线性代数

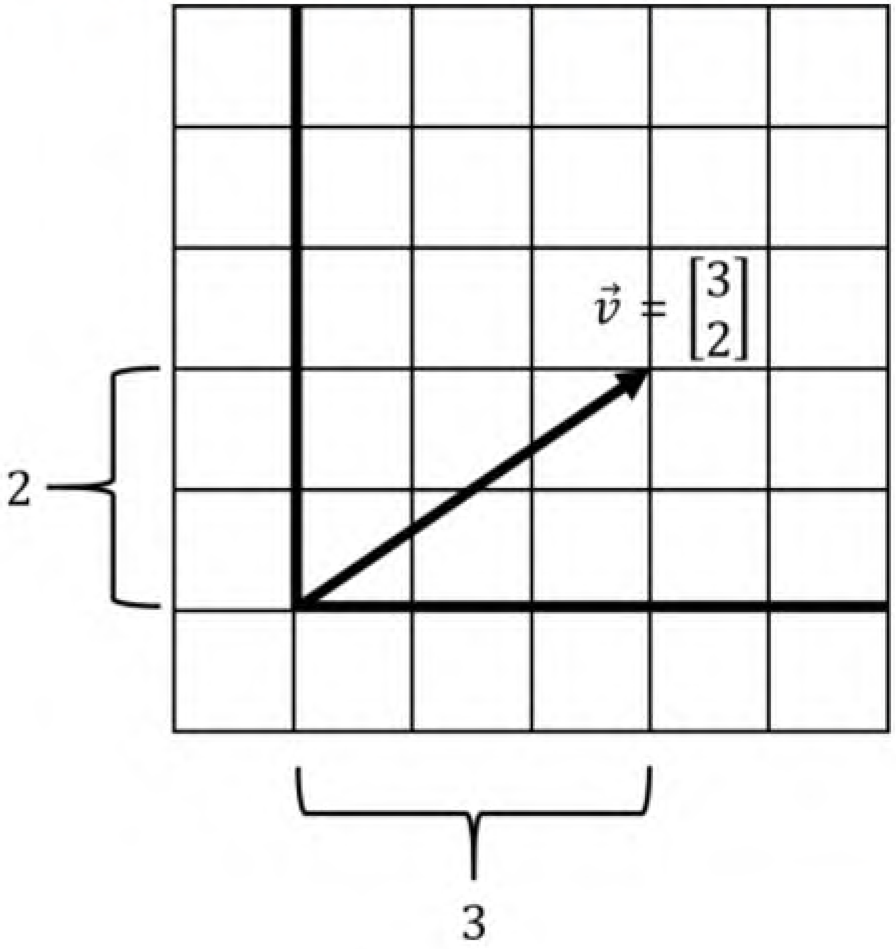
顾立平

01什么是向量？

- 有时人们将线性代数与基本代数混淆，认为它可能与使用代数函数y=mx+b绘制直线有关。这是为什么线性代数应该被称为“向量代数”或“矩阵代数”的原因：它更抽象。线性系统发挥了作用，但以一种更形而上学的方式。
- 什么是线性代数呢？线性代数关注线性系统，但通过向量空间和矩阵来表示它们。线性代数是数学、统计学、运筹学、数据科学和机器学习等许多应用领域的基础。
- 使用R或者Python可以完成许多任务，如果想在这些黑盒子后面获得直觉，并更有效地处理数据，了解线性代数的基础：向量，是不可避免的。

