Manual of VIKAASA
An Application Capable of Computing and Graphing Viability Kernels for Simple Viability Problems

JACEK B. KRAWCZYK, ALASTAIR PHARO

Abstract. This manual introduces and provides usage details for an application we have developed called VIKAASA, as well as the library of functions underlying it. VIKAASA runs in GNU Octave or MATLAB®, using the numerical computing and graphing capabilities of those packages to approximate, visualise and test viability kernels for viability problems involving a differential inclusion of two or more dynamic variables, a rectangular constraint set and a single scalar control.

Contents

VIKAASA stands for: **Vi**ability **K**ernel **A**pproximation, **A**nalysis and **S**imulation **A**pplication. The Sanskrit word *vikaasa* (विकास) means "progress" or "development" – we believe that our application represents विकास in the process of understanding and application building in viability theory.

1. A brief introduction to viability theory

This document details the usage of VIKAASA, an application for undertaking analysis of viability kernels.[1] Since viability theory is still new, this document first provides a brief introduction to viability theory and the specific subset of viability problems that can be analysed using VIKAASA.

Viability problems, as defined by [1], involve systems that evolve dynamically over time, where the concern is to identify *viable evolutions* – trajectories whereby the system in question does not violate some set of viability constraints over a given (possibly infinite) time-frame. A *viability kernel*, which is the set of all possible initial states from which viable trajectories exist, hence becomes a useful tool for analysing such problems. VIKAASA is a tool which can be used to create approximate viability kernels for a certain class of viability problems.

Example Box A. Introducing the fisheries model

In these boxes, which appear throughout the manual, we will demonstrate VIKAASA by using it to analyse a viability problem similar to the one presented in [2]. This problem concerns a marine resource which is being exploited by a fishing fleet. Viability here means both the sustainability of the resource (i.e., avoiding extinction), and the sustainability of fishing (i.e., the fleet remaining profitable). The evolution of the resource biomass is modelled over time, subject to the "harvesting flow," or rate at which fish are caught.

The question is then to discover what initial conditions, and what harvesting policies are sustainable (or viable), in the sense given above. Alternatively, the question can be formulated as: "are the system's dynamics compatible with the geometry of the constraint set?" It is supposed that regulatory instruments are available which can affect how many fish are caught by altering the level of "effort" that the fleet expends on fishing, either increasing or decreasing it by some amount. This "effort" has various interpretations. Here we follow [3] in supposing that it represents the proportion of some period of time (e.g., years, months, weeks, etc.) that the fishing fleet is allowed to engage in fishing. This level can be driven upwards or downwards, but only at a limited rate, perhaps due to political pressures.

Thus the problem can be framed as a type of *control problem* – the viability of any given state-space point depends on whether sufficient control over effort levels exists to prevent violation of the sustainability constraints as the system evolves from that state. This problem can then be analysed using viability theory, and VIKAASA can be used as a tool in that analysis. The problem's viability kernel would provide us with those initial conditions from which it is possible to sustain the system, possibly by enacting some control.

[1] As mentioned in the abstract, VIKAASA handles viability problems involving a differential inclusion of two or more dynamic variables, a rectangular constraint set and a single scalar control. In Appendix A we briefly outline how VIKAASA could be extended to deal with more general viability problems.

The basic notion of the viability kernel is that it provides us with the information necessary to determine whether or not a given state-space position has a viable trajectory proceeding from it, i.e., whether starting at that position, the system can be maintained within its constraints, or not. In what follows, we give a more technical explanation of viability theory, including a formal definition of the *viability kernel*. For those who are new to viability theory however, it may be beneficial to first read [4], which gives some more examples.

1.1. **General formulation of a viability problem.** The core ingredients of a viability problem[2] are:

(1) A continuum of time values, $\Theta \equiv [0, T] \subseteq \mathbb{R}_+$, where T can be finite or infinite.
(2) A vector of n real-valued state variables, $x(t) \equiv [x_1(t), x_2(t), \ldots, x_n(t)]' \in \mathbb{R}^n$, $t \in \Theta$ that together represent the dynamic system in which we are interested. The vector $x(t)$ is called the *state-space representation* of these variables.[3]
(3) A *constraint set*, $K \subset \mathbb{R}^n$, which is a *closed set* representing some normative constraints to be imposed on these state variables. Violation of these constraints means that the system has become non-viable. Thus in seeking viable trajectories, we want to ensure that $\forall t(t \in \Theta \rightarrow x(t) \in K)$.
(4) A vector of real-valued controls, $u(t) \equiv [u_1(t), u_2(t), \ldots, u_m(t)]' \in \mathbb{R}^m$, $t \in \Theta$. We call $u(t)$ the *control vector*.
(5) Some normative constraints on the controls, so that
$$\forall\ t \forall x\ (t \in \Theta \wedge x \in K \rightarrow u(t) \in U(x(t)) \subset \mathbb{R}^m).$$
So, $U : \mathbb{R}^n \rightsquigarrow \mathbb{R}^m$ is a set-valued function, which gives the set of control vectors available in each state. Thus the control vector at time t is constrained according to the state, $x(t)$ of the system.
(6) A set of real-valued first-order differential inclusions,

$$\dot{x}(t) = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \vdots \\ \dot{x}_n(t) \end{bmatrix} \in \left\{ f(x,u) = \begin{bmatrix} f_1(x,u) \\ f_2(x,u) \\ \vdots \\ f_n(x,u) \end{bmatrix} \right\}_{u \in U} \tag{1}$$

Each function $f_i : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}, i = 1, 2 \ldots n$ specifies the velocity of the corresponding variable x_i, for any pair (x, u), where $x \in \mathbb{R}^n$ is a position in the state space, and $u \in \mathbb{R}^m$ is a control choice.

Note that we have formulated viability problems above in terms of *differential inclusions*, as in [1], whereby the evolution of some or all of the system's variables is *set-valued*. That is, for a given $x(t)$ we have an array of possible controls to choose from in $U(x(t))$ and hence have a *set* of points in the state space which can be reached in time $t + \gamma$ (where $\gamma > 0$ is small). We are then concerned with finding those members of the set U for which the trajectories are viable.

[2]Here, we formulate a viability problem in continuous time. A similar formulation could be made for a viability problem in discrete time.

[3]Throughout this text we distinguish between "dynamic" variables, which together determine the state-space position of the system, and so-called "additional" variables, which may represent other matters of interest, but which are completely determined by the dynamic variables, and so are not necessary considerations for the construction of the viability kernel.

Given such a problem, we can attempt to find one or more *viability domains*, $D \subseteq K$, where each viability domain is a set of initial conditions $x(0)$, for which there exist viable trajectories. That is, for every element $x(0) \in D$, there exists a function (or feed-back rule) $g : \mathbb{R}^n \mapsto \mathbb{R}^m$, such that for each element, k, of constraint set $K \subset \mathbb{R}^n$, $g(k) \in U(x)$ and $\forall t \in T, x(t) \in K$ where $x(t)$ is a solution to Equation 1 with $u(t) = g(x(t))$. In other words, for every initial state in D, there must exist sufficient control from U to prevent violation of the viability set, K, over $t \in \Theta$. The problem's *viability kernel*, $V \subseteq K$ is then the *largest* possible viability domain (or the union of all viability domains), giving all initial conditions in K, for which a set of controls in U exists to prevent the system from exiting K over $t \in \Theta$.

Example Box B. Specifying the fisheries viability problem

Following the numbering used above, we will now specify the fisheries problem of [2] in the same terms:

(1) The model is concerned with an infinite time horizon, so $\Theta = [0, \infty]$.

(2) The system is described by of two dynamic variables: fish biomass, $b(t)$ and effort, $e(t)$. Effort is exerted by the fishing fleet to extract the resource (i.e., fish). This is a fixed fleet-size model, so there is no variation in capital to consider.

A "catchability coefficient" q is defined to determine the quantity of biomass that each unit of effort extracts, relative to the total size of the biomass at the time. Thus, for some biomass level $b(t)$, an effort level of $e(t)$ yields $qe(t)b(t)$ in resource biomass.

(3) Three constraints are given. The first constraint concerns the ecological sustainability of the resource. To this end, a "safe minimum biomass level," $b_{min} > 0$ is specified, below which it is believed that the resource will become extinct. Thus, sustainability requires that $\forall t(t \in \Theta \rightarrow b(t) \geqslant b_{min})$.

The second constraint concerns the economic sustainability of the fleet, and requires that fishing remain profitable. Profits are given by

$$R(b(t), e(t)) = pqe(t)b(t) - ce(t) - C, \tag{2}$$

where p is the price of a unit of biomass (fixed in this model), and as explained above, $qe(t)b(t)$ is the catch size, making $pqe(t)b(t)$ the revenue gained, whilst C is some fixed cost, and c is a variable cost for each unit of effort. The profitability requirement is then that $R(b, e) > 0$, meaning that revenue must be at least as big as the combination of fixed and variable costs.

The third constraint is not normative, but rather concerns the physical capabilities of the fishing fleet. That is, it is supposed that $\forall t \in \Theta \cdot e(t) \in [0, e_{max}]$, where e_{max} is the maximum possible effort exertable by the fleet. Given the fixed size of the fleet, this is the only input into fleet behaviour.

Thus the constraint set is:

$$K = \{(b, e) : b \geqslant b_{min} \wedge pqeb - ce - C \geqslant 0 \wedge e \in [0, e_{max}]\}. \tag{3}$$

It should be noted that this constraint set is *not* closed, as there is no upper limit on b. However, there is an implicit upper limit imposed by the differential inclusion for $b(t)$ (see item 6 below).

(4) As per Example box A, it is supposed that regulatory instruments can be used to increase or decrease the level of effort exerted by the fleet. Thus, the system can be modelled as having a single scalar control, $u(t) \in \mathbb{R}$, which determines effort variation; $u(t) = \dot{e}(t)$. Given that viability problems concern the existence (or not) of sufficient control, we are

here engaged with finding state-space points where effort is at a sustainable level and no variation is required, or where it can be changed (given the available control-set) to become sustainable without any violation of constraints.

(5) Effort variation is bounded by $U = [u^-, u^+]$, where $u^- < 0$ and $u^+ > 0$. Thus, where $e(t)$ is too high (entailing imminent extinction), or too low (meaning that fishing is not profitable), it may not be possible to increase or decrease $e(t)$ fast enough (depending on the sizes of u^- and u^+, which determine to speed of changes of $e(t)$) to maintain the viability of the system.

(6) The differential inclusion for $b(t)$ is:

$$\dot{b}(t) \in \left\{ rb(t)\left(1 - \frac{b(t)}{l}\right) - qe(t)b(t) \right\}_{u(t)\in U}. \tag{4}$$

As this inclusion is not dependent on $u(t)$, it simplifies to the following *differential equation*:

$$\dot{b}(t) = rb(t)\left(1 - \frac{b(t)}{l}\right) - qe(t)b(t). \tag{5}$$

This equation is based on [5], and is common in models of population growth. The resource grows at a rate proportional to r, up to the "limit carrying capacity," l of the resource's ecosystem, less the size of the catch, $qe(t)b(t)$.

As mentioned in item 4, the differential inclusion for e is:

$$\dot{e}(t) \in U = [u^-, u^+]. \tag{6}$$

1.2. **Viability theory, bounded rationality and sustainability.** Here, we highlight some links between viability theory, bounded rationality and sustainability. Briefly, the existence and importance of these links explicates an economic interest in viability theory.[4]

Herbert A. Simon, 1978 Economics Nobel Prize laureate, argued that there are *bounds* on economic agents' "rationality" and that economists really need "satisficing," (his neologism, see [17]) rather than optimising solutions. We share Simon's view in that we believe that some economic agents may not seek unique optimal solutions. Take for example the central bank governor's task in a country where the allowable inflation band has been legislated; or a national park director who is responsible for biodiversity of the fauna; or an international body seeking multi-country adherence to some standards. Each of these agents will strive to satisfy several goals, many of them consisting of ensuring that the key outcomes (e.g., inflation, or the number of bears, or the amount of some noxious substance) remain within some normative bounds. The bounds might have been derived from some felicity function optimisation, but the governor (or park director, or the international body) will perceive them as exogenously specified. We think that an economic theory that follows the Simon prescription may bring modelling closer to how these people actually behave; and we contend that viability theory provides a mathematical platform to deal with Simon's concerns analytically.

[4]Viability theory has been successfully applied to environmental economics problems see [6], [7] and [8]; for applications to financial analysis see [9] and the references provided there. Along with [4], [10], [11], [12], [13], [14] and [15] deal with viable solutions to macroeconomic problems; see [16] for a microeconomic problem solution.

In the case of the fisheries model of [2], viability theory moreover provides us with a means of considering the *sustainability* of the system as a whole, without our needing to assign relative weights to our two concerns, as would be necessary if we were to attempt to find an optimal state-space position for our problem.

1.3. **Computing viability kernels.** For our example model, a general analytic solution for the viability kernel has been obtained in [2]. Generally speaking however, computation of a viability kernel can be a very complex task, and the level of complexity increases with the dimensionality of the problem. Various approaches have been used to cope with this complexity. For instance, using algorithms (e.g., [18], [19]) or heuristics, the papers above cited in footnote 4 provide examples of viability kernels in two- and three-dimensional state spaces. Another example is given by [20], where a method based on some results in [21] is utilised for a four-dimensional problem, resulting in a viability kernel approximated by those state-space locations for which the value function realisations of an *auxiliary* cost-minimising optimal control problem are small.

It is because of this complexity that we have developed VIKAASA. VIKAASA makes use of simple numerical methods (namely Euler's method) for solving differential equations, making it possible to approximate the true system's dynamics without explicitly solving the equations. These methods are then used by VIKAASA to infer the viability kernel by identifying a discrete sample of initial conditions from which there exists a progression to a near-steady state in finite time, without the system leaving the constraint set in doing so.

This set of points, which strictly speaking is a viability domain, can then be interpolated to provide an approximation of a problem's "true" kernel, as would be obtained by solving a problem analytically. Thus, for those viability problems to which it is suited, VIKAASA can be used to construct an approximate viability kernel, without getting involved in solving the equations themselves.[5]

2. Introducing VIKAASA

As mentioned, VIKAASA can be used to generate approximate viability kernels using numerical methods without the need to solve problems analytically. VIKAASA works with viability problems of the type alluded to above in Section 1.1, albeit with some important limitations. In the following subsections, the capabilities and limitations of VIKAASA are outlined.

2.1. **Formulation of a VIKAASA viability problem.** In what follows, a general description of a viability problem that is compatible with VIKAASA is given. The list below is enumerated so that each item matches an item from Section 1.1. Thus, the two lists can be compared to see how viability problems that are compatible with VIKAASA are limited with respect to the general specification.

For a viability problem to be analysed with VIKAASA, the following requirements must be met:

[5]These methods will not be suitable for all classes of viability problems. We do not attempt any formal proofs in this manual concerning what sorts of problems this method is suited to, so we ask that analysts using VIKAASA exercise caution.

(1) The continuum of time values, Θ must be infinite. That is, $\Theta = [0, \infty)$.[6] This implies that the differential inclusions for the system must be *autonomous*.
(2) There are no technical limits to how many dynamic variables can be specified.[7] However, in order for visualisation of the resulting viability kernel to be possible, there must be at least two.
(3) By default the constraint set, K needs to be a "rectangular" set involving all dynamic variables. That is, $K \equiv [\underline{x}_1, \overline{x}_1] \times [\underline{x}_2, \overline{x}_2] \times \cdots \times [\underline{x}_n, \overline{x}_n]$, where $\underline{x}_i$ is the lower bound of the i^{th} variable, and $\overline{x}_i$ is the upper bound. This limitation can be alleviated somewhat however through the specification of a *custom constraint set function* which filters the points in the rectangular constraint set by testing them against some additional criterion, such as an inequality. See Section 2.2.
(4) $m = 1$. That is, the control, $u \in \mathbb{R}$ is a single *scalar* variable.
(5) The control set, $U(x)$ must be the same for all values of x, and must be symmetrical about zero. That is, given that u is a scalar, $\forall x(U = [-c, c])$, where $c \in \mathbb{R}_+$. We therefore write U instead of $U(x)$.
(6) VIKAASA can only work with *deterministic* autonomous system's dynamics. For any given point in the state space, and any given control choice, there can only be one possible trajectory. That is, VIKAASA cannot model stochastic processes. Additionally, it should be noted that the kernel approximation algorithm may not perform well with highly non-linear differential inclusions/equations, due to the simple numerical methods employed to solve them.

Because VIKAASA uses numerical methods, all parameters except for the state-space vector $x(t)$, and the control $u(t)$ must be specified as specific numbers. That includes $\overline{x}_i|_{i=1}^n$, $\underline{x}_i|_{i=1}^n$ and c.

2.2. **Custom constraint set functions (CCSFs).** As mentioned above, it is possible to specify a *custom constraint set function* (CCSF) for a viability problem in order to get around the problem of needing to specify a rectangular constraint set for viability problems in VIKAASA. The CCSF is a Boolean-valued function, $CCSF : \mathbb{R}^n \to \mathbb{B}$, which given any state-space vector, $x(t)$ in the rectangular constraint set, $K \equiv [\underline{x}_1, \overline{x}_1] \times [\underline{x}_2, \overline{x}_2] \times \cdots \times [\underline{x}_n, \overline{x}_n]$, returns "true" if and only if that state-space point does not violate the viability constraints. Note that the CCSF is an additional constraint on top of the rectangular ones imposed by K. Hence, $CCSF(\cdot)$ need only operate over values within K.

[6] Thus, we exclude "viability-with-target" problems. VIKAASA is not able to ascertain whether a particular target was reached within a set time-frame. However, one should also note that when determining the viability of a point, VIKAASA will do so by searching for a steady state over some finite number of time intervals $[0, \frac{1}{h}, \frac{2}{h}, \ldots, \frac{N}{h}]$, where h is the step-size (see below), and by default $N = 46,000$. Thus, in systems where there is only one steady state position, the viability algorithm employed by VIKAASA does indeed resemble a "viability-with-target" approach.

[7] The scheme we propose will however suffer from *the curse of dimensionality*: the amount of time required to compute a kernel approximation increases exponentially with the number of dimensions. For instance, with a discretisation (see Section 2.3) of 10 in every dimension, a two-dimensional problem considers 100 points, a three-dimensional problem considers 1,000 points, and a four-dimensional problem considers 10,000 points. It therefore quickly becomes desirable to take advantage of the parallel computing facilities offered in MATLAB® and Octave (see Section 3.2).

This function is used in Algorithm 1 (see below) to filter elements from consideration. For simplicity, where a CCSF is not supplied by the user, we implicitly define a default CCSF (identified by a superscript "d"), $CCSF(\cdot) = CCSF^d(\cdot) \equiv \text{true}$.

Example Box C. Specifying the fisheries problem for VIKAASA

We now modify the specification given in Example box B to make it work with VIKAASA. item 1, item 2, item 4 and item 6 of Section 2.1 do not cause any difficulties. However, the profitability constraint, $R(b(t), e(t)) = pqe(t)b(t) - ce(t) - C \geqslant 0$, cannot be represented by a rectangular constraint set, so we will need to use a CCSF (as described in Section 2.2). Also, item 5 means that $u^- = -u^+$. That is, the maximum increase in effort must be the same as the absolute maximum decrease.

Lastly, until now we have only specified the problem in general terms. In order to use VIKAASA we will need to give specific numbers to all the parameters. The values we will work with are:

- Catchability coefficient, $q = \frac{1}{2}$.
- Cost per unit of effort, $c = 10$.
- Fixed cost, $C = 100$.
- Growth rate, $r = \frac{2}{5}$.
- Limit carrying capacity, $l = 500$.
- Maximum effort, $e_{max} = 1$.
- Maximum effort variation, $c = u^+ = -u^- = \frac{1}{100}$.
- Safe minimum biomass level, $b_{min} = 5$.
- Unit fish price, $p = 8$.

Thus, the differential inclusions[8] become:

$$\dot{b}(t) = \frac{2}{5}b(t)\left(1 - \frac{b(t)}{500}\right) - \frac{1}{2}e(t)b(t) \tag{7}$$

and:

$$\dot{e}(t) \in U = \left[-\frac{1}{100}, \frac{1}{100}\right] \tag{8}$$

Our rectangular constraint set is:

$$K = [5, 500] \times [0, 1] \tag{9}$$

We supplement this with a custom constraint set function:

$$CCSF(b, e) = (4eb \geqslant 10e + 100) \tag{10}$$

Thus, the *effective* constraint set consists of all the elements of K for which $CCSF(\cdot, \cdot)$ returns true:

$$K_{effective} = K \cap \{(b, e) : CCSF(b, e) = \text{true}\} \tag{11}$$

We will see in the next example box how to enter this viability problem into VIKAASA.

2.3. **Solving viability problems with VIKAASA.** VIKAASA takes problems of the kind specified in Section 2.1 and approximates viability kernels for them using numerical methods. In this section we explain the general algorithm employed by VIKAASA, the

additional parameters required by the algorithm, and how these affect the goodness of viability kernel approximations produced.

In addition to the core ingredients as specified in Section 1.1 and Section 2.1, VIKAASA requires the following additional parameters in order to compute viability kernel approximations:

(1) A *discretisation*, $\delta = [\delta_1, \delta_2, \ldots, \delta_n]' \in \mathbb{Z}^n$, which determines the finite subset of K to be examined by the algorithm.

VIKAASA takes a vector of δ_i evenly spaced values from each dimension i of K, starting at $\underline{x}_i$ and finishing at $\overline{x}_i$.[9] These vectors are then combined using a Cartesian product to make a discretised version of the constraint set, $K_\delta \subset K$, containing a total of $\prod_{i=1}^{n} \delta_i$ points. VIKAASA will then consider each of these points individually to see whether it is viable or not.

(2) A *stopping tolerance*, $\epsilon \in \mathbb{R}_+$ is used as the criterion for "near-steadiness" of the system.

The velocity of the system in state x, subjected to some control u is calculated using the Euclidean norm of the system velocities at that point, $|f(x,u)| = \sqrt{\dot{x}_1(x,u)^2 + \dot{x}_2(x,u)^2 + \cdots + \dot{x}_n(x,u)^2}$. Near-steadiness is then said to obtain when this norm is less than what the system would have if its velocity were ϵ in every direction. That is, when $|f(x(t),u(t))| < \sqrt{n \cdot \epsilon^2}$.

(3) A *step size*, $h \in \mathbb{R}_+$ is needed by the approximation algorithm in order to compute the system trajectories using the Euler method. That is, given some state $x(t)$, and a control choice $u(t)$, the "next" value will be $x(t+h) = x(t) + h \cdot f(x(t),u(t))$.[10]

(4) A *control algorithm*, $u^* : \mathbb{R}^n \to U$, which is a time-independent feed-back rule, responsible for slowing the system velocity to below the stopping tolerance, $\sqrt{n \cdot \epsilon^2} = \sqrt{n} \cdot \epsilon$. VIKAASA takes this control rule, and uses it to choose $u(t) = u^*(x(t))$ at each time realisation. By default a one-step forward-looking numerical norm-minimisation algorithm is used, so that $u^*(x) = \arg\min_u^G \{|f(x + h \cdot f(x,0),u)|\}$.[11] This algorithm is only suited to cost functions where any local minimum is guaranteed to be a global minimum. Where this is not the case, it may be necessary to consider other algorithms. The choice of a good control algorithm for kernel determination is very important, and is described in more detail in Section 3.8.

2.4. **The viability kernel approximation algorithm.** The above ingredients are used by VIKAASA to approximate viability kernels. For each point in K_δ, VIKAASA calls a numerical simulation routine, using the above parameters to see if a "near-steady" state can be found. Algorithm 1 below provides a simplified version of how this determination is made.

[9] Each vector is constructed using the MATLAB® function, linspace(xmin(i), xmax(i), d(i)), where xmin(i) is the i^{th} lower bound, xmax(i) is the i^{th} upper bound, and d(i) is the discretisation.

[10] This means that derivatives are replaced by differences. Other methods could also be used e.g., Runge-Kutta.

[11] $\min^G$ refers to the numerical method of function minimisation employed. VIKAASA uses the function u = fminbnd(fn, −c, c), present in both MATLAB®and Octave, to choose the norm-minimising control, where fn is a handle to a function which gives the norm of the system velocity at the present position for a given u, and c is the absolute maximum size of the control. fminbnd makes use of a golden ratio search algorithm, and stops searching when the distance between realisations of fn are less than some *control tolerance*, ξ.

Algorithm 1 Determination of the viability of a point, x in the state-space

if $CCSF(x)$ returns "false" **then**
  return not viable
end if
$t \leftarrow 0$
repeat
  $x \leftarrow x + h \cdot f(x, u^*(x))$
  $t \leftarrow t + 1$
until $|f(x, u^*(x))| \leqslant \sqrt{n} \cdot \epsilon$ **or** $x \notin K$ **or** $CCSF(x)$ returns "false" **or** $t = 46,000$
if $x \in K$ **and** $t < 46,000$ **and** $CCSF(x)$ returns "true" **then**
  return viable
else
  return not viable
end if

To determine the approximate viability kernel then, it remains only to run this algorithm on each point in K_δ, and identify the set of points S, ("S" for "steady") for which "viable" is returned. This is described in Algorithm 2:

Algorithm 2 Construction of approximate viability kernel, S from a constraint set K

$K_\delta \leftarrow discretise(K)$
$S \leftarrow \varnothing$
for all $x \in K_\delta$ **do**
  if running Algorithm 1 on x returns "viable" **then**
    $S \leftarrow S \cup \{x\}$
  end if
end for
return S

It should be clear from this that there are some important shortcomings in this algorithm. In particular, no attempt is made by VIKAASA to verify whether the results obtained have the characteristics of a viability domain or not.[12] For this reason, some remarks on how to test the "goodness" of an approximation, and how one's choices of the parameters: δ, ϵ, h and u^*, can affect this are given in the following sections.

2.5. **Determining the goodness of an approximation.** In order for some S to be a good approximate viability kernel, it should be the case that S closely resembles a discretised version of the "true" kernel, V. That is, $S \approx V \cap K_\delta$. Of course it is impossible

[12]For instance, no check is performed to see whether the paths from each initial condition in S remain "inside" any viability domain interpolated from S. Similarly, no effort is made to check whether points that are just outside of S have potential paths pointing into S, thus making it possible to expand S to include those points. A sketch of what such a check might consist of is provided in what follows: for points that are deemed non-viable for the selected parameters, but which are "close" (perhaps adjacent in K_δ) to the viable points, a check could be performed to see whether a velocity emanating from this point would form an obtuse angle with the normal of viability domain's frontier. For each point where this was the case, the viability domain could then be expanded. This idea is explored in [22].

to test for this without already knowing V, which is why VIKAASA cannot automatically check for the goodness of its approximations. Nevertheless, some general observations can be made about how the parameters affect the kernel approximation process for better or for worse. These are explored in the following subsections.

2.5.1. *Discretisation, δ.* VIKAASA approximates viability kernels by considering a discrete subset of the points in K. The number of points considered is determined by the user-defined *discretisation* vector, δ. As a heuristic, we have found that a discretisation of around 10 in each dimension makes for a good initial indication of kernel size and shape, without taking too long to compute.

A higher discretisation means that there will be more points to consider, and thus more time will be required to compute the kernel approximation. Higher discretisation is desirable however because it should provide a more accurate interpolation of the "true" kernel. This can be seen by considering the "boundary" or "frontier" of a viability kernel, $fr(V)$, which is the set of points that are not completely surrounded by other points in the true kernel (i.e., the points that are closest to the non-viable points). Because the true kernel is not known, the exact whereabouts of the frontier is not known either. However, given some kernel approximation S, then for any two points, $A \in S$ and $B \in K_\delta \backslash S$, where A and B are adjacent in K_δ, provided that Algorithm 1 has correctly determined the viability of A and the non-viability of B, it must be the case that $fr(V)$ passes between A and B. Thus, the closer the points sampled from K are to one another (i.e., the larger is each δ_i), the less uncertainty there will regarding there whereabouts of the frontier. This means that approximations made with a higher discretisation should afford a more accurate visual representation of the viability kernel, and that policies which make use of the kernel for decision-making should be better informed.[13]

There is an important caveat to the above claim that a higher discretisation is desirable however, and that is that increasing the discretisation can in no way make up for problems introduced by one's choices of ϵ, h and u^*. Discretisation only affects Algorithm 2, and not Algorithm 1, so if Algorithm 1 does not correctly identify the viability/non-viability of a point, there is nothing that the discretisation can do to correct this.

2.5.2. *Stopping tolerance, ϵ.* The stopping tolerance is required because the finite and imprecise nature of numerical calculation, as well as the inability to take limits at $t = \infty$, make it impossible for the method employed in Algorithm 1 to achieve a system velocity of zero in most cases. For this reason, a velocity which is "close enough" to zero must be specified.

As mentioned, this "close enough" criterion is determined by comparing the Euclidean norm of the system velocity, $|f(x,u)| = \sqrt{\dot{x}_1(x,u)^2 + \dot{x}_2(x,u), \ldots, \dot{x}_n^2}$ to the Euclidean norm of a movement of ϵ in every direction, $\sqrt{n} \cdot \epsilon$. If the norm of the system velocity is less than this, then the system is considered to be in a near-steady state, and provided

[13] As this discussion suggests, it may make sense to think about the frontier of a viability kernel approximation, $fr(S)$ as being "thick," given there will be a range of points through which the "true" frontier might possibly pass. Increasing the discretisation of the kernel thus reduces the "thickness" of the frontier. Note though that the kernel visualisation tools provided by VIKAASA are not currently able to display a thick frontier.

that the state is also within K, the initial condition which gave rise to this state will be marked as stable.

Thus, the main consideration to take into account when choosing a value for ϵ is what is a sufficient condition for $\sqrt{n} \cdot \epsilon$ to be "close enough." A value of ϵ that is too large risks producing a viability kernel with "false positives" in it – that is, Algorithm 1 might identify a point as viable when there do not in fact exist controls that can prevent the system from violating the constraint set eventually. Thus, a value of ϵ that is as small as possible (without being zero) would seem to be desirable. There are (again) important caveats to this however. Firstly, in the case where a point has a viable trajectory, reducing ϵ generally increases the time taken by Algorithm 1, because more iterations are required to slow the system velocity sufficiently. Secondly, the control algorithm $u^*(x)$ employed in Algorithm 1 may take longer to produce controls which slow the system sufficiently. Specifically, those control algorithms which make use of numerical cost-minimising methods may need to reduce their *control tolerance* ξ (see Section 3.8.2).

