              pkRealVectorGetName(c) );
98
99      puts( "   %");
100     puts( "   % Labelling the simple functions.");
101     puts( "   %");
102     puts( "   \\path");
103     x = bitMore( pkRealVectorGetComponent(b1)[0], pkRealVectorGetComponent(c)[0], 0.2 );
104     printf( "      (" FLTFMT "," FLTFMT ")"
```

```
105             " node[simplefunctioncolor,above right]{$%s$}\n",
106             x,
107             simpleFuncAt(f1,x),
108             simpleFuncGetName(f1) );
109     x = bitLess( pkRealVectorGetComponent(b1)[0], pkRealVectorGetComponent(c)[0], 0.8 );
110     printf( "      (" FLTFMT "," FLTFMT ")"
111             " node[simplefunctioncolor,above]{$%s$};\n",
112             x,
113             simpleFuncAt(f2,x),
114             simpleFuncGetName(f2) );
115
116     puts( "   %");
117     puts( "   % Labelling the plateau functions.");
118     puts( "   %");
119     puts( "   \\path");
120     x = bitMore( pkRealVectorGetComponent(b1)[0], pkRealVectorGetComponent(c)[0], 0.3 );
121     printf( "      (" FLTFMT "," FLTFMT ")"
122             " node[plateaufunctioncolor,opaquelabel]{$%s$}\n",
123             x,
124             plateauFuncAt(F[0],x),
125             plateauFuncGetName(F[0]) );
126     x = bitLess( pkRealVectorGetComponent(b1)[0], pkRealVectorGetComponent(c)[0], 0.2 );
127     printf( "      (" FLTFMT "," FLTFMT ")"
128             " node[plateaufunctioncolor,opaquelabel]{$%s$};\n",
129             x,
130             plateauFuncAt(F[1],x),
131             plateauFuncGetName(F[1]) );
132
133     puts(   "   %");
134     puts(   "   % Labelling the plateau approximation.");
135     puts(   "   %");
136     puts(   "   \\path[->]");
137     x = bitMore( pkRealVectorGetComponent(b1)[0], pkRealVectorGetComponent(c)[0], 0.2 );
138     printf( "      (" FLTFMT "," FLTFMT ")"
139             " node[plateauapproxcolor,opaquelabel]{$%s$};\n",
140             x,
141             plateauApproxAt(pa,x),
142             plateauApproxGetName(pa) );
143
144     return;
145 }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
146 static void _diagram(void)
147 {
148     const PKMATHREAL accuracy = 0.01;
149     PKREALVECTOR *e1,
150                  *e2,
151                  *a,
152                  *b1,
153                  *b2,
154                  *c;
155     PKMATHREAL x[3];
156     SIMPLEFUNC *f[2];
157     PLATEAUAPPROX *pa;
158
```

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

```
159     e1 = pkRealVectorAlloc1( "x", 2, 7.0, 0.0 );
160     e2 = pkRealVectorAlloc1( "y", 2, 0.0, 5.0 );
161     //pkRealVectorScale(e1,1.2);
162     //pkRealVectorScale(e2,1.2);
163
164     printf( "\\long\\gdef\\egTwoAccuracy{%g}\n", accuracy );
165
166     /*
167      * The base points.
168      */
169     x[0] = -0.8 * pkRealVectorGetComponent(e1)[0];
170     x[1] = 0.0;
171     x[2] = -x[0];
172
173     /*
174      * Important positions in the '1.2' plane.
175      */
176     a = pkRealVectorAlloc1( "x\\undr{1}=-1", 2, x[0], 0.0 );
177     b1 = pkRealVectorAlloc1( "x\\undr{2}=0", 2, x[1], 0.8 * pkRealVectorGetComponent(e2)[1]
178     b2 = pkRealVectorAlloc1( "",             2, x[1], 5.0 / 8.0 * pkRealVectorGetComponent
179     c = pkRealVectorAlloc1( "1=x\\undr{3}",  2, x[2], 0.0 );
180
181     f[0] = simpleFuncAlloc0( "f\\undr{1}(x)=1+x", linear, a,  b1, 1.0 );
182     f[1] = simpleFuncAlloc0( "f\\undr{2}(x)=\\myFiveEighth(1-x)", linear, b2, c,  1.0 );
183     pa = plateauApproxAlloc1( "F(x)", 3, x, (const SIMPLEFUNC **)f, accuracy );
184     //plateauApproxPrint(pa);
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
185     puts(   "\\begin{PkTikzpicture}[scale=1.0]");
186     _drawCoordinates(e1,e2);
187     _drawBasisVectors(e1,e2);
188     simpleFuncDraw( f[0],
189                     bitLess( x[0], x[1], 0.2 ),
190                     bitMore( x[1], x[2], 0.2 ),
191                     2 );
192     simpleFuncDraw( f[1],
193                     bitLess( x[1], x[0], 0.8 ),
194                     bitMore( x[2], x[1], 0.2 ),
195                     2 );
196     plateauFuncDraw( plateauApproxGetPf(pa)[0],
197                      bitLess( x[0], x[1], 0.2 ),
198                      bitMore( x[2], x[1], 0.2 ),
199                      80 );
200     plateauFuncDraw( plateauApproxGetPf(pa)[1],
201                      bitLess( x[0], x[1], 0.2 ),
202                      bitMore( x[2], x[1], 0.2 ),
203                      80 );
204     plateauApproxDraw( pa,
205                        bitLess( x[0], x[1], 0.2 ),
206                        bitMore( x[2], x[1], 0.2 ),
207                        50 );
208     _drawSomeLabelling(pa);
209     puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
210      pkRealVectorFree1(e1);
211      pkRealVectorFree1(e2);
212      pkRealVectorFree1(a);
213      pkRealVectorFree1(b1);
214      pkRealVectorFree1(b2);
215      pkRealVectorFree1(c);
216      simpleFuncFree0(f[0]);
217      simpleFuncFree0(f[1]);
218      plateauApproxFree1(pa);
219
220      return;
221   }