01什么是向量？

- 向量是空间中具有特定方向和长度的箭头，通常表示一段数据。它是线性代数的核心构建块，包括矩阵和线性变换。



- 在其基本形式中，它没有位置概念，因此始终假设其尾部从笛卡尔平面（0,0）的原点开始。如左图所示。
- 在数学上，声明一个向量：

$$\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$$

- 在Python里，声明一个向量：

```
import numpy as np
v = np.array([3, 2])
print(v)
```

01什么是向量？

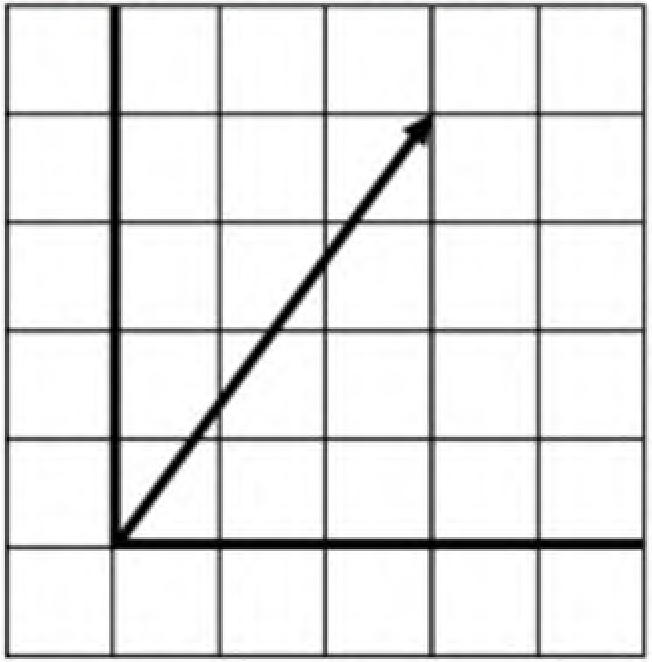
- 当我们开始使用向量进行数学计算时，特别是在执行机器学习等任务时，我们使用NumPy库，它比Python（如上所示，以列表方式处理数据）更有效。当小数变得不方便时，可以使用SymPy执行线性代数运算。
- 使用NumPy（的array（ ）函数）在Python中声明向量。

```
import numpy as np
v = np.array([3, 2])
print(v)
```

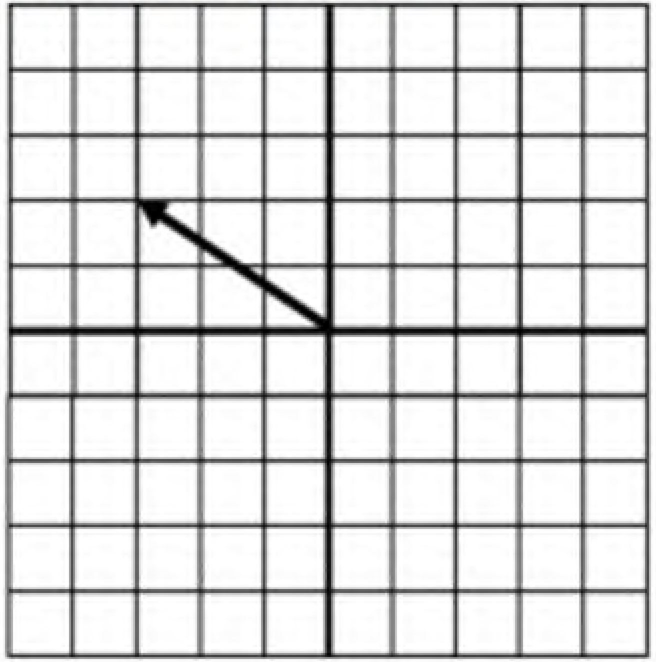
- 向量有无数的实际应用。在物理学中，向量通常被认为是方向和大小。在数学中，它是XY平面上的方向和比例，有点像运动。在计算机科学中，它是存储数据的一组数字。