2.5.3. *Step size, h.* Reducing the step size, h, will make state-space values produced by the Euler method employed in Algorithm 1 closer to their "true" continuous-time values. However, a smaller step-size will most likely mean that it takes more iterations to determine whether a state-space point is viable or not, thereby increasing the amount of time required to compute the approximate viability kernel. Thus, a higher step-size may be desirable if it does not distort the system's dynamics too much. Also, a step-size of $h = 1$ can be used to allow VIKAASA to deal with discrete-time models based on differences instead of derivatives.

2.5.4. *Control algorithm, $u^*(x)$.* The control policy used must be able to bring (almost) all points in V to a steady state in finite time, without violating the constraints in doing so. If it cannot, then points that have viable trajectories will be excluded from S, resulting in $S \not\approx V \cap K_\delta$. Selection of a good control algorithm is thus highly important.

As mentioned in Section 2.3 VIKAASA uses a forward-looking one-step norm-minimising control algorithm by default, but a number of other possible algorithms are provided as well, including the possibility to write your own cost function for minimisation. Section 3.8 below gives a detailed account of what control algorithms come with VIKAASA, and what options exist for customising them. Section 3.8.6 outlines how you can write your own control algorithm.

3. Using VIKAASA to approximate and visualise viability kernels

VIKAASA can be used in two different ways: via a graphical user interface (GUI), or programmatically through a library of functions. The graphical interface is only available in MATLAB®, but the library interface works under either GNU Octave or MATLAB®.

3.1. **The VIKAASA graphical user interface (GUI).** The VIKAASA GUI is fairly powerful and is meant to make the task of approximating a viability kernel easier. Furthermore, little or no knowledge of MATLAB® is required in order to use the GUI. If however the GUI does not contain the features you require, or if you want to integrate the functionality of VIKAASA into another application, then you will need to make use

of the library. The library has been written with extensibility in mind, and so allows for the functionality of VIKAASA to be extended well beyond what is offered in the GUI. A screenshot of the GUI is given in Figure 1.

In the following sections, we explain how to use VIKAASA *interactively*, both through the GUI and from the Octave or MATLAB® command-line. As such, this section does not give exhaustive detail about the VIKAASA library, but it should provide sufficient information for most standard tasks. If you are interested in interfacing with the VIKAASA library *programmatically*, please see Appendix D for a more detailed description of the available functionality.

3.2. **Requirements for using VIKAASA.** The GUI has been tested on MATLAB® R2009b. It is not known how it will behave on earlier versions of MATLAB®, but it should work in more recent versions.

The library has been tested with version 3.4.1 of Octave.[14] as well as MATLAB® R2009b. It should behave roughly the same under both systems.[15]

Additionally, VIKAASA is able to make use of the parallel processing facilities available in both MATLAB® and Octave to speed up computation time on computers that have more than one processor, or multiple cores. In MATLAB®, VIKAASA needs the parfor operator, which is available in `http://www.mathworks.com/products/parallel-computing/`.[16] In Octave, VIKAASA needs the parcellfun function, which is available from `http://octave.sourceforge.net/general/index.html`.

3.3. **Starting VIKAASA.** To start the GUI in MATLAB®, invoke the vikaasa script. This can be done by right-clicking the `vikaasa.m` file, and selecting "Run," or by entering vikaasa into the MATLAB® command window.

Upon starting, you should see the main window, as shown below in Figure 1.

If you are using the VIKAASA via the Octave or MATLAB® command-line, you need to invoke the vikaasa_cli script in order to set-up paths, etc. On either platform, entering vikaasa_cli at the command-line should accomplish this. If you are using a *NIX platform such as Linux, you can also start Octave and VIKAASA together by running the `vikaasa` shell script. If things are working correctly, you should see a start-up message indicating that VIKAASA has been loaded.

3.4. **Working with projects.** Data in VIKAASA is organised around the concept of a *project*, which is a data structure containing information pertaining to a viability problem in a format that can be saved and loaded from either Octave or MATLAB®. Specifically, a project contains:

[14] See `http://www.gnu.org/software/octave/index.html` for more information. Some of the plotting features in VIKAASA do not work well without the FLTK graphics toolkit in Octave, which may not be available in some current releases. In this case, it may be necessary to use a current development snapshot instead.

[15] Octave appears to be able run the kernel approximation routines faster than MATLAB®, but does not contain MATLAB®'s extensive plotting functionality.

[16] VIKAASA only requires that the toolkit be installed. Starting and stopping of worker pools is handled automatically.

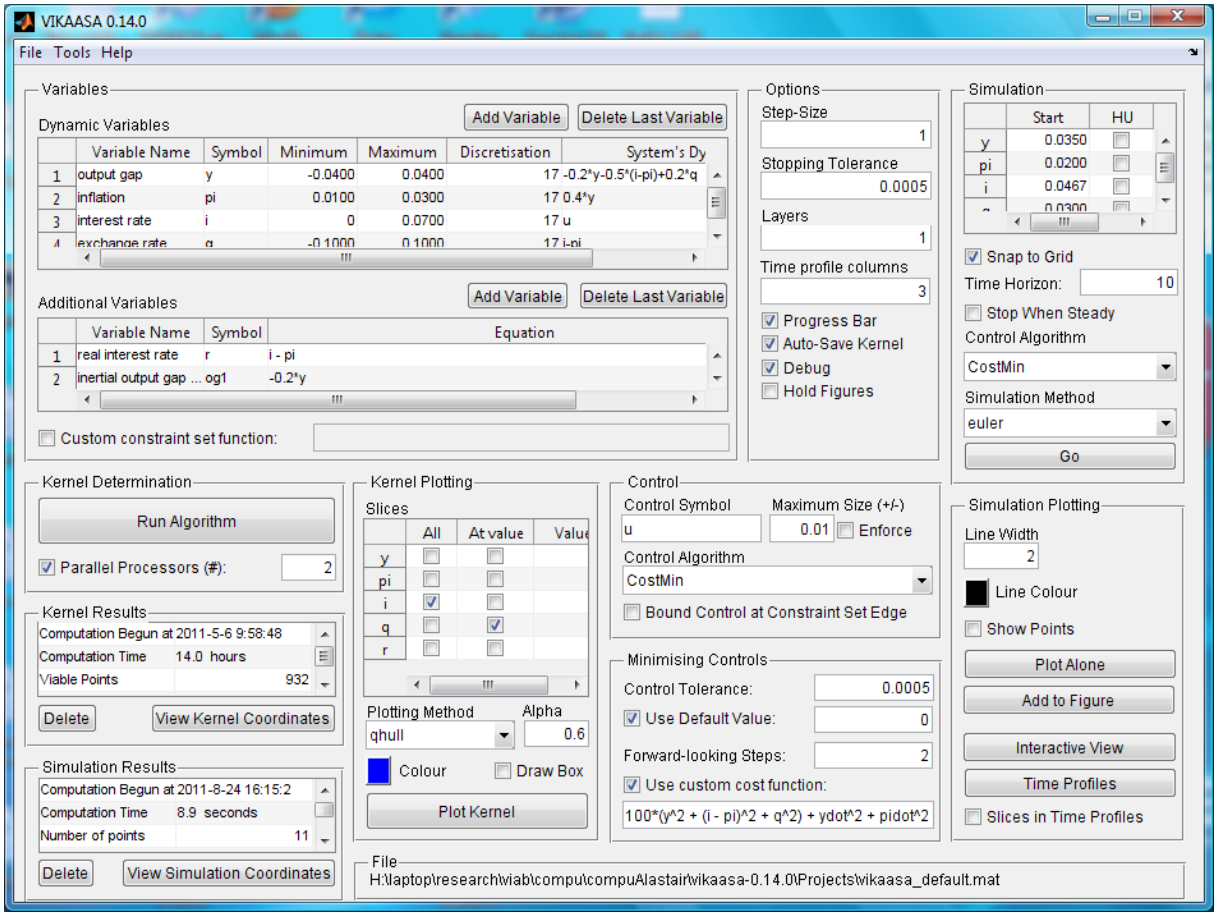


FIGURE 1. The Main Window

- all details of the variables for the viability problem, and associated parameters for solving the problem (i.e., the contents of the "Control," "Minimising Controls," "Options" and "Variables" panels in the GUI);
- the most recently calculated (or loaded) viability kernel, and related information (as shown in the "Kernel Results" panel);
- settings relating to the display of the viability kernel (as shown in the "Kernel Plotting" panel); and
- the most recent simulation that was made, along with simulation settings (as shown in the "Simulation" and "Simulation Plotting" panels).

Projects are stored in `.mat` files[17], enabling the files to be saved and loaded easily from either MATLAB® or Octave. The files are kept in the "Projects" folder by convention, although this is not a strict necessity.

If you are using the GUI, it is not usually necessary to understand how the project files are organised. If you are using the VIKAASA library interface however (from the command-line or otherwise), then all changes to projects need to be made manually. This is not difficult, but it requires that you be aware of the name and format of each project field that you need to change. Some information concerning this is given in the following subsections, and more detail is given in Appendix B.

[17]The version 7 format is used, as this format can be used from both MATLAB® and Octave.

When you start the VIKAASA GUI, `vikaasa_default.mat` in the "Projects" folder is automatically loaded. In a new installation of VIKAASA, this file will contain a copy of the fisheries model from [2], but you can replace this file with anything you like. You can tell what project is currently loaded by looking at the "File" panel in the bottom-right corner of the main window.

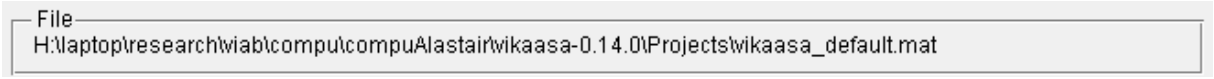


FIGURE 2. The "File" panel

The GUI is only able to work with one project at a time, so make sure to save your changes before loading a new project. If you need to copy settings from one project to another, then you need to use the command-line.

3.4.1. *Creating a new project.* To create a new project in the GUI, go to the "File" menu at the top of the main window, select "New Project", see Figure 3. A new project with some default values will then be loaded into the interface. You can then modify it as you please. We have already noted that the GUI can only work with one project at a time, so creating a new project will overwrite any existing project information currently loaded into the GUI.

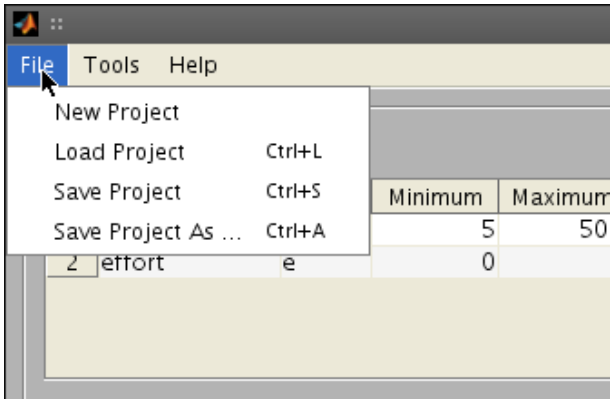


FIGURE 3. Creating a new project from the "File" menu

If you are using the command-line, you can create a new project with the vk_project_new function. This function returns a *struct* (or record) representing the newly created project. As with the GUI, projects created in this manner are pre-populated with default values (although some fields that the GUI pre-populates with "dummy" information are not pre-populated in the library). Below is a transcript of what running vk_project_new in Octave returns.[18]

[18]This example makes use of the fact that a command entered into either MATLAB® or Octave without a terminating semicolon will display its result on the command-line. Note however that MATLAB® and Octave differ in how they display these results. On both platforms the disp command can also be used to display results.

```
octave:1> vk_project_new
ans =

  scalar structure containing the fields:

    numvars = 2
    numaddnvars = 0
    alpha = 0.90000
    addnlabels = {}(0x1)
    addnsymbols = {}(0x1)
    addneqns = {}(0x1)
    addnignore = [](0x1)
    autosave = 0
    controlalg = ZeroControl
    controldefault = 0
    c = 0.0050000
    controlbounded = 0
    controlenforce = 0
    controlsymbol = u
    controltolerance = 0.0010000
    custom_cost_fn =
    custom_constraint_set_fn =
    debug = 0
    diff_eqns =
    {
      [1,1] = [](0x0)
      [2,1] = [](0x0)
    }
    discretisation =

       11
       11

    drawbox = 0
    h = 1
    holdfig = 0
    K =

       0 0 0 0

    labels =
    {
      [1,1] = [](0x0)
      [2,1] = [](0x0)
```

```
}
layers = 1
parallel_processors = 2
plotcolour =

   1 1 0

plottingmethod = qhull
progressbar = 1
sim_controlalg = ZeroControl
sim_hardlower = [](0x0)
sim_hardupper = [](0x0)
sim_iterations = 10
sim_line_colour =

   0 0 1

sim_line_width = 2
sim_method = ode
sim_use_nearest = 0
sim_showpoints = 0
sim_showkernel = 0
sim_start =

   0
   0

sim_stopsteady = 0
sim_timeprofile_cols = 2
slices = [](0x0)
steps = 1
stoppingtolerance = 0.0010000
symbols =
{
  [1,1] = [](0x0)
  [2,1] = [](0x0)
}
use_controldefault = 0
use_custom_cost_fn = 0
use_custom_constraint_set_fn = 0
use_parallel = 0
```

In this example, the newly created project has been placed into the project variable. Thus, each of the fields can be accessed and altered using the standard "dot" accessor. For

instance project.use_parallel = 1; would enable parallel processing; disp(project.K); would display the rectangular constraint set of the viability problem.

When using VIKAASA via the library, one is not limited to having a single project open. For instance, the following initialises two new projects in the variables project1 and project2:

```
project1 = vk_project_new;
project2 = vk_project_new;
```

You can also specify project settings as parameters when you call vk_project_new. For instance, the following creates a new project with three variables:[19]

```
proj = vk_project_new( ...
  'numvars', 3, ...
  'discretisation', [9, 11, 7], ...
  'labels', {{'x'; 'y'; 'z'}}, ...
  'symbols', {{'x'; 'y'; 'z'}}, ...
  'diff_eqns', {{'0.2*x - 0.5*y'; '0.5*y + 0.5*z'; 'u'}} ...
);
```

3.4.2. *Loading and saving existing projects.* If you have a previously created project, you can load it into the GUI by going to the "File" menu, selecting "Load," browsing to the folder containing the `.mat` file for your project, and clicking "Open." You will find that the "File" panel in the bottom-right corner of the GUI displays the name and location of this file. As said, loading a project into the GUI overwrites any project information previously loaded.

To save any changes you have made through the GUI, select either "Save Project" or "Save Project As …" from the "File" menu. The former option overwrites your existing project file (as specified in the "File" panel; see Figure 2), whereas the latter option will present you with a dialog box through which you can specify the name and location of a different file.

If you are not using the GUI, projects can be loaded and saved using the vk_project_load and vk_project_save commands, respectively. For instance, to load the file `vikaasa1.mat` in the "Projects" folder and store it in the variable default_project, type:

```
default_project = vk_project_load('Projects/vikaasa1.mat');
```

Note that you can also use the cd command to change the relative path, so that the following pair of commands accomplishes the same thing:

```
cd Projects
default_project = vk_project_load('vikaasa1.mat');
```

[19]Note that double braces are used to delimit cell arrays when they are in a list context.

To save this project to the `MyProject.mat` file in the `MyProjects` folder (assuming that this folder exists), type:

```
vk_project_save(default_project, 'MyProjects/MyProject.mat');
```

As with loading, you can use `cd` to navigate to the folder first.

3.5. **Dynamic variables.** If you are using the GUI, information concerning the names and system's dynamics of the viability problem's dynamic variables – as well as the upper and lower bounds of the rectangular constraint set, and the discretisation to be used – are all held in the "Dynamic Variables" table in the "Variables" panel in the main window, as shown in Figure 4. Each variable is numbered on the left from one through to the number of dynamic variables in the project, and has a name, a symbol, a minimum, a maximum, a discretisation and a function body specifying its dynamics.

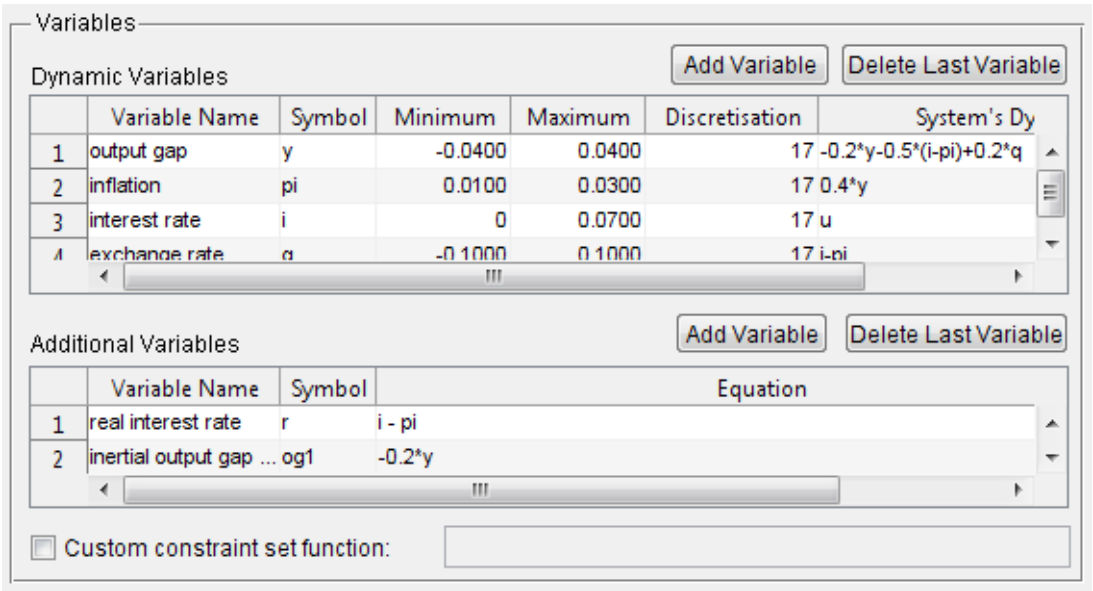


FIGURE 4. The "Variables" panel

If you are using VIKAASA directly through the library, then you can specify the equivalent fields through the `labels`, `symbols`, `K`, `discretisation` and `diff_eqns` fields of your project's struct. Examples are given in the following subsections.

3.5.1. *Names / labels.* The name of the variable serves to identify the quantity being represented. Variable names are automatically added as axis labels when using VIKAASA's plotting features.

VIKAASA stores projects' variable names in the `labels` field, as a cell array of strings in column order. Thus, if you had a project with three variables in the variable `p`, you could specify the labels as follows:

```
p.labels = {'First Variable'; 'Second Variable'; 'Third Variable'};
```

Note that semicolons are used as separators, because the data needs to be in column form.

3.5.2. *Symbols.* The symbol of a variable is the set of characters by which the variable will be referred to in any functional context (e.g., specification of system's dynamics, CCSF, or cost function). It can be any combination of letters and numbers that is permitted by either Octave or MATLAB® variable syntax,[20] so it is up to the user whether to use letters such as y, or to use more descriptive variables such as OutputGap.

VIKAASA stores projects' variable symbols in the symbols field, which is a cell array of strings in column order. An example of setting the symbols field in some project p would be:

```
p.symbols = {'var1'; 'var2'; 'var3'};
```

Note that semicolons are used as separators, because the data needs to be in column form.

3.5.3. *Minimum and maximum / constraint set.* The minimum and maximum values correspond to the upper and lower bounds, respectively, of the rectangular constraint set K, as discussed in Section 2.1.

VIKAASA stores the rectangular constraint set of each project in the K field, which is a row vector of length $2n$, arranged as a list of minimum-maximum pairs (making it easy to use with the axis command). As an example, to make a square constraint set centred around $\left(\frac{1}{2}, -\frac{3}{4}\right)$ with unit length and width, enter the following:

```
p.K = [0, 1, -1.25, 0.25];
```

The first half of this vector (i.e., [0, 1]), gives the minimum and maximum values of the first dimension, and the second half gives the values for the second dimension. Similarly, to create a three-dimensional constraint set, one would specify a row vector of length six.

3.5.4. *Discretisation.* Discretisation is used by the viability kernel approximation algorithm, as described in Section 2.3, to determine which points in K to examine. Factors determining a good value for discretisation are considered in Section 2.5.1. The default for each dimension is 11.

VIKAASA stores discretisation information for each project in the discretisation field, which is a column vector of length n. For instance, setting discretisation of all dimensions to 20 in a project, p with 10 variables, one could type:

```
p.discretisation = 20*ones(10,1);
```

3.5.5. *System's dynamics.* The problem's differential inclusions are specified to VIKAASA in terms of the system's dynamics, $f_1(x,u), f_2(x,u), \ldots, f_n(x,u)$ (as in item 6 of Section 1.1). Each function $f_i(x,u)$ should be a valid Octave or MATLAB® expression, written in terms of the dynamic variable symbols (as explained in Section 3.5.2), as well as the control symbol (see Section 3.7; by default it is u), and evaluating to a numeric

[20] See http://www.gnu.org/software/octave/doc/interpreter/Variables.html or http://www.mathworks.com/help/techdoc/ for more information on allowed variable names. Generally, they should start with a letter and may contain alpha-numeric symbols only.

value.[21] Thus, as an example, 0 (i.e., $f_i(x,u) = 0$) would be an admissible function body, as would rand(1) (i.e., random movement between 0 and 1). If one had defined x and y as variable symbols, 0.5*x − y*u in the first row would represent the function $f_1(x(t),u(t)) = \frac{1}{2}x_1(t) - x_2(t)u(t)$.

VIKAASA stores these functions in the diff_eqns field, which is a cell array of strings in column form. Thus, to create a two-variable system, with differential inclusions, $\dot{x}(t) = \frac{1}{2}x(t) - y(t)$ and $\dot{y}(t) \in x(t)u(t)_{u(t)\in U}$, one could enter:

```
proj = vk_project_new;
proj.symbols = {'x';'y'};
proj.diff_eqns = {'0.5*x - y'; 'x*u'};
```

Note that semicolons are used as separators, because the data needs to be in column form.

3.5.6. *Custom constraint set function.* A custom constraint set function (CCSF), as explained in Section 2.2 can also be specified in the "Variables" panel of the GUI. To do so, check the box next to "Custom constraint set function" and then enter the function body into the text area to the right.

To specify a CCSF directly through the library, there are two fields that need to be set. The first is the custom_constraint_set_fn field, which should contain a string which evaluates[22] to a valid MATLAB® or Octave expression, giving the CCSF. The second is the use_custom_constraint_set_fn field, which is a Boolean, telling VIKAASA whether to evaluate the CCSF (when use_custom_constraint_set_fn is set to true), or not.

The CCSF (when evaluated by VIKAASA) should be a valid Octave or MATLAB® expression which returns a Boolean. For instance true would constitute a valid (but useless) CCSF. Symbols associated with the system's dynamic variables (as described in Section 3.5.2) may be used in the CCSF. For instance, if one had defined the system symbols as x and y, then x^2 + y^2 <= 0.5 would be an admissible CCSF, one that imposed a circular constraint centred around the origin. To specify this same example from the command-line, for some project proj, one would enter:

```
proj.custom_constraint_set_fn = 'x^2 + y^2 <= 0.5';
proj.use_custom_constraint_set_fn = true;
```

3.5.7. *Adding and removing dynamic variables.* When you create a new project in VIKAASA it will have two dynamic variables by default. To add more variables using the GUI, click the "Add Variable" button above the "Dynamic Variables" table in the "Variables" panel. A new row will appear at the bottom of the list of dynamic variables, which you can then populate with information. You can also delete the last row in the list by clicking the "Delete Last Variable" button.

[21] Note that only the *dynamic* variable symbols may be used. Symbols representing so-called "additional" variables, as described in Section 3.6, may not be used for describing the system's dynamics.

[22] VIKAASA uses the inline function to evaluate the custom_constraint_set_fn field with the symbols field supplying the inline function parameters.

To specify the number of dynamic variables through the VIKAASA library, one needs to do two things. Firstly, the numvars field of the project needs to be set to the desired number of variables. Secondly, all fields whose contents are dependent on the number of variables need to be updated. These are labels, symbols, K, discretisation and diff_eqns. Once you have set numvars, you can specify these as illustrated in the preceding sections. Alternatively, you can use the vk_project_sanitise function to adjust the lengths of all variables as appropriate, and then simply specify the values of the new variables. For instance, if you had a project, p, with some (possibly unknown) number of dynamic variables, the following would increment the number by one, and set-up the new variable:

```
% Increment the number of dynamic variables:
p.numvars = p.numvars + 1;

% Adjusts field lengths
p = vk_project_sanitise(p);

% Specify settings related to the new dynamic variable:
p.labels{end} = 'New Variable Name';
p.symbols{end} = 'z';
p.K(end-1:end) = [0 1];
p.discretisation(end) = 11;
p.diff_eqns{end} = 'z*u';
```

Example Box D. Creating a project for the fisheries problem

In this box we will create a new project and specify our (VIKAASA-compatible) fisheries problem in it, using the numbers and equations that we determined in Example box C. First we will walk through creating the project using the GUI, then provide the commands to accomplish the same thing using the library:

- Launch the VIKAASA GUI as described in Section 3.3.
- Firstly, we will create a new project by going to the "File" menu, and selecting "New Project" (see Figure 3). This gives us a blank project with two variables, which is exactly the number that we need.
- Now we need to define our variables in the "Variables" panel. As mentioned in Example box B, there are two variables for this problem: fish biomass and effort. So, enter these under the "Variable Name" column. You can enter the variable names in either order – it doesn't matter, so long as you enter the other parameters in the same order.
- The symbols that we used in Example box B were $b(t)$ for biomass, and $e(t)$ for effort. So, you could enter "b" and "e" into the "Symbols" column, or you can choose some other pair of symbols if you prefer.
- As we discovered in Equation 9, an appropriate rectangular constraint set for our problem was $K = [5, 500] \times [0, 1]$, so put these numbers into the "Maximum" and "Minimum" spaces. 5 is the minimum fish biomass, and 500 is the maximum fish biomass. Similarly for effort.
- In the "System's Dynamics" column, we need to put the right-hand sides of our differential inclusions (as specified in Example box C) in a format that MATLAB® will

understand. To do so, we should use the symbols we entered into the "Symbols" column, and "u" for the control symbol (or, if you prefer, you can change the control symbol to something else – see Section 3.7). If you used "b" and "e" as above, then Equation 7 will become `0.4*b*(1 − b/500)− 0.5*e*b` and Equation 8 will become `u`.

- We will leave the discretisation with the default values of 11 in each dimension for the time being.
- We need to specify a custom constraint set function (CCSF) for this problem, as the constraint set is not rectangular (see Equation 11). We need to specify the CCSF (given in Equation 10) in a format that MATLAB® will understand, so assuming that the variables "b" and "e" were chosen above, the CCSF should be: `4*e*b >= 10*e + 100`. To set this, check the "use custom constraint set function" check-box, and then enter the inequality into the text box to the right.
- Lastly, we should specify the absolute maximum value of the control, $c = \frac{1}{100}$. To do this, put `0.01` into the "Maximum Size (+/-)" field of the "Control" panel (see Figure 6).

Now that we have entered our information, we should save the project file. To do this, go to the "File" menu, select "Save As …" and choose a file name for the project. "Fisheries Example," perhaps. a `.mat` extension should be automatically added to the file when you save it. By convention we normally put project files into the "Projects" folder, but this is not strictly necessary.

To accomplish the same thing using the library interface, we could enter the following commands at the Octave or MATLAB® command-line, or place them into a `.m` file:

```
% Initialise the environment
vikaasa_cli

% Create the project
fisheries_proj = vk_project_new( ...
  'numvars', 2, ...
  'labels', {{'fish biomass'; 'effort'}}, ...
  'symbols', {{'b'; 'e'}}, ...
  'K', [5 500 0 1], ...
  'discretisation', [11 11], ...
  'diff_eqns', {{'0.4*b*(1 - b/500) - 0.5*e*b'; 'u'}}, ...
  'use_custom_constraint_set_fn', 1, ...
  'custom_constraint_set_fn', '4*e*b >= 10*e + 100', ...
  'c', 0.01);

% Save the project in the "Projects" folder
vk_project_save(fisheries_proj, 'Projects/Fisheries Example.mat');
```

3.6. **Additional variables.** VIKAASA also allows for so-called "additional" variables to be specified. These are variables which give data of interest, but which are completely determined by the other variables in the system, and so are not necessary for the purposes of determining the evolution of the system or for computing the viability of state-space points. Specifying additional variables therefore does not affect the kernel approximation routine, but can be useful for plotting the results of the approximation, for instance by

constructing aggregates where a system is highly dimensional, or by providing an alternative viewpoint. For instance, in a system that models nominal interest and inflation, we could specify "real interest" as the difference between these two. Specifying this as an additional variable then means that we can see how the viability of our system is affected by real interest instead of nominal interest. In Example box H we give some examples of using this feature with our fisheries problem.

To specify additional variables in the GUI, you first need to add an additional variable by clicking the "Add Variable" button above the "Additional Variables" table (see Figure 4). This will add a line to the table, where you can enter your information. Similar to the dynamic variables, additional variables take symbols and labels – the same requirements apply as outlined in Section 3.5.1 and 3.5.2. Next, in the "Equation" column, you need to give the right-hand side of the equation specifying the additional variable in terms of the values of the dynamic variables. For our real interest example then, assuming that we had i and pi as the symbols for our (dynamic) interest rate and inflation, and we wanted to specify real interest as $r = i - \pi$, we would put "real interest rate" into the "Label" column, r into the "Symbol" column, and i − pi into the "Equation" column.

You can add as many additional variables as you like. Because they can get in the way when creating plots, an "Ignore" option exists which will remove them from consideration if required.

From the command-line, the corresponding settings are the addnsymbols, addnlabels, addneqns and addnignore fields of any project. Similar to the procedure for adding dynamic variables, you can add or remove additional variables by altering the value of numaddnvars and then calling vk_project_sanitise. The following example, for some project, p, gives an example of how one might increment the number of additional variables:

```
% Increment the number of additional variables:
p.numaddnvars = p.numaddnvars + 1;

% Update all the variables to reflect this:
p = vk_project_sanitise(p);

% Set the value of the last additional variable, assuming that 'r' and 'pi'
% correspond to dynamic variables.
p.addnsymbols{end} = 'r';
p.addnlabels{end} = 'real interest rate';
p.addneqns{end} = 'i - pi';

% Ignore the new variable for the time being (i.e. don't display it in plots):
p.addnignore(end) = 1;
```

All of the fields are cells (with entries in column form), except for addnignore, which is a column vector.

