Kernels

Kernels in Set language

Given Ω-modules V and W and a linear map $\varphi : V \to W$,

$$\ker \varphi = \{v \in V \mid \varphi(v) = 0\}.$$

This definition is concise because it's meaning is outsourced to the concept of a set. Let us see how far this gets us. For example, suppose we are given φ by a matrix

$$\Phi = \begin{bmatrix} 1 & 1 & 0 & 3 \\ 1 & 0 & 2 & 5 \\ 1 & 1 & 0 & 3 \end{bmatrix}$$

So what is in $\ker \varphi$? The set offers no answer.

Kernels in Computer Algebra Systems

We can begin by inputting a linear mapping (as a matrix) into a computer algebra system.

In [1]
```
Phi = [ 1 1 1; 1 0 1; 0 2 0; 3 5 3 ]
print Phi
```

Out [1]
```
[ 1 1 0 3 ]
[ 1 0 2 5 ]
[ 1 1 0 3 ]
```

Research shows humans make 3-6 errors per hour, no matter what the task is. Why waste any of them on miscalculating? Let us ask a computer for the kernel.

In [2]
```
K = Kernel(Phi)
print K
```

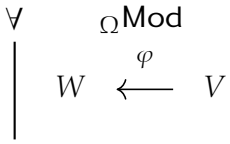
Out [2]
```
[ -2 -5 ]
[  2  2 ]
[  1  0 ]
[  0  1 ]
```

This may be a surprise. This is a matrix, not a set. Why?

Kernels in Diagram language

To decode the difference between the set-wise concept of a kernel and what a program provides, we start with an alternative definition of kernels using diagrams and focussed on what it means to put data into the kernel and to get data out of the kernel.

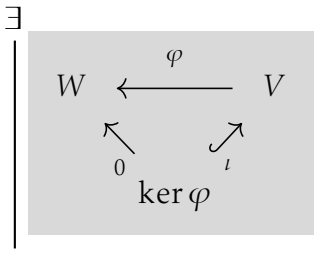
We start with a linear map, and so that gives us our first diagram. It is introduced with $\forall$ because it applies to *all* linear maps. We should also indicate somewhere that that context is modules, which we do once at the start by writing ${}_\Omega\mathsf{Mod}$. Often we authors skip that step letting context be know implicitly.



Kernels exist for every linear map and will appear as a new structure denoted $\ker\varphi$. This on its own would have no relation to φ in the diagram, so we draw out that relationship by adding new arrows, new linear functions that is. One arrow is just 0, and the second ι is injective.

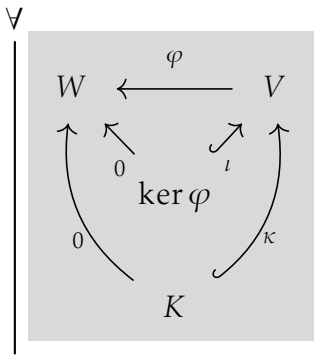
The 0 arrow should be explained a bit, it just means $0(x) = x$. This is an uneventful step but its role will unfold as necessary because it will offer the constraint to the equations that follow. Without it there would be no equations and thus no solutions. Note, some authors prefer to take two steps for zero functions pausing to pass through the zero module $\{0\}$. Somewhat confusingly that space is also written as 0. Get used to it but prepared to explain your own missuses of 0 should anyone ask you. So instead of writing $0 : A \to B$ some authors will write a sequence of arrows $A \to 0 \to B$.

The real star is the arrow denoted by ι which feeds into the arrow φ. Lining up the arrows indicates we can compose these two functions $\varphi \circ \iota$. This should equal the other arrow reaching the same point, namely 0. So $\varphi \circ \iota = 0$, or rather $\varphi(\iota(k)) = 0$ for $k \in \ker\varphi$. We say the diagram *commutes* and we indicate this by shading the background.



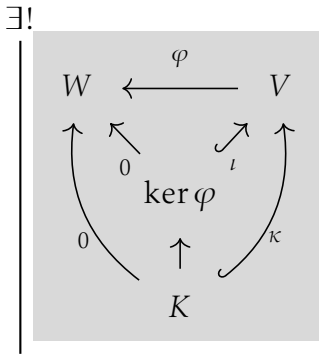
The ι in this diagram is a new linear map, and so it could be given by a matrix. In the example given earlier, the matrix Φ produced a kernel function ι whose matrix was the matrix that our computer produced. So through our new lens of kernels the computer output is correct, we should get a matrix not a set.

Unfortunately we cannot stop here even though we have the promised kernel. This is because many spaces K and functions $\kappa : K \to V$ could play the role of $\ker\varphi$ as shown in the above diagram without actually being the kernel we have in mind. For example, $K = \{0\}$ certainly would do the same but the matrix Φ given earlier we expect a different answer. So this cannot be a complete understanding of kernels. We need $\ker\varphi$ to be "as big as possible". Even saying that we find a puzzle because what would it mean for a function to be "as big as possible". To resolve this let us add to our diagram an other data and functions that can match what we already know about kernels.



