The Four Spaces 1

$\varphi(v) = \Phi v$	Codomain	Domain
Quotient	coker $\varphi \cong$ Null $\Phi^\dagger$	coim $\varphi \cong$ Span $\Phi^\dagger$
Submodule	im $\varphi \cong$ Span Φ	ker $\varphi \cong$ Null Φ

Table 1.1: The 4 spaces of a linear map.

方程 **Fang cheng**

Algorithms appearing in China around 2000 years ago depict the use of linear algebra on matrices to solve problems. The Chinese called these matrices *Făng chéng*. A thousand years later in Iraq and Iran where inventing algebra and using it to solve systems of linear equations. In the late 1600's Leibniz recorded matrix ideas for European scholars and may have even been aware of the Chinese Fang cheng before doing so. A number of textbooks today call this process "Gaussian Elimination" after the 19th century math polymath Carl Friedrich Gauss who systemized the mechanics. Now that the Chinese origins are better known the best practice in history would promote this attribution instead.

Definition 1.1. For a natural number d, $[d] = \{1, \ldots, m\}$ with $[0]$ the empty set.

- A *coordinate vector* is function $v : [d] \to \Delta$ also denoted $v : \Delta^d$. We access its i-th value by writing v_i.
- An $d_1 \times d_2$-*matrix* Φ over Δ is a function $\Phi : [d_1] \times [d_2] \to \Delta$. These are also denoted $\Phi : \Delta^{d_1 \times d_2}$. Evaluating that function as (i, j) is denoted Φ_{ij}. The coordinates i are called *rows* and the j's are called *columns*.
- A *coordinate tensor* (also called a *multiway array*) is function $t : [d_1] \times \cdots \times [d_\ell] \to \Delta$. The $i : [\ell]$ are called *axes*.

Of the many important classes of of matrices.

$$e_i : \Delta^n \qquad (e_i)_j = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \tag{e_i}$$

$$E_{ij} : \Delta^{m \times n} \qquad (E_{ij})_{kl} = \begin{cases} 1 & i = k, j = l \\ 0 & \text{else} \end{cases} \tag{E_{ij}}$$

These are sometimes called *matrix units*. For example for $m = 3, n = 4$ we have

$$E_{23} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \qquad E_{22} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad E_{32} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Proposition 1.2. Every $(m \times n)$-matrix Φ is the sum of scaled matrix units,

$$\Phi = \sum_i \sum_j \Phi_{ij} E_{ij}.$$

For $m = n$ we have the following essential matrices.

$$(I_n)_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \tag{I_n}$$

$$I_n + \alpha E_{ij} \quad i \neq j \tag{Transvection}$$

$$Diag(a_1, \ldots, a_n)_{ij} = \begin{cases} a_i & i = j \\ 0 & \text{else} \end{cases} \tag{Diagonal}$$

$$\Sigma_{ij} \neq 0 \Rightarrow (\Sigma_{ij} = 1) \& (k \neq i \Rightarrow \Sigma_{kj} = 0) \& (j \neq k \Rightarrow \Sigma_{ik} = 0). \tag{Permutation}$$

These last three families are known as *elementary matrices*. Each is invertible.

1.1 Clearing a column

Consider what we might do to clear a column in the following matrix.

$$\begin{bmatrix} 4 & 12 \\ 6 & 30 \\ 27 & 24 \end{bmatrix}$$

Ready...set...WAIT! Lets examine what rules we are prepared to except.

The first moves in linear algebra are to organize the data, swapping what can be swapped. Think of a larger chalkboard with the numbers in place. Would you really want the chore of copying the data over just to move it? Computers likewise rarely move the data either. Instead they just compose the function $M : [d_1] \times [d_2] \to \Delta$ with permutations of $[d_1]$, respectively $[d_2]$ and leave the data exactly as it was but allow the user of the matrix to interact with the data as if it was moved. Said another way, lets just leave permutations off the menu as they are mostly cosmetic.

The next move in linear algebra is to turn a nonzero into a 1 so we can use it as a pivot. Indeed in Linear Algebra we saw by Schur's lemma that the data can be written with coefficients in a division ring. So nonzero values have inverses. But there are two reasons to question this move. When we rescale we can build really small or really large numbers and this causes rounding problems. Addition is far more stable. Secondly, there are numerous applications where numbers are restricted and division is not permitted. So let us set aside rescaling until we need it.

Subtract $2\times$ row 1 to from row 2 and once from row 3.

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -6 & 0 & 1 \end{bmatrix}\begin{bmatrix} 4 & 12 \\ 6 & 30 \\ 27 & 24 \end{bmatrix} = \begin{bmatrix} 4 & 12 \\ 2 & 18 \\ 3 & -48 \end{bmatrix}$$

Subtract row 2 from row 3.