222   int main( const int argc, const char *argv[] )
223   {
224      //logMsg("----------------------------| %s |-----------------------------", argv[0]
225      _diagram();
226      //memPrintf();
227      exit(0);
228   }
```

4.2.11 The `example5.c` file

A listing of the `example5.c` file follows:

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "common.h"
20  #include "simplefunc.h"
21  #include "plateaufunc.h"
22  #include "plateauapprox.h"
```

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

```
23  static void _drawCoordinates( const PKREALVECTOR *e1,
24                                const PKREALVECTOR *e2 )
25  {
26     if ( !e1 || !e2 )
27        return;
28
29     printf( "   %%\\draw[help lines]\n"
30             "      (" FLTFMT "," FLTFMT ")"
31             " grid (" FLTFMT "," FLTFMT ");\n",
32             -pkRealVectorGetComponent(e1)[0] - 0.2, -0.2,
33             2.0 * pkRealVectorGetComponent(e1)[0] + 0.2, pkRealVectorGetComponent(e2)[1] +
34     puts(   "   %");
35     puts(   "   % Some coordinates.");
36     puts(   "   %");
37     puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
38
39     return;
40  }
```

The `_drawBasisVectors` private function simply `printf()` to standard output Ti*k*Z commands to typeset the $\hat{\mathbf{1}}$ and $\hat{\mathbf{2}}$ basis vectors.

```
41  static void _drawBasisVectors( const PKREALVECTOR *e1,
42                                 const PKREALVECTOR *e2 )
43  {
44     if ( !e1  ||  !e2  )
45        return;
46
47     puts(   "   %");
```

```
48      puts(   "   % The '1' and '2' basisvectors.");
49      puts(   "   %");
50      printf( "   \\draw[pktikzbasisvector,->]\n"
51              "      (" FLTFMT "," FLTFMT ") -- (" FLTFMT "," FLTFMT ") node[right]{$%s$};\n
52              -pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
53              2.0 * pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
54              pkRealVectorGetName(e1) );
55      printf( "   \\draw[pktikzbasisvector,->]\n"
56              "      (origin) +(0,-4pt) -- (" FLTFMT "," FLTFMT ") node[above]{$%s$};\n",
57              pkRealVectorGetComponent(e2)[0], pkRealVectorGetComponent(e2)[1],
58              pkRealVectorGetName(e2) );
59
60      return;
61  }
```

The `_drawSomeLabelling()` private function simply `printf()`s to standard output a body of Ti*k*Z source code which may be used to typeset some labelling in the figure.

```
62  static void _drawSomeLabelling( const PLATEAUAPPROX *pa )
63  {
64      SIMPLEFUNC *f1,
65                 *f2,
66                 *f3;
67      PLATEAUFUNC **F;
68      PKREALVECTOR *a,
69                   *b,
70                   *c,
71                   *d;
72      PKMATHREAL x;
73
74      if ( !pa )
75         return;
76
77      F = plateauApproxGetPf(pa);
78      f1 = plateauFuncGetSimpleFunc(F[0]);
79      f2 = plateauFuncGetSimpleFunc(F[1]);
80      f3 = plateauFuncGetSimpleFunc(F[2]);
81      a = simpleFuncGetP(f1);
82      b = simpleFuncGetP(f2);
83      c = simpleFuncGetP(f3);
84      d = simpleFuncGetQ(f3);
85
86      puts(   "   %");
87      puts(   "   % Labelling on the x-axis.");
88      puts(   "   %");
89      puts( "   \\draw");
90      printf( "      (" FLTFMT ",0) +(0,-3pt)"
91              " node[below right=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
92              pkRealVectorGetComponent(a)[0],
93              pkRealVectorGetName(a) );
94      printf( "      (" FLTFMT ",0) +(0,-3pt)"
95              " node[below right=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
96              pkRealVectorGetComponent(b)[0],
97              pkRealVectorGetName(b) );
98      printf( "      (" FLTFMT ",0) +(0,-3pt)"
99              " node[below left=0pt and -1.4ex]{$%s$} -- +(0,4pt)\n",
100             pkRealVectorGetComponent(c)[0],
101             pkRealVectorGetName(c) );
102     printf( "      (" FLTFMT ",0) +(0,-3pt)"
103             " node[below left=0pt and -1.4ex]{$%s$} -- +(0,4pt);\n",
104             pkRealVectorGetComponent(d)[0],
```

```
105             pkRealVectorGetName(d) );
106
107     puts( "   %");
108     puts( "   % Labelling the simple functions.");
109     puts( "   %");
110     puts( "   \\path");
111     x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
112     printf( "      (" FLTFMT "," FLTFMT ")"
113             " node[simplefunctioncolor,above right]{$%s$}\n",
114             x,
115             simpleFuncAt(f1,x),
116             simpleFuncGetName(f1) );
117     x = bitLess( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
118     printf( "      (" FLTFMT "," FLTFMT ")"
119             " node[simplefunctioncolor,above left]{$%s$}\n",
120             x,
121             simpleFuncAt(f2,x),
122             simpleFuncGetName(f2) );
123     x = bitMore( pkRealVectorGetComponent(d)[0], pkRealVectorGetComponent(c)[0], 0.2 );
124     printf( "      (" FLTFMT "," FLTFMT ")"
125             " node[simplefunctioncolor,above]{$%s$};\n",
126             x,
127             simpleFuncAt(f3,x),
128             simpleFuncGetName(f3) );
129
130     puts( "   %");
131     puts( "   % Labelling the plateau functions.");
132     puts( "   %");
133     puts( "   \\path");
134     x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
135     printf( "      (" FLTFMT "," FLTFMT ")"
136             " node[plateaufunctioncolor,opaquelabel]{$%s$}\n",
137             x,
138             plateauFuncAt(F[0],x),
139             plateauFuncGetName(F[0]) );
140     x = bitLess( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
141     printf( "      (" FLTFMT "," FLTFMT ")"
142             " node[plateaufunctioncolor,opaquelabel]{$%s$}\n",
143             x,
144             plateauFuncAt(F[1],x),
145             plateauFuncGetName(F[1]) );
146     x = bitMore( pkRealVectorGetComponent(c)[0], pkRealVectorGetComponent(d)[0], 0.2 );
147     printf( "      (" FLTFMT "," FLTFMT ")"
148             " node[plateaufunctioncolor,opaquelabel]{$%s$};\n",
149             x,
150             plateauFuncAt(F[2],x),
151             plateauFuncGetName(F[2]) );
152
153     puts(   "   %");
154     puts(   "   % Labelling the plateau approximation.");
155     puts(   "   %");
156     puts(   "   \\path[->]");
157     x = bitMore( pkRealVectorGetComponent(b)[0], pkRealVectorGetComponent(c)[0], 0.2 );
158     printf( "      (" FLTFMT "," FLTFMT ")"
159             " node[plateauapproxcolor,opaquelabel]{$%s$};\n",
160             x,
161             plateauApproxAt(pa,x),
162             plateauApproxGetName(pa) );
163
164     return;
165 }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
166  static void _diagram(void)
167  {
168     const PKMATHREAL accuracy = 0.00001;
169     PKREALVECTOR *e1,
170                  *e2,
171                  *a,
172                  *b,
173                  *c,
174                  *d;
175     PKMATHREAL x[4];
176     SIMPLEFUNC *f[3];
177     PLATEAUAPPROX *pa;
178
```