01什么是向量？

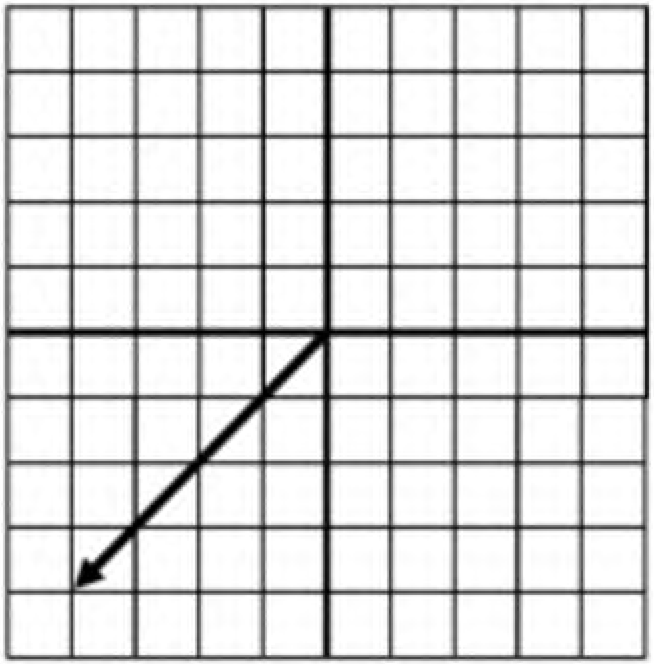
- 一些矢量在X和Y刻度上具有负方向。当我们稍后组合负方向的向量时，它们将产生影响，本质上是相减而不是相加。



$\vec{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$



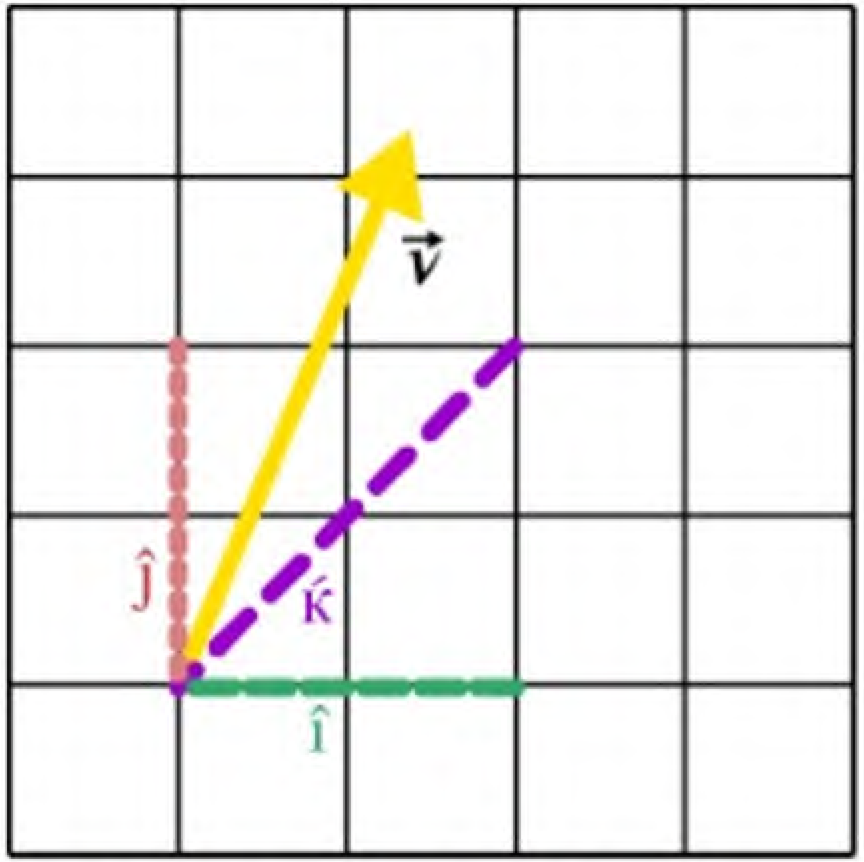
$\vec{v} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$



$\vec{v} = \begin{bmatrix} -4 \\ -4 \end{bmatrix}$

- 在进行统计和机器学习工作时，数据通常被导入并转化为数值向量，以便我们可以使用它。视频游戏和飞行模拟器也使用向量和线性代数来建模。

01什么是向量？



- 向量可以存在于两个以上的维度上。我们沿轴x、y和z声明三维向量：

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 4 \end{pmatrix}$$

- 使用NumPy在Python中声明三维向量：

```
import numpy as np
v = np.array([3, 6, 9])
print(v)
```

- 如图所示，创建这个向量，在x方向上单步执行四个步骤，一个在它们的y方向上，两个在z方向上，将之可视化。
- 此处，不再在二维网格上显示向量，而是具有三个轴的三维空间：x、y和z啦