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & -1 & 1\end{bmatrix}\begin{bmatrix}4 & 12\\2 & 18\\3 & -48\end{bmatrix}=\begin{bmatrix}4 & 12\\2 & 18\\1 & -64\end{bmatrix}$$

Use row 3 to subtract from rows 2 and 4.

$$\begin{bmatrix}1 & 0 & -4\\0 & 1 & -2\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}4 & 12\\2 & 18\\1 & -64\end{bmatrix}=\begin{bmatrix}0 & 268\\0 & 146\\1 & -64\end{bmatrix}$$

It took more work but we achieved the goal of clearing the first column with no division and no permutations.

Now lets try to clear one more column using only transvections. We suppress these matrices.

$$\begin{bmatrix}4\\6\\12\end{bmatrix}\sim\begin{bmatrix}4\\2\\0\end{bmatrix}\sim\begin{bmatrix}0\\2\\0\end{bmatrix}$$

Here we have to stop because we have refused to divide. This could of course just indicated that we tried the wrong strategy. Or it could be a symptom of an obstacle. Which is it? Well notice the following.

$$\begin{bmatrix}4\\6\\12\end{bmatrix}=2\begin{bmatrix}2\\3\\6\end{bmatrix}$$

So whatever our transformations $Tv = T(2u) = 2(Tu)$. So the result will be a multiple of 2. So unless it is all 0 we are at our best to get to $2e_i$. This is a general principal.

Proposition 1.3. If $v = \lambda u$ and T is a product of transvections then Tv is a multiple of λ.

With this limitation in mind we can consider writing all the values in the vector with no common divisor (meaning if $\lambda \mid v_i$ for each i then λ is invertible). This is not possible in all number systems but that is for another course entirely. Let us make one further assumption that there is an algorithm to compute greatest common divisors (the Euclidean algorithm will do). This also allows us to find a vector u where $u \cdot v = 1$.

For example, $v = (5, 17)$ then we may use $u = (5, -2)$ so that $7 \cdot 5 + -2 \cdot 17 = 1$. Using this we have a means to clear.

$$\begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ -3 & 1 \end{bmatrix}\begin{bmatrix} 5 \\ 17 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 5 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Now let us multiply these matrices together.

$$\begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ -3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -2 \\ -2 & 5 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ -3 & 1 \end{bmatrix} = \begin{bmatrix} 7 & -2 \\ -17 & 5 \end{bmatrix}.$$

Now we see the anatomy of this solution. Because $u \cdot v = 1$ we put u in the first row. Then we simply write down a basis of vector perpendicular to v for the remaining rows.

Definition 1.4. A matrix is *unimodular* if it is a product of transvections.

Proposition 1.5. If we have a Euclidean algorithm then for every nonzero $v : \Delta^n$ there is a unimodular matrix T, a scalar λ, and $i : [n]$ such that $Tv = \lambda e_i$.

Tableaux and Normal Forms

We all know the Reduced Row Echelon Form (RREF). This is merely a recursive use of clearing columns as above. But it would help not to go in circles and have a target.

Definition 1.6. A matrix Φ is in *Reduce Row Echelon Form (RREF)* if there is a permutation matrix Σ of the columns such that

$$\Phi\Sigma = \begin{bmatrix} I_r & M \\ 0 & 0 \end{bmatrix}.$$

The *Hermite Normal Form* is a permutation matrix Σ and numbers $a_1|\cdots|a_r$ such that

$$\Phi\Sigma = \begin{bmatrix} D & M \\ 0 & 0 \end{bmatrix} \qquad D = \mathsf{Diag}(a_1, \ldots, a_r).$$

So how can we make this outcome? Augment the matrix Φ by I_m to create a tableaux $T = [I_m|\Phi]$. The first m columns are called *artificial*. Now we scan the Φ for a column j which we can clear. So if it is 0 we skip it, and if not we scan for the column whose greatest common divisor divides the greatest common divisor of the entire matrix. Then repeat on the remaining basic columns.

Let us look at this in the case of a division ring we clear the column with pivot in row 1.

$$\begin{array}{|ccc|c|c|c|}
\hline
 & & & & j & \\
\hline
1 & & & & v_1 & \\
 & \ddots & & * & \vdots & * \\
 & & 1 & & v_m & \\
\hline
\end{array}$$

becomes

$$\begin{array}{|ccc|c|c|c|}
\hline
 & & & & j & \\
\hline
* & & & & 1 & \\
* & \ddots & & * & 0 & * \\
\vdots & & 1 & & \vdots & \\
\hline
\end{array}$$

Because we had a pivot in row 1 the clear occurred only from row 1 and so only the first column was affected. It is said that the first column is *leaving* and the j-th column is *entering*. We proceed with every artificial column. The columns that enter become the identity of the RREF.

Proposition 1.7. Every matrix over a division ring has an RREF. In the general ring case with Euclidean algorithm there is a Hermite Normal Form.