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

```
179     e1 = pkRealVectorAlloc1( "x", 2, 5.0, 0.0 );
180     e2 = pkRealVectorAlloc1( "y", 2, 0.0, 4.0 );
181     //pkRealVectorScale(e1,1.2);
182     //pkRealVectorScale(e2,1.2);
183
184     printf( "\\long\\gdef\\egFourAccuracy{%f}\n", accuracy );
185
186     /*
187      * The base points.
188      */
189     x[0] = -0.8 * pkRealVectorGetComponent(e1)[0];
190     x[1] = 0.0;
191     x[2] = -x[0];
192     x[3] = -2.0 * x[0];
193
194     /*
195      * Important positions in the '1.2' plane.
196      */
197     a = pkRealVectorAlloc1( "x\\undr{1}=-1", 2, x[0], 0.0 );
198     b = pkRealVectorAlloc1( "x\\undr{2}=0",  2, x[1], 0.8 * pkRealVectorGetComponent(e2)[1]
199     c = pkRealVectorAlloc1( "x\\undr{3}",  2, x[2], 0.0 );
200     d = pkRealVectorAlloc1( "x\\undr{4}",  2, x[3], pkRealVectorGetComponent(b)[1] );
201
202     f[0] = simpleFuncAlloc0( "f\\undr{1}(x)=1+x", linear, a, b, 1.0 );
203     f[1] = simpleFuncAlloc0( "f\\undr{2}(x)=1-x", linear, b, c, 1.0 );
204     f[2] = simpleFuncAlloc0( "f\\undr{3}(x)=x-1", linear, c, d, 1.0 );
205     pa = plateauApproxAlloc1( "F(x)", 4, x, (const SIMPLEFUNC **)f, accuracy );
206     //plateauApproxPrint(pa);
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
207     puts(   "\\begin{PkTikzpicture}[scale=1.0]");
208     _drawCoordinates(e1,e2);
209     _drawBasisVectors(e1,e2);
210     simpleFuncDraw( f[0],
211                     bitLess( x[0], x[1], 0.2 ),
212                     bitMore( x[1], x[2], 0.2 ),
```

```
                    2 );
    simpleFuncDraw( f[1],
                    bitLess( x[1], x[0], 0.2 ),
                    bitMore( x[2], x[1], 0.2 ),
                    2 );
    simpleFuncDraw( f[2],
                    bitLess( x[2], x[1], 0.2 ),
                    bitMore( x[3], x[2], 0.2 ),
                    2 );
    plateauFuncDraw( plateauApproxGetPf(pa)[0],
                     bitLess( x[0], x[1], 0.2 ),
                     bitMore( x[3], x[2], 0.2 ),
                     80 );
    plateauFuncDraw( plateauApproxGetPf(pa)[1],
                     bitLess( x[0], x[1], 0.2 ),
                     bitMore( x[3], x[2], 0.2 ),
                     80 );
    plateauFuncDraw( plateauApproxGetPf(pa)[2],
                     bitLess( x[0], x[1], 0.2 ),
                     bitMore( x[3], x[2], 0.2 ),
                     80 );
    plateauApproxDraw( pa,
                       bitLess( x[0], x[1], 0.2 ),
                       bitMore( x[3], x[2], 0.2 ),
                       50 );
    _drawSomeLabelling(pa);
    puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
    pkRealVectorFree1(e1);
    pkRealVectorFree1(e2);
    pkRealVectorFree1(a);
    pkRealVectorFree1(b);
    pkRealVectorFree1(c);
    pkRealVectorFree1(d);
    simpleFuncFree0(f[0]);
    simpleFuncFree0(f[1]);
    simpleFuncFree0(f[2]);
    plateauApproxFree1(pa);

    return;
}


int main( const int argc, const char *argv[] )
{
    //logMsg("----------------------------| %s |-------------------------------", argv[0]
    _diagram();
    //memPrintf();
    exit(0);
}
```