The quantifier $\forall$ here now ranges over $\kappa : K \to V$ so we are setting ourselves up to compare how our chosen $\ker\varphi$ compares to any other possible solution.

The conclusion you might have guessed is that if our solution should be at least as capable as any other, and to diagram that we simply need that every alternative solution can be mapped into our own $\ker\varphi$. So there exists a unique new arrow $K \to \ker\varphi$ transforming any competitor data into data of our own type, as shown below.



Like frames in a graphic novel, these four diagrams should be read as a timeline, see Figure 0.1.

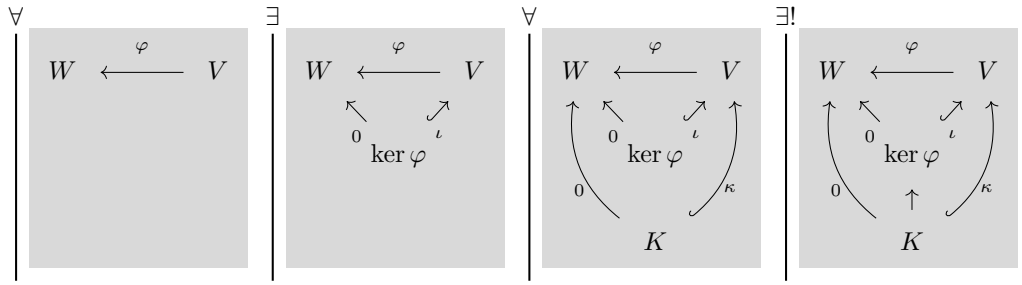


Figure 0.1: The diagram description of kernels.

This pattern of "∀∃∀∃!" will be repeated many times in similar constructions. In fact the longer we work with logical puzzles we will find the steady use of the pattern

$$\Pi_n \equiv \overbrace{\forall\exists\cdots\forall\exists}^{n} \qquad\qquad \Sigma_n \equiv \overbrace{\exists\forall\cdots\exists\forall}^{n}.$$

These organizations of logical sentences was introduced by Kleene and Mostowski and is known today as the *Arithmetical Hierarchy*. Programmers who ask how hard it is to prove statements in that Hierarchy will find they have invented the *Polynomial-time Hierarchy*. So it is worth getting comfortable with the meaning, but we warned that answering questions in these hierarchies could earn you a million Euro prize and your name in the newspaper. One of the leading questions is this tower may one day collapse, meaning that you only need to go to some fixed value of n before you know everything.

Kernels in Typed language

The diagram language clarifies how kernels can be thought of as functions and functions that have a maximal quality. Yet, programs do not think in pictures, we do. So we need to translate the same ideas to a syntax we can turn into a program. Here is how.

First the question depends on Ω-modules V and W and a linear map $\varphi : V \to W$ known first from context, denoted ctx or Γ. Using the notation $P \vdash Q$ to say "P leads to Q", also denoted $\frac{P}{Q}$, then we can state this as forming the kernel under a list of assumed knowledge.

$$\frac{\mathsf{ctx} \vdash V, W :_{\Omega} \mathsf{Mod} \qquad \varphi : \mathsf{Lin}_{\Omega}(V, W)}{\mathsf{ctx} \vdash \ker \varphi : \mathsf{Type}} \qquad (\mathsf{F}_{\ker})$$

The label $\mathsf{F}_{\ker}$ stands for *formation*. Programs write this in many different ways usually by introducing some keywords like "import" and "use X from Y'. To introduce a new type of data a keyword such as "class" or "type" is used. For example, the following pseudo-code reflects the content of $(\mathsf{F}_{\ker})$ but in a dialect similar to several modern procedural programming languages such as C++ and Java.

```
using V,W:Mod[Omega], Phi:Lin[V,W] from ctx
class Ker[Phi] {...}
```

For those using functional programming languages like OCaml or Haskell the following syntax offers as similar translation.

```
import V,W:Mod Omega, Phi:Lin V W from ctx
type Ker Phi
```

Listing 0.0.1 An introduction of data to a kernel.

```
// Procedural style code
class Ker[Phi](k:K,kappa:K->V) where (Phi(kappa(k)) == 0)
// usage
Phi = ...; k = ...; kappa = ...;
x = new Ker[Phi](k,kappa)

--- Functional style code
type Ker Phi
null: (k:K)-> (kappa:K->V)-> (Phi kappa k == 0)-> Ker Phi
--- usage
Phi= ...; k= ...; kappa= ...;
x= null Phi k kappa --- system checks Phi kappa k == 0
```