3.7. **Specifying kernel solution settings.** As Section 2.3 explains, the kernel solution algorithm employed by VIKAASA requires parameters in addition to those covered in the

previous section. In the GUI, stopping tolerance, ϵ and step size, h are specified via the corresponding fields in the "Options" panel (see Figure 5), and the control algorithm and related options are specified in the "Control" panel (see Figure 6), including the control symbol, and whether to restrict the control algorithm in certain circumstances.

If you are using VIKAASA through the library interface, the corresponding settings can be specified via the stoppingtolerance, h, controlalg, controlsymbol, controlbounded and controlenforce fields of the project struct.

3.7.1. *Stopping tolerance.* Stopping tolerance, ϵ is used to determine when the system is "close enough" to steady. The process is described in Section 2.5.2. It can be any positive number, and is stored in the stoppingtolerance field of each project. For instance, 0.001 or 1e−3 could be entered as stopping tolerance values.

To set the stopping tolerance of some project p, one could enter:

```
p.stoppingtolerance = 1e-3;
```

3.7.2. *Step size.* The step size, h is described in Section 2.5.3. It can be any positive number, and is stored in the h field of each project. For instance, to set the step size of some project p to $\frac{1}{2}$, one could enter:

```
p.h = 0.5;
```

3.7.3. *Control symbol.* In the GUI, the symbol to use in differential equations to represent the current control choice can be set in the "Control Symbol" field of the "Control" panel. From the command-line, you can set the controlsymbol field, as in the following example:

```
p.controlsymbol = 'MyControl';
```

By default the control symbol is u (as it is assumed to be in most of the examples in this manual), but it can be any valid variable name.

3.7.4. *Control set.* As discussed in item 5 of Section 2.1, the specification of a viability problem's control set reduces to defining the absolute maximum size of the control, c. To specify this in the GUI, you can edit the "Maximum size (+/-)" field in the "Control" panel. To set it via the command-line, set your project's c field, as in the following example:

```
p.c = 0.05;
```

This maximum value is not strictly enforced by VIKAASA unless the "Enforce" checkbox is selected, or the controlenforce field is set to true. That is, if the "enforce" option is not selected, it would be possible for a control algorithm to select a control outside of the $[-c, c]$ range. The control algorithms that are included with VIKAASA all adhere to the control set limits, making this option unnecessary. However, it may be useful to turn this option on when using a custom-built control function. If it is enabled, then any control choice outside of $[-c, c]$ will be replaced with either $-c$ or c, depending on which

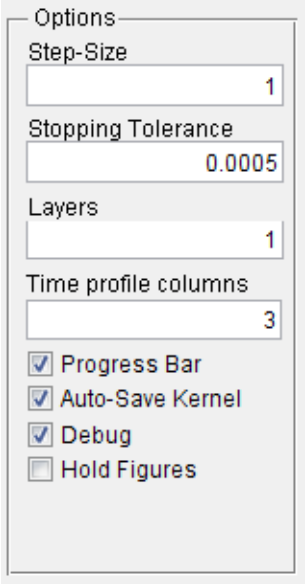


FIGURE 5. The "Options" panel

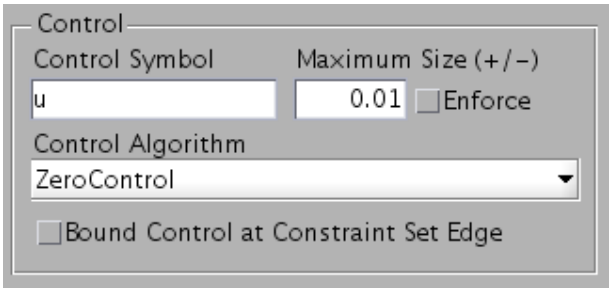


FIGURE 6. The "Control" panel

is closest. To enable enforcing of the control set limits from the command-line for some project p, one would enter:

```
p.controlenforce = 1;
```

3.7.5. *Bounding the control algorithm at the constraint set edge.* VIKAASA can also attempt to limit the control algorithm's choices when the system is close to the boundary of the rectangular constraint set.[23] For instance, consider a system where one of the system's variables, y is governed by the differential equation, $\dot{y}(t) = u(t)$, and at time τ, $y(\tau) = \underline{y}$. That is, y is right on the lower boundary. In this circumstance, choosing $u(\tau) < 0$ will cause a violation of the constraints, resulting in the point being marked as non-viable. VIKAASA can detect that this is the situation and "massage" any choices that will cause an immediate crash into ones that will instead keep (just) within the constraints. For instance, if $u^*(\underline{x}) = -c$, in the above example, then VIKAASA will wrap this control choice in another function $u^{**}(\cdot)$, so that $u^{**}(\underline{x}) \approx 0$. Algorithm 3 gives an outline of how this is done.

[23]This option only works with rectangular constraint sets, and so is always disabled when a CCSF is in use.

Algorithm 3 $u^{**}(\cdot)$ – bound control choices to prevent easily avoidable constraint violations

Require: $CCSF(\cdot) = CCSF^d(\cdot)$
  $l \leftarrow -c$ **and** $r \leftarrow c$ {Initialise upper and lower bounds for use below.}
  $u \leftarrow u^*(x)$
  {Check to see if u^* has caused a constraint set violation:}
  if $x + h \cdot f(x,u) \notin K$ **then**
    {Create a set, O of all the dimensions in which violation occurred:}
    $O \leftarrow \{i : x_i + h \cdot f_i(x,u) > \overline{x}_i\} \cup \{i : x_i + h \cdot f_i(x,u) < \underline{x}_i\}$
    for all $i \in O$ **do**
      {Minimise the distance to the closest edge in that dimension:}
      if $x_i + h \cdot f_i(x,u) > \overline{x}_i$ **then**
        $u_i \leftarrow \arg\min^G_{u \in [l,r]} (x_i + h \cdot f_i(x,u) - \overline{x}_i)$
        $d_i \leftarrow x_i + h \cdot f_i(x,u_i) - \overline{x}_i$
      else
        $u_i \leftarrow \arg\min^G_{u \in [l,r]} (x_i + h \cdot f_i(x,u) - \underline{x}_i)$
        $d_i \leftarrow x_i + h \cdot f_i(x,u_i) - \underline{x}_i$
      end if
      {Reduce the size of the control set for the next iteration:}
      if $u_i > 0$ **then**
        $l_i \leftarrow l$ **and** $r_i \leftarrow u_i$
      else if $u_i < 0$ **then**
        $l_i \leftarrow u_i$ **and** $r_i \leftarrow r$
      else
        $l_i \leftarrow u_i$ **and** $r_i \leftarrow u_i$
      end if
      $O_i \leftarrow \{i : x_i + h \cdot f_i(x,u_i) > \overline{x}_i\} \cup \{i : x_i + h \cdot f_i(x,u_i) < \underline{x}_i\}$
      if $d_i \approx 0$ **and** $|O_i| < |O|$ **then**
        $u \leftarrow u_i$ **and** $O \leftarrow O_i$ **and** $l \leftarrow l_i$ **and** $r \leftarrow r_i$
      end if
    end for
  end if
  if using a default value, u^d **then**
    repeat the above for u in $x + h \cdot f(x + h \cdot f(x,u), u^d)$
  end if
  return u

Note firstly that in the case where u^* produces no constraint violation, $u^{**}(x) = u^*(x)$. Where there is violation however, u^{**} works by attempting to reduce the number of dimensions in which constraint violation occurs. It does this by gradually reducing the control set available, starting with $[-c, c]$, and shrinking it each time a control is found which reduces the number of constraint set violations.[24] This shrinking can only make

[24] As indicated in Algorithm 3, controls are found by minimising the distance from the boundary of the constraint set along the dimension in question, and then checking that that distance is approximately zero. This is done numerically using fzero function, and the degree of precision is subject to the *control tolerance* parameter, ξ. See Section 3.8.2.

sense where the system's dynamics are fairly linear, so this bounding may not be advisable in all contexts.

Note also that u^{**} performs more checks if the project in question is using a default value (see Section 3.8.4). Thus, these two options work well together, as the bounding code will be more effective if it is able to preempt constraint violations earlier.

To enable this option in the GUI, check the "Bound Control at Constraint Set Edge" option. In the library interface, set the controlbounded field to true. The following example renames the control symbol and enables the "enforcing" and "bounding" options for some project p:

```
p.controlsymbol = 'v';
p.controlenforce = 1;
p.controlbounded = 1;
```

3.8. **Control algorithms.** As indicated in Section 2.3, VIKAASA depends on an effective control algorithm u^* to determine the viability or non-viability of state-space points by using u^* to slow the velocity of the system to a near-steady state. Two flexible control algorithms, CostMin and CostSumMin, are provided with VIKAASA for this purpose. In the following sections we explain firstly the workings control algorithms in VIKAASA, before explaining the usage of CostMin and CostSumMin in more depth. All control algorithms provided with VIKAASA also have additional documentation in Appendix C.

3.8.1. *"Simple" control algorithms.* The simplest control algorithms provided with VIKAASA are MaximumControl, MinimumControl and ZeroControl. All three of these simply apply a constant control (c, $-c$ and 0, respectively) regardless of the system state. These algorithms are therefore not generally useful for approximating viability kernels, but may be useful for experimentation. See Section 4 for details on experimenting with control algorithms.

Additionally, another "simple" algorithm, ManualControl, is also available. This control algorithm prompts the user to enter a control choice at each iteration. It is certainly not useful for computing viability kernels, but again may be useful for experimentation, as illustrated in Section 4.

3.8.2. *Cost-minimising control algorithms.* VIKAASA provides three generic numerical cost-minimising algorithms. The most simple of these is NormMin1Step. This function simply chooses $u^*(x) = \arg\min^G_{u\in[-c,c]} |f(x + h \cdot f(x,u), u^d)|$, where min^G is the numerical golden ratio minimisation function – fminbnd in MATLAB® and Octave – and u^d is some default control value. If a "default value" is specified (see Section 3.8.2), then this is used as the value for u^d. Otherwise, $u^d = 0$.

NormMin1Step thus attempts to minimise the system's velocity, one Euler-step into the future. This is almost always preferable to simply minimising $|f(x,u)|$, as doing so cannot capture any of the dynamic effects of the control choice. For instance, in our fisheries example, using the systems dynamics worked out in Example box C, choosing $u^*(x) = \arg\min_u |f(x,u)|$ will always result in $u^*(t) = 0$, as can be seen in Example box E.

As an improvement on the performance provided by NormMin1Step, two additional algorithms are provided: CostMin and CostSumMin. These algorithms are quite similar – both extend the forward-looking behaviour of NormMin1Step to any arbitrary (positive) number of forward-looking steps, and both also allow for a custom cost function to be specified, so that instead of minimising the Euclidean norm of the system velocity, you can instead choose to minimise some other function if you wish. You could minimise the distance from some analytically determined steady state, for instance. The difference between CostMin and CostSumMin is that CostMin focuses on minimising the cost at the final forward-looking step (or the "scrap cost"), whereas CostSumMin attempts to minimise the sum of the costs from each step. The basic recursive cost algorithms for each function are given below:

(12)
$$cost^S(x,u,r) = \begin{cases} g(x,u) & r = 0 \\ \arg\min^G_{v\in[-c,c]} cost^S(x + h\cdot f(x,u), v, r-1) & \text{otherwise} \end{cases}$$

(13)
$$cost^\Sigma(x,u,r) = \begin{cases} g(x,u) & r = 0 \\ g(x,u) + \arg\min^G_{v\in[-c,c]} cost^\Sigma(x + h\cdot f(x,u), v, r-1) & \text{otherwise} \end{cases}$$

$cost^S$ gives the "scrap" cost of being at x, after r steps; $cost^\Sigma$ gives the cost of being at each x in the steps up to and including r. Using these functions, simplified versions of CostMin and CostSumMin are given in what follows:

Algorithm 4 CostMin – Minimise the scrap cost over some arbitrary number of steps.

Require: $x \in \mathbb{R}^n$
Require: $s \in \mathbb{Z}_+$ {s is the number of forward-looking steps.}
Require: $g : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}$ {g is the cost function. It operates over $x \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$.}

return $\arg\min^G_{u\in[-c,c]} cost^S(x,u,s)$

Algorithm 5 CostSumMin – Minimise the sum cost over some arbitrary number of steps.

Require: $x \in \mathbb{R}^n$
Require: $s \in \mathbb{Z}_+$
Require: $g : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}$
return $\arg\min^G_{u\in[-c,c]} cost^\Sigma(x,u,s)$

Both control algorithms use a recursive algorithm to perform minimisation, roughly analogous to solving an optimisation problem in discrete time using backwards induction. Given that both algorithms employ numerical minimisation methods (i.e., fminbnd), the time required to compute any control choice, u increases exponentially with the number of forward-looking steps, s. In practice it has been found that on a modern PC computer, more than 2 forward-looking steps is not practical, because of the extremely long wait times involved. Numerical minimisation also means that both functions are sensitive

to the *control tolerance*, ξ. Making ξ smaller should result in any numerically-computed minimum values in VIKAASA (i.e., $\arg\min^G f(\cdot)$) more closely approximating the "true" minimum values (i.e., $\arg\min f(\cdot)$).

3.8.3. *Settings for cost-minimising control algorithms.* The VIKAASA GUI has a specific "Minimising Controls" panel for settings pertaining to the three cost-minimising control algorithms (see Figure 7). To select the number of forward-looking steps in the VIKAASA GUI, simply enter it into the "Forward-looking Steps" field in this panel.

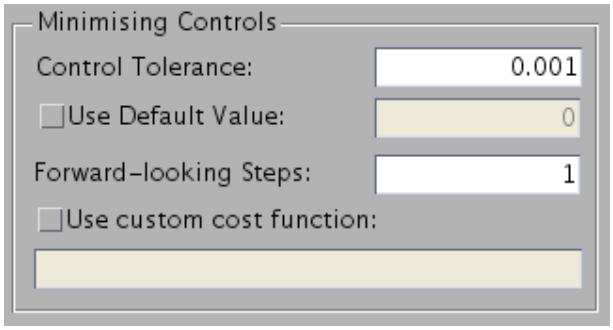


FIGURE 7. The "Minimising Controls" panel

You can also set the *control tolerance* here, which affects the performance of any function in VIKAASA which makes use of either fminbnd or fzero. Although this option is therefore mainly used by NormMin1Step, CostMin and CostSumMin, it is also used by the control bounding code, if this is enabled (see Section 3.7.5).

The "Minimising Controls" panel also has a field for a custom cost function to be specified. If this option is not enabled, then an implicit cost function $g(x,u) = |f(x,u)|$ is used in CostMin and CostSumMin.[25] If you want to specify your own cost function to minimise, you can check the "Use custom cost function" box, and then enter the function body into the field below. The field should be a valid MATLAB® or Octave expression, similar to how the system's dynamics are specified (see Section 3.5.5), or the custom constraint set function (see Section 3.5.6). VIKAASA makes the current value of each *dynamic* variable available via the variable symbols (as described in Section 3.5.2), as well as the current velocity of each variable, for which variables are created by concatenating "dot" onto the end of each symbol name. Thus, if you have symbols x and y, then xdot and ydot would give the velocity of the respective variables, and 0.7*(x^2 + y^2)+ 0.3*(xdot^2 + ydot^2) would be a valid custom cost function.

In order to specify CostMin as your control algorithm, to set the number of forward-looking steps, and to set the control tolerance from the command line, one could enter the following:

```
% Create a new project with 2 variables:
p = vk_project_new;

% Set the variable symbols to be 'w' and 'z':
```

[25] This means that for one forward-looking step, the behaviour of CostMin should be the same as for NormMin1Step when a custom cost function is not specified.

```
p.symbols = {'w';'z'};

% Set 'CostMin' as the control algorithm:
p.controlalg = 'CostMin';

% Two forward-looking steps with CostMin:
p.steps = 2;

% Custom cost function which imposes quadratic costs:
p.use_custom_cost_fn = 1;
p.custom_cost_fn = '(w - z)^2';
```

3.8.4. *Using a default value.* There is an option in VIKAASA called "use default value," which is provided to improve the performance of CostMin and CostSumMin. As explained in Algorithm 4 and Algorithm 5, these algorithms work by minimising recursive cost functions, $cost^S$ and $cost^\Sigma$. As numerical minimisation is used, the number of forward-looking steps can drastically increase the computation time required to determine a control choice. Furthermore, as demonstrated in Equation 23 (below), the final step, represented by minimising the value of $cost^{\cdot}(x,u,0) = g(x,u)$ may be trivially easy to solve, and may not depend on x at all. The "use default value" option therefore exists that this final step can be skipped. The recursive cost functions then become:

(14)
$$cost^{S,d}(x,u,r) = \begin{cases} g(x + h \cdot f(x,u), u^d) & r = 1 \\ \arg\min^G_{v\in[-c,c]} cost^{S,d}(x + h \cdot f(x,u), v, r-1) & \text{otherwise} \end{cases}$$

(15)
$$cost^{\Sigma,d}(x,u,r) = \begin{cases} g(x,u) + g(x + h \cdot f(x,u), u^d) & r = 1 \\ g(x,u) + \arg\min^G_{v\in[-c,c]} cost^{\Sigma}(x + h \cdot f(x,u), v, r-1) & \text{otherwise} \end{cases}$$

That is, the number of recursive optimisation steps is decreased by one. The result is that a cost-minimising control for s forward-looking steps, and using the "use default value" option will take roughly the same time to compute as a control for $s-1$ forward-looking steps, without the option enabled. Thus, where this option makes sense to use,[26] it is recommended that you do so, given the considerable savings in computation time that it should afford.

[26]For cost functions or problems where $\arg\min_u |f(x,u)|$ does not solve to a constant, as it does in Equation 23, it can still often make sense to use zero as the default value. Firstly, consider that for $s > 0$, the value of u fed into $cost^{\cdot}(x,u,0)$ (i.e., the final step in the recursive solution to $cost^S$ or $cost^\Sigma$), given that it does not take into consideration any of the dynamic effects of this choice, is unlikely to reflect the value of $u^*(x)$ when evaluated at the same point. The time-consuming computation required to determine $\arg\min_u cost^{\cdot}(x,u,0)$ is therefore of questionable use, given that it is difficult to know what this minimal value reflects. A zero default value on the other hand can be thought of as representing the mean value in a uniform distribution over $[-c,c]$, and so can be interpreted as an uninformed guess concerning the future choice of u, giving a clearer interpretation than that yielded from not using the "default value" option.

To enable this option in the VIKAASA GUI, check the "Use Default Value" check-box, and then enter the value you want (e.g., 0) into the field to the right. To accomplish the same from the command-line for some project, p, one could enter:

```
p.use_controldefault = 1;
p.controldefault = 0;
```

3.8.5. *Selecting a control algorithm for kernel approximation.* To select a particular control algorithm in the VIKAASA GUI, you can simply select it from the "Control Algorithm" drop-down box in the "Control" panel. The list should include all algorithms present in the "ControlAlgs" folder, including any that you have made yourself. If you have only just created your algorithm though, then you may need to restart VIKAASA before it appears in the list.

To specify a control algorithm through the library, you should change the controlalg field of your project to a string containing the function's name. For instance, to select NormMin1Step as your control algorithm in a project, p, enter the following:

```
p.controlalg = 'NormMin1Step';
```

3.8.6. *Designing custom control algorithms.* In this section we detail the process involved in writing your own control algorithms for use in VIKAASA. Each control algorithm is a function adhering to a particular functional signature, contained inside of it's own `.m` file and residing in the "ControlAlgs" folder. The simplest example is ZeroControl:

```
function u = ZeroControl(x, K, f, c, varargin)
    u = 0;
end
```

As we can see, the first argument that control algorithms take is x, which is a column vector giving the current state-space position of the system. It is expected that the control algorithm will react somehow to this x in determining its choice of u. The other arguments provide additional information to assist the algorithm in doing this:

- K, the system's rectangular constraint set, as described in Section 3.5.3. The custom constraint set function is also accessible if needed, through the options argument (as described below).
- f, the functional representation of the system's dynamics. This function takes two parameters, x, a column vector representing a state space position and u, a scalar representing the control choice.

Clearly, ZeroControl simply selects $u = 0$, regardless of the value of x. A more complex example is given by (a simplified version of) NormMin1Step:

```
function u = NormMin1Step(x, K, f, c, varargin)
    % Initialise the 'options' struct.
    options = vk_options( K, f, c, varargin{:});
```

```
    h = options.h;
    ud = options.controldefault;

    % Return the control that minimises the norm.
    u = fminbnd(@(u) norm(f(x + h*f(x, u), ud)), -c, c);
end
```

The first thing that happens in this function is that a structure called options gets initialised with a call to the vk_options function. If you need to make use information in your control algorithm other than x, K, f and c, then you should initialise this structure. It means that your function can flexibly accept options in a variety of formats, and that any options that are not specified will be replaced by their default values. Most of the settings specified in your project have analogues in the options structure. See vk_options_make for information on this.

In our example then, having initialised options, we extract fields of interest. They could also be used as-is (e.g., you could directly use the value of options.h instead of initialising a variable called h and copying the value into it – this is up to you). Then, using this information, the control choice is determined by using fminbnd.[27]

The options structure contains many more settings. For instance, you can access the viability problem's custom constraint set function (CCSF) via options.custom_constraint_set_fn . A full list of the available options is given in the appendix.

A further example of writing a custom control function is given in Example box F.

3.9. **Running the kernel approximation algorithm.**

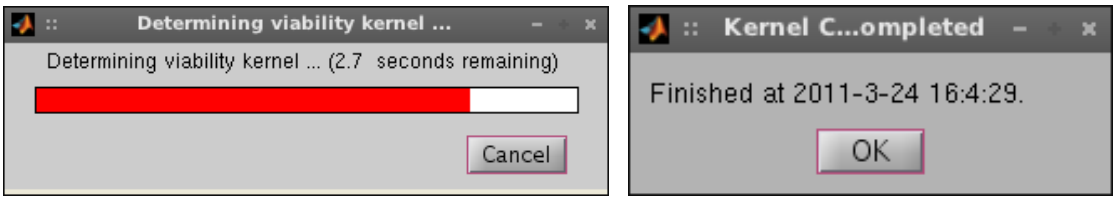
3.9.1. *Running the algorithm in the GUI.* Once all of the variables and options have been specified, you are ready to run the kernel approximation algorithm. In the GUI this is done by clicking the "Run Algorithm" button in the main window. The algorithm can take some time to complete, depending on the dimensionality[28] of your problem, and the options you have specified. The GUI can display a progress bar while the algorithm is running, which includes an estimation of how much longer the algorithm will take to complete. The progress bar also contains a "Cancel" button, which will stop computation and discard the partially computed viability kernel.[29]

[27]The actual NormMin1Step algorithm does not use fminbnd directly in the way illustrated, but instead makes use of the options.min_fn field, which provides a function handle to a minimisation function expected to work like fminbnd. It is therefore possible to alter the options structure so that a different function is used, if desired.

[28]As mentioned, the implemented algorithm suffers from the curse of dimensionality and the computation times can be tens of hours. For example, a four-dimensional kernel with 17 points along each dimension has been known to take 14 hours to compute. On the other hand, the fisheries management problem kernel in two dimensions will take less than 20 seconds to complete.

[29]The progress bar is implemented using the MATLAB® waitbar function. If you choose to use MATLAB®'s Parallel Computing Toolbox to speed up kernel computation time, then the progress bar cannot be used for technical reasons. You may wish to select the "Debug" option in this case in order to still see some feedback while kernel approximation is underway.

The progress bar is only displayed if the "Progress Bar" check-box in the "Options" panel is checked. The reason that the progress bar is optional is that we have experienced situations in which it causes MATLAB® to crash with a segmentation fault. This appears to happen when the number of points under consideration is very high. In such cases, you will want to run the approximation algorithm without the progress bar.



(A) The progress bar

(B) The dialogue box displayed upon completion

FIGURE 8. Running the algorithm

If you choose not to use the progress bar, it is still possible to cancel computation at any time by pressing Control-C while the MATLAB® command window is in focus. If you terminate computation prematurely, the partially-computed viability kernel will be discarded.

After the algorithm completes, you should see a dialogue box on your screen, informing you of the time at which the algorithm completed (see Figure 8b). Some brief facts about the set of viable points computed will be displayed in the "Kernel Results" panel (see Figure 9). This panel also contains a "Delete" button, which will remove any existing computed kernel from the current project; and a "View Kernel Coordinates" button, which will open the set of viable points, V, in MATLAB®'s Variable Editor.

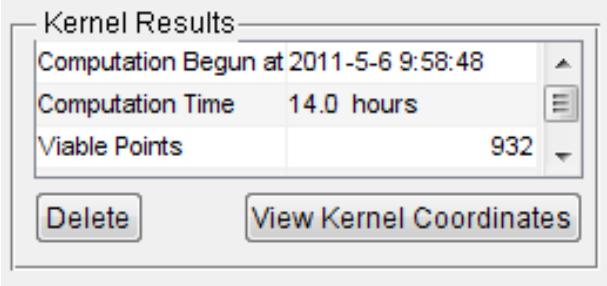


FIGURE 9. The "Kernel Results" panel

3.9.2. *Using vk_kernel_run.* If you are using VIKAASA from the command-line, you can use the vk_kernel_run command to run the approximation algorithm. This command can be run in a number of possible ways. For instance, you can run it directly on a project file:

```
% Load 'Fisheries Example.mat', run the kernel algorithm, and then save the
% results back to that same file.
vk_kernel_run('Projects/Fisheries Example.mat');
```

This will read the kernel approximation settings from the specified `.mat` file, run the kernel approximation algorithm, and then save the results back into the same file, overwriting any existing kernel information. If you want to preserve the file, and save your results to a new file instead, you can enter the following:

```
% Load 'Fisheries Example.mat', run the kernel algorithm, and then save the
% results to 'New File.mat'.
vk_kernel_run('Projects/Fisheries Example.mat', 'Projects/New File.mat');
```

vk_kernel_run can also take a project structure (such as is returned by vk_project_load or vk_project_new) as an argument. In this case, it will return another structure, representing the same project, but with the kernel information added in:

```
proj = vk_project_load('Projects/vikaasa_default.mat');
% Run the algorithm and store the results in proj2 (so that proj is unchanged):
proj2 = vk_kernel_run(proj);

% Run the algorithm and store the results back into proj:
proj = vk_kernel_run(proj);

% Save the results.
vk_project_save(proj2, 'Projects/New File.mat');
```

If the progressbar field of the project is set to 1, then vk_kernel_run will display the percentage competed on the command-line.[30]

```
% Enable the progress display.
p.progressbar = 1;
% Run the approximation algorithm.
p = vk_kernel_run(p);
```

The resulting set of viable points from running vk_kernel_run (or from running the algorithm through the GUI) are stored in the V field of the project in question. V is a $n \times v$ matrix, where n is the number of dimensions in the problem (e.g., 2 in the fisheries problem), and v is the number of viable points found. Each row in V therefore represents a state-space point that is deemed viable. Additional information is also stored in the comp_time and comp_datetime fields. To get information similar to what would be displayed in the "Kernel Results" panel of the GUI, you can use the vk_kernel_results function:

```
octave:1> p = vk_project_load('Projects/vikaasa_default.mat');
octave:2> vk_kernel_results(p)
ans =
{
```

[30] Note that in order for this to work correctly in Octave you may need to turn off screen paging, by entering: page_screen_output(0);.

```
  [1,1] = Computation Begun at
  [2,1] = Computation Time
  [3,1] = Viable Points
  [4,1] = Percentage Viable
  [1,2] = 2011-5-6 9:58:48
  [2,2] = 14.0 hours
  [3,2] = 932
  [4,2] = 1.1159
}
```

3.9.3. *Options related to running the kernel approximation algorithm.* In addition to the various options already discussed, there are is a "Debug" option, which when enabled causes the state-space coordinates of each point under consideration to be printed to the command window while the algorithm is running (in addition to some other information). This can be useful for solving problems with control algorithms, and it is also suggested that you enable this option if you are using the Parallel Computing Toolkit.

In the GUI, this option can be enabled by checking the check-box in the "Options" panel. From the command-line, it is enabled by setting the debug field of a project to 1:

```
% Enable debugging in the project 'proj':
proj.debug = 1;

% Run the kernel algorithm with debugging turned on:
proj = vk_kernel_run(proj);
```

An "Auto-Save Kernel" option also exists for use with the GUI. If this is enabled (via the check-box in the "Options" panel), then upon successful completion of the kernel approximation algorithm, your project will be saved into the `autosave.mat` file in the "Projects" folder. To accomplish the same thing from the command-line, use vk_kernel_run with two parameters, as explained above. For example:

```
cd Projects
vk_kernel_run('vikaasa_default.mat', 'vikaasa_autosave.mat');
```

3.9.4. *Using vk_kernel_compute.* The functionality given by vk_kernel_run, although convenient, may not be sufficient for your needs. Specifically, if you wish to run the kernel computation algorithm with a highly customised options structure, then you will need to use the lower-level vk_kernel_compute function instead. This function takes three basic arguments: K, the problem's rectangular constraint set; f, a function, $f : \mathbb{R}^n \times \mathbb{R} \mapsto \mathbb{R}^n$, representing the system's dynamics; and $c \in \mathbb{R}_+$, representing the absolute maximum size of the control. It returns a set of viable points, V, a $n \times v$ matrix, as described in the previous section. A simple example would be the following (representing the fisheries problem):

```
% Specify the problem:
K = [5, 500, 0, 1];
f = @(x,u) [0.4*x(1)*(1 - x(1)/500) - 0.5*x(2)*x(1); u];
c = 0.01;

% Run the algorithm, using the default (zero) control algorithm:
V = vk_kernel_compute(K, f, c);
```

You can also specify options to this function in a variety of ways. If there are a small number of additional options that you want to specify, then you can do so by specifying name, value pairs after the three mandatory parameters, as in the following example:

```
% Run the algorithm, using the CostMin control algorithm and with control
% bounding enabled:
V = vk_kernel_compute(K, f, c, ...
    'control_fn', @CostMin, ...
    'controlbounded', 1);
```

Or, if you have a large number of options and want to keep them together, use vk_options to initialise an options structure, as in the following:

```
% Create options structure:
options = vk_options(K, f, c, ...
    'control_fn', @CostMin, ...
    'controlbounded', 1);

% Run the algorithm, using these options.
V = vk_kernel_compute(K, f, c, options);
```

If you choose to use functions such as vk_kernel_compute directly, then you lose the ability to keep your work organised in a project file, but you are able make many more customisations than are available through vk_kernel_run.