4.2.12 The `example6.c` file

A listing of the `example6.c` file follows:

```
1   #include <pkfeatures.h>
2
3   #include <stddef.h>
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <unistd.h>
7   #include <stdarg.h>
8   #include <string.h>
9   #include <math.h>
10  #include <float.h>
11
12  #include <pkmemdebug.h>
13  #include <pkerror.h>
14  #include <pktypes.h>
15  #include <pkstring.h>
16  #include <pkmath.h>
17  #include <pkrealvector.h>
18
19  #include "common.h"
20  #include "simplefunc.h"
21  #include "plateaufunc.h"
22  #include "plateauapprox.h"
```

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

```
23  static void _drawCoordinates( const PKREALVECTOR *e1,
24                                const PKREALVECTOR *e2 )
25  {
26     if ( !e1 || !e2 )
27        return;
28
29     printf( "   %%\\draw[help lines] (-0.2,-0.2) grid (" FLTFMT "," FLTFMT ");\n",
30             pkRealVectorGetComponent(e1)[0] + 0.2,
31             pkRealVectorGetComponent(e2)[1] + 0.2 );
32     puts(   "   %");
33     puts(   "   % Some coordinates.");
34     puts(   "   %");
35     puts(   "   \\pktikzSetUncircledPoint{(0,0)}{origin};" );
36
37     return;
38  }
```

The `_drawBasisVectors` private function simply `printf()` to standard output Ti*k*Z commands to typeset the $\hat{\mathbf{1}}$ and $\hat{\mathbf{2}}$ basis vectors.

```
39  static void _drawBasisVectors( const PKREALVECTOR *e1,
40                                 const PKREALVECTOR *e2 )
41  {
42     if ( !e1  ||  !e2  )
43        return;
44
45     puts(   "   %");
46     puts(   "   % The '1' and '2' basisvectors.");
47     puts(   "   %");
```

```
48      printf( "   \\draw[basisvector,->]\n"
49              "      (" FLTFMT "," FLTFMT ") -- (" FLTFMT "," FLTFMT ") node[right]{$%s$};\n
50              -pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
51              pkRealVectorGetComponent(e1)[0], pkRealVectorGetComponent(e1)[1],
52              pkRealVectorGetName(e1) );
53      printf( "   \\draw[basisvector,->]\n"
54              "      (origin) +(0,-4pt) -- (" FLTFMT "," FLTFMT ") node[above]{$%s$};\n",
55              pkRealVectorGetComponent(e2)[0], pkRealVectorGetComponent(e2)[1],
56              pkRealVectorGetName(e2) );
57
58      return;
59  }
```