Next the diagram above captured the high-level movement of that data without ever considering the actual data. The programs will certainly need these data. The premise from the diagram is that any data $k : K$ which is found to have $\kappa(k) : V$ where $\varphi(\kappa(j)) = 0$ (see the diagrams above) must produce data in $\ker\varphi$. Any such data $k : K$ is meant to produce data in the kernel, because the kernel is the largest such structure. So we include such a rule.

$$\frac{k : K \qquad \kappa : K \to V \qquad pf : \varphi(\kappa(k)) =_W 0}{\mathsf{null}(\kappa(k)) : \ker\varphi} \qquad (\mathrm{I}_{\ker})$$

The $\mathrm{I}_{\ker}$ here is for *introduction* because data is being introduced of the desired type. Most readers will not be prepared for the meaning of symbols like:

$$pf : \varphi(\kappa(k)) =_W 0$$

Programmers however are uniquely well-positioned to guess the meaning. We want some data pf that has the type $\varphi(\kappa(k)) = 0$ in W. Said another way, we need someone to provide a proof

of that equality. In programs this can be done by several tricks most common are what are known as *guards* or *rails*. These are a type of documentation added to a program to let the programming language enforce that data is used in restricted ways. In this case, no one can introduce a term in the kernel without proof. In code this can be captured in a number of ways, Listing 0.0.1 is one option.

Now it is time to use data in the kernel. It is clear how this should proceed, anything in the kernel can be mapped to 0 in W or to a value in V which will map to 0 under φ. The rules are therefore as follows.

$$\frac{x : \ker \phi}{0 : W} \qquad \frac{x : \ker \phi}{\iota(x) : V} \tag{$\mathsf{E}_{\ker}$}$$

The name $\mathsf{E}_{\ker}$ stands for *elimination* as we are eliminating the kernel type to get to new types. In code this might be done as shown in the code fragment Listing 0.0.3.

Listing 0.0.2 Using of data of kernel type.

```
// Procedural style code
class Ker[Phi](k:K,kappa:K->V) where (Phi(kappa(k))== 0){
    def iota:V = ...
    def zero:W = ...
}
// usage
x = new Ker[Phi](k,kappa)
v = x.iota

--- Functional style code
iota: Ker Phi -> V
...
zero: Ker Phi -> W
...
--- usage
x= null Phi k kappa --- system checks Phi kappa k == 0
v = iota x
```

Finally we need to do some computing somewhere and we learn what to compute by inspecting the condition of "commutative diagrams".

$$\frac{k : K \qquad \kappa : K \to V \qquad pf : \varphi(\kappa(j)) =_W 0}{\iota(x) := \kappa(k)} \qquad (\mathsf{C}_{\mathrm{ker}})$$

All together this comes together in software in many different ways each designed around different techniques to improve how we read and execute code. Listing **??** provides some of the options.

Listing 0.0.3 Using of data of kernel type.

```
// Procedural style code
class Ker[Phi](k:K,kappa:K->V) where (Phi(kappa(k))== 0){
    def iota:V = kappa(k)
    def zero:W = 0
}
// usage
x = new Ker[Phi](k,kappa)
v = x.iota

--- Functional style code
iota: Ker Phi -> V
iota x = kappa k where x = null k kappa
zero: Ker Phi -> W
zero x = 0
--- usage
x= null Phi k kappa --- system checks Phi kappa k == 0
v = iota x
```

Computing a kernel

Algorithms appearing in China around 2000 years ago appear to depict a version of computing kernels. A thousand years later in Irag and Iran the inventers of algebra were solving systems of

Listing 0.0.4 A complete data type for kernels

```
using V,W:Mod[Omega], Phi:Lin[V,W] from ctx
class Ker[Phi] {
    private v:V

    def null(j:J, kappa:J->V,
        require Phi(kappa(j)) == 0) {
        v = kappa(j)
    }
    def iota:V = v
    def zero:W = 0
}
```

linear equations which would necessitate the ability to solve kernels as well. In the 1800's Gauss's many contributions to math included a systematic algorithm to solve for kernels. Once you have learned Gauss's method it becomes impossible to think of any other likely solution. For that reason the historic examples are usually conjectured to be the same algorithm.

Let us consider a matrix in which a subset of the columns are an identity matrix. That is, up to possibly permuting the columns the matrix has the form

$$\begin{bmatrix} I_r & M \\ 0 & 0 \end{bmatrix} \in \Delta^{m\times n}$$

Then an answer would be written down with formula requiring no computation:

$$\begin{bmatrix} I_r & M \\ 0 & 0 \end{bmatrix}\begin{bmatrix} -M^\dagger \\ I_{n-r} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\Phi = \begin{bmatrix} 1 & 1 & 0 & 3 \\ 0 & -1 & -2 & -2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Uniqueness questions