Example Box E. Creating a viability kernel for the fisheries problem

VIKAASA requires that we specify a control algorithm, or *feed-back rule*, $u^*(x)$ for our fisheries problem. This rule needs to take a state-space representation of our problem,

$$x(t) = \begin{bmatrix} b(t) \\ e(t) \end{bmatrix} \tag{16}$$

and provide a control,

$$u \in [-c, c] = \left[-\frac{1}{100}, \frac{1}{100}\right] \tag{17}$$

Essentially then, we are trying to solve an infinite horizon constrained optimal control problem in continuous time. We could therefore attempt to solve:

$$\min_{u(t)} \int_0^\infty e^{-}\rho t|f(x(t),u(t))|dt \tag{18}$$

The solution to this problem may however be difficult to obtain. Furthermore, it is probably unnecessary, given that we are not really concerned with establishing an optimal path, but only with steadying the system in a manner that does not affect the system's *viability*. This can be done with a bit trial and error, trying different approaches to see if they produce plausible viability kernels.

As a first guess, we could try $u^*(x) = 0$, a control rule which says to "do nothing," regardless of what levels effort or biomass are at. We can then compute a viability domain, consisting of points in K_δ for which this control choice results in the system slowing to a near-steady state, by using selecting the ZeroControl algorithm in VIKAASA, and running the kernel algorithm (as described in Section 3.9). To do so through the GUI, select the ZeroControl algorithm from the drop-down list in the "Control" panel, and then click "Run Algorithm." Note that we are using the default 11-point discretisation setting still, so we will be examining 121 points. The resulting viable points are displayed in Figure 10a (see Section 3.10 for details on producing plots). These points represent the starting state combinations of fish biomass, b and effort e, for which making no changes to effort (i.e., choosing $\dot{e}(t) = 0$ for all t) results in a near-zero velocity being achieved without violating the constraints (as in Equation 11).

We can then interpolate this viability domain using the *convex hull* method to arrive at an approximation of all the points in $K_{effective}$ for which no control results in a viable trajectory, represented by the yellow area in Figure 10b. This area corresponds roughly to the *viability niches* identified in [2] – the points from which the system will drift, unassisted to a steady state within the viability constraints.

According to [2] though, this viability niche does not necessarily represent the viability *kernel* for our problem. This is because for a certain set of points above or below the area we have found, it should be possible to decrease, or increase e so as to arrive inside this area. We therefore need non-zero control to establish the viability of these points.

As a next step then, we could try to seek out steady state-space points by choosing the control that minimises the current system velocity: $u^*(x) = \arg\min_{u\in[-c,c]} |f(x,u)|$. We can easily solve this problem to obtain a feed-back rule, using the following first-order condition:

$$\frac{\partial}{\partial u^*}|f(x(t),u^*)| = 0 \tag{19}$$

$$\frac{\partial}{\partial u^*}\left|f\left(\begin{bmatrix} b(t) \\ e(t) \end{bmatrix}, u^*\right)\right| = 0 \tag{20}$$

$$\frac{\partial}{\partial u^*}\sqrt{\left(\frac{2}{5}b(t)\left(1-\frac{b(t)}{500}\right)-\frac{1}{2}b(t)e(t)\right)^2+(u^*)^2} = 0 \tag{21}$$

$$2u^* \cdot \frac{1}{2}\left(\left(\frac{2}{5}b(t)\left(1-\frac{b(t)}{500}\right)-\frac{1}{2}b(t)e(t)\right)^2+(u^*)^2\right)^{-\frac{1}{2}} = 0 \tag{22}$$

$$u^* = 0 \tag{23}$$

This problem therefore solves for $u^*(x) = 0$, which we have already determined is not sufficient to find the viability kernel. The reason that Equation 23 solves for this is that any non-zero

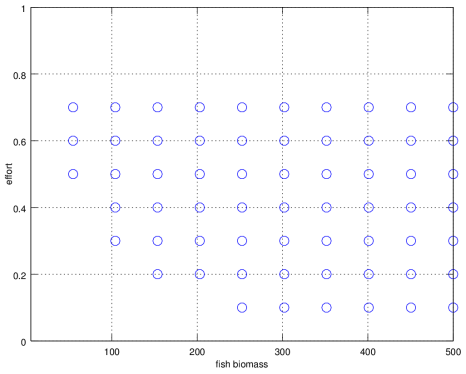
choice of u will increase the immediate system velocity, in that it has no immediate effect on $\dot{b}(t)$ (i.e., $\frac{\partial b}{\partial u} = 0$), and it dictates the velocity of $e(t)$.

What about the dynamic effects of choosing $u(t) \neq 0$ though? We argued above that increasing or decreasing $e(t)$ could help to steer the system into a position where a steady state was achievable. Thus, what happens if we search around the current state-space position, x, for a reachable state-space position at which point the velocity is lower (and therefore closer to a steady state)? That is, what if we attempt to minimise $|f(x + h \cdot f(x, u_1), u_2)|$?

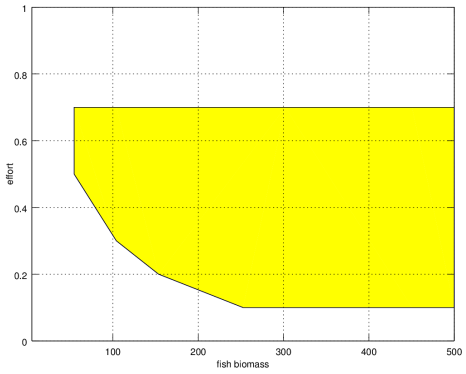
As we already know from Equation 23 that the velocity-minimising value of u_2 will be zero, we can attempt to answer this by modelling the control, $u^*(x) = \arg\min_{u \in [-c,c]} |f(x + h \cdot f(x, u), 0)|$, which is (approximately) the algorithm employed by NormMin1Step. Let's run NormMin1Step on our problem then, and see what the resulting viability domain looks like. In the GUI, select NormMin1Step from the drop-down in the "Control" panel, and then click the "Run Algorithm" button again. The resulting interpolated kernel is shown in Figure 10c, where the blue area represents the viability domain we computed using $u^*(x) = 0$, and the yellow area represents the *additional* points which are now considered viable using the one step norm minimising algorithm.

This is looking more promising. Whereas before there were no viable points above $e = 0.7$, we now have viable points for $e = 0.8$, and 0.9, indicating that NormMin1Step must have successfully steered these points downwards. Now, let's increase the discretisation, δ from $[11\ 11]'$ to $[50\ 50]'$, run the algorithm again, and plot the result. You should get a graph like Figure 10d. The higher discretisation results in smoother edges, as well as a larger overall area of viable points.

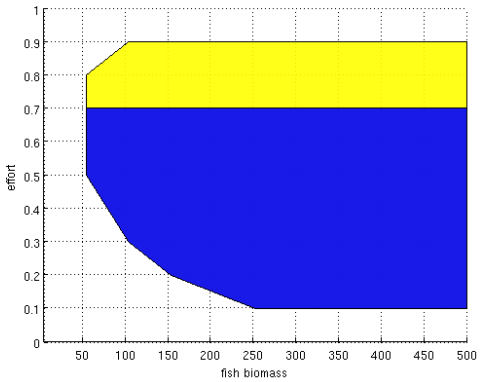
In Example box F below, we will see how we can improve on this result even further by designing a customised control algorithm.



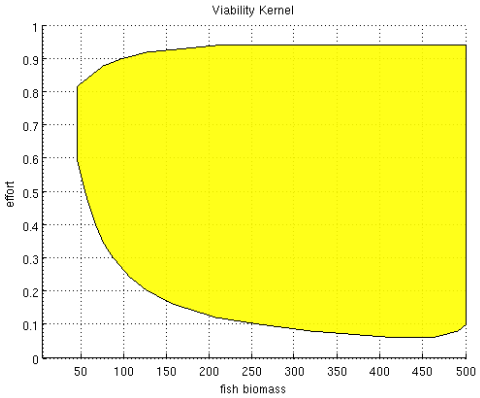
(A) A viability domain for the fisheries problem using $u^*(x) = 0$



(B) An interpolation of the viability domain in Figure 10a



(C) The yellow area represents the additional points determined viable using NormMin1Step, over ZeroControl (shown in blue).



(D) An interpolation of the viable points found using NormMin1Step for a higher discretisation.

FIGURE 10. Viability domains for the fisheries problem (see Example box E).

EXAMPLE BOX F. CREATING A CUSTOM CONTROL ALGORITHM FOR THE FISHERIES PROBLEM

In Example box E we first discovered a viability domain using the ZeroControl control algorithm in Figure 10b, and then extended it using NormMin1Step in Figure 10c. The question now is: can we do any better, or have we found the (approximate) viability kernel?

This is in general a difficult question to answer. In this particular case though, we are lucky enough to have some theory to guide us. We know from [2] and from Figure 10b that for $e \in [0.1, 0.7]$, no change to e is necessary for a point to be shown viable (or otherwise) – between these upper and lower bounds, we are in the so-called "viability niche," wherein state-space points drift left or right, gradually slowing as they do so, without any control being exerted, and without violating the viability constraints in doing so.

We can use this knowledge to design a simple control algorithm for this problem, one that does nothing for $e(t) \in [0.1, 0.7]$, exerts $u(t) = -c$ for $e(t) > 0.7$ and $u(t) = c$ for $e(t) < 0.1$.

Let us call this algorithm FisheriesControl. To create this algorithm, open a new file called `FisheriesControl.m` in the "ControlAlgs" folder, and enter the following code:

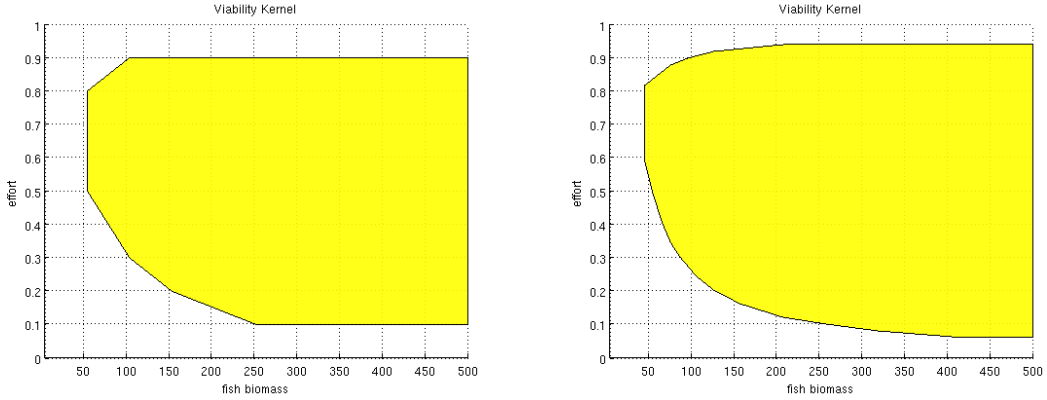
```
function u = FisheriesControl(x, K, f, c, varargin)
    %% Extract information from the state−space vector:
    b = x(1);
    e = x(2);

    %% Define our upper and lower bounds for e:
    e_upper = 0.7;
    e_lower = 0.1;

    %% Determine the control to use.
    % Note that we are calling ZeroControl, MaximumControl and
    % MinimumControl, purely to demonstrate how you can chain existing
    % control algorithms into your custom ones. We could just as well use
    % '0', '−c' and 'c' ourselves.
    if (e > e_upper)
        u = MinimumControl(x, K, f, c, varargin{:});
        % or: u = −c;
    elseif (e < e_lower)
        u = MaximumControl(x, K, f, c, varargin{:});
        % or: u = c;
    else
        u = ZeroControl(x, K, f, c, varargin{:});
        % or: u = 0;
    end
end
```

Because of the robust reasoning behind this algorithm, we can be fairly confident that kernels generated using it will approximate the problem's viability kernel. Moreover, because the algorithm makes no use of numerical optimisation tools such as fminbnd, it should perform much faster than using NormMin1Step did.[31]

Having created this algorithm then, we can run it in the GUI. If you already have the GUI open, you will need to restart it (and then reload the fisheries model) before the algorithm becomes available from the drop-down in the "Control" panel. Select it, and then click the "Run Algorithm" button again. Once the computation has completed, click the "Plot Kernel" button to view the result. You should see an area like the one in Figure 11a. This area is in fact exactly the same as the one in Figure 10c. Given our current coarse discretisation then, both NormMin1Step and FisheriesControl appear to produce the same set of viable points. This shows that in the right situation, a simple, generic algorithm like NormMin1Step may be all you need to obtain a reasonable approximation of the viability kernel.

In Figure 10d we increased the discretisation to $\delta = [50\ 50]'$, and ran NormMin1Step again. The resulting area was larger, and smoother. Interestingly though, there was an area in the bottom-right, where NormMin1Step failed to find viable points. When we run FisheriesControl with this same discretisation, we see get the figure shown in Figure 11b. Although the difference is quite small, we can see then that this algorithm is actually more successful. This illustrates the

benefits of designing your own algorithm, based on a sound understanding of how the problem works.



(A) An interpolation of the viability domain for FisheriesControl using $\delta = 11$ in each dimension.

(B) An interpolation of the viability domain for FisheriesControl using $\delta = 50$ in each dimension.

FIGURE 11. Viability kernel for the fisheries problem at various levels of discretisation (see Example box F).

3.10. **Visualisation tools.** VIKAASA includes an interface to the plotting features of MATLAB® and Octave in order to visualise viability kernel approximations and viability domains. This functionality is accessible in the GUI through the "Kernel Plotting" panel of the main window.

3.10.1. *Creating a kernel visualisation.* To view two- or three-dimensional kernels, you can simply click the "Plot Kernel" button in the GUI. Assuming you have not changed any of the visualisation settings from their defaults, this should display your visualisation in a new window, as either a two-dimensional area, or a three-dimensional volume.

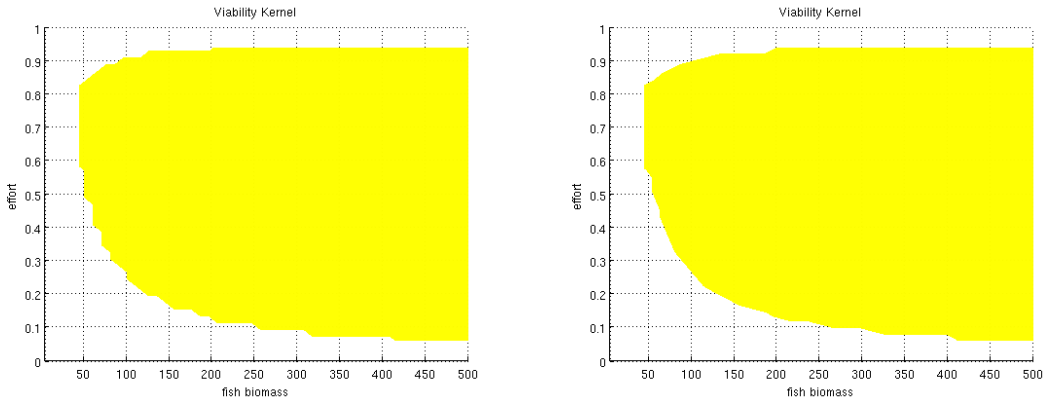
To achieve the same thing from the command-line, you can use the vk_kernel_view command. Like clicking "Plot Kernel," this command takes the plotting settings in your project, and uses them to create a figure. For instance, assuming that the file, `Fisheries Example.mat` contains the project we created in Example box D, the following should display the same area as in Figure 10b:

```
cd Projects

% Load the project into 'p'.
p = vk_project_load('Fisheries Example.mat');

% Run the kernel algorithm, storing the result back into 'p' -- only necessary
% if the project does not already contain a kernel.
p = vk_kernel_run(p);
```

```
% View the result.
vk_kernel_view(p);
```

3.10.2. *Plotting methods.* VIKAASA can use a number of different methods for displaying kernel visualisations, each of which comes with its own advantages and disadvantages. The two most commonly used plotting methods, which are fully supported supported under both MATLAB® and Octave are the qhull method, which draws a convex hull around the points, and the scatter method, which draws a scatter plot of the points.[32] This is the functionality that we refer to when we speak of "interpolating" any discretised set of viable points into smooth area. You should only use the convex hull plotting method if it makes sense to, however – if the set of viable points do not resemble a convex set (if there are any "holes," or concave edges), then drawing a convex hull around the points will make the viable area look bigger than it actually is. In such situations, you should use the scatter plot method instead.

Two additional plotting methods exist, both of which make use of the isosurface function in MATLAB® or Octave.[33] Unlike the convex hull method, isosurface does not assume that the area it is drawing a surface around is convex. Instead, the surface will appear to be quite "jagged." See Figure 12a as an example, which is the same set of points as in Figure 11b, but using the isosurface method, instead of the convex hull method. MATLAB® also offers the ability to smooth areas produced by isosurface, making them less jagged. See for instance Figure 12b.



(A) The fisheries viability kernel with isosurface

(B) The fisheries viability kernel with a smoothed isosurface

FIGURE 12. Examples using isosurface for plotting.

In the GUI, the available options are in the "Plotting Method" drop-down in the "Kernel Plotting" panel (see Figure 13). From the command-line you can set the plotting method by changing the plottingmethod field of your project structure. For instance:

[32]The convex hull functionality in both platforms is provided by the QHull library. See http://www.qhull.org/.

[33]This functionality is somewhat more sophisticated in MATLAB® than it is in Octave. The example plots given in this manual for the isosurface method are produced using MATLAB®.

```
p.plottingmethod = 'scatter';
```

In either case, the plotting methods available are the same:

- qhull, the convex hull method;
- isosurface;
- isosurface−smooth, the same as the isosurface method, but with the jagged edges smoothed (only works in MATLAB®);
- scatter, the scatter-plot method, using circles;
- scatter−x, the scatter-plot method, using crosses; and
- scatter−+, the scatter-plot method, using pluses.

3.10.3. *Slices.* Visualisation is only possible in either two or three dimensions. Where a viability kernel or domain, V has more dimensions than this (or where you want to view a two-dimensional cross-section of a three-dimensional area), VIKAASA is able to "slice" V along one or more of its axes, creating a two- or three-dimensional representation of part of the volume.

Each slice involves a single axis of values, $s \in \{1, 2, \ldots, n\}$, and can be of two types. Firstly, it is possible to slice "for all" values of x_s, which effectively means ignoring the variable x_s, and reducing V to a $(n-1)$-dimensional kernel, W by taking all the viable points in V, and removing the s^{th} element. The resulting kernel will no longer have any information about x_s in it, which can be useful for viewing the extent of a kernel with high dimensionality.

The second type of slice involves slicing for some the variable, x_s, "at" some particular value, $v \in [\underline{x}_s, \overline{x}_s]$. In this case, VIKAASA constructs W by finding only those points in V whose value of x_s is close to v, and then removing x_s from them, so that W is again $(n-1)$-dimensional. A value of x_s is considered to be close to v when it's value is less than half the distance between points in K_δ (in the s^{th} dimension) away from v. This means that you don't need to select v explicitly with respect to δ_s in order to make a slice. However, it does mean that for coarser discretisations, W has the potential to be somewhat misleading, as elements in W may not in fact be viable at $x_s = v$, but only at some value close to v. This problem diminishes with higher discretisations however, and you can make sure it does not affect you by selecting v to be exactly at a discretisation point.

Slicing for more than one value is achieved by iteratively applying the two algorithms above to the kernel. Thus, you can mix the two slicing methods together depending on your needs.

In the GUI, slicing is determined using the "Slices" table in the "Kernel Plotting" panel (see Figure 13). To mark a dimension for slicing, place a check in either the "All" or "At value" column, by the variable in question. If you select "At value," then you should enter a value (i.e., some v, as described above) in the field to the right.

From the command-line, you can accomplish the same thing by manipulating the slices field of your project structure. This field is a $q \times 3$ matrix, where q is the number of slices to be performed. The first column of slices gives the number of the variable. For instance,

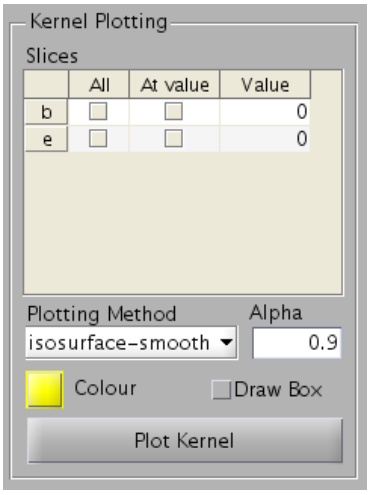


FIGURE 13. The "Kernel Plotting" panel

a value of 4 in the first column means that you want to slice through the fourth variable in the state space. The second column gives the value v to slice at. If you want to slice through all values, then you should put NaN (not a number) into this field. The third column gives the distances between points in K_δ in this dimension, and is only important if the second column did not have NaN entered. The distances can easily be calculated using vk_kernel_distances, as in the following example involving a four-variable problem:

```
octave:5> vk_kernel_distances(p.K, p.discretisation)
ans =

   0.0050000 0.0012500 0.0043750 0.0125000
```

The return from vk_kernel_distances is a row vector of length n, each element of which gives the distance between points in K_δ, along the n^{th} axis. In the case of the above values, suppose we wanted to slice through the third and fourth variables, using $v = 0$ for the third, and slicing through all values of the fourth. To do this, we could enter the following:

```
% Specify two slices.
p.slices = [ ...
  3, 0, 0.0043750; ...
  4, NaN, NaN];

% Plot the resulting kernel.
vk_kernel_view(p);
```

The order that the slices are given in does no matter, so we could also just as well enter the slices in reverse, as in:

```
p.slices = [ ...
  4, NaN, NaN; ...
```

```
3, 0, 0.0043750];
```

Example Box G. Extending the fisheries model to include capital

In order to demonstrate some of VIKAASA's more complex visualisation features, including the slice functionality just mentioned, we need a model with more dimensions than the one we developed in Example box C. For this reason, we will extend that model by adding an additional variable to the system: capital (or fleet size), k.

We will suppose that additional capital increases the size of the catch, but at a decreasing rate (i.e., there are *decreasing returns to scale*), and that the variable cost is proportional to both capital and effort. Also, we will suppose that effort continues to affect the catch as before. Thus the differential inclusion for fish biomass (previously given in Equation 5) becomes:

$$\dot{b}(t) = rb(t)\left(1 - \frac{b(t)}{l}\right) - qe(t)b(t)k(t)^{\frac{1}{\gamma}}, \tag{24}$$

where $\gamma > 1$ gives the extent of the decreasing returns. Similarly, the new equation for profit (previously given in Equation 2) is given by:

$$R(b(t), e(t), k(t)) = pqe(t)b(t)k(t)^{\frac{1}{\gamma}} - ce(t)k(t) - C. \tag{25}$$

This is the same as in Equation 2, if we suppose that $k(t)$ had an implicit value of one there. Now though, as $k(t)$ increases, revenue increases on account of the increased catch size; and variable costs also increase, proportional to the size of $k(t)$. As before, we will suppose that the moment fishing becomes unprofitable, the fleet will be dismantled. Thus, as before we require:

$$R(b(t), e(t), k(t)) \geqslant 0. \tag{26}$$

Although it may be clear from the above profit equation that there will be an optimal level of capital for any given combination of effort and fish biomass, we will suppose that investment in capital is constrained, so that the optimal level is not necessarily attained in all circumstances. Specifically, suppose that the investment level at any given time, $\iota(t)$ is exactly equal to the level of profit at that time: [34]

$$\iota(t) = R(b(t), e(t), k(t)). \tag{27}$$

Now, supposing that each unit of capital costs some price, s, and that capital depreciates at a rate of ρ, the differential inclusion for capital is:

$$\dot{k}(t) \in \left\{ \frac{R(b(t), e(t), k(t))}{s} - \rho k(t) \right\}_{u(t) \in U}. \tag{28}$$

Substituting in Equation 25 and noting that as in Equation 4, $u(t)$ does not appear in the differential inclusion, we can simplify it to the following differential *equation*:

$$\dot{k}(t) = \frac{pq}{s} e(t)b(t)k(t)^{\frac{1}{\gamma}} - \frac{c}{s} e(t)k(t) - \frac{C}{s} - \rho k(t). \tag{29}$$

The equation for $\dot{e}(t)$ will remain as in Equation 6.

As we did in Example box C, we now need to make the model concrete by specifying values for all of the model parameters; and we need to specify a rectangular constraint set, as well as a custom constraint set function (CCSF) for our profit constraint. We will re-use the parameter values from Example box C, give values for the three new parameters we have introduced as follows:

- The degree of diminishing returns to capital, $\gamma = 4$.
- The price per unit of capital, $s = 100$.
- The rate of depreciation on capital, $\rho = \frac{1}{10}$.

This gives us equations as follow:

$$\dot{b}(t) = \frac{2}{5}b(t)\left(1 - \frac{b(t)}{500}\right) - \frac{1}{2}e(t)b(t)k(t)^{\frac{1}{4}}, \tag{30}$$

$$\dot{e}(t) \in U = \left[-\frac{1}{100}, \frac{1}{100}\right] \tag{31}$$

and

$$\dot{k}(t) = \frac{1}{25}e(t)b(t)k(t)^{\frac{1}{4}} - \frac{1}{10}e(t) - \frac{1}{100}C - \frac{1}{10}k(t). \tag{32}$$

We will set a lower limit on $k(t)$ of one, which we justify by claiming that operation is not possible with less than one boat. The upper limit for $k(t)$ is more difficult to determine, however. We can discover it though, by observing that as $k(t)$ gets larger, two things occur: firstly, the size of the catch increases (albeit at a diminishing rate); secondly, beyond an "optimal" level of capital, profitability falls. Thus, for any given combination of $b(t)$ and $e(t)$, the maximum possible $k(t)$ is given by the lesser of either the level of capital for which all fish become extinct immediately (determined by $\dot{b}(t) = -b(t)$), and the level of capital for which profits become negative (assuming that that level is greater than the optimal level of capital). This gives rise to an upper bound on capital depicted in Figure 14. This upper bound is clearly not rectangular; in fact it is not even convex – the corner the upper bound where $e(t)$ and $b(t)$ are large is clearly concave. As before then, we will need to use a custom constraint set function (CCSF) to specify this upper bound on capital. From the diagram (and by sampling the function which produced the diagram), we can see that the absolute maximum value that $k(t)$ could have under any circumstances, without violating the constraints is somewhere around 1,130 units. We therefore use this value as the upper bound for our rectangular constraint set.

The rectangular constraint set for the problem thus becomes $K = [5, 500] \times [0, 1] \times [1, 1130]$. The CCSF for the problem will be similar to that in Equation 10, but using Equation 25. We could also include the requirement that $\dot{b}(t) > -b(t)$, but because we are using a step-size of 1 for approximating the kernel, this is not necessary, as the system will crash in one step under such circumstances anyway.

$$CCSF(b, e, k) = (4ebk^{\frac{1}{4}} \geqslant 10ek + 100) \tag{33}$$

We leave it to the reader to set up a project in VIKAASA for this problem. In Figure 15, we present a set of figures for the viability *niche* for this problem (i.e., produced using $u^*(x) = 0$), using a discretisation of 51 in each dimension, and using two different plotting methods. Note that because the resulting area has a concave edge, using the qhull plotting method as in Figure 15b, produces misleading results. As in Example box E, we should expect that we can improve on this niche by choosing a better control algorithm – an exercise which is left to the reader to attempt.

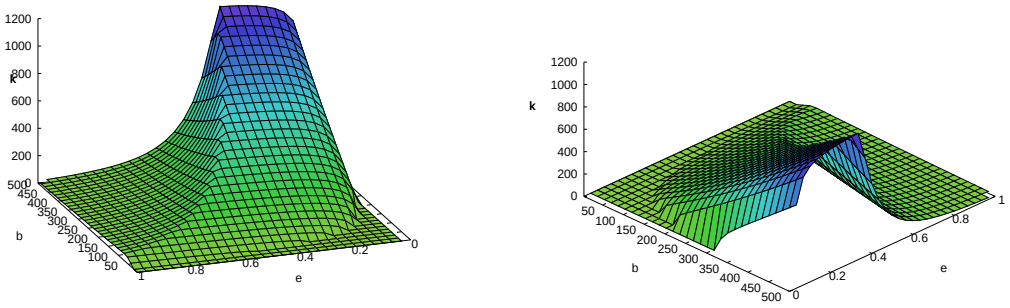


FIGURE 14. The upper bound on capital in the fisheries model, shown from two different angles.

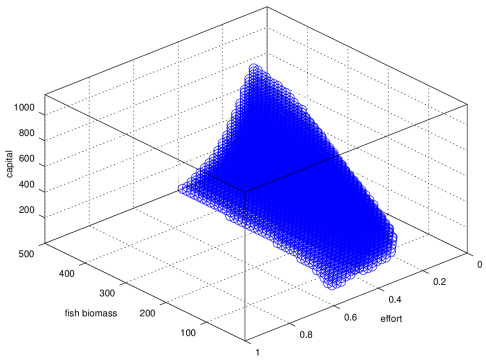
EXAMPLE BOX H. SLICING THE EXTENDED FISHERIES MODEL

In Example box G we found a three-dimensional *viability niche* for our augmented fisheries problem. In this box we show how we can use the slicing functionality in VIKAASA to get some other views of this area.