The `_drawSomeLabelling()` private function simply `printf()`s to standard output a body of Ti*k*Z source code which may be used to typeset some labelling in the figure.

```
60  static void _drawSomeLabelling( const PLATEAUAPPROX *pa )
61  {
62      PKMATHREAL *basepoint,
63                 w;        /* Simple buffer. */
64      PLATEAUFUNC **F;
65      SIMPLEFUNC *f1,
66                 *f2;
67      PKREALVECTOR *x[3],  /* Positions on the simple functions */
68                           /*    above the basepoints. */
69                   *p[4];  /* Positions defining the simple functions. */
70      int i;
71
72      if ( !pa )
73         return;
74
75      basepoint = plateauApproxGetBasepoint(pa);
76      F = plateauApproxGetPf(pa);
77      f1 = plateauFuncGetSimpleFunc(F[0]);
78      f2 = plateauFuncGetSimpleFunc(F[1]);
79
80      p[0] = simpleFuncGetP(f1);
81      p[1] = simpleFuncGetQ(f1);
82      p[2] = simpleFuncGetP(f2);
83      p[3] = simpleFuncGetQ(f2);
84
85      x[0] = pkRealVectorAlloc1( "", 2, basepoint[0], plateauFuncAt( F[0], basepoint[0] ) );
86      x[1] = pkRealVectorAlloc1( "", 2, basepoint[1], plateauFuncAt( F[0], basepoint[1] ) );
87      x[2] = pkRealVectorAlloc1( "", 2, basepoint[2], plateauFuncAt( F[1], basepoint[2] ) );
88
89      puts(   "   %");
90      puts(   "   % Labelling on the x-axis.");
91      puts(   "   %");
92      puts(   "   \\draw");
93      for ( i = 0; i < 3; i++ )
94         printf( "      (" FLTFMT ",0) +(0,-2pt) node[below]{$x\\undr{%d}$} -- +(0,4pt)%s\n"
95                 pkRealVectorGetComponent(x[i])[0],
96                 i + 1,
97                 ( i == 3 - 1 ) ? ";" : "" );
98      puts(   "   \\draw[pktikzdimension]");
99      for ( i = 0; i < 3; i++ )
100        printf( "      (" FLTFMT ",0) +(0,2pt) -- ++(0," FLTFMT ")%s\n",
101                pkRealVectorGetComponent(x[i])[0],
102                pkRealVectorGetComponent(x[i])[1],
103                ( i == 3 - 1 ) ? ";" : "" );
104
```

```
105     pkRealVectorFree1( x[0] );
106     pkRealVectorFree1( x[1] );
107     pkRealVectorFree1( x[2] );
108
109     puts( "   %");
110     puts( "   % Labelling the simple functions.");
111     puts( "   %");
112     puts( "   \\path");
113     w = bitLess( pkRealVectorGetComponent(p[0])[0], pkRealVectorGetComponent(p[1])[0], 0.2
114     printf( "      (" FLTFMT "," FLTFMT ")"
115             " node[simplefunctioncolor,opaquelabel]{$%s$}\n",
116             w,
117             simpleFuncAt(f1,w),
118             simpleFuncGetName(f1) );
119     w = bitMore( pkRealVectorGetComponent(p[2])[0], pkRealVectorGetComponent(p[3])[0], 0.0
120     printf( "      (" FLTFMT "," FLTFMT ")"
121             " node[simplefunctioncolor,opaquelabel]{$%s$};\n",
122             w,
123             simpleFuncAt(f2,w),
124             simpleFuncGetName(f2) );
125
126     puts( "   %");
127     puts( "   % Labelling the plateau functions.");
128     puts( "   %");
129     puts( "   \\path");
130     w = bitMore( basepoint[1], basepoint[0], 0.2 );
131     printf( "      (" FLTFMT "," FLTFMT ")"
132             " node[plateaufunctioncolor,opaquelabel]{$%s$}\n",
133             w,
134             plateauFuncAt(F[0],w),
135             plateauFuncGetName(F[0]) );
136     w = bitLess( basepoint[1], basepoint[2], 0.15 );
137     printf( "      (" FLTFMT "," FLTFMT ")"
138             " node[plateaufunctioncolor,opaquelabel]{$%s$};\n",
139             w,
140             plateauFuncAt(F[1],w),
141             plateauFuncGetName(F[1]) );
142
143     puts(   "   %");
144     puts(   "   % Labelling the plateau approximation.");
145     puts(   "   %");
146     puts(   "   \\path[->]");
147     w = bitMore( basepoint[1], basepoint[0], 0.8 );
148     printf( "      (" FLTFMT "," FLTFMT ")"
149             " node[plateauapproxcolor,opaquelabel]{$%s$};\n",
150             w,
151             plateauApproxAt(pa,w),
152             plateauApproxGetName(pa) );
153
154     puts(   "   %");
155     puts(   "   % Labelling the important positions.");
156     puts(   "   %");
157     puts(   "   \\path");
158     for ( i = 0; i < 4; i++ )
159        printf( "      (" FLTFMT "," FLTFMT ") coordinate[pktikzpoint] node[right]{$%s$}%s\
160                pkRealVectorGetComponent(p[i])[0],
161                pkRealVectorGetComponent(p[i])[1],
162                pkRealVectorGetName(p[i]),
163                ( i == 4 - 1 ) ? ";" : "" );
164
165     return;
166 }
```

Initialise the diagram's primary landscape parameters, and then draw the landscape. The `_diagram()` private function below specifies the diagram's landscape. The function prints to standard output a body of Ti*k*Z source code which may be used to typeset the diagram's landscape in TEX.

```
167  static void _diagram(void)
168  {
169     const PKMATHREAL accuracy = 0.001,
170                      sIntensity = 0.4,
171                      tIntensity = 0.19;
172     const PKMATHREAL e1Len = 14.0,
173                      e2Height = 7.0;
174     PKREALVECTOR *e1,
175                  *e2,
176                  *a,
177                  *b,
178                  *c,
179                  *d;
180     PKMATHREAL basepoint[3];
181     SIMPLEFUNC *f[2];
182     PLATEAUAPPROX *pa;
183
```

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

```
184     e1 = pkRealVectorAlloc1( "x", 2, e1Len, 0.0 );
185     e2 = pkRealVectorAlloc1( "y", 2, 0.0,   e2Height );
186
187     printf( "\\long\\gdef\\egFiveAccuracy{%g}\n",    accuracy );
188     printf( "\\long\\gdef\\egFiveSintensity{%g}\n", sIntensity );
189     printf( "\\long\\gdef\\egFiveTintensity{%g}\n", tIntensity );
190
191     /*
192      * The base points.
193      */
194     basepoint[0] = 1.0 / 14.0 * e1Len;
195     basepoint[1] = 5.5 / 14.0 * e1Len;
196     basepoint[2] = 13.0 / 14.0 * e1Len;
197
198     /*
199      * Important positions in the '1.2' plane.
200      */
201     a = pkRealVectorAlloc1( "\\veca", 2,  3.0 / 14.0 * e1Len, 3.0 / 7.0 * e2Height );
202     b = pkRealVectorAlloc1( "\\vecb", 2,  8.0 / 14.0 * e1Len, 4.0 / 7.0 * e2Height );
203     c = pkRealVectorAlloc1( "\\vecc", 2, 12.0 / 14.0 * e1Len, 5.0 / 7.0 * e2Height );
204     d = pkRealVectorAlloc1( "\\vecd", 2,  4.0 / 14.0 * e1Len, 4.0 / 7.0 * e2Height );
205
206     f[0] = simpleFuncAlloc0( "f\\undr{1}(x)", quadratic, a, b, sIntensity );
207     f[1] = simpleFuncAlloc0( "f\\undr{2}(x)", quadratic, c, d, tIntensity );
208     pa = plateauApproxAlloc1( "F(x)",
209                               3, basepoint,
210                               (const SIMPLEFUNC **)f,
211                               accuracy );
212     //plateauApproxPrint(pa); exit(0);
```