1.2 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 the 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. In any case the figure we start with is below.

$$\forall \qquad {}_{\Omega}\mathsf{Mod}$$
$$\begin{tikzcd} W & V \arrow[l, "\varphi"'] \end{tikzcd}$$

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, displayed with a hook.

$$\exists$$
$$\begin{tikzcd} W & & V \arrow[ll, "\varphi"'] \\ & \ker\varphi \arrow[ul, "0"] \arrow[ur, hook, "\iota"'] & \end{tikzcd}$$

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 for 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 any other data and functions that can match what we already know about kernels.

$\forall$

$$
\begin{tikzcd}
W & & V \arrow[ll, "\varphi"'] \\
& \ker\varphi \arrow[ul, "0"] \arrow[ur, hook, "\iota"'] & \\
& K \arrow[uul, bend left, "0"] \arrow[uur, hook, bend right, "\kappa"'] &
\end{tikzcd}
$$

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 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.

$\exists!$

$$
\begin{tikzcd}
W & & V \arrow[ll, "\varphi"'] \\
& \ker\varphi \arrow[ul, "0"] \arrow[ur, hook, "\iota"'] & \\
& K \arrow[u] \arrow[uul, bend left, "0"] \arrow[uur, hook, bend right, "\kappa"'] &
\end{tikzcd}
$$

Like frames in a graphic novel, these four diagrams should be read as a timeline, see Figure 1.1.

This pattern of "$\forall\exists\forall\exists!$" 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

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

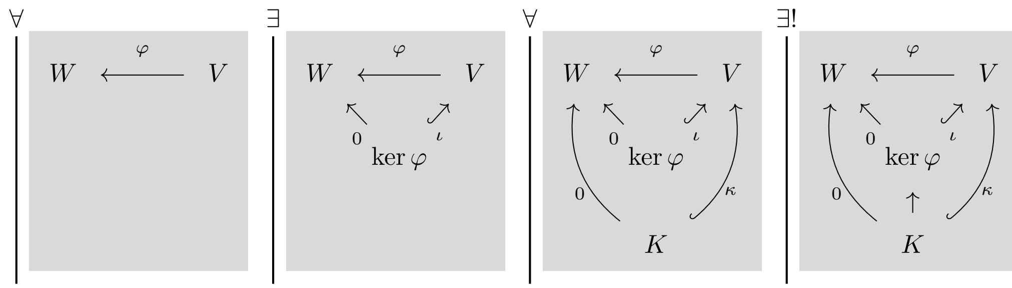

Figure 1.1: The diagram description of kernels.

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 be 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 if this tower may one day collapse, meaning that you only need to go to some fixed value of n before you know everything.

Computing the kernel

The description above achieved the idea of kernel without cleari-fying how to get it. We can return to our RREF to get that answer. 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:

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

Proposition 1.8. For every $(m \times n)$-matrix Φ over a division ring Δ, there is an invertible matrix R and a permutation matrix Σ such that

$$R\Phi\Sigma = \begin{bmatrix} I_r & M \\ 0 & 0 \end{bmatrix}.$$

Furthermore

$$\Sigma \begin{bmatrix} -M \\ I_{n-r} \end{bmatrix}$$

is a kernel map $\iota : \Delta^{n-r} \to \Delta^n$.

Proof. Augment the matrix Φ by I_m to create a tableaux $T = [I_m | \Phi]$. The *admissible* rows are initially $\{1, \ldots, m\}$. Choose an admissible row i. Scan the Φ block for a column j with $\Phi_{ij} \neq 0$. Since we are over a division ring Φ_{ij} is therefore invertible.

Select a non-zero column j in the Φ block that is non-zero.

$$\left[\Phi_{ij}^{-1} I_n\right]$$

□

```
def enteringCol():Fin[n]
```

Suppose

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

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.

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

The label $\mathsf{F}_{\mathrm{ker}}$ 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 $(\mathsf{F}_{\mathrm{ker}})$ 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
```

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.

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

The $\mathrm{I}_{\mathrm{ker}}$ 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:

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

Listing 1.2.1 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
```

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 1.2.1 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.

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

The name $\mathsf{E}_{\mathrm{ker}}$ 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 1.2.2.

Finally we need to do some computing somewhere and we learn what to compute by inspecting the condition of “commutative di-

Listing 1.2.2 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
```

agrams".

$$\frac{k:K \qquad \kappa:K\to V \qquad pf:\varphi(\kappa(j)) =_W 0}{\iota(x) := \kappa(k)} \tag{C_{ker}}$$

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 1.2.3 provides some of the options.

Problems

1.1. *Identify the type of elementary matrix displayed below.*

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \alpha \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 42 & 0 \\ 0 & \sqrt[128]{17} \end{bmatrix} \quad D = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Listing 1.2.3 Complete data type for kernels.

```
// 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
```

1.2. *Complete the reduction of the matrix*

$$\begin{bmatrix} 4 & 12 \\ 6 & 30 \\ 27 & 24 \end{bmatrix}$$

1.3. *Find access to a computer algebra system and compute kernels of random* 1000×1200, 1000×1000 *and* 1200 *matrices.*

1.4. *Prove that any two kernels are isomorphic.*