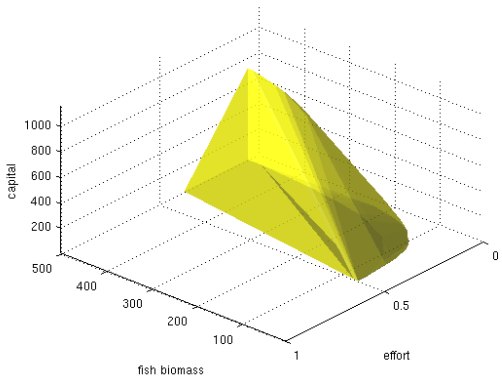
Firstly, we may be interested in seeing what combinations of biomass and effort are viable for some particular level of capital. This will provide us with some two-dimensional views of our viability niche similar to those we generated for our two-dimensional problem in Figure 10b, above. In particular, we can consider the two-dimensional problem to be a simplified case of the three-dimensional problem, with no capital dynamics (i.e., $\dot{k}(t) = 0$) and the level of capital fixed at $k(t) = 1$. Thus, slicing through the k axis can give us some idea of how changes to both the level of capital and to the dynamics of capital have affected the viability of our system.

in VIKAASA by taking a slice through capital at the values that we are interested in, and plotting the resulting two-dimensional slice. In the GUI, this would be done by checking the "at value" option next to "k" in the "Slices" table in the "Kernel Plotting" panel of the main window, entering some value (e.g., "100" for $k(t) = 100$) into the space to the right of the check-box, and then clicking the "Plot" button to view the result. The same thing can be accomplished from the command-line with the following, for some project p:

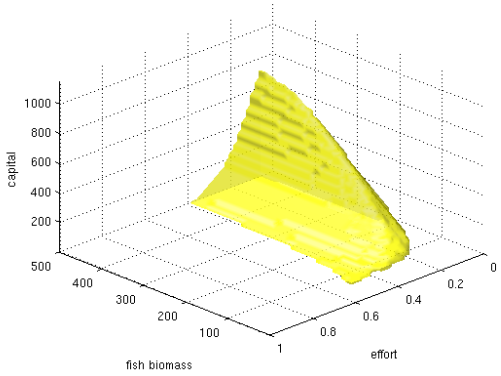
```
% Work out the distance between points in the kernel, according to the
% constraint set and the discretisation.
distances = vk_kernel_distances(p.K, p.discretisation);

% Specify a slice through the third dimension of the state space, at a value of
% 100 +/- half the distance between points in that dimension.
p.slices = [3, 100, distances(3)];

% We use 'isosurface' here, due to concerns that the space might be concave.
p.plottingmethod = 'isosurface';
vk_kernel_view(p);
```



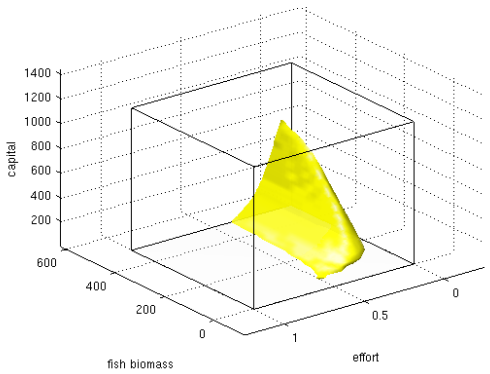
(A) Using the scatter plotting method



(B) Using the qhull plotting method



(C) Using the isosurface plotting method



(D) Using the isosurface−smooth plotting method, with a box representing the rectangular constraint set

FIGURE 15. Various views of the viability niche for the fisheries problem with capital.

Figure 16 gives some slices of our viability niche for various different values of k, and using a variety of plotting methods.[35] Firstly, looking at the slice through $k(t) = 1$ in Figure 16a, we can see that compared to Figure 10b that there appear to be fewer viable points, in particular for combinations of higher effort and higher fish stocks. As we noted, the only difference in the system when $k(t) = 1$ is the change to the capital dynamics, so we can conjecture that the reason that certain combinations of effort and biomass are no longer viable is they cause capital to grow to some non-viable level by generating large profits.

In Figure 17 we have superimposed the slice for $k(t) = 1$ over the two-dimensional viability niche, using the scatter and isosurface methods. These plots make it clear that in addition to higher levels of effort no longer being viable, there is also a small area at the lower bound that is no longer viable either. As this necessarily is a result of the introduction of capital dynamics, we can surmise that this must be because when effort and fish stocks become too low, profits are no sufficient to cover depreciation costs, causing $k(t)$ to fall below the lower bound of 1.

We can also see from Figure 16a that the viability nice is clearly concave along its upper edge when $k(t) = 1$. This, we can suppose arises from the fact that the constraint set is concave, as we showed in Figure 14. As the subsequent plots show, this concavity diminishes as capital increases. Practically, this means that we cannot use the qhull plotting method for small values of $k(t)$, but we can use it for larger values, once the area has become convex.

Overall, we can see that the higher capital is, the smaller the viability kernel slice becomes. This indicates that at least without exerting any control (i.e., for $u(t) = 0$ – which is what makes this a viability *niche*, rather than a kernel), higher levels of capital impose additional constraints on the viability of the system. This is roughly what we would expect, given the constraints we imposed on capital in Example box G.

Finally, we may be interested in viewing the viability niche from a different perspective, for instance, the set of viable catch sizes relative to the size of the fish biomass. For this purpose, we can define an "additional" variable, as described in Section 3.6. In order to do so, we need an equation for the catch size in terms of the other variables in the system. We will use $\chi(t)$ to represent the catch, so our equation is:

$$\chi(t) = qe(t)b(t)k(t)^{\frac{1}{\gamma}}. \tag{34}$$

Substituting in our values for q and γ, we get:

$$\chi(t) = \frac{1}{2}e(t)b(t)k(t)^{\frac{1}{4}}. \tag{35}$$

Now, we need to enter this equation as an additional variable. In the GUI, this can be done by clicking the "Add Variable" button above the "Additional Variables" table. We will enter chi for the variable name, "catch size" for the label, and 0.5*e*b^0.25 as the equation. If we leave the "Ignore" option unchecked, then you should immediately see chi listed in the "Slices" table of the "Kernel Plotting" window. From the command-line, assuming your project is called p, the following should achieve the same thing:

```
% Increment the number of additional variables:
p.numaddnvars = p.numaddnvars + 1;

% Update all the variables to reflect this:
p = vk_project_sanitise(p);

% Define the catch-size variable.
p.addnsymbols{end} = 'chi';
p.addnlabels{end} = 'catch size';
p.addneqns{end} = '0.5*e*b^0.25';
```

In order to create our visualisation of viable combinations of fish biomass to catch size, we need to create slices through all variables except for b and chi. Assuming that we are interested in *all* possible combinations of these two variables, we should slice through all values of e and k. In the GUI, we do this by checking the "All" column next to each of the two variables. From the command-line, assuming that e and k are the second and third variables, we set the slices field of the project as follows:

```
p.slices = [2, NaN, NaN; 3, NaN, NaN];
```

The resulting plots are displayed in Figure 18.

3.10.4. *Plot colour.* As you may have noticed, VIKAASA allows you to colour plots according to your needs. From the GUI, this is done simply by clicking on the button labelled "Plot Colour", and selecting a colour from the pop-up window that appears. The button should display the currently selected colour. From the command-line, the colour is set by altering the plotcolour field of one's project, as in the following example:

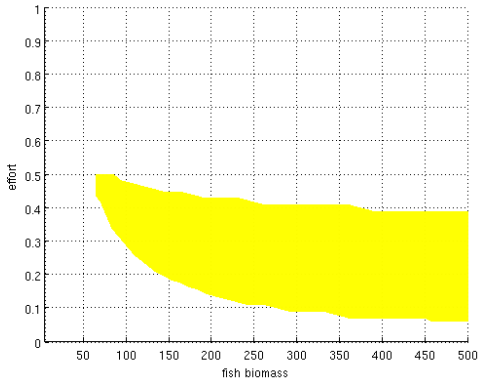
```
% Set the plot colour to yellow
p.plotcolour = [1 1 0];

% Plot the kernel in this colour.
vk_kernel_view(p);
```

The colour can be given as a triple, representing levels of red, green and blue respectively, or as one of a select number of strings, such as `'k'` for black. See the Octave or MATLAB® manuals for more information on plotting colours.

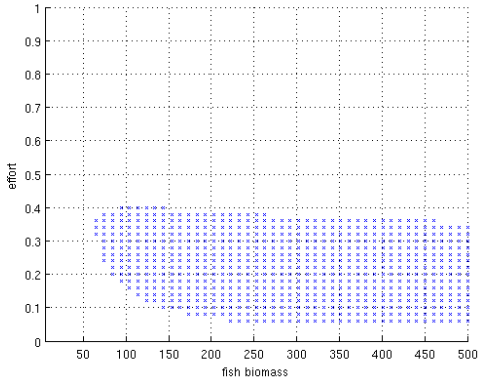
3.10.5. *Boxing plots.* VIKAASA can draw a rectangular box around two- or three-dimensional viability kernels, representing the rectangular constraint set K of the viability problem. Similarly, it can draw a box around any kernel slice as well, representing the relevant dimensions of K. If you are using the VIKAASA GUI, you can turn this feature on by checking the "Draw Box" option in the "Kernel Plotting" panel of the main window. Once this option is checked, subsequent plots (made by clicking "Plot Kernel") will be boxed. See Figure 15d and Figure 16f for examples. To accomplish the same functionality from the command-line, set the drawbox field in your project structure:[36]

```
% Load some project into p:
p = vk_project_load('project.mat');

% Enable box drawing:
p.drawbox = 1;

% Plot kernel (or slice) with box:
vk_kernel_view(p);

% Disable box drawing:
p.drawbox = 0;
```

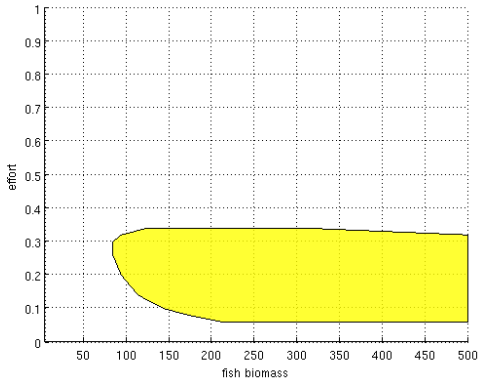
[36]Note that in Octave, plots are always drawn with a box around them where the axis limits are. The drawbox option causes the axis limits to extended, so that it will appear as if two boxes have been drawn. Also note that the drawbox option does not appear to work in Octave when using Gnuplot for plotting.



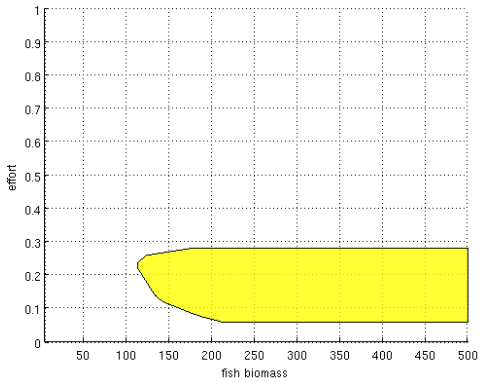
(A) Slice through $k(t) = 1$ using the isosurface−smooth method



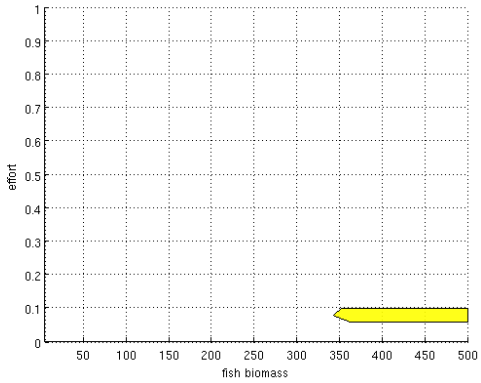
(B) Slice through $k(t) = 30$ using the scatter method



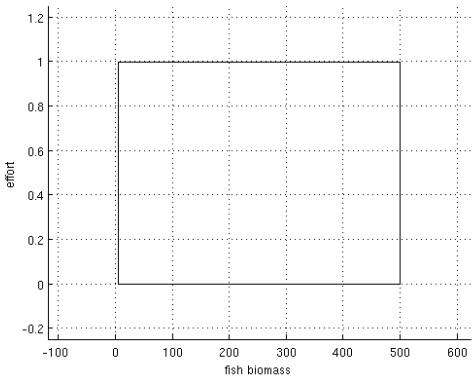
(C) Slice through $k(t) = 50$ using the qhull method



(D) Slice through $k(t) = 100$ using the qhull method



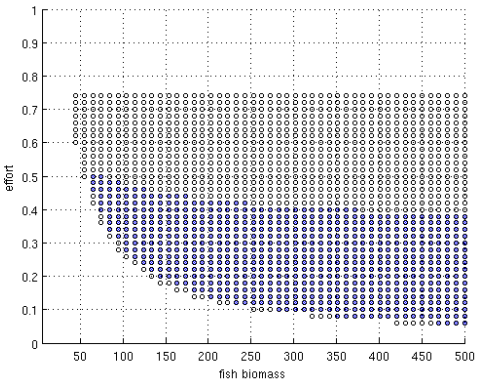
(E) Slice through $k(t) = 500$ using the qhull method



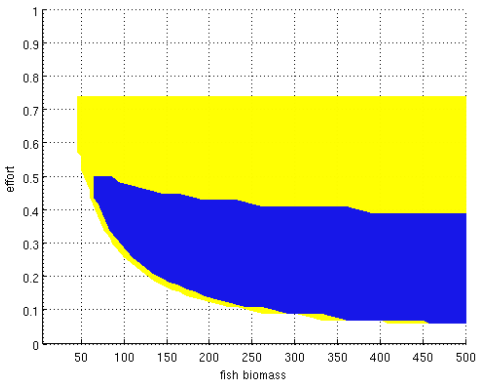
(F) Slice through $k(t) = 1000$ (no viable points), with a box representing the rectangular constraint set

FIGURE 16. Slices of the viability niche for the fisheries problem with capital.

3.10.6. *Superimposing figures.* Both MATLAB® and Octave offer the ability to "hold" a figure, so that multiple plots can be placed into the same figure for comparison. This is

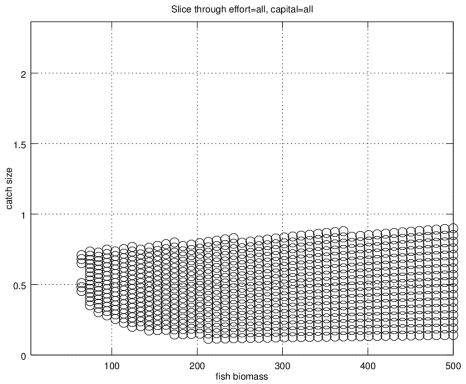


(A) Using the scatter method (circles for the two-dimensional niche; crosses for the three-dimensional slice).

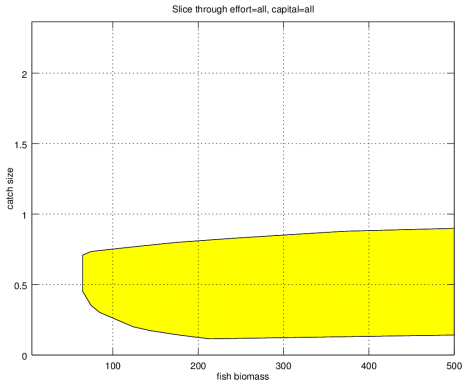


(B) Using the isosurface method (two-dimensional niche in yellow; three-dimensional slice in blue).

FIGURE 17. Viability niche for the three-dimensional problem at $k(t) = 1$ superimposed over the viability niche for the two-dimensional problem.



(A) Using the scatter method



(B) Using the qhull method

FIGURE 18. Viable combinations of catch size and biomass for the viability niche with capital.

useful for instance in Figure 17, where we use it to show how the two-dimensional and three-dimensional versions of the fisheries models give different sets of viable points, by first plotting the viability niche for the two-dimensional problem, and then plotting a slice from the viability niche for the three-dimensional problem over it. VIKAASA offers some functionality to tie into these features.

In the VIKAASA GUI, checking the "Hold Figures" option in the will mean that the next time a plot is produced, it will be drawn into the current figure, where "current" means the last figure to have focus (i.e., a mouse click).[37]

[37]Note that this does not necessarily correspond to the value of calling gcf. This is because the VIKAASA GUI itself counts as a figure in MATLAB®, and so gcf will usually point to the main GUI window instead of a figure. For this reason the VIKAASA GUI employs its own method to keep track of which

The current figure is remembered for the duration that VIKAASA is running, so it is possible for instance using the GUI to load a project into VIKAASA, plot something using that project, and then load a second project, and plot something else into the same figure. This is how Figure 17 was produced for instance. first a project file containing the viability niche for the two-dimensional fisheries problem (see Example box E) was loaded into the GUI, a plotting method and colour was selected, and a new figure was created by clicking "Plot Kernel" (i.e., the "Hold Figures" option was *not* checked to make this first figure). Then, without closing the resulting figure, a second project file was loaded into the GUI, "Hold Figures" was selected, a different plotting colour was chosen, and a slice through k was specified. When "Plot Kernel" was clicked, the slice was added to the figure containing the niche from the two-dimensional problem.

From the command-line, the holdfig option has no effect, but you can use vk_kernel_view with an additional "handle" parameter instead, as in the following example:

```
% Load two projects:
p1 = vk_project_load('project1.mat');
p2 = vk_project_load('project2.mat');

% ... Perhaps change the plotting colours, or something here ...

% Plot the kernel from p1, and get a handle to the figure back:
fig = vk_kernel_view(p1);

% Plot p2 into the same figure by specifying the figure as a second parameter:
vk_kernel_view(p2, fig);
```

This feature does not offer any additional checks, so it is quite easy to use it to produce non-sensical plots, for instance by plotting two slices along different axes in the same area.

3.11. **Plotting additional variables.** As described in Section 3.6, VIKAASA offers the ability to define variables which are not needed for kernel computation, but which may be useful for visualisation as "additional" variables. When an additional variable is defined, it will become available as an axis for plotting in VIKAASA, and as such will appear in the "Slices" table in the "Kernel Plotting" panel of the GUI. When a viability kernel is plotted (either using the GUI, or by making a call to vk_kernel_view), any additional variable that is not ignored (i.e., by checking the "Ignore" column in the "Additional Variables" table, or by setting the value of the corresponding row of the addnignore field to 1) will then be treated just like any other variable. This means that you can for instance create a three-dimensional plot by adding an additional variable to a two-dimensional viability problem. It also means though, that you need to either ignore or slice through additional values in some cases, where the number of dimensions is too high.

Example box H gives an example of using the additional variable functionality in VIKAASA to create a plot. Note that slicing through particular values of additional variables

figure is "current". Notably this includes tracking the current time profile window (see Section 4.4.2) separately from the current state-space plot.

does not work well, due to these variables not being evenly distributed according to a discretisation, as the other variables are.

4. Using VIKAASA to simulate dynamic dynamic trajectories

Because VIKAASA depends so extensively on one's choice of control algorithms, and on the choice of an appropriate stopping tolerance for choosing velocities which are "close enough" to steady, it is often highly useful to be able to see how a particular control algorithm works in practice, and how it is that VIKAASA has determined that a particular state-space point is viable or otherwise. For this reason, VIKAASA includes functionality to facilitate "simulations" of a system's dynamic evolution over time, as described in this section.

4.1. **Components of a simulation.** VIKAASA includes the information pertaining to a simulation in a structure similar to that used to store projects (see Section 3.4). If you are using the VIKAASA GUI, then the most recently run simulation is kept with the project file, so that saving and loading the project also includes the the most recently run simulation. If you are using the command-line, you can accomplish the same thing in a manner that is compatible with the GUI by storing simulation structures into the sim_state field of your project (examples given below).

A simulation in VIKAASA contains the following information:

(1) the *start state* of the system: a vector of length n giving the state-space position of the system at the beginning of the simulation (i.e., when $t = 0$);
(2) a vector of time values T, giving the values of $t \in \Theta$ at which the value of the state of the system has been sampled;
(3) the "velocity" (i.e., the Euclidean norm of $[\dot{x}_1(t), \dot{x}_2(t), \ldots, \dot{x}_n(t)]$) of the system at each time value, $t \in T$;
(4) the value of the control choice, $u(t)$ for each time value, $t \in T$;
(5) the state-space position $x(t)$ of the system at each time value, $t \in T$; and
(6) whether VIKAASA considers this state-space point to be inside the constraint set (including consideration of the CCSF, if applicable) or not.

Additional information is stored in the simulation if the project has a viability kernel associated with it. See Section 4.5.

4.2. **Creating a simulation.** To create a simulation in the MATLAB® GUI, enter the initial values of the system's state at $t = 0$ into the "Start" column of the table in the "Simulation" panel (see Figure 19), enter a time horizon, $t_N \in \mathbb{Z}_+$, giving the point in time at which simulation will cease, and choose a control algorithm for the simulation from the "Control Algorithm" drop-down. The list of control algorithms will be the same as in the "Control" panel for kernel approximation (see Section 3.8.5), except that there will be additional "kernel-aware" algorithms available as well, as described below in Section 4.5. The following gives an example of how the same options would be set from the command-line given some project, p with two dynamic variables:

```
% Specify the initial state of the system as a column vector:
p.start = [0.5; 1];
```

```
% Choose a time horizon (called "iterations" for historical reasons only):
p.sim_iterations = 10;

% Use the same control algorithm that was used for kernel approximation:
p.sim_controlalg = p.controlalg;
```

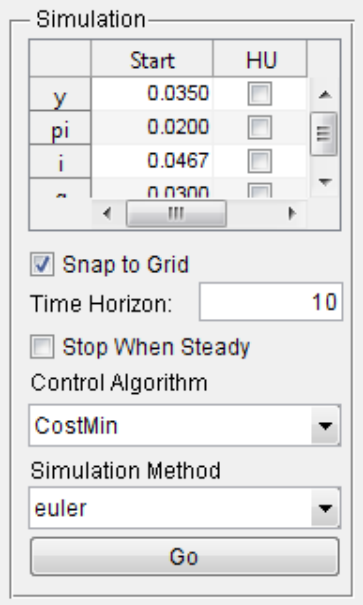


FIGURE 19. The "Simulation" panel

With the exception of the choice of control algorithm, which is made as described above, VIKAASA uses the same settings for simulation that it does for kernel approximation. For instance, the choice of whether to bound the control choice at the constraint set edge (see Section 3.7.5), the number of forward-looking steps to be employed by the CostMin algorithm (see Section 3.8.2), or the step-size, h (see Section 2.5.3) will be treated in the same manner for both kernel approximation and simulation.

To run a simulation from the GUI, click the "Go" button in the "Simulation" panel. You should see a progress bar similar to the one shown in Figure 8a. From the command-line, for some project p, enter:

```
% Run the simulation:
simulation = vk_sim_make(p);

% Store the result in the project (useful if you want to share the simulation
% with someone using the GUI):
p.sim_state = simulation;
```

The resulting simulation structure will contain the information described above in Section 4.1. Parallel processing is not used for simulations.

Some information about the simulation is shown in the GUI in the "Simulation Results" panel (see Figure 20). This information is also available from the command-line, as in the following example:

```
octave:1> % Run the simulation, and store the result in the project:
octave:1> p.sim_state = vk_sim_make(p);
octave:2> % View the results:
octave:2> vk_sim_results(p)
ans =
{
  [1,1] = Computation Begun at
  [2,1] = Computation Time
  [3,1] = Number of points
  [4,1] = Lowest velocity
  [5,1] = Average velocity
  [1,2] = 2011−8−17 22:29:42
  [2,2] = 3.3 seconds
  [3,2] = 51
  [4,2] = 6.3311e−04
  [5,2] = 3.0556
}
```

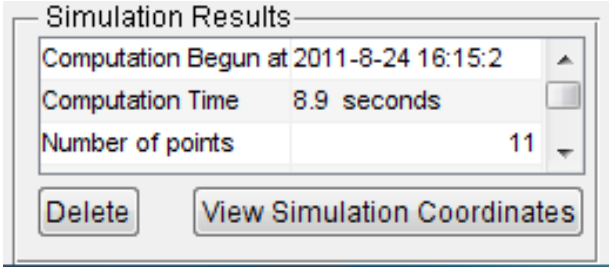


FIGURE 20. The "Simulation Results" panel

4.3. **Additional simulation options.** In addition to the options outlined in the previous section, there exist three further options, which affect the way that VIKAASA performs simulations. These are described in the following subsections.

4.3.1. *Snap to grid.* It is often the case that one is concerned with repeating the result arrived at by VIKAASA in determining whether a point is viable or not. For this reason it is often useful to begin a simulation from a point which corresponds to one of the discretised points in K_δ. To be sure that a starting state entered does indeed correspond to a starting point, you can use the "Snap to grid option", which will find the point in K_δ which is closest to the starting state you have entered. This option is located in the "Simulation" panel of the GUI, and can be changed from the command-line by setting the sim_use_nearest field.

4.3.2. *Simulation method.* There are two different methods available in VIKAASA for computing the dynamic evolution of the system: euler and ode. The euler simulation method makes use of Euler's method for solving differential equations, which is the same method used to approximate the viability kernel (see Section 1.3). Thus, it makes sense to use this method where one is concerned about replicating the results of the kernel approximation algorithm (e.g., to see how a particular state-space point considered to be in the kernel was determined to be viable).

The ode method uses either the ode45[38] in MATLAB®, or the lsode[39] method in Octave. These functions provide more sophisticated methods for solving differential equations, and are thus expected to produce "truer" trajectories than those obtained using Euler's method.

To change the simulation method using the GUI, select the appropriate option from the "Simulation Method" drop-down in the "Simulation" panel (see Figure 19). From the command-line, for some project, p, enter:

```
% Switch to the "ode" method:
p.sim_method = 'ode';

% Switch to the "euler" method:
p.sim_method = 'euler';
```

4.3.3. *Stop when steady.* Normally a simulation will continue until the time horizon is reached. However, if you are only interested in seeing the evolution of the system to it's near-steady state (as determined by VIKAASA according to the stopping tolerance, as described in Section 2.3), you can set the "stop when steady" option. This means that you can for instance specify a large time horizon, and only get back a vector of time values T, up to and including the point in time where the system attains it's near-steady state. Of course, if the system never attains a near-steady state, the simulation will continue until it reaches the time horizon.

4.3.4. *Hard constraints.* Normally VIKAASA does not halt a simulation when the constraints are violated, but only when the time horizon is arrived at. However, it may be the case that although it is possible according to the system's dynamics for a given constraint to be violated, that it does not make sense for this to happen in reality. In economics for instance, negative wages might be considered an example of this. Thus, if we had a system which described the evolution of wages over time, we would probably want to disallow negative wages from appearing in our calculations. Not only would it lack any interpretation, but it might cause other problems. If the square root of the wage rate was used anywhere in the system's dynamics for instance, then by Euler's method, the system's state would become complex when wages became negative – a further problem for meaningful interpretation.

When approximating the viability kernel, VIKAASA always stops when constraints are violated, so that provided the constraints are "sensible", the problems mentioned here

[38] See the MATLAB® manual for more information.

[39] See the Octave manual for more information.

will not appear. For simulations however, it is necessary to tell VIKAASA to consider certain bounds as "hard", meaning that simulation cannot proceed if they are violated. When a "hard" constraint is violated, the simulation results are truncated to the point where the violation occurred. For the euler simulation method, this can be detected while the simulation is proceeding, meaning that using hard constraints can save time, where you are not concerned about the system's dynamics outside of the constraint set. For the ode method however, this is not possible. Instead, the full simulation must be run, and any violation worked out retrospectively.

Hard constraints can be either "upper" or "lower". These apply to variables violating the upper or lower bounds of the rectangular constraint set, respectively. As such, they do not apply to any violations of any custom constraint set that may be specified for the project.

In the GUI, hard constraints are set by checking the box in the "HU" or "HL" columns (short for "hard upper" and "hard lower", respectively), next to the variable in question. From the command-line, the same thing is achieved by manipulating the sim_hardupper and sim_hardlower column arrays, as in the following example:

```
% Set hard upper constraints on variables 4 and 5:
p.sim_hardupper = [4; 5];

% Set hard lower constraints on variables 3 and 4:
p.sim_hardlower = [3; 4];
```

4.4. **Viewing a simulation trajectory.** Once you have completed a simulation, there are a number of ways to view the result using VIKAASA. The following subsections outline each of those options in turn.

4.4.1. *Viewing the raw data.* Firstly, one can access the raw coordinates representing the simulation path over time. To do this using the GUI, click the "View Simulation Coordinates" button in the "Simulation Results" panel (see Figure 20). This will display the variables as a sequence of columns using the MATLAB® variable editor.

From the command-line, you can access the path data directly by querying the simulation structure produced by vk_sim_make. See the entry for vk_sim_make in Appendix D for more information.

4.4.2. *Time profiles.* One of the easiest ways to view the results of a simulation is to plot each variable individually over time in its own plot. VIKAASA facilitates this by offering functionality which displays time profiles together in a figure, along with some optional additional information.

In the GUI, once you have created a simulation, you can create a time profile for it by clicking the "Time Profiles" button in the "Simulation Plotting" panel (see Figure 21). By default the time profiles will be displayed in two columns, as in Figure 22a, but you can change this via the "Time profile columns" field in the "Options" panel. From the command-line, you can use the vk_figure_timeprofiles_make function, as in the following example, for some project, p:

```
% Set−up a simulation and run it:
p.sim_start = [0; 0.5; 1];
p.sim_iterations = 10;
p.sim_state = vk_sim_make(p);

% Plot time profiles for the resulting simulation:
vk_figure_timeprofiles_make(p);
```

To simplify the following procedure, VIKAASA also offers the vk_sim_timeprofiles_from function, which creates a simulation from a given initial state, and then plots time profiles afterwards:

```
% Create a simulation using the information in p, but starting from a specified
% position. Save the resulting simulation back into p, and then display a time
% profile.
p = vk_sim_timeprofiles_from(p, [0; 0.5; 1]);
```

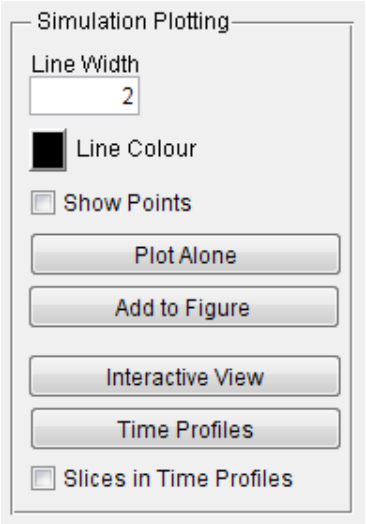


FIGURE 21. The "Simulation Plotting" panel

An additional option exists for displaying the extent of the viability kernel in time profiles. It works by slicing the viability kernel $n-1$ times for each time value $t \in T$ and for each variable $x_i(t)$ for $i \in \{1,2,\ldots,n\}$. The result is a series of lines, representing the extent of the kernel for any given $x_i(t)$. This can be used to see in which dimensions the trajectory can be considered "inside" or "outside" of the kernel. This option is turned on the GUI by checking the "Slices in Time Profiles" option in the "Simulation Plotting" panel. From the command-line, for some project, p, you can enter:

```
p.sim_showkernel = 1;
```

Time profiles for additional variables will also be displayed, provided the "Ignore" option for a given additional variable is not selected. Note however that the "Slices in Time

Profiles" option described does not work well with additional variables, for the same reason that slicing through additional variables does not work well, as described in Section 3.6. For this reason, if you want to use this feature and you have additional variables defined, it is recommended that you check the "ignore" option to disable them.