Prepare the Ti*k*Z commands for typesetting the diagram.

```
213     puts(   "\\begin{PkTikzpicture}[scale=1.0]");
```

```
214     _drawCoordinates(e1,e2);
215     drawBasisVectors(e1,e2);
216     simpleFuncDraw( f[0],
217                     bitMore( basepoint[0], basepoint[1], 0.1 ),
218                     bitMore( basepoint[1], basepoint[0], 0.7 ),
219                     20 );
220     simpleFuncDraw( f[1],
221                     bitLess( basepoint[1], basepoint[2], 0.4 ),
222                     bitLess( basepoint[2], basepoint[1], 0.03 ),
223                     20 );
224     plateauFuncDraw( plateauApproxGetPf(pa)[0],
225                      bitLess( basepoint[0], basepoint[1], 0.15 ),
226                      bitMore( basepoint[1], basepoint[0], 0.5 ),
227                      80 );
228     plateauFuncDraw( plateauApproxGetPf(pa)[1],
229                      bitLess( basepoint[1], basepoint[2], 0.4 ),
230                      bitMore( basepoint[2], basepoint[1], 0.13 ),
231                      80 );
232     plateauApproxDraw( pa,
233                        bitLess( basepoint[0], basepoint[1], 0.15 ),
234                        bitMore( basepoint[2], basepoint[1], 0.13 ),
235                        50 );
236     _drawSomeLabelling(pa);
237     puts(   "\\end{PkTikzpicture}");
```

Finally, clean up.

```
238     pkRealVectorFree1(e1);
239     pkRealVectorFree1(e2);
240     pkRealVectorFree1(a);
241     pkRealVectorFree1(b);
242     pkRealVectorFree1(c);
243     pkRealVectorFree1(d);
244     simpleFuncFree0(f[0]);
245     simpleFuncFree0(f[1]);
246     plateauApproxFree1(pa);
247
248     return;
249 }


250 int main( const int argc, const char *argv[] )
251 {
252     //logMsg("----------------------------| %s |-----------------------------", argv[0]
253     _diagram();
254     //memPrintf();
255     exit(0);
256 }
```

4.2.13 The common.h and common.c files

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

```
1   #ifndef _COMMON
2   #define _COMMON
```

Inclusions.

```
3   #include <pkfeatures.h>
4
5   #include <stddef.h>
6   #include <stdlib.h>
7   #include <stdio.h>
8   #include <unistd.h>
9   #include <stdarg.h>
10  #include <string.h>
11  #include <math.h>
12  #include <float.h>
13
14  #include <pkmemdebug.h>
15  #include <pkerror.h>
16  #include <pktypes.h>
17  #include <pkstring.h>
18  #include <pkmath.h>
19  #include <pkrealvector.h>
```

Macro definitions.

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

Function declarations.

```
21  extern PKMATHREAL bitLess( const PKMATHREAL x1, const PKMATHREAL x2, const PKMATHREAL del
22  extern PKMATHREAL bitMore( const PKMATHREAL x1, const PKMATHREAL x2, const PKMATHREAL del
23  extern PKMATHREAL realMin( const PKMATHREAL a, const PKMATHREAL b );
24  extern PKMATHREAL linear( const PKMATHREAL x,
25                            const PKMATHREAL p1,
26                            const PKMATHREAL p2,
27                            const PKMATHREAL q1,
28                            const PKMATHREAL q2,
29                            const PKMATHREAL r );
30  extern PKMATHREAL quadratic( const PKMATHREAL x,
31                               const PKMATHREAL p1,
32                               const PKMATHREAL p2,
33                               const PKMATHREAL q1,
34                               const PKMATHREAL q2,
35                               const PKMATHREAL a );
36  extern PKMATHREAL exponential( const PKMATHREAL x,
37                                 const PKMATHREAL p1,
38                                 const PKMATHREAL p2,
39                                 const PKMATHREAL q1,
40                                 const PKMATHREAL q2,
41                                 const PKMATHREAL r );
42  extern PKMATHREAL upFunc( const PKMATHREAL x, const PKMATHREAL base );
43  extern PKMATHREAL doFunc( const PKMATHREAL x, const PKMATHREAL base );
44  extern PKMATHREAL udFunc( const PKMATHREAL x,
```

```
45                              const PKMATHREAL x1,
46                              const PKMATHREAL x2,
47                              const PKMATHREAL base );
48  extern void drawBasisVectors( const PKREALVECTOR *e1,
49                                const PKREALVECTOR *e2 );
50  extern void drawInterestFunction(void);

51  #endif
```

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

```
1   #include "common.h"
2
3   #include <pkfeatures.h>
4
5   #include <stddef.h>
6   #include <stdlib.h>
7   #include <stdio.h>
8   #include <unistd.h>
9   #include <stdarg.h>
10  #include <string.h>
11  #include <math.h>
12
13  #include <pkmemdebug.h>
14  #include <pkerror.h>
15  #include <pktypes.h>
16  #include <pkstring.h>
17  #include <pkmath.h>
18  #include <pkrealvector.h>

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

The `bitLess()` function returns a `PKMATHREAL` number less than the specified `x1`. Specifically, it returns

$$x_1 - \delta\,|x_2 - x_1|$$

```
20  PKMATHREAL bitLess( const PKMATHREAL x1, const PKMATHREAL x2, const PKMATHREAL delta )
21  {
22     return( x1 - delta * fabs( x2 - x1 ) );
23  }
```

The `bitMore()` function returns a `PKMATHREAL` number more than the specified `x2`. Specifically, it returns

$$x_1 + \delta\,|x_2 - x_1|$$

```
24  PKMATHREAL bitMore( const PKMATHREAL x1, const PKMATHREAL x2, const PKMATHREAL delta )
25  {
26     return( x1 + delta * fabs( x2 - x1 ) );
27  }

28  PKMATHREAL realMin( const PKMATHREAL a, const PKMATHREAL b )
29  {
30     return( ( a < b ) ? a : b );
31  }
```

The `linear()` function simply returns the `PKMATHREAL` value

$$f(x; p_1, p_2, q_1, q_2, a) = p_2 + (\frac{q_2 - p_2}{q_1 - p_1})(x - p_1)$$

By doing so, it returns the value of a linear function at the domain value x, and where the function passes through the positions $\mathbf{p} = p_1\hat{\mathbf{1}} + p_2\hat{\mathbf{2}}$ and $\mathbf{q} = q_1\hat{\mathbf{1}} + q_2\hat{\mathbf{2}}$. Note that the specified `r` is not used. It's there because `linear()`'s function signature must comply.

```
32  PKMATHREAL linear( const PKMATHREAL x,
33                     const PKMATHREAL p1,
34                     const PKMATHREAL p2,
35                     const PKMATHREAL q1,
36                     const PKMATHREAL q2,
37                     const PKMATHREAL r )
38  {
39     return( p2 + ( q2 - p2 ) / ( q1 - p1 ) * ( x - p1 ) );
40  }
```

The `quadratic()` function simply returns the `PKMATHREAL` value

$$f(x; p_1, p_2, q_1, q_2, a) = a(x - p_1)(x - q_1) + (\frac{x - p_1}{q_1 - p_1})(q_2 - p_2) + p_2$$

By doing so, it returns the value of a quadratic function at the domain value x, and where the function passes through the positions $\mathbf{p} = p_1\hat{\mathbf{1}} + p_2\hat{\mathbf{2}}$ and $\mathbf{q} = q_1\hat{\mathbf{1}} + q_2\hat{\mathbf{2}}$.

```
41  PKMATHREAL quadratic( const PKMATHREAL x,
42                        const PKMATHREAL p1,
43                        const PKMATHREAL p2,
44                        const PKMATHREAL q1,
45                        const PKMATHREAL q2,
46                        const PKMATHREAL a )
47  {
48     return(  a * ( x - p1 ) * ( x - q1 )
49            + ( x - p1 ) / ( q1 - p1 ) * ( q2 - p2) +  p2 );
50  }
```

The `exponential()` function simply returns the `PKMATHREAL` value

$$f(x; p_1, p_2, q_1, q_2, r) = \left(\frac{e^{-rx} - e^{-rp_1}}{e^{-rq_2} - e^{-rp_1}}\right)(q_2 - p_2) + p_2$$

By doing so, it returns the value of an exponential function at the domain value x, and where the function passes through the positions $\mathbf{p} = p_1\hat{\mathbf{1}} + p_2\hat{\mathbf{2}}$ and $\mathbf{q} = q_1\hat{\mathbf{1}} + q_2\hat{\mathbf{2}}$.

```
51  PKMATHREAL exponential( const PKMATHREAL x,
52                          const PKMATHREAL p1,
53                          const PKMATHREAL p2,
54                          const PKMATHREAL q1,
55                          const PKMATHREAL q2,
56                          const PKMATHREAL r )
57  {
58     return( ( exp(-r*x)  - exp(-r*p1) )
59           / ( exp(-r*q1) - exp(-r*p1) )
60           * ( q2 - p2 ) + p2 );
61  }
```

The `upFunc()` function simply returns the `PKMATHREAL` value

$$\mathrm{up}(x; b) = \frac{1}{1 + b^{-x}}$$

where $b =$ `base`. It is a rising function in that, for $b > 1$, $\mathrm{up}(x; b) \to 0$ as $x \to -\infty$, and $\mathrm{up}(x; b) \to 1$ as $x \to \infty$.

```
62  PKMATHREAL upFunc( const PKMATHREAL x, const PKMATHREAL base )
63  {
64     if ( base <= 0.0 )
65        return(0.0);
66
67     return( 1.0 / ( 1.0 + pow(base,-x) ) );
68  }
```

The `doFunc()` function simply returns the `PKMATHREAL` value $\mathrm{do}(x;b) = \mathrm{up}(-x;b)$, where $b =$ `base`. It is a falling function in that, for $b > 1$, $\mathrm{do}(x;b) \to 1$ as $x \to -\infty$, and $\mathrm{do}(x;b) \to 0$ as $x \to \infty$.

```
69  PKMATHREAL doFunc( const PKMATHREAL x, const PKMATHREAL base )
70  {
71     return( upFunc(-x,base) );
72  }
```

The `udFunc()` function simply returns the `PKMATHREAL` value

$$\mathrm{ud}(x;x_1,x_2,b) = \mathrm{up}(x-x_1;b)\mathrm{do}(x-x_2;b)$$

where $b =$ `base`. It is a rising then falling function in that, for $b > 1$, the $\mathrm{up}(x-x_1;b)$ factor works to raise $\mathrm{ud}(x)$ for $x > x_1$, and the $\mathrm{do}(x-x_2;b)$ factor works to lower $\mathrm{ud}(x)$ for $x > x_2$.