Red lines are drawn into each time profile, representing the upper- and lower-bounds of the rectangular constraint set in each dimension. For additional variables, upper and lower values are guessed by evaluating the additional variables' equations at the various upper- and lower-bounds.

Two additional plots are always included in any time profile. They are the system "velocity", represented by the Euclidean norm of the system's dynamics; and the value of the control choice, $u(t)$ at each time value $t \in T$. These two are always plotted last. For the control choice, red lines are drawn in, showing the upper- and lower-bounds of the control value; for the velocity, a lower red line is drawn, representing the velocity below which the system will be considered steady.

See Example box I and J for some examples of creating time profiles.

4.4.3. *Plotting simulations in two- or three-dimensional space.* In addition to time profiles, simulations can be plotted as trajectories moving through two- or three-dimensional space. In this case, information concerning time values is absent, but you are able to view the system's path through the state space (or a subset of it). Trajectory plotting uses VIKASA's slice functionality, so that a four-dimensional problem can be sliced along one of its axes in order to visualise the system's evolution as movement in the remaining three dimensions; or the problem could be sliced twice to visualise it as movement in two dimensions. Note though that slicing is in this case always done through "all" values, rather than the slice being at some particular value – that is, trajectories are sliced by "flattening" them into the requisite number of dimensions, meaning that information concerning the dimensions that have been sliced is missing.

From the GUI, you can view the evolution of a trajectory by clicking either the "Plot Alone" or the "Add to Figure" options. The former of these will create a new figure and plot the path of the simulation into it, whilst the latter will plot the simulation directly into the "current" figure (as defined in Section 3.10.6). To achieve the same thing from the command-line, for some project p, use the following:

```
% Create a simulation, and store it in the project:
p.sim_state = vk_sim_make(p);

% Plot the simulation alone (according to the project's slices):
vk_sim_view(p);

% Plot the kernel, and get a handle for the figure:
fig = vk_kernel_view(p);

% Plot the simulation into the kernel figure:
vk_kernel_view(p, fig);
```

Note that when using slices, plotting a simulation into the same figure as a kernel slice can be misleading, because viable points represented by the slice will only be relevant for some particular values. Thus it can appear that a trajectory is viable, according to the viability kernel, when a time profile view for instance would reveal that it is not.

4.4.4. *Additional simulation plotting options.* In addition to the functionality outlined above, you can also change the colour and width of the line that plots are drawn in. These options are given in the "Kernel Plotting" panel of the GUI. From the command-line, they are altered by setting the sim_line_width and sim_line_colour fields of the project.

Finally, there is an option to make VIKAASA draw coloured points along the trajectory, one for each time value, $t \in T$. These points are coloured as follows:

- If there is a kernel associated with the project, and the current position, $x(t)$ is considered to be "inside" that kernel (as described below in Section 4.5.1, then the a *green* point will be drawn.
- If instead, the point is considered to be on the "edge", then a *blue* point will be shown.
- If the point is inside the constraint set (and satisfies any CCSF), but considered "outside" the kernel, or if there is no kernel, then an *orange* point will be shown.
- If the point is outside of the constraint set in real space, then a *red* point will be shown.
- If the point is outside of the constraint set in imaginary space, then a *purple* point will be shown.

Additionally, whenever the system is deemed by VIKAASA to have reached a near-steady state, then if the "show points" option is enabled, green dots will be displayed in the "velocity" plot in time profiles.

This option can be enabled in the GUI by checking the "Show Points" option in the "Kernel Plotting" panel. From the command-line it can be set by altering the value of the sim_showpoints field of the project.

Example Box I. Creating and plotting simulations for the two-dimensional fisheries problem

In this box we will give some brief examples of simulation plotting in VIKAASA for the two fisheries problems we have been working with. We start with the two-dimensional problem. In what follows, we assume that you have a project with the two-dimensional kernel that we constructed in Example box F loaded either in the GUI or on the command-line.

Firstly, let's pick a point from the two-dimensional problem that is in the viability kernel (see Figure 11a), but not in the viability *niche* (see Figure 10b). We have chosen $b = 200$, $e = 0.8$ as one such point. If you are using the GUI, then enter this into the "Start" column in the "Simulation" panel. We will start with a time horizon of 20, and use the euler method for the simulation. First, we will run the simulation using the ZeroControl algorithm, and the FisheriesControl. Firstly then, select the ZeroControl algorithm from the drop-down box in the "Simulation" panel, then click "Go". From the command-line, enter the following:

```
% Set the start state, etc.
p.sim_start = [200; 0.8];
p.sim_iterations = 20;
p.sim_method = 'euler';

% Make two copies of the project, one with the ZeroControl set, and the other
% with the FisheriesControl:
pz = p;
pz.sim_controlalg = 'ZeroControl';
pf = p;
pf.sim_controlalg = 'FisheriesControl';

% Run both simulations, and store the result back into the respective projects:
pz.sim_state = vk_sim_make(pz);
pf.sim_state = vk_sim_make(pf);
```

Once your simulation (or simulations) has finished, let's view the time profiles. In the GUI, click the "Time Profiles" button in the "Simulation Plotting" panel. You should see something like in Figure 22a. From the command-line, enter:

```
vk_figure_timeprofiles_make(pf);
```

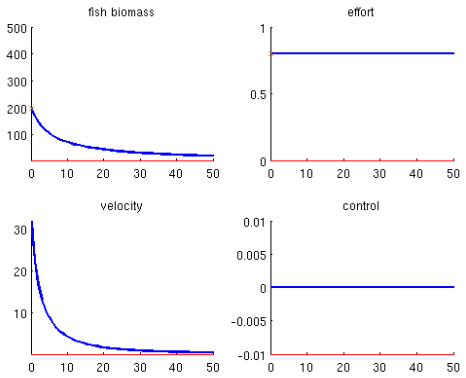
From this figure, it appears that nothing is in fact drastically wrong – although fish stocks are declining, they are doing so at a decreasing rate, and they have not fallen below the lower threshold. The falling velocity of the system indicates that things are slowly stabilising. However, if we plot the time profiles again, with "show points" enabled,[40] as in Figure 22a, we will see that around $t = 30$, the points become red, indicating that VIKAASA regards them as non-viable. Because this cannot be attributable to violation of the rectangular constraint set, it must be the case that the CCSF has been violated – that is, with such low fish stocks, and such high levels of effort, fishing has become unprofitable.

We can get another view on this now by superimposing the simulation into a plot of the viability niche. From the GUI, first plot the viability niche by clicking the "Plot Kernel" button the "Kernel Plotting" panel. Once the figure is present, click the "Add to Figure" button in the "Simulation Plotting" panel. From the command-line, enter the following:

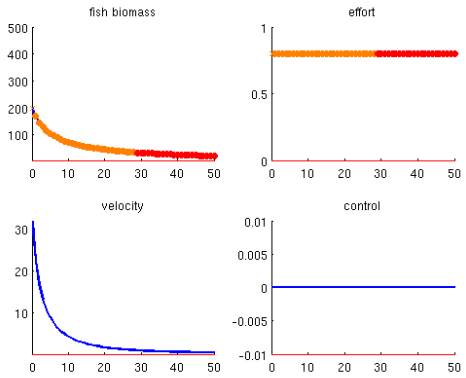
```
% Plot the kernel.
fig = vk_kernel_view(pz);

% Turn on points.
pz.sim_showpoints = 1;

% Plot the simulation into the same figure:
vk_sim_make(pz, fig);
```

You should see a result like that in Figure 22c. This can be seen as further evidence that the system has strayed into an unprofitable area – the system had moved so far to the left that there are no longer any viable points in the niche, for the given level of fish biomass.

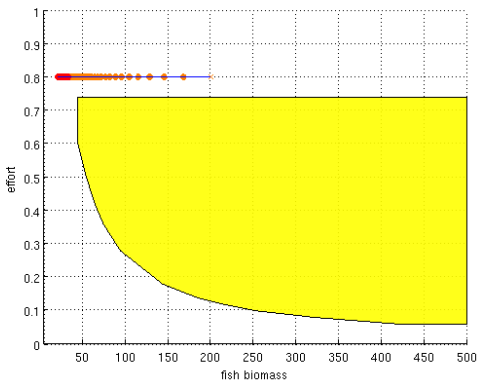
Now let's run the simulation again, this time using FisheriesControl. After you have run the simulation, plot the time profiles again, with the "Show points" option enabled. You should see a plot like the one in Figure 22d. The major difference here is that the control path starts of with $u(t) = -c$ until around $t = 10$, when it returns to zero. This has the effect of reducing effort by some amount, and according to the coloured dots, avoiding the unprofitable levels of fish depletion that we saw in Figure 22c. Note that the system has not become steady yet, as evidenced by the fact that the "velocity" line is still above the cut-off. One supposes that if we ran the simulation with a longer time-horizon, we would reach this steady velocity – an exercise left to the reader to check.



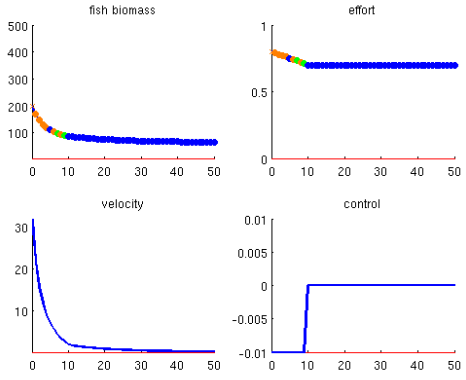
(A) Time profile for ZeroControl with "show points" disabled



(B) Time profile for ZeroControl with "show points" enabled



(C) Plot of ZeroControl with viability niche and "show points" enabled



(D) Time profile using FisheriesControl with "show points" enabled

FIGURE 22. Simulations from $x(0) = [200, 0.8]'$ in the two-dimensional fisheries model.

Example Box J. Creating and plotting simulations for the three-dimensional fisheries problem

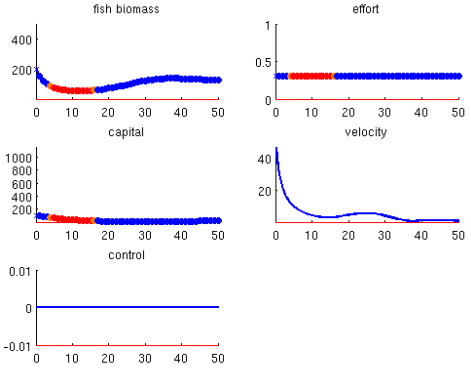
Let us move on to the viability niche for the three-dimensional problem now. In this problem, as we saw in Figure 17, there were fewer combinations of effort and fish biomass for which viable trajectories existed. Moreover, we saw in Figure 14 that for higher levels of effort, only very low levels of capital would suffice to stop the system from destroying all the fish stock effectively immediately. Let's start again with a point that is close to the viability nice, but outside, and see what happens. Our starting point will be $x(0) = [200, 0.3, 100]'$. Running a simulation with a time horizon of 50, using the Euler method, and ZeroControl should result in time profiles like those in Figure 23a. Again, the system appears to have crashed because fishing has become unprofitable as fish stocks dwindle. We can also plot the viability niche using the isosurface method, and then add the simulation trajectory to the plot, as we did in Example box I. You should get a three-dimensional plot like the one in Figure 23b. Although it's hard to see on paper, the system becomes unprofitable once fish fall below a certain level. We can see this somewhat clearer by slicing through all values of capital, and adding plotting the trajectory into the result plot again, as in Figure 23c. From the GUI, simply check the "All" column next to k in the "Slices" table of the "Kernel Plotting" panel. Then, plot the kernel again (into a new figure), and click "Add to Figure" in the "Simulation Plotting" panel to add in the simulation trajectory. From the command-line, you can enter the following (assuming that p contains the project for the three-dimensional problem):

```
% Slice through all values of k.
p.slices = [3, NaN, NaN];

% Plot the (sliced) figure.
fig = vk_kernel_view(p);

% Add the simulation.
vk_sim_view(p, fig);
```

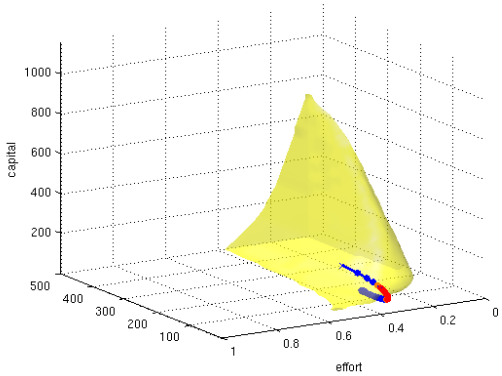
As in Figure 22b, the level of biomass falls until fishing is no longer profitable. What is different here though, is that we can see in Figure 23b that the system would hypothetically recover, if we weren't treating any encounter with unprofitable conditions to be the death of the system.

We don't have a specialised control algorithm for this case, but we can try using NormMin1Step, to see if purposefully slowing the system is enough to stop the crash we witnessed above from happening. Change the control algorithm and re-run the simulation. The results should look like those in Figure 24. We can see that they look more promising – although at $t = 50$, the velocity is still not slow enough for us to consider the system steady, we have steered into the inside of the viability niche (as indicated by the green dots). We might therefore try to run run the simulation for longer, to see if the system eventually stabilises. Also, it might be interesting to see a kernel approximation made using NormMin1Step.

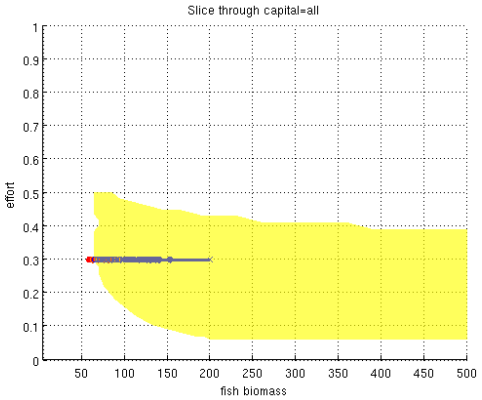
4.5. **Viability kernel information in simulations.** Where simulations are undertaken in the context of a viability kernel approximation S (i.e., where an approximate viability kernel has been calculated, and is stored in the current project), the simulation also records whether or the state-space point at each $t \in T$ is considered by VIKAASA to be



(A) Time profile with "show points"



(B) Three-dimensional plot with simulation



(C) Slice through capital with simulation

FIGURE 23. Simulations from $x(0) = [200, 0.3, 100]'$ using ZeroControl

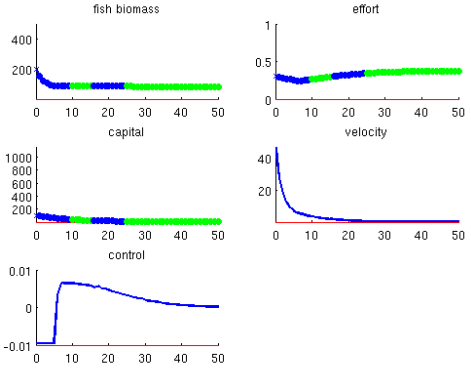
"inside" the continuous interpolation of the kernel, $V \supset S$, and if not whether it is on the "edge" of V or not. How VIKAASA determines these two states is explained below.

4.5.1. *Being "inside" the kernel, or on the "edge".* For some state-space point $x(t)$, being "inside" means that VIKAASA can find points in S that surround $x(t)$, according to:

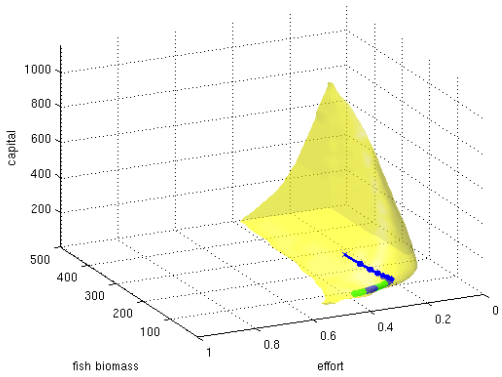
(1) the discetisation vector δ, that was used to determine the points in S, and
(2) a "layers" setting, $\lambda \in \mathbb{Z}_+$.

In order to determine if $x(t)$ is "inside", VIKAASA searches S for a λ-layered n-dimensional "rectangle" of points, such that in each dimension which surrounds $x(t)$. If such a rectangle can be found, the point is considered to be "inside". Otherwise, if some points in S can be found satisfying the criteria, but not enough to construct a complete rectangle, the point is considered to be on the "edge" of $V \supset S$.[41] The greater the number of layers, λ, the more difficult this requirement is to fulfil, and the fewer points will be considered

[41]This means that being "inside" V and being on the "edge" of V are mutually exclusive states.



(A) Time profile with "show points"



(B) Three-dimensional plot with simulation

FIGURE 24. Simulations from $x(0) = [200, 0.3, 100]'$ using NormMin1Step

inside. Thus, the layers setting can be seen as making the criterion for being "inside" more stringent.

4.5.2. *Control algorithms which use the kernel information.* VIKAASA allows you to make use of kernel information from within the context of a control algorithm for the purposes of simulation (but not kernel determination). The intended use of this is to allow one to text out kernel-aware policies or control rules – that is, to simulate the use of the viability kernel as information for decision-making.

Control algorithms which make use of this additional information are kept in the "VControlAlgs" folder in VIKAASA. Two examples of such algorithms come with VIKAASA – SatisficeCostMin and SatisficeMaxMin, both of which follow a similar policy of doing nothing so long as they are "inside" the kernel (in the sense described above), and then taking evasive action when the kernel edge is arrived at.

You can add you own by writing a function and placing it in the folder. You should consult the examples to see how the function signature of these functions differs from that of the standard control algorithms.

Appendices

A. Potential extensions to VIKAASA

A.1. **Increasing the number of controls.** VIKAASA is currently limited in that it can only work with a scalar control. A potentially very useful extension to VIKAASA then would be to extend its functionality to deal with a vector of controls. We have not had time to implement this ourselves, so this section briefly outlines how this could be done.

In order to be able to compute viability kernels, at a bare minimum, changes would need to by made to vk_viable, which is the function which determines the viability or otherwise of an individual state-space point (see Algorithm 1), and the functions that it depends on, such as vk_options.[42] These functions could then be run as described in Section 3.9.4; that is, you would not be able to make use of the GUI or the project-related functionality, but you would be able to compute kernel approximations.

Additionally, a control algorithm would need to be written which produced a vector, rather than a scalar. Such an algorithm could perhaps be written using the fmincon function from MATLAB®, or otherwise could be custom-built (after the manner described in Section 3.8.6) for the problem at hand.

A.2. **Improving or replacing the kernel approximation algorithm.** VIKAASA makes use of a very primitive method for approximating viability kernels, consisting of simply trying to slow the system in some amount of time. As mentioned in the manual, this algorithm clearly has a number of shortcomings. It's asymptotic behaviour is very bad, because each point can take up to 46,000 steps to complete; and it can incorrectly identify non-viable points as viable and vice versa. For this reason it is expected that the norm minimisation algorithm used by VIKAASA should be augmented (e.g., by adding additional passes, adding or removing points), or entirely replaced.

In both cases, the functions that need to be examined are vk_kernel_compute, which manages the collecting of viable points, as described in Algorithm 2, and vk_viable, described in Algorithm 1. These functions represent the core viability approximation algorithm used by VIKAASA.

B. The VIKAASA library structure

This appendix gives a brief overview of the structure used to organise the various functions employed by VIKAASA.

B.1. **Getting help on a particular function.** Each function in VIKAASA should have its purpose documented at the top of its `.m` file. From within MATLAB® or Octave, after you have initialised the environment (as described in Section 3.3), you should be able to get information for a given function by using the help command. For instance, help vk_kernel_view should give the help page for the vk_kernel_view function.

B.2. **Dependency graph of functions.** Each function has a "Requires" section in its header, which lists the other from within the VIKAASA library which it depends on to operate. The intention of this is that if you only want a specific part of VIKAASA, you can extract the required function, along with the functions that it depends on. A graph based on these dependencies is given in the "Docs" folder of VIKAASA (it is too large and unwieldy to display here). Each box in the graph represents a function, and an arrow going from one function to another indicates that the former depends on the latter.

[42] You can consult the "Requires:" line in the header of `vk_viable.m` to see what other functions it depends on. Also, a graph of dependencies is given in the "Docs" folder of VIKAASA.

B.3. **Directory layout of functions.** The library of functions in VIKAASA can be found in the "Libs" folder. Within that folder there are a number of sub-folders, each containing groups of functions that fulfil certain types of tasks. In addition, a number of functions which do not neatly fit into any of the other categories reside in the "Libs" folder proper. To find out where a particular function resides, you only need to look at its name. All VIKAASA functions start with `vk_`, followed by the name of the category that they belong to (e.g., `project` for functions that deal with projects), and then followed by a description (e.g., vk_kernel_view is for viewing kernels).

C. Control algorithms provided with VIKAASA

C.1. **CostMin.** Apply the control that minimises the specified cost function.

Synopsis. This function attempts to minimise the cost function at some number of steps in the future. The function can work with any arbitrary number of forward-looking steps, but becomes exponentially slower for each one.

Notes.

- The cost function is given in options.cost_fn.
- The number of forward-looking steps is given by options.steps.
- If options.use_controldefault is set to 1, then the algorithm will not bother finding an optimal control for the final step, but will instead apply 'options.controldefault'.

Usage.

```
% Standard use with required arguments:
u = CostMin(x, K, f, c)
```

See ControlAlgs for informaton on the required parameters, and the return value.

```
% With additional options structure passed in. 'options' is either a list
% of name,value pairs, or a structure created by vk_options.
u = CostMin(x, K, f, c, options)
```

Requires. vk_costmin_recursive, vk_options

See also. ControlAlgs, options

❁

C.2. **CostSumMin.** Find the control which minimises the sum of costs

Synopsis. This function finds the control that minimises the sum of cost function realisations for some set number of steps.

- The cost function is given in options.cost_fn.
- The number of forward-looking steps is given by options.steps.

- If options.use_controldefault is set to 1, then the algorithm will not bother finding an optimal control for the final step, but will instead apply 'options.controldefault'.

Usage.

```
% Standard use, with required parameters:
u = CostSumMin(x, K, f, c)
```

```
% With optional parameters. options is either a list of name,value
% pairs, or a structure created by vk_options.
u = CostSumMin(x, K, f, c, options)
```

Requires. vk_costsum_recursive, vk_options

See also. CostMin, NormMin1Step

❁

C.3. **ManualControl.** Manually choose control from the command line

Synopsis. This function allows the user to manually specify control at each position, and also allows the user to test different possible control paths.

Usage.

```
% Standard usage:
u = ManualControl(x, K, f, c)
```

```
% With an options structure attached:
u = ManualControl(x, K, f, c, options)
```

options is either a structure created by vk_options, or otherwise a series of 'name', value pairs.

Requires: vk_options, vk_viable_exited

❁

C.4. **MaximumControl.** Apply maximum control.

Synopsis. This control rule simply returns the maximum control, regardless of position, etc.

Usage.

```
% Standard usage
u = MaximumControl(x, K, f, c);
```

```
% With options
u = MaximumControl(x, K, f, c, options);
```

See also. MinimumControl, ZeroControl, vk_viable

❁

C.5. **MinimumControl.** Apply minimum control (i.e., $-c$)

Synopsis. This function returns the largest negative control available, regardless of size.

Usage.

```
% Standard usage:
u = MinimumControl(x, K, f, c);
```

```
% With additional options
u = MinimumControl(x, K, f, c, options);
```

See also. MaximumControl, ZeroControl, vk_viable

❁

C.6. **NormMin1Step.** Fast 1-step norm-minimising control function

Synopsis. This function returns the control that minimises the norm of the system velocity in one step.

It has the advantage that it is faster than using COSTMIN, but it is less flexible. Firstly, it cannot be used for more than one step. Secondly, it uses 'controldefault' to avoid the issue of having to optimise for the control-in-one-step. This is fast, but it may not make sense in a non-linear dynamic system.

Usage.

```
% Standard use with required arguments:
u = NormMin1Step(x, K, f, c)
```

```
% With options passed in. options is either a list of name, value pairs,
% or a structure created by vk_options.
u = NormMin1Step(x, K, f, c, options);
```

Requires. vk_options

See also. CostMin, CostSumMin

❁

C.7. **ZeroControl.** Apply control of zero

Synopsis. This function chooses a control of zero every time.

Usage.

```
% Standard usage:
u = ZeroControl(x, K, f, c)
```

See also. MaximumControl, MinimumControl

❁

C.8. **SatisficeCostMin.** Satisficing control algorithm that uses CostMin at the edge

Synopsis. This algorithm is similar to SatisficeMaxMin, except that it uses CostMin to determine what control to use when at the kernel edge.

Usage.

```
% Standard usage.
u = SatisficeCostMin(info, x, K, f, c);
```

```
% With options
u = SatisficeCostMin(info, x, K, f, c, options);
```

info is a structure, as described in SatisficeMaxMin.

Requires. vk_kernel_inside, vk_options

See also. vk_sim_simulate_euler, vk_sim_simulate_ode, vk_viable, SatisficeMaxMin

❁

C.9. **SatisficeMaxMin.** Satisficing Viabilty Control Algorithm

Synopsis. This control rule does nothing unless it finds itself are in an 'edge' scenario, in which case either the max or min control available is used – whichever is less costly, according to the cost function.

Usage.

```
% To get the statisficing control rule for point x:
u = SatisficeMaxMin(info, x, K, f, c);
```

info is a structure that must contain:

- V: A viability kernel
- distances: An array of distances FIXME
- layers: How many layers of points before the algorithm considers that it is in an 'edge' scenario.

Requires. vk_kernel_inside, vk_options

See also. vk_simulate_euler, vk_simulate_ode, vk_viable

❁

D. The VIKAASA library function reference

D.1. **vk_control_bound.** Bound a control choice to prevent the system from crashing, where possible.

Synopsis. This function takes a state-space point and a control choice, and checks to see if that control choice will cause the system to exit the constraint set in zero, one or two steps.

Zero steps means that the point is already outside the constraint set.

One step means that after applying u, the system violates the constraint set.

Two steps makes use of the controldefault option. This is only checked if the use_controldefault is specified. In this case, the uncontrolled system will crash in the next step.

If vk_control_bound is not able to prevent a crash, it will attempt to minimise the number of variables that crash.

Usage.

```
% Simple usage:
u = vk_control_bound(x, u, K, f, c)
```

See vk_kernel_compute for information on the format of the input parameters.

The return value will be the bounded version of the u specified in the input list. Where potential violations were detected that were salvagable, the new u will differ from the original one.

```
% Getting more informaton:
[u, crashed] = vk_control_bound(x, u, K, f, c)
[u, crashed, exited_on] = vk_control_bound(x, u, K, f, c)
```

- crashed is a boolean value that indicates whether a crash occurred despite any efforts to avoid one.
- exited_on is matrix of the form described by vk_viable_exited.

```
% Specifying options:
[u, crashed] = vk_control_bound(x, u, K, f, c, options)
```

Here options is either a structure created by vk_options, or otherwise a list of 'name', value pairs.

Caveats. When the use_custom_constraint_set option is specified, vk_control_bound will not be able to improve the control choice for real (non-imaginary) violations. In this case, it simply checks for constraint set violations.

Requires. vk_distance_fn, vk_newcontrol, vk_options, vk_viable_exited

See also. vk_control_enforce, vk_viable

❁

D.2. **vk_control_cost_fn.** compose cost functions that take vector parameters.

Synopsis. This function is needed by certain cost-minimising control functions in VIKAASA in order to be able to minimise using an un-named vector of variables, instead of a tuple.

This function is used by VIKAASA to transform a cost function which takes a tuple of named variables into a function that instead takes a pair of vectors.

Usage.

```
% Evaluate the cost of being in state x, with velocity xdot:
cost = vk_control_cost_fn(cost_fn, x, xdot)
```

```
% Construct a cost function, given cost_fn:
cfn = @(x, xdot) vk_control_cost_fn(cost_fn, x, xdot)
```

cost_fn should be a handle to a function that takes $2n$ parameters, where n is the number of variables in the state space. The first n variables should represent the state space, and the second n variables should give the velocities. For instance, if there were three variables in the state space, then the following might be a valid cost function:

```
% Construct a cost function for use in vk_control_cost_fn:
cost_fn = @(x,y,z,xdot,ydot,zdot) x^2 + y^2 + z^2 + norm([xdot, ydot, zdot]);
```

See also. vk_control_eval_fn

❁

D.3. **vk_control_enforce.** Simple function that makes sure that u is in $[-c, c]$

Synopsis. This function is used by VIKAASA to filter control choices that are outside of the permitted $[-c, c]$ range. This is done by clipping control choices outside of this range to the nearest value.

Usage.

```
% Make sure u ∈ [-c, c].
u = vk_control_enforce(u, c)
```

See also. vk_control_bound

❁

D.4. **vk_control_make_fn.** Returns a handle to a control function from a given string

Synopsis. Control functions can have one of two signatures. Either they take info first (those control algorithms in the VControlAlgs folder), or they don't. This function works out which is the case, and adds info (which needs to be specified as a second argument) if necessary.

Usage.

```
% Given some function name, and possibly an optional second argument
% containing an info structure, return a handle to a function:
control_fn = vk_control_make_fn(fn_name, varargin);
```

Requires. vk_error, vk_make_control_fn

See also. ControlAlgs, VControlAlgs

❁

D.5. **vk_control_wrap_fn.** Wrap a control algorithm with bounding code, etc.

Synopsis. This function wraps the result of a control choice in up to two functions. Firstly, if enforcement of the control range is in place, then the outcome of calling the function will be passed through vk_control_enforce to make sure that it lies within the control set $[-c, c]$. Secondly, if bounding of control choices at the constraint set edge is enabled, then the control choice (or the result from calling vk_control_enforce) is passed through vk_control_bound.

Usage.

```
% Get a handle to a function which is wrapped in one or both of the above
% functions, depending on the options structure:
fn = vk_control_wrap_fn(control_fn, K, f, c, options);
```

```
% Call that function on some x, and u:
fn(x, u)
```

Requires. vk_options

See also. vk_control_make_fn

❁

D.6. **vk_diff_fn.** Returns the vector of derivatives for a state-space and control.