```
73  PKMATHREAL udFunc( const PKMATHREAL x,
74                     const PKMATHREAL x1,
75                     const PKMATHREAL x2,
76                     const PKMATHREAL base )
77  {
78     return( upFunc(x-x1,base) * doFunc(x-x2,base) );
79  }
```

The `drawBasisVectors` function simply `printf()`s to standard output Ti*k*Z commands to typeset the $\hat{\mathbf{1}}$ and $\hat{\mathbf{2}}$ basis vectors.

```
80  void drawBasisVectors( const PKREALVECTOR *e1,
81                         const PKREALVECTOR *e2 )
82  {
83     if ( !e1  ||  !e2  )
84        return;
85
86     puts(   "   %");
87     puts(   "   % Basisvectors.");
88     puts(   "   %");
89     printf( "   \\draw[pktikzbasisvector,<->]\n"
90             "      (" FLTFMT "," FLTFMT ") node[right]{$%s$}"
91              " -- (origin) -- (" FLTFMT "," FLTFMT ") node[above]{$%s$};\n",
92             pkRealVectorGetComponent(e1)[0],
93             pkRealVectorGetComponent(e1)[1],
94             pkRealVectorGetName(e1),
95             pkRealVectorGetComponent(e2)[0],
96             pkRealVectorGetComponent(e2)[1],
97             pkRealVectorGetName(e2) );
98
99     return;
100 }
```

`drawInterestFunction()` `printf()`s Ti*k*Z commands for typesetting our function of interest, $f(x)$.

```
101  void drawInterestFunction(void)
102  {
103     puts( "   %");
104     puts( "   % The interest function 'f(x)'.");
105     puts( "   %");
106     puts( "   \\pktikzSetUncircledPoint{(0.4,4)}{a};" );
107     puts( "   \\pktikzSetUncircledPoint{(1,4.7)}{b};" );
108     puts( "   \\pktikzSetUncircledPoint{(2.5,2.5)}{c};" );
109     puts( "   \\pktikzSetUncircledPoint{(4,1.7)}{d};" );
110     puts( "   \\pktikzSetUncircledPoint{(6,2.5)}{e};" );
111     puts( "   \\pktikzSetUncircledPoint{(8,6)}{f};" );
112     puts( "   \\pktikzSetUncircledPoint{(10,5)}{g};" );
113     puts( "   \\pktikzSetUncircledPoint{(11.5,1)}{h};" );
114     puts( "   \\pktikzSetUncircledPoint{(13,1)}{i};" );
115     puts( "   \\pktikzSetUncircledPoint{(14,5)}{j};" );
116     puts( "   \\pktikzSetUncircledPoint{(14.7,5.5)}{k};" );
117     puts( "   \\draw[function] plot[smooth] coordinates {");
118     puts( "      (a) (b) (c) (d) (e) (f) (g) (h) (i) (j) (k) };");
119     puts( "   \\node[below] at (a) {$f(x)$};");
120
121     return;
122  }
```

4.3 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.

```
1   all: dimensionality-of-reality.pdf
2
3   clobber: latexclobber
4           @rm -f *.o
5           @rm -f *.run spherefigure.tex
6           @rm -f *.core
7
8   backup: clobber
9           @PACKDIR=‘basename \‘pwd\‘‘  &&  cd ..  &&  tar -czvf ${TARPATH} $${PACKDIR}
10
```

File based Make targets.

```
11
12  dimensionality-of-reality.pdf: spherefigure.tex \
13                                 Makefile.demo \
14                                 dimensionality-of-reality.bib
15
```

Implicit rule targets.

```
16
17  .SUFFIXES: .c .o .run .tex
18  .c.o:
19          clang -c -DDEBUG=2 -I/usr/local/pklib/include -DFreeBSD -o ${@} ${<}
20  .o.run:
21          clang -DDEBUG=2 -I/usr/local/pklib/include -DFreeBSD -o ${@} ${<} \
22                /usr/local/pklib/lib/libpk.a \
23                /usr/local/pklib/lib/libpkmath.a \
24                -lm
25  .run.tex:
26          ./${<} > ${@}
27
```

Incorporate PKLATEXMAKE[9].

```
28
29  # Added by 'pklatexmake.mk'.  Do not delete. 26Jul16
30  .include "/usr/local/pklatexmake/lib/pklatexmake.mk"
```

5 Acknowledgments

As always Mels, thanks for being such a close friend and supportive partner. You cannot be interpolated.

References

[1] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C*. Number 0-521-43108-5. Cambridge University Press, 1992.

[2] Brice Carnahan, H.A. Luther, and James Wilkes. *Applied Numerical Methods*. Number 0-471-13507-0. John Wiley & Sons, 1969.

[3] Foley. *Computer Graphics—Principles and Practice*. Number 0-201-84840-6. Addison-Wesley, 2nd edition.

[4] Till Tantau. PGF and Ti*k*Z—Graphic systems for TEX. `https://sourceforge.net/projects/pgf/`.

[5] TEX User Group. TEX Live. `http://tug.org/texlive/`.

[6] Comprehensive TeX Archive Network. `http://www.ctan.org/`.

[7] CTAN: TEX Live—A comprehensive distribution of TeX and friends. `http://www.ctan.org/pkg/texlive`.

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

[9] Paul Kotschy. The pkLATEXmake package. paul.kotschy@gmail.com.