Synopsis. This function returns a column vector of length n of derivatives. You don't need to use this function directly. Instead, call vk_make_diff_fn.

Usage.

```
% Get derivatives:
xdot = vk_diff_fn(f, x, u)
```

- f is a function that takes $n + 1$ inputs – i.e., the set of state variables, plus the scalar control.
- x is a column vector of length n, giving the state-space representation of the system.
- u is a scalar control.

Examples.

```
% Make a function (or use vk_make_diff_fn for this):
f = @(a,b,c,d) a*b + c*d;
fn = @(x,u) vk_diff_fn(f,x,u);
```

See also. vk_make_diff_fn

❁

D.7. **vk_diff_make_fn.** Construct a MATLAB function from an array of strings

Synopsis. This function returns a function which returns the array of derviatives for a given state-space point (represented as a column vector), and a control choice (represented by a scalar).

Usage.

```
% Given some project, create a function:
diff_fn = vk_diff_make_fn(project)
```

Requires: vk_diff_fn

❁

D.8. **vk_figure_data_insert.** Add data into a figure handle.

Synopsis. This function is used by VIKAASA to remember the current limits and slices in a given figure. The limits give either the maximum and minimum values in each dimension, or the values of the rectangular constraint set. In this way, trajectories can be added to a figure at a later time, and the axes of the figure readjusted without clipping any other information in the figure. This function inserts the data into the figure; it can then be retrieved with vk_figure_data_retrieve.

Usage.

```
% For some figure, h:
vk_figure_data_insert(h, limits, slices)
```

- limits is a row vector of length 4 (for a two-dimensional plot) or 6 (for a three-dimensional plot). It is the same format used to represent the rectangular constraint set, K.
- slices is a data structure of the type compatible with vk_kernel_slice.

Examples.

```
% Create a figure, and then insert information for a two−dimensional
% constraint set and no slices:
h = figure;
K = [0, 1, 5, 500];
vk_figure_data_insert(h, K, []);
```

See also. vk_kernel_slice, vk_figure_data_retrieve

❁

D.9. **vk_figure_data_retrieve.** Retrieve data previously stored in figure

Synopsis. This function is used by VIKAASA to remember the current limits and slices in a given figure. The limits give either the maximum and minimum values in each dimension, or the values of the rectangular constraint set. In this way, trajectories can be added to a figure at a later time, and the axes of the figure readjusted without clipping any other information in the figure. This function retrieves data previously associated with a figure using vk_figure_data_insert.

Usage.

```
% For some figure, h, get the limits of the figure and the slices.
[limits, slices] = vk_figure_data_retrieve(h);
```

- limits is a row vector of length 4 (for a two-dimensional plot) or 6 (for a three-dimensional plot). It is the same format used to represent the rectangular constraint set, K.
- slices is a data structure of the type compatible with vk_kernel_slice.

See also. vk_figure_data_retrieve

❁

D.10. **vk_figure_make.** Plots a viability kernel.

Synopsis. This function will plot the given viability kernel in an existing figure, as specified by the handle. Based on the number of dimensions in the kernel, vk_figure_make determines whether to plot an area or a volume.

Usage.

```
% Standard usage:
vk_figure_make(V, K, labels, colour, method, box, alpha_val, handle)
```

- V: The complete viability kernel.
- K: The constraint set
- labels: Labels for the axes
- colour: The colour to draw th kernel
- method: The method for drawing the kernel
- box: Whether or not to draw a box around the kernel
- alpha_val: The degree of transparency
- handle: the handle to display the figure in.

Requires. vk_error, vk_figure_data_insert, vk_plot, vk_plot_box

See also. vk_kernel_view

❁

D.11. **vk_figure_make_slice.** Slices a viability kernel and then plots the result.

Synopsis. This function slices the given kernel and then plots it into the figure provided by handle. Aside from slicing, functionality is identical to vk_figure_make.

Usage.

```
% Standard usage:
vk_figure_make_slice(V, slices, K, labels, colour, method, box, alpha_val, handle)
```

- V: The complete viability kernel
- slices: A nx3 matrix of [axis, point,distance] triples (see vk_kernel_slice).
- K: The constraint set

- labels: Labels to display on the axes.
- colour: The colour to draw the kernel.
- method: Which method to use in drawing the kernel.
- box: Whether or not to draw a box around the kernel.
- alpha_val: The transparency to give the kernel (certain drawing methods only)
- handle: The handle to display the figure in.

Requires. vk_error, vk_figure_data_insert, vk_kernel_slice, vk_plot, vk_plot_box

See also. vk_figure_make, vk_kernel_view

❁

D.12. **vk_figure_timeprofiles_make.** Construct a time profile figure from a project

Synopsis. This function creates a figure (or uses an existing one, if one is supplied) and draws time profile sub-plots into it, using the information and settings from the given project.

Usage.

```
% Plot the time profiles using the 'sim_state' field in 'project' in a new
% figure, returned as 'handle'.
handle = vk_figure_timeprofiles_make(project);
```

```
% Plot the time profile in a pre-existing figure, h.
vk_figure_timeprofiles_make(project, 'handle', h);
```

```
% Plot the time profile in a pre-existing figure, but use a different
% simulation to the one in the project:
handle = figure;
sim = vk_sim_make(project);
vk_figure_timeprofiles_make(project, 'simulation', sim, 'handle', handle);
```

Requires: vk_figure_timeprofiles_plot, vk_kernel_augment, vk_kernel_augment_constraints , vk_sim_augment

❁

D.13. **vk_figure_timeprofiles_plot.** Plot time profiles for a given simulation.

Synopsis. Places time profile subplots into the given figure handle. If there are already subplots present, they are not overwritten (i.e., hold on is set).

Usage.

```
% Create a handle, and plot time profiles into it:
h = figure;
vk_figure_timeprofiles_plot(labels, K, discretisation, c, V, ...
    plotcolour, line_colour, width, simulation, h);
```

All values are as in the projects.

Requires. vk_kernel_distances, vk_kernel_slice, vk_plot_path

See also. vk_figure_timeprofiles_plot_make, vk_project_new

❁

D.14. **vk_gui_figure_close.** Catch close events in the VIKAASA GUI

Synopsis. This function is used by the VIKAASA GUI to keep track of close events being sent to windows. It is unlikely that you will need to use it yourself.

Usage.

```
% Close a figure.
vk_gui_figure_close(h, event);
```

```
% Close a timeprofile figure.
vk_gui_figure_close(h, event, 'tp');
```

See also. vk_gui_figure_focus, vikaasa

❁

D.15. **vk_gui_figure_focus.** Catch focus events in the VIKAASA GUI.

Synopsis. This function is used by the VIKAASA GUI to keep track of focus events being sent to windows. It is unlikely that you will need to use it yourself.

Usage.

```
% Adjust internal VIKAASA GUI settings when a window is focussed.
vk_gui_figure_focus(h, event);
```

```
% The same, for a time profile.
vk_gui_figure_focus(h, event, 'tp');
```

See also. vk_gui_figure_close, vikaasa

❁

D.16. **vk_gui_make_waitbar.** Constructs a waitbar

Synopsis. This function constructs a progress bar. Used by the VIKAASA GUI to display the progress of viability kernel computations, and of simulations; and to capture cancel events.

Usage.

```
% Create a handle to a waitbar called 'wb'
wb = vk_gui_make_waitbar('Please wait');
```

See also. waitbar

❁

D.17. **vk_gui_project_load.** Setup the GUI-related fields after a file is loaded.

Synopsis. This function wraps the CLI command, vk_project_load, providing some additional support in the VIKAASA GUI.

Requires: vk_gui_update_inputs, vk_project_load

❁

D.18. **vk_gui_set_vartable.** Update the vartable to reflect the project

Synopsis. The vartable is the table of variables in VIKAASA that contains the main dynamic variables. This function inserts information into that table (and into the addnvartable too) from a given project. You don't need to use it yourself.

Usage.

```
% The project should be in 'handles.project'.
handles = vk_gui_set_vartable(hObject, handles);
```

See also. vikaasa

❁

D.19. **vk_gui_simgui.** GUI for interactive view of simulations.

Synopsis. This function creates a separate GUI window in which one can view the dynamic evolution of a simulation in relation to the viability kernel, or a slice through the kernel. The slice will be updated as the simulation proceeds to reflect the appropriate slice.

Requires. vk_figure_data_insert, vk_gui_simgui_, vk_gui_simgui_drawstep, vk_gui_simgui_setup , vk_kernel_slice, vk_plot, vk_plot_box, vk_plot_path, vk_plot_path_limits

See also. vikaasa

❁

D.20. **vk_gui_update_inputs.** Set the GUI inputs to the values recorded in the system state

Synopsis. This function is used by the VIKAASA gui to update all of the inputs in the main window to reflect the values stored in a given project.

Usage.

```
% 'handles' needs to contain the inputs for the main window, as well as the
% project at project.handles.
handles = vk_gui_update_inputs(hObject, handles);
```

Requires. vk_gui_set_vartable, vk_kernel_results, vk_sim_results

See also. vikaasa

❁

D.21. **vk_kernel_augment.** Augment the kernel with 'additional variable' data.

Synopsis. Where a project has specified additional variables, these are evaluated for each point in the kernel.

Usage.

```
% Augment the kernel, store the result in V, using data from p:
V = vk_kernel_augment(p);
```

```
% Using the viability kernel V, and taking other options from p:
V = vk_kernel_augment(p, V);
```

See also. vk_kernel_augment_constraints

❁

D.22. **vk_kernel_augment_constraints.** Augment the kernel constraint set.

Synopsis. Where additional values are specified, construct dummy constraint set values around them, using all possible combinations of the real constraint set.

Usage.

```
% Augment the constraint set, store the result in K, using data from p:
K = vk_kernel_augment_constraints(p);
```

```
% Specifying K, but taking other variables from p:
K = vk_kernel_augment_constraints(p, K);
```

See also. vk_kernel_augment

❁

D.23. **vk_kernel_augment_slices.** Augment the list of slices, and remove any ignored

Synopsis. Where additional values are specified, make sure that any that are set to 'ignore' are not included in the list of slices, and then add 'all' slices for them, so that they will be eliminated.

Usage.

```
% Augment the slices, store the result in slices, using data from p:
slices = vk_kernel_augment_slices(p);
```

See also. vk_kernel_augment, vk_kernel_augment_constraints

❁

D.24. **vk_kernel_compute.** Compute a viability kernel approximation

Synopsis. This function takes a constraint set, an array of differential equations, and a maximum absolute control magnitude, and attempts to compute an approximate viability kernel, by dividing the state-space into a discretised set of points (according to the discretisation option specified in options), and calling a viability-determination algorithm (usually vk_viable) against each point.

In addition to the three arguments that this function accepts, additional options can be passed in either as (name, value) pairs, or as a structure generated by vk_options.

vk_kernel_compute makes use of cellfun by splitting the problem space into discretisation-many sub-problems, which are then passed into cellfun. This is useful because in GNU Octave, parcellfun can be used as a drop-in replacement for cellfun to simultaneously consider the viability of multiple points, using parallel processing. For MATLAB®, we have written an implementation of cellfun called vk_cellfun_parfor which facilitates parallel

processing when the Parallel Toolkit is available. The choice of cell function to use can be altered by changing the cell_fn option (see vk_options).

Usage.

```
% Standard way of calling:
V = vk_kernel_compute(k, f, c)
```

- K is the constraint set, a row vector twice as long as the number of variables,

```
% Passing in an options structure, constructed by vk_options:
V = vk_kernel_compute(K, f, c, options)
```

```
% Using the default options, except for some specified here:
V = vk_compute(K, f, c, ...
    'name1', value1, ...
    'name2', value2 [, ...])
```

```
% Using an options structure, and modifying some parameters:
V = vk_kernel_compute(K, f, c, options, ...
    'name1', value1, ...
    'name2', value2 [, ...])
```

Examples.

```
% Compute a simple viability kernel
K = [0, 1, 0, 1] % Two dimensions, each with the same upper and
                 % lower bounds.
f = @(x, u) [1/2*x(1) + x(2)*u; u];
V = vk_kernel_compute(K, f, 0.001);
```

```
% Compute the same kernel with a higher discretisation
V = vk_kernel_compute(K, f, 0.001, 'discretisation', [50, 50]);
```

```
% Compute the same kernel again, but this time using PARCELLFUN
V = vk_kernel_compute(K, f, 0.001, ...
    'discretisation, [50, 50], ...
    'cell_fn', @(varargin) parcellfun(2, varargin{:}, 'UniformOutput', false));
```

Requires. vk_kernel_compute_recursive, vk_options

See also. cellfun, parcellfun, vk_cellfun_parfor, vk_viable

D.25. **vk_kernel_convert.** Converts viability kernels from an old format to new.

Synopsis. This function is for converting viability kernels stored in an old legacy format to the current format. It takes a cell array of axes, and a multi-dimensional array of points, dispgrid. dispgrid represents the viability kernel. If there is a zero in the (i, j, k)th element of dispgrid, then the point '[ax1(i), ax2(j), ax3(k)]' was identified as viable.

The replacement format, V is a $n \times dim$ array, where n is the of viable points; dim is the number of dimensions. Each row thus represents a viable point directly.

Usage.

```
% Standard usage:
V = vk_convert({xax, yax, zax}, dispgrid);
```

Requires: vk_convert_recursive

❁

D.26. **vk_kernel_delete_results.** Delete the results of a kernel approximation from the project

Synopsis. This function deletes a kernel and associated computation info from project. The fields removed are:

- V,
- comp_datetime,
- comp_time.

Usage.

```
% The resulting project will have no V field, etc.
project = vk_kernel_delete_results(project);
```

❁

D.27. **vk_kernel_distances.** Calculate distances between points in each dimension.

Synopsis. This function returns a row array, giving the distance between points in K, given the discretisation. This can then be used to work out which elements neighbour which others, for instance.

This information is also needed to produce slices. See vk_kernel_slice.

Usage.

```
% Store the distance information in 'd':
d = vk_kernel_distances(K, discretisation);
```

discretisation is a column vector of length n, where n is the number of dimensions/variables in the viability problem.

K is a column vector with length $2n$, and should be viewed as consisting of a sequence of paired values. Each pair of values in K represents the upper and lower bounds of the rectangular constraint set in that dimension.

See also. vk_kernel_slice, vk_kernel_frontier, vk_kernel_inside, vk_kernel_neighbours

❁

D.28. **vk_kernel_inside.** Test to see whether the given point is inside the kernel

Synopsis. This function determines whether the point x lies inside of V or not.

A point x is considered to be inside (for some distances and layers) if x is surrounded neighbour points in V (i.e., for a 3D problem, there would need to be 8 points in V around x). See vk_neighbours for a definition of "neighbour points".

If the point is not inside, but it still has some neighbours in V, then it is considered an "edge" point instead.

Usage.

```
% Standard usage:
[inside, edge] = VK_KERNEL_INSIDE(x, V, distances, layers)
```

- x is a column-vector, representing a point in the state space (see vk_viable for more information).
- V is a viability kernel. See vk_compute for the format of this.
- distances is a row-vector. Each element gives the distance between points in V in that dimension. For some kernel discretisation, d, the distance between points in the i-th dimension should be calculable as: distances(i)= (upper(i)− lower(i))/ (d −1), where upper and lower represent the upper- and lower-bounds of the constraint set.
- layers is an integer greater than zero. See vk_neighbours for information on how this is used.

Requires. vk_kernel_inside_rec, vk_kernel_neighbours

See also. VControlAlgs, vk_viable, vk_kernel_compute

❁

D.29. **vk_kernel_make_slices.** Construct a slice array from a cell array.

Synopsis. The cell array conforms to the format displayed by the VIKAASA GUI. From the command-line it is not really necessary to use this function, as you can just make the slice array by hand.

Usage.

```
% For some n × 3 cell array,
slices = vk_kernel_make_slices(data, K, discretisation);
```

Requires. vk_kernel_distances

See also. vk_kernel_slice

❁

D.30. **vk_kernel_middle.** Find a point which represents the "middle" of the kernel.

Synopsis. This function finds the middle of a viability kernel approximation by averaging the points. Note that the middle may not actually represent a point in the kernel approximation.

Usage.

```
% C will be a vector giving the postion of the middle.
C = vk_kernel_middle(V);
```

❁

D.31. **vk_kernel_neighbours.** Find a point's neighbours in a viability kernel.

Synopsis. This function goes through the kernel V, looking for all points that are at most $layers \cdot distances(dim)$ away, and greater than $(layers - 1) \cdot distances(dim)$ away, along each axis.

Usage.

```
% Standard usage:
neighbour_elts = vk_kernel_neighbourS(x, V, distances, layers)
```

- x is a column-vector, representing a point in the state space (see vk_viable for more information).
- V is a viability kernel. See vk_kernel_compute for the format of this.
- distances is a row-vector. Each element gives the distance between points in V in that dimension. For some kernel discretisation, d, the distance between points in the i-th dimension should be calculable as: $distances(i) = (upper(i) -$

$lower(i))/(d-1)$, where *upper* and *lower* represent the upper- and lower-bounds of the constraint set.

- layers is an integer > 0. As explained above, this variable is used to filter out elements in V that are too close to V. Thus, a the higher the layers, the fewer possible points can be considered within V. This can be used to make algorithms that care about whether they are on the edge of the kernel or not take action "sooner" (something akin to being more risk-averse).

See also. vk_kernel_compute, vk_viable, VControlAlgs

❁

D.32. **vk_kernel_results.** Returns the results of a kernel approximation.

Synopsis. This function returns a cell array giving an overview of a kernel approximation run. It is the same information that is displayed in the "Kernel Results" panel of the GUI.

Usage.

```
% place the information into a cell.
results = vk_kernel_results(project);
```

Requires: vk_timeformat

❁

D.33. **vk_kernel_run.** Run a kernel calculation from a file or structure. This function takes as input either a filename containing a project, or a structure representing a project, and runs the viability kernel calculation contained within. Then, it either returns the result, or saves it into a file.

vk_kernel_run(FILENAME) Runs the project contained in FILENAME, and when complete saves the result back into that file.

vk_kernel_run(FILE1, FILE2) Runs the project contained in FILE1, and when complete, saves the result into FILE2.

proj2 = vk_kernel_run(PROJ1) Runs the project represented by PROJ1 and returns a new structure.

Examples

```
% Load a file into a structure
proj = vk_project_load('Projects/vikaasa_default.mat');
% Change some settings.
proj.controlalg = 'CostMin';
proj.steps = 2;
```

```
proj.use_controldefault = 1;
proj.controldefault = 0;
% Re-run the kernel.
proj = vk_kernel_run(proj);
% Save the result.
vk_project_save(project, 'Projects/newproject.mat');
```

Requires. vk_diff_make_fn, vk_kernel_compute, vk_options, vk_options_make, vk_project_load

See also. vikaasa_cli

❁

D.34. **vk_kernel_slice.** Slice a viability kernel according to a slices array

Synopsis. Slices a viability kernel through any number of axes.

Usage.

```
% Store the resulting sliced kernel in SV.
SV = vk_kernel_slice(V, slices);
```

V is the viability kernel; slices is a $n \times 3$ array of '[dimension, point, distance]' rows.

- dimension (> 0): the index of the dimensions to eliminate.
- point: the position to do the slice at. NaN means all points
- distance (> 0): the "width of the blade" – i.e., the space to either side of the point that will be considered within the range.

Requires. vk_kernel_slice_helper

See also. vk_kernel_make_slices

❁

D.35. **vk_kernel_view.** View the viability kernel contained in the given file.

Synopsis. This function opens a project, or takes a project structure, and displays the kernel contained within it, using the settings contained in the project.

Usage.

```
% Viewing a kernel from within a project file:
vk_view_kernel('project.mat');
```

```
% Getting a handle to the resulting figure:
fig = vk_view_kernel(proj);
```

```
% Specifying an existing figure to plot into:
vk_view_kernel('project.mat', fig);
% Or:
p = vk_project_load('project.mat');
vk_view_kernel(p, fig);
```

Examples.

```
% Loading a project file, changing some settings and plotting the result:
% First, load a project into a structure.
proj = vk_project_load('Projects/vikaasa_default.mat');
% Change some of the settings:
proj.alpha = 0.4;
proj.drawbox = 1;
proj.plottingmethod = 'isosurface';
% Now, display the kernel:
vk_view_kernel(proj);
```

```
% Loading two different projects, and plotting both kernels into a single
% figure:
p1 = vk_project_load('project1.mat');
p2 = vk_project_load('project2.mat');
fig = vk_kernel_view(p1);
vk_kernel_view(p2, fig);
```

```
% The same thing again, but all in a single line:
vk_kernel_view('project2.mat', ...
    vk_kernel_view('project1.mat'));
```

Requires. vk_figure_make, vk_figure_make_slice, vk_kernel_augment, vk_kernel_augment_constraints, vk_kernel_augment_slices, vk_project_load

See also. vikaasa

❁

D.36. **vk_options.** Create an options structure for use with the VIKAASA library.

Synopsis. This function generates an options structure which can be used with a large number of VIKAASA library functions to modify their behaviour.

Usage.

```
% Create an options structure that contains all of the default settings:
options = vk_options(K, f, c)
```

```
% Create an options structure, overriding the settings for the
% configuration variables, 'name1' and 'name2':
options = vk_options(K, f, c, ...
    'name1', value1, ...
    'name2', value2 [, ...])
```

```
% Update an existing options structure, changing one or more variables:
options = vk_options(K, f, c, options,
    'name1', value1, ...
    'name2', value2 [, ...])
```

Available options (default values in brackets):

- bound_fn (vk_control_bound): Function handle specifying the function to use for bounding when the controlbounded option is set to 1.
- cancel_test (0): Whether to test to see if the user has interrupted computation. (e.g., by pressing "Cancel" in the VIKAASA GUI. See VIKAASA) If this option is set to 1, then the handle specified by cancel_test_fn will be called from time to time.
- cancel_test_fn (@()0): This is a function that takes no options, and returns either 0 to indicate that the system should continue, or else 1 to indicate that computation should be cancelled. If cancel_test is set to 1, then this option is used by vk_kernel_compute and vk_sim_simulate_euler/vk_sim_simulate_ode.
- cell_fn (cellfun): The "cell function" used by vk_kernel_compute to divide up the first dimension of the discretised constraint set sample when testing for viability. This is an option because it is possible to replace this function with a parallel version and thereby make vk_compute operate on multiple processors. The cell function that is chosen needs to use the option UniformOutput set to zero (see cellfun for more information).
- controlbounded (0): When set to 1, VIKAASA will attempt to prevent the system from crashing by limiting the control choice when close to the boundary.
- controldefault (0): The default control (should be a number in $[-c, c]$) – used in some cost-minimising control algorithms when use_controldefault is enabled (See CostMin for an example.)
- controlenforce (0): When set to 1, VIKAASA will check to ensure that the control choice is within $[-c, c]$.
- controlsymbol (u): A string representing the symbol used to denote the control in the differential equations.
- controltolerance (1e-3): Used by optimising control algorithms to decide when enough samples of the cost-function have been made. The smaller the number, the closer the control will be to the "true" optimum (See CostMin for an example.)
- cost_fn (@(x,xdot)norm(xdot)): This function is used by cost-minimising control algorithms such as CostMin to determine what control to use. The default behaviour

of this function is to consider the size of the velocity of the system, norm(xdot) solely, which may be quite inferior in many cases. This is therefore a very important option. See the examples.

- custom_constraint_set_fn (@(x)1): This function is used by vk_kernel_inside if the use_custom_constraint_set_fn option is set to 1. If specified, it should give a function that returns 1 when the specified point is in the constraint set, and zero otherwise. This functionality can be used to specify non-rectangular constraint sets. See the examples.
- debug (0): Turn on "debug mode." When this is enabled various data are printed into the MATLAB® Command Window during execution.
- discretisation (column vector of 10s): State-space discretisation. There should be one value for each variable in the viability problem. When the kernel is being computed, the constraint set is sampled for $\prod_{i=1}^{n} \delta_i$ points, each of which is then individually tested for viability. Lower discretisation is faster, but less useful. See
- enforce_fn (vk_control_enforce): The function to use when controlenforce is set to 1. See above.
- h (1): The step-size used in numerical approximation of the differential equations.
- maxloops (46000): Maximum number of loops performed by vk_viable before it gives up on a point. This option is present to prevent infinite loops.
- min_fn (fminbnd): The function used by cost-minimising algorithms such as CostMin to find the control which entails the least cost, as specified by the cost function (see the cost_fn option.) The default is to use fminbnd which uses a golden ratio search. This minimisation algorithm is therefore only suitable for cost functions that have a single global minimum in the range $[-c, c]$. By default this function is sensitive to the controltolerance option. See the examples, below. Also, note that VIKAASA comes with an alternative minimisation function that does a linear search instead. See vk_fminbnd.
- next_fn (@(x,u)x + h*f(x,u)): This function is used by vk_viable and vk_simulate_euler to work out the next point to consider. By default a 1st-order Euler approximation is used.
- norm_fn (norm): The function used to calculate the size of the system velocity. Used by vk_viable to decide when the system is slow enough to be considered steady. This function should take a single argument, which is a (column) vector of velocities, and should return a single numeric result.
- numvars: Gives the number of variables in the viability problem. This is usually calculated as half the length of the constraint set, K. You shouldn't change this unless you know what you are doing.
- ode_solver (defaults to a function handle making use of ode_solver_name): A function handle used by vk_simulate_ode to compute a numerical solution to the differential system of equations. This function is by default rigged to interact with cancel_test_fn and the progress_fn, and has MaxStep equal to the h option. Rather than changing this function, it may be best to change ode_solver_name.
- ode_solver_name (ode45): This string specifies the name of the function used by ode_solver (see above), without altering that function's use of MaxStep, etc. Unless you are doing something fancy, this is probably what you want to use.
- parallel_processors (2): Used in conjunction with use_parallel, this option specifies how many processors (or MATLAB workers) to create.

- progress_fn (@(x)1): A function that gets called periodically by vk_kernel_compute and vk_sim_simulate_euler/vk_sim_simulate_ode when report_progress is set to 1. It takes one parameter, which is either the number of points that have been assessed for viability (under vk_kernel_compute), or otherwise the number of time-frames that have been simulated.
- report_progress (0): Whether vk_kernel_compute and vk_sim_simulate_euler/vk_sim_simulate_o should call a progress report function (specified by progress_fn) to indicate how far they are through their tasks.
- steps (1): The number of forward-looking steps used by finite-time optimising control algorithms such as CostMin.
- sim_fn (vk_sim_simulate_ode): The "simulation function" to use. This is only really an important option if you decide to use vk_sim_make. It should be either vk_simulate_ode or vk_simulate_euler.
- sim_hardupper ('[]'): A column vector giving the indices of the variables which have "hard" upper bounds. When a hard upper bound is violated, simulation is halted.
- sim_hardlower ('[]'): Same as sim_hardupper, but for lower bounds.
- sim_stopsteady (0): Used by vk_sim_simulate_euler/vk_sim_simulate_ode to decide whether to continue the simulation on to the end, or to stop once the near-steady state has been determined.
- small (1e-3): Used by vk_sim_simulate_euler/vk_sim_simulate_ode and vk_viable to decide when to consider a system state to be "steady-enough" to be viable. This value is compared to the value of function specified by the norm_fn option, evaluated over the size of the differential equations at the given point. If the function value is less than or equal to the value of small, then the point will be considered viable. Thus, a smaller value of small is in theory more accurate, but may lead to much longer computation times.
- use_controldefault (0): See controldefault above for an explanation of this option.
- use_custom_constraint_set_fn (0): See the custom_constraint_set_fn option for more information on this.
- use_parallel (0): If set to 1, vk_kernel_compute will try to use a parallel implementation of cellfun (either parcellfun or vk_cellfun_parfor) so as to compute viability kernels in parallel. In MATLAB you need to have the Parallel Computing Toolbox for this to work. In GNU Octave parcellfun is available from Octave-Forge.
- viable_fn (vk_viable): This is the function used by vk_kernel_compute to determine whether a point is viable or not. The default is to use vk_viable; however, this could potentially be replaced with a different implementation.
- zero_fn (fzero): A function handle used by vk_control_bound to solve situations where the control should be chosen to prevent the system leaving the constraint set. It is sensitive to the controltolerance option.

Examples.

```
% Specifying a cost function:
options = vk_options( ...
    'cost_fn', @(x, xdot) (x(1) - x(2))^2);
```

```
% Specifying a circular custom constraint set:
options = vk_options( ...
    'custom_constraint_set_fn', @(x) (x(1)^2 + x(2)^2 < 100);
```

```
% Specifying an optimising function which is sensitive to controltolerance:
options = vk_options(options, ...
    'min_fn', @(f, min, max) fminbnd(f, min, max, ...
        struct('TolX', options.controltolerance)));
```

Requires. vk_cellfun_parfor, vk_control_bound, vk_control_enforce, vk_lsode_wrapper, vk_ode_outputfcn, vk_sim_simulate_ode, vk_viable

See also. CostMin, CostSumMin, cellfun, fminbnd, fzero, norm, ode45, vikaasa, vk_compute, vk_fminbnd, vk_kernel_inside, vk_sim_simulate_euler

D.37. **vk_options_make.** Creates an options structure from a project file.

Synopsis. This function is used to wrap vk_options in VIKAASA. It reads options out of project and feeds them into vk_options.

Usage.

```
% Standard usage:
options = vk_options_make(project, f)
```

- project should be a VIKAASA project.
- f should be a function, as created with vk_make_diff_fn

```
% Optionally, a waitbar can also be specified:
options = vk_options_make(project, f, wb, numcomputations, message)
```

Requires: vk_control_cost_fn, vk_options, vk_sim_simulate_euler, vk_sim_simulate_ode, vk_timeformat

D.38. **vk_plot.** Draw a two- or three-dimensional kernel

Synopsis. Draws a viability kernel using one of the methods available, or 'scatter' as a fallback. If V has two dimensions, then an _area_ function will be used. If it has three dimensions, then a _surface_ function will be used.

Requires. vk_plot_

See also. vk_figure_make, vk_figure_make_slice

❁

D.39. **vk_plot_area_isosurface.** Plots a 2D viability kernel using the isosurface function.

Synopsis. This function uses the isosurface method to plot a flat 2D kernel. This is done by building a fake 3D kernel, and then only displaying the first two dimensions of it (i.e., by lying it on its side). It requires the isocaps method to work properly, which at the time of writing was not available in Octave.

Usage.

```
% Plot into the current figure.
vk_plot_area_isosurface(V, colour);
```

```
% Plotting with 50% transparency
vk_plot_area_isosurface(V, colour, 0.5);
```

```
% Specify a ''smooth'' isosurface:
vk_plot_area_isosurface(V, colour, 0.5, 'smooth', 1);
```

See also. isosurface, isocaps, alpha

❁

D.40. **vk_plot_area_qhull.** Plots a 2D viability kernel as a convex area

Synopsis. This function uses the convhull function to make a convex area from the points in the viability kernel. This area is then filled using the fill function.

Usage.

```
% Standard:
vk_plot_area_qhull(V, colour);
```

```
% With transparency:
vk_plot_area_qhull(V, colour, 0.5);
```

See also. convhull, alpha

❁

D.41. **vk_plot_area_scatter.** Plots a 2D viability kernel as a scatter plot

Synopsis. This function plots a 2D kernel as a scatter plot, using the scatter function.

Usage.

```
% Standard:
vk_plot_area_scatter(V, colour);
```

```
% Using a different marker (the one is for alpha, which does not affect
% this function).
vk_plot_area_scatter(V, colour, 1, 'marker', 'x');
```

See also. scatter

❁

D.42. **vk_plot_box.** Draw a box around a kernel

Synopsis. This function boxes the given figure according to the constraint set. If there are slices, then these are considered too.

Usage.

```
% Plot the box in the current figure, and get back limits, which can be
% used with vk_kernel_data_insert.
limits = vk_plot_box(K);
```

```
% Plot the box in the current figure, using a slice:
limits = vk_plot_box(K, slices);
```

See also. vk_kernel_slice, vk_kernel_data_insert

❁

D.43. **vk_plot_path.** Draw a trajectory into a viability kernel window.

Synopsis. This function takes information from a simulation and plots it into the current figure.

Usage.

```
% Standard:
vk_plot_path(T, path, viablepath, showpoints);
```

```
% With colour and line width:
vk_plot_path(T, path, viablepath, showpoints, 'k', 2);
```

T, path and viablepath should be as they would be if they were produced by vk_sim_make.

See also. vk_sim_make

❁

D.44. **vk_plot_path_limits.** Calculate the extended limits of a kernel

Synopsis. When a path is being plotted, it is possible that it will travel outside of the constraint set in doing so. This function calculates the size of the necessary display window.

Usage.

```
% Given some limits, e.g., produced by vk_plot_box, see if they need to be
% expanded.
limits = vk_plot_path_limits(limits, path);
```

- path is a simulation path, as produced by vk_sim_make.

See also. vk_plot_path, vk_plot_box, vk_sim_make

❁

D.45. **vk_plot_surface_isosurface.** Plot a 3D kernel using isosurface.

Synopsis. This function plots a 3D kernel using the isosurface function, as well as isocaps, if available. It uses the current figure, and an alpha level may optionally be specified. If lighting functionality is present, the figure will be shaded. Otherwise, edges are drawn in black.

Usage.

```
% For some kernel V, and colour, c:
vk_plot_surface_isosurface(V, c);
```

```
% Specifying alpha level of 0.5:
vk_plot_surface_isosurface(V, c, 'alpha', 0.5);
```

See also. vk_plot, vk_plot_area_isosurface

❁

D.46. **vk_plot_surface_qhull.** Plot a 3D kernel using convex hull method.

Synopsis. Draws a 3D representation fo a kernel (or kernel slice) using the convhulln function. It uses the current figure. The colour can be either a string, or a triple, like '[1 1 0]'. An alpha level can optionally be specified.

Usage.

```
% For some kernel V and colour, c:
vk_plot_surface_qhull(V, c);
```

```
% Create a figure, and then plot a blue kernel in it:
h = figure;
vk_plot_surface_qhull(V, 'b');
```

```
% Optionally specify an alpha setting of 0.5:
vk_plot_surface_qhull(V, c, 0.5);
```

See also. vk_plot_surface

❁

D.47. **vk_plot_surface_scatter.** Plot a 3D scatter plot of a kernel.

Synopsis. This function plots the points given in a kernel, or kernel slice in 3D space. The current figure is used.

Usage.

```
% For some kernel, V and some colour, c:
p = vk_plot_surface_scatter(V, c);
```

```
% With a different marker (1 is for alpha, which is not used here).
p = vk_plot_surface_scatter(V, c, 1, '+');
```

See also. vk_plot, vk_plot_area_scatter

❁

D.48. **vk_project_load.** Loads a file and returns a project structure.

Synopsis. This function loads a `.mat` file into a structure, and checks to make sure that it represents a consistent project, by calling vk_project_sanitise on it. It also performs checks to see if the file is in the old format. If it is, then it is converted.

See vk_project_sanitise for a comprehensive list of what should be in a project.

Usage. p = vk_project_load('filename.mat');

Requires. vk_error, vk_kernel_convert, vk_project_sanitise

See also. vk_project_save

❁

D.49. **vk_project_new.** Creates a new project structure.

Synopsis. This function returns a newly initialised project structure.

Usage.

```
% Initialising a project, and storing it in p.
p = vk_project_new;
```

```
% Initialising a project, and setting fields at the same time:
p = vk_project_new( ...
  'numvars, 3, ...
  'symbols', {{'x'; 'y'; 'z'}}
);
```

Requires. vk_project_sanitise

See also. vk_project_load

❁

D.50. **vk_project_sanitise.** Set default values for the project, if missing.

Synopsis. This function checks a project for consistency, and updates any erroneous information as necessary.

Usage.

```
% Check that a project is ok:
project = vk_project_sanitise(project);
```

Fields. The following fields should be in every project. Default values are in brackets.

- alpha (0.9): the level of transparency used when plotting areas and surfaces.

- addnlabels (empty cell array of length numaddnvars): the labels for the additional variables. Should be a cell array with one column, and one row per variable.
- addnsymbols (empty cell array of length numaddnvars): the symbols for the additional variables. Should be a cell array in column form.
- addneqns (empty cell array of length numaddnvars): the right-hand sides of the eqautions for the additional variables. Should be a cell array in column form.
- addnignore (a column vector of zeros, of length numaddnvars): whether or not to ignore each additional variable. A one means it will be ignored.
- autosave (0): whether to auto-save kernel results after computing a viability kernel or not.
- controlalg (ZeroControl): a string representing the control algorithm to be used for kernel computation. Should be the name of a function residing in the "ControlAlgs" folder.
- controldefault (0): the value of the "default control" to use if use_controldefault is set to 1.
- c (0.005): The absolute maximum size of the scalar control.
- controlbounded (0): whether or not to use the control-bounding functionality (see vk_control_bound).
- controlenforce (0): wheter or not to enforce the c setting (see vk_control_enforce).
- controlsymbol (u): a string giving the symbol to use to represent the choice of control.
- controltolerance (1e-3): the tolerance to use with numerical cost-minimising control algorithms (see vk_options).
- custom_cost_fn (empty string): a string representing the right-hand side of a custom cost function (see the section in the manual on cost-minimising controls).
- custom_constraint_set_fn (empty string): a string giving the custom constraint set function (CCSF) for the problem, if there is one.
- debug (0): whether to display additonal debugging information or not.
- diff_eqns (an empty cell array of length numvars): a column of cells giving the system's dynamics for the viability problem.
- discretisation (a column vector of length numvars with value 11 in every field): the discretisation to use in seeking viable points.
- drawbox (0): whether to draw a box around points when plotting or not.
- h (1): the step-size to use with Euler's method for solving differential equations.
- holdfig (0): Whether to hold figures, so that subsequent plots are superimposed over previous ones.
- K (row vector of zeros, of length 2*numvars): the rectangular constraint set.
- labels (empty cell array, of length numvars, in column form): labels for the dynamic variables.
- layers (1): used by vk_kernel_inside to determine whether inside the kernel or not.
- parallel_processors (2): number of processors to use in parallel when use_parallel is set to 1.
- plotcolour ('[1 1 0]'): colour to plot viability kernels in.
- plottingmethod ("qhull"): a string giving the current plotting method. See vk_plot for more information on admissible plotting methods.
- progressbar (1): whether to display a progress bar while performing computations or not.

- sim_controlalg ("ZeroControl"): a string representing the name of a function to use for simulation. Should be a functon residing in either the "ControlAlgs" or the "VControlAlgs" folders.
- sim_hardlower (empty array): an array of indices giving the index numbers of variables that hard hard lower constraints.
- sim_hardupper (empty array): an array of indices giving the index numbers of variables that hard hard upper constraints.
- sim_iterations (10): the "time horizon" for simulations.
- sim_line_colour ('[0 0 1]'): the colour to plot lines in simulation plots.
- sim_line_width (2): the width of the lines to plot.
- sim_method ("ode"): a string giving the method to use for simulation. Should be one of "ode" or "euler".
- sim_use_nearest (0): whether to make sure that simulations start from points in the discretised constraint set K_δ or not.
- sim_showpoints (0): whether to show coloured dots indicating the viability of the system in simulation plots and time profiles.
- sim_showkernel (0: whether to display kernel slices in time profiles or not.
- sim_start (column vector of zeros of length numvars): the initial state for simulation.
- sim_stopsteady (0): whether or not to stop simulations when a near-steady state is achieved.
- sim_timeprofile_cols (2): the number of columns to display time profiles in within a figure.
- slices (empty array): the slice array. See vk_kernel_slice.
- steps (1): the number of forward-looking steps. Used by multi-step forward-looking cost-minimisation algorithms.
- stoppingtolerance (1e-3): used to calculate velocity at which the system will be considered "near-steady". See vk_viable.
- symbols (empty cell array of length numvars): column array of cells, each containing a string representing the symbol to be used in equations to represent that variable.
- use_controldefault (0): whether or not to use a default control with forward-looking cost-minimising algorithms.
- use_custom_cost_fn (0): whether to use a custom cost function with cost-minimising algorithms. If not, then norm-minimisation will be used.
- use_custom_constraint_set_fn (0): whether or not to use a custom constraint set function to augment the rectangular constraint set.
- use_parallel (0): whether or not to use parallel processors when computing viability kernels.

See also. vikaasa, vk_project_new

D.51. **vk_project_save.** Save a project to a specified file.

Synopsis. Saves a given project to a specified `.mat` file, ensuring that the file is in MATLAB®'s version 7 format (so that it can be read from either Octave or MATLAB®). If the file already exists, it will be overwritten.

Usage.

```
% Save the project structure into a file
vk_project_save(project, 'project.mat');
```

Notes. If saving is not successful, an error will be thrown.

Requires. vk_error

See also. vikaasa, vk_project_load

❁

D.52. **vk_sim_augment.** Augment the simulation structure in the given project

Synopsis. Where a project has specified additional variables, this function works to augment the sim_state information to include those variables as well.

Usage.

```
% Augment the kernel, store the result in V, using data from p:
sim_state = vk_sim_augment(p);
```

```
% Specifying some other simulation:
sim_state = vk_sim_augment(p, sim_state);
```

Requires: vk_kernel_augment, vk_kernel_augment_constraints

❁

D.53. **vk_sim_delete_results.** Removes data belonging to the simulation from project

Synopsis. This function deletes the sim_state field from a project.

Usage. p = vk_sim_delete_results(p);

See also. vk_kernel_delete_results

❁

D.54. **vk_sim_make.** Create the sim_state structure.

Synopsis. Makes a call to either vk_sim_simulate_euler or vk_sim_simulate_ode and returns the results in a structure, along with the important input arguments.

Usage.

```
% Standard Usage. In this case, the simulation is created using options
% from the project.
project.sim_state = vk_sim_make(project);
```

```
% Or, create an options structure yourself, then use it to make a
% simulation:
simulation = vk_sim_make(project, options);
```

```
% Similar, but with an additional setting:
simulation = vk_sim_make(project, options, 'sim_fn', @vk_sim_simulate_euler);
```

The project should be a standard VIKAASA project structure. options can be a structure created with vk_options, or a series of name:value pairs, or both (the former before the latter). sim_state should contain the following properties (see See vk_sim_simulate_euler for more details):

❁

D.55. **vk_sim_results.** Returns the results a simulation in a cell array.

Synopsis. This function returns a cell array giving informaton about the simulation information stored in the sim_state field of the given project. It is the same information that is displayed in the "Simulation Results" panel of the GUI.

Usage. results = vk_sim_results(project);

Requires. vk_timeformat

See also. vk_kernel_results

❁

D.56. **vk_sim_simulate_euler.** Simulate system trajectory using Euler approximation

Synopsis. This function simulates the path that the system would take over some number of iterations, given a starting point, and using some specified control algorithm.

Usage.

```
% Standard usage:
[T, path, normpath, controlpath, viablepath] = vk_sim_simulate_euler(...
```

```
    x, time_horizon, control_fn, V, distances, layers, ...
    K, f, c)
```

Return arguments are:

- T: Row-vector of time values, starting with zero.
- path: $n \times |T|$ matrix. Each column represents a the state-space position of the system at the time in the correponding element in T. Thus for instance, The first column will equal x.
- normpath: A row-vector, of same length as T, giving the value of the norm (as specified by the norm_fn option in vk_options) velocity of the system at each point in time.
- controlpath: A row-vector, of same length as T, giving the control choice at each point in time.
- viablepath: $5 \times |T|$ matrix. Each column represents four information flags (1 or 0) for that point in time: (i) whether or not the point is inside the kernel, V (See vk_kernel_inside for how this is computed); (ii) whether or not the point is considered to be an "edge' point (See vk_kernel_inside for info on this); (iii) whether or not the point is outside of the constraint set in a real dimension; (iv) whether or not the poimnt is outside the constraint set in a complex dimension; and (v) whether or not the system velocity is slow enough for the point to be considered steady.

Input arguments are:

- x: A column vector of length numvars, specifying the starting point of the simulation.
- time_horizon: A number greater than zero, specifying the end time of the simulation. The start time is always zero.
- control_fn: A control algorithm. See vk_control_wrap_fn.
- V: A viability kernel. This is used to give information about the system trajectory in relation to the kernel (through the viablepath return variable). If you don't care about this, you can specify an empty matrix, [] as the kernel.
- distances: A row vector of length numvars, giving the distance between points in V in each dimension. See vk_kernel_inside for an explanation of this.
- layers: See vk_kernel_inside for an explanation.
- K: A constraint set. See vk_kernel_compute for the format of this.
- f: A function that gives the velocity of each variable in the system, given some state space vector, x and some control choice, u.
- c: The absolute maximum size of the control.

```
% Usage with additional options:
[T, path, normpath, controlpath, viablepath] = vk_sim_simulate_euler(...
    x, time_horizon, control_fn, V, distances, layers, ...
    K, f, c, options)
```

options is either a structure created by vk_options, or a set of ('name', value) pairs, or both.

Notes. An imporant option for vk_sim_simulate_euler is sim_stopsteady, which causes the simulation to terminate as soon as a steady state is encountered, instead of waiting.

Requires. vk_control_wrap_fn, vk_kernel_inside, vk_options, vk_viable_exited

See also. ControlAlgs, VControlAlgs, vk_kernel_compute, vk_sim_simulate_ode

D.57. **vk_sim_simulate_ode.** Simulate system trajectory using an ODE solver

Synopsis. Simulates the path that the system would take over some number of periods, given a starting point, and using some specified control algorithm.

This function takes and returns identical arguments to vk_sim_simulate_euler, but it uses an ODE solver (e.g., ode45) instead of an Euler approximation. This means that it is generally slower, but more accurate for most purposes.

The ODE solver that this function uses is given by the ode_solver_name option (see vk_options). By default this is set to ode45. See the examples below for how you could change this to use a different solver.

It is also possible to entirely replace the wrapper by specifying the ode_solver option. See the examples.

Usage.

```
% Standard usage:
[T, path, normpath, controlpath, viablepath] = VK_SIM_SIMULATE_ODE(...
    x, time_horizon, control_fn, V, distances, layers, ...
    K, f, c)
```

```
% With optional extras added:
[T, path, normpath, controlpath, viablepath] = VK_SIM_SIMULATE_ODE(...
    x, time_horizon, control_fn, V, distances, layers, ...
    K, f, c, OPTIONS)
```

See vk_sim_simulate_euler for a description of all the arguments.

Examples.

```
% Use ode23 instead of ode45:
[T, path, normpath, controlpath, viablepath] = vk_sim_simulate_ode(...
    x, time_horizon, control_fn, V, distances, layers, ...
    K, f, c, 'ode_solver_name', 'ode23');
```

```
% Or, with other options:
options = vk_options(K, f, c, ...
    'ode_solver_name', 'ode23', ...
    'stepsize', 0.5, ...
    'sim_stopsteady', 1);
[T, path, normpath, controlpath, viablepath] = vk_sim_simulate_ode(...
    x, time_horizon, control_fn, V, distances, layers, ...
    K, f, c, options);
```

```
% Use a custom-made ODE solver:
myodesolver = @(fn, T, x0) somefunction(fn, x0);
[T, path, normpath, controlpath, viablepath] = vk_sim_simulate_ode(...
    x, time_horizon, control_fn, V, distances, layers, ...
    K, f, c, 'ode_solver', myodesolver);
```

Requires. vk_control_wrap_fn, vk_kernel_inside, vk_options, vk_sim_simulate_ode_helper, vk_viable_exited

See also. vk_kernel_compute, vk_sim_simulate_euler

❁

D.58. **vk_sim_start.** Determine where the starting position is.

Synopsis. This function reads a start state out of the given project. If the project has sim_use_nearest checked, then the nearest "grid" point (according to the discretisation) is used instead of the given one.

Usage.

```
% Work out the start state.
start = vk_sim_start(project)
```

See also. vk_sim_make

❁

D.59. **vk_sim_timeprofiles_from.** Create a simulation and view its time profile

Synopsis. This function is short-hand for creating a simulation and then plotting time profiles from it; here it is done in a single step. This is equivalent to calling vk_sim_make, followed by vk_figure_timeprofiles_make. The start state used is stored into the sim_start field of the project.

Usage.

```
% Return an updated project structure with the new simulation information
% in it, and display the time profiles.
project = vk_sim_timeprofiles_from(project, start);
```

❁

D.60. **vk_sim_view.** Draw a two- or three-dimensional simulation trajectory

Synopsis. Draws a simulation trajectory, either into a new figure, or into an existing one.

Usage.

```
% Plot the simulation into a new figure and return a handle to it.
h = vk_sim_view(project);
```

```
% Plot the simulation in an existing figure.
vk_sim_view(project, h);
```

Requires. vk_figure_data_insert, vk_figure_data_retrieve, vk_kernel_augment_constraints, vk_kernel_augment_slices, vk_plot_box, vk_plot_path, vk_plot_path_limits, vk_sim_augment

See also. vk_figure_make, vk_figure_make_slice

❁

D.61. **vk_viable.** Determine the viability of a point in the state space

Synopsis. The algorithm attempts to bring the system to a near-steady state by applying a bounded control.

The function returns one or two values. The second value is a structure that gives information about how the point was determined viable or non-viable.

Examples.

```
% Standard usage:
isviable = vk_viable(x, K, f, c);
```

```
% With an options structure created by vk_options:
isviable = vk_viable(x, K, f, c, options);
```

```
% Returning additional information:
[isviable, paths] = vk_viable(x, K, f, c, options);
```

Requires. vk_control_wrap_fn, vk_options, vk_viable_exited

See also. ControlAlgs, vk_kernel_compute

❁

D.62. **vk_viable_exited.** Indicate whether a point has exited the constraint set.

Synopsis. This function returns a $n \times 2$ matrix, where n is the number of dimensions in the problem, the first column gives the direction of any real violation of the constriant set for that dimension (or NaN if there was no real violation), and the second column gives the direction of any imaginary violation (or NaN if there was no imaginary violation).

Because VIKAASA only deals with real-valued problems, any complex value at all is considered a violation – that is, the only imaginary value for which there is no violation is 0*i.

In each case, if the number is negative, the lower bound has been violated; if the number is positive, the upper bound is violated. The actual value gives the distance from the lower or upper bound respectively (or zero for the complex dimension).

It is also possible for this function to return zeros in the first column. Zero means that a custom constraint set function was used, in which case it is impossible to know which axis the (real) violation occurred on.

Usage.

```
% Standard usage:
exited_on = vk_viable_exited(x, K, f, c);
```

```
% With optional params:
exited_on = vk_viable_exited(x, K, f, c, options);
```

exited_on is a $n \times 2$ matrix of numbers. Each number is either zero, or it represents an axis. If the number is negative, then that indicates that the lower bound was violated. If it's positive then the upper bound was violated.

```
% Checking how many constraints were violated:
exited_on = vk_viable_exited(x, K, f, c);
violated = any(any(~isnan(exited_on)));
count = sum(sum(~isnan(exited_on)));
```

```
% Checking whether a custom constraint set violation occured:
exited_on = vk_viable_exited(x, K, f, c);
ccsf = all(exited_on(:,1) == 0);
```

Requires. vk_options

See also. vk_control_bound, vk_viable

❁

D.63. **vk_cellfun_parfor.** An implementation of CELLFUN that uses PARFOR

Synopsis. This s a MATLAB equivalent of parcellfun for GNU Octave. It is used by vk_kernel_compute to approximate viability kernels in parallel. You should not need to use it individually.

This function automatically starts and stops workers in order to perform computation, so you shouldn't do this yourself.

Usage.

```
% Run with two processors -- call fn(x, y, z) on each combination of
% elements from x, y, z, and give their return values in the matrix, v.
v = vk_cellfun_parfor(2, fn, x, y, z);
```

```
% If fn2 returns a non-scalar, then use the "UniformOutput" option. In this
% case, v will be a cell.
v = vk_cellfun_parfor(2, fn2, x, y, z, 'UniformOutput', false);
```

See also. cellfun, parcellfun, parfor

❁

D.64. **vk_error.** Display an error using either errordlg or error.

Synopsis. This function is used by VIKAASA to display error messages differently, depending on the capabilities of the platform.

It will attempt to display an error window, using errordlg, and then also throw a real error. This means that unless this function is called inside of a try .. catch block, it will halt computation.

Usage.

```
% Throw an error.
vk_error('An error occurred');
```

See also. error, errordlg

❁

D.65. **vk_fminbnd.** A naive (slow) minimisation function.

Synopsis. This function is similar to fminbnd, except that instead of using a golden ratio search, it searches linearly through [*minvar, maxvar*]. fminbnd is faster, so this function is generally not used.

It takes the same arguments as fminbnd, except that it takes an additional tolerance option, which indicates the distance between points polled (i.e. minvar:tolerance:maxvar is polled).

Usage.

```
% Find the minimum of x^2 between −1 and 1:
min = vk_fminbnd(@(x) x^2, −1, 1, 0.01);
```

See also. fminbnd

D.66. **vk_help.** Display a help message. Type vk_help to see the message.

D.67. **vk_init.** Initialise the VIKAASA environment.

Synopsis. This script initialises the variables vikaasa_version, and vikaasa_copyright, which should be populated with the contents of the VERSION and NOTICE files, respectively; and adds the directories containing the VIKAASA library functions and control algorithms to the path. It is used by the vikaasa or vikaasa_cli commands.

Usage.

```
% Run the script
vk_init
% Afterwards, this should display the version of vikaasa:
vikaasa_version
```

See also. vikaasa, vikaasa_cli

D.68. **vk_lsode_wrapper.** Wrap the lsode function so that it works like ode45.

Synopsis. Because lsode does not conform to the functional signature of ode45, it is necessary to wrap it using this function in order to use it with VIKAASA.

Usage.

```
% Use lsode like you would use ode45:
[T, Y] = vk_lsode_wrapper(odefun, [0, time_horizon], x0);
```

See also. lsode, vk_options

❁

D.69. **vk_ode_outputfcn.** Used to make various checks during ODE solver runs.

Synopsis. This function is usually fed by vk_options into ode45 (or similar) so that certain checks can be undertaken while the solver is running. Currently, those checks are testing to see if a steady state has been acheived and testing to see if the user has issued a cancel command.

Usage.

```
% Standard usage -- 'outputfcn' would then be suitable for feeding into
'ode45'.
outputfcn = @(T, Y, flag) vk_ode_outputfcn(T, Y, flag, ...
    K, f, c);
odeopts = odeset('OutputFcn', outputfcn);
[T, Y] = ode45(odefun, T, Y, odeopts);
```

```
% Specifying options:
outputfcn = @(T, Y, flag) vk_ode_outputfcn(T, Y, flag, ...
    K, f, c, options);
```

Requires. vk_options

See also. vk_sim_simulate_ode

❁

D.70. **vk_timeformat.** Helper function that formats seconds into something human readable.

Synopsis. Given some number of seconds, this function returns a string containing a number, and a unit (one of 'seconds', 'minutes', 'hours' or 'days'). The unit will be accurate to one decimal place. The decision concerning which units to use is made by choosing the smallest possible, such that the unit is greater than one.

Usage.

```
% Displays '60 seconds'
vk_timeformat(60)
```

```
% Displays '1 minutes'
vk_timeformat(61)
```

❁

D.71. **vikaasa.** The VIKAASA GUI.

Synopsis. VIKAASA stands for VIability Kernel Approximation, Analysis and Simulation Application.

The VIKAASA graphical user interface (GUI) provides a front-end to the VIKAASA library for computation and analysis of viability kernels with two or more variables and a scalar control.

Usage. Running VIKAASA opens the VIKAASA GUI, or raises the GUI window if VIKAASA is already open (only one instance of the VIKAASA GUI can be open at a time).

Requires. vk_diff_make_fn, vk_figure_data_insert, vk_figure_data_retrieve, vk_figure_make , vk_figure_make_slice, vk_figure_timeprofiles_make, vk_gui_figure_close, vk_gui_figure_focus , vk_gui_make_waitbar, vk_gui_project_load, vk_gui_simgui, vk_gui_update_inputs, vk_init , vk_kernel_augment, vk_kernel_augment_constraints, vk_kernel_augment_slices, vk_kernel_compute , vk_kernel_delete_results, vk_kernel_make_slices, vk_kernel_results, vk_options_make, vk_plot_box , vk_plot_path, vk_plot_path_limits, vk_project_new, vk_project_sanitise, vk_project_save , vk_sim_augment, vk_sim_delete_results, vk_sim_make

See also. gui, project, vikaasa_cli

❁

D.72. **vikaasa_cli.** Initialise the VIKAASA enviroment for use from the commandline.

Requires. vk_help, vk_init

See also. vikaasa

❁

References

[1] Aubin, J.-P. (1997) *Dynamical Economic Theory: A Viability Approach*. Springer.

[2] Béné, C., Doyen, L., and Gabay, D. (2001) A viability analysis for a bio-economic model. *Ecological Economics*, **36**, 385–396.

[3] Martinet, V., Thébaud, O., and Rapaport, A. (2010) Hare or tortoise? trade-offs in recovering sustainable bioeconomic systems. *Environmental Modeling and Assessment*, **15**, 503–517.

[4] Krawczyk, J. B. and Kim, K. (2009) Satisficing solutions to a monetary policy problem: A viability theory approach. *Macroeconomic Dynamics*, **13**, 46–80.

[5] Schaefer, M. B. (1954) Some aspects of the dynamics of populations. *Bull. Int. Am. Trop. Tuna Comm.*, **1**, 26–56.
[6] Martinet, V. and Doyen, L. (2007) Sustainability of an economy with an exhaustible resource: A viable control approach. *Resource and Energy Economics*, **29**, 17–39.
[7] De Lara, M., Doyen, L., Guilbaud, T., and Rochet, M.-J. (2006) Is a management framework based on spawning stock biomass indicators sustainable? A viability approach. *ICES Journal of Marine Science*.
[8] Martinet, V., Thébaud, O., and Doyen, L. (2007) Defining viable recovery paths toward sustainable fisheries. *Ecological Economics*, **in press**, [doi:10.1016/j.ecolecon.2007.02.036].
[9] Pujal, D. and Saint-Pierre, P. (2006) Capture basin algorithm for evaluating and managing complex financial instruments. *12th International Conference on Computing in Economics and Finance*, Cyprus, June, conference maker.
[10] Bonneuil, N. and Saint-Pierre, P. (2008) Beyond optimality: Managing children, assets, and consumption over the life cycle. *Journal of Mathematical Economics*, **44**, 227–241.
[11] Bonneuil, N. and Boucekkine, R. (2008) Sustainability, optimality and viability in the ramsey model, manuscript, 39 pages.
[12] Krawczyk, J. B. and Kim, K. (2004) A viability theory analysis of a macroeconomic dynamic game. *Eleventh International Symposium on Dynamic Games and Applications*, Tucson, AZ, US, December.
[13] Krawczyk, J. B. and Sethi, R. (2007) Satisficing solutions for new zealand monetary policy. Discussion Paper DP2007/03, Reserve Bank of New Zealand, available at: http://www.rbnz.govt.nz/research/discusspapers/dp07_03.pdf.
[14] Clément-Pitiot, H. and Saint-Pierre, P. (2006) Goodwin's models through viability analysis: some lights for contemporary political economics regulations. *12th International Conference on Computing in Economics and Finance*, Cyprus, June, conference maker.
[15] Clément-Pitiot, H. and Doyen, L. (1999), Exchange rate dynamics, target zone and viability.
[16] Krawczyk, J. B. and Serea, O. S. (2009) A viability theory approach to a two-stage optimal control problem of technology adoption. Discussion Paper 2009/4, CORE, available at: http://www.uclouvain.be/cps/ucl/doc/core/documents/coredp2009_4.pdf on 3/09/2009.
[17] Simon, H. A. (1955) A behavioral model of rational choice. *Quarterly Journal of Economics*, **69**, 99–118.
[18] Bonneuil, N. (2006) Computing the viability kernel in large state dimension. *J. Math. Anal. Appl.*, **323**, 1444–1454.
[19] Quincampoix, M. and Saint-Pierre, P. (1995) An algorithm for viability kernels in hölderian case: Approximation by discrete dynamical systems. *Journal of Mathematical Systems, Estimation and Control*, **5**, 1–13.
[20] Krawczyk, J. B. and Serea, O.-J. (2009) Computation of viability kernels for three and four meta-state monetary-policy macroeconomic problems. *Workshop on Perturbations, Game Theory, Stochastics, Optimisation and Applications*, University of South Australia, September.
[21] Gaitsgory, V. and Quincampoix, M. (2009) Linear programming approach to deterministic infinite horizon optimal control problems with discounting. *SIAM J. Control Optim.*, **forthcoming**.
[22] Krawczyk, J. B. and Serea, O. S. (2007) A viability theory approach to a two-stage optimal control problem. *Joint Meeting of the AMS - NZMS*, **2007**.

List of Figures

Index