01什么是向量？

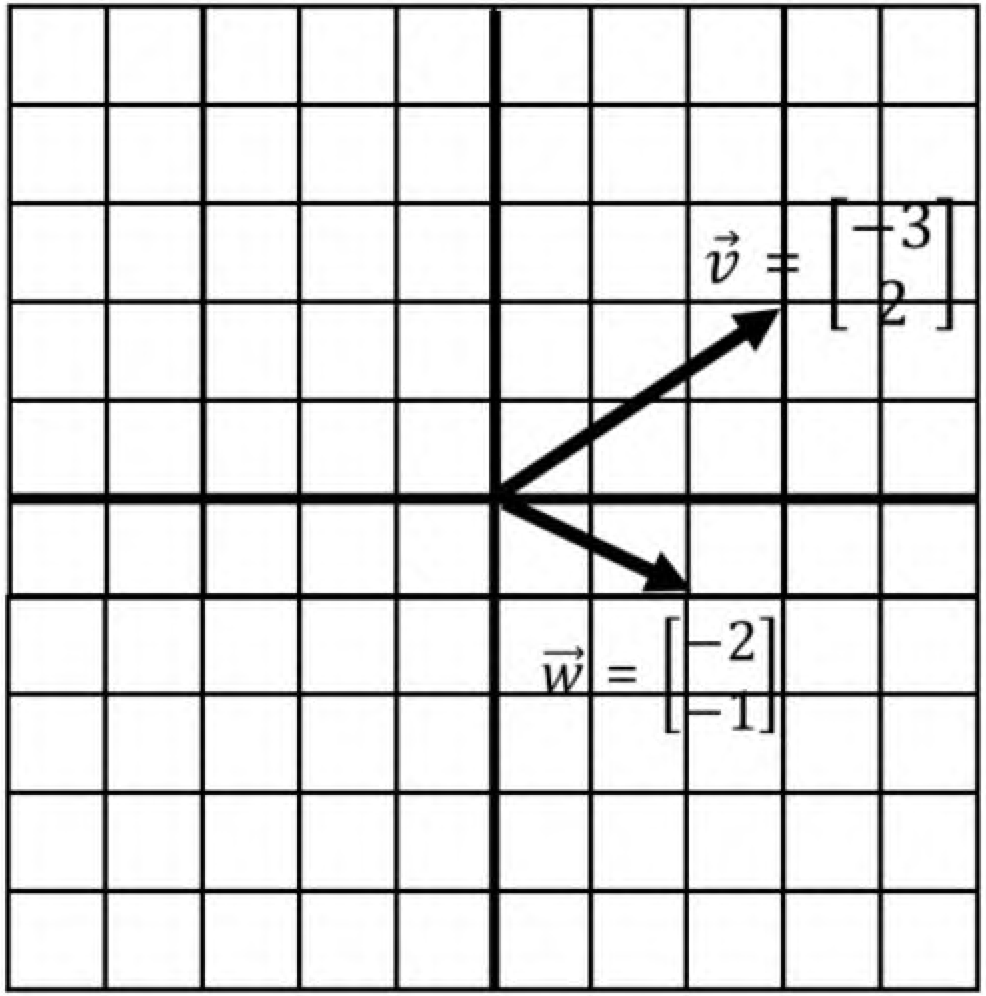
- 像许多数学模型一样，可视化三维以上是一项挑战。但从数字上讲，它仍然很简单。

$$\vec{v} = \begin{pmatrix} u \\ w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \\ 6 \\ 4 \\ 5 \end{pmatrix}$$

- 使用NumPy在Python中声明五维向量

```
import numpy as np
v = np.array([3, 1, 6, 4, 5])
print(v)
```

02添加和组合向量



- 向量表达了方向和大小，有点像空间中的运动。
- 两个向量的运动组合为单个向量，称为向量加法。
- 组合两个向量，包括它们的方向和比例，从数值上，这很简单，只需将相应的x值，然后将y值添加到新向量中。

02添加和组合向量

$$\vec{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$\vec{w} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

$$\vec{v} + \vec{w} = \begin{bmatrix} 3+2 \\ 2+\ -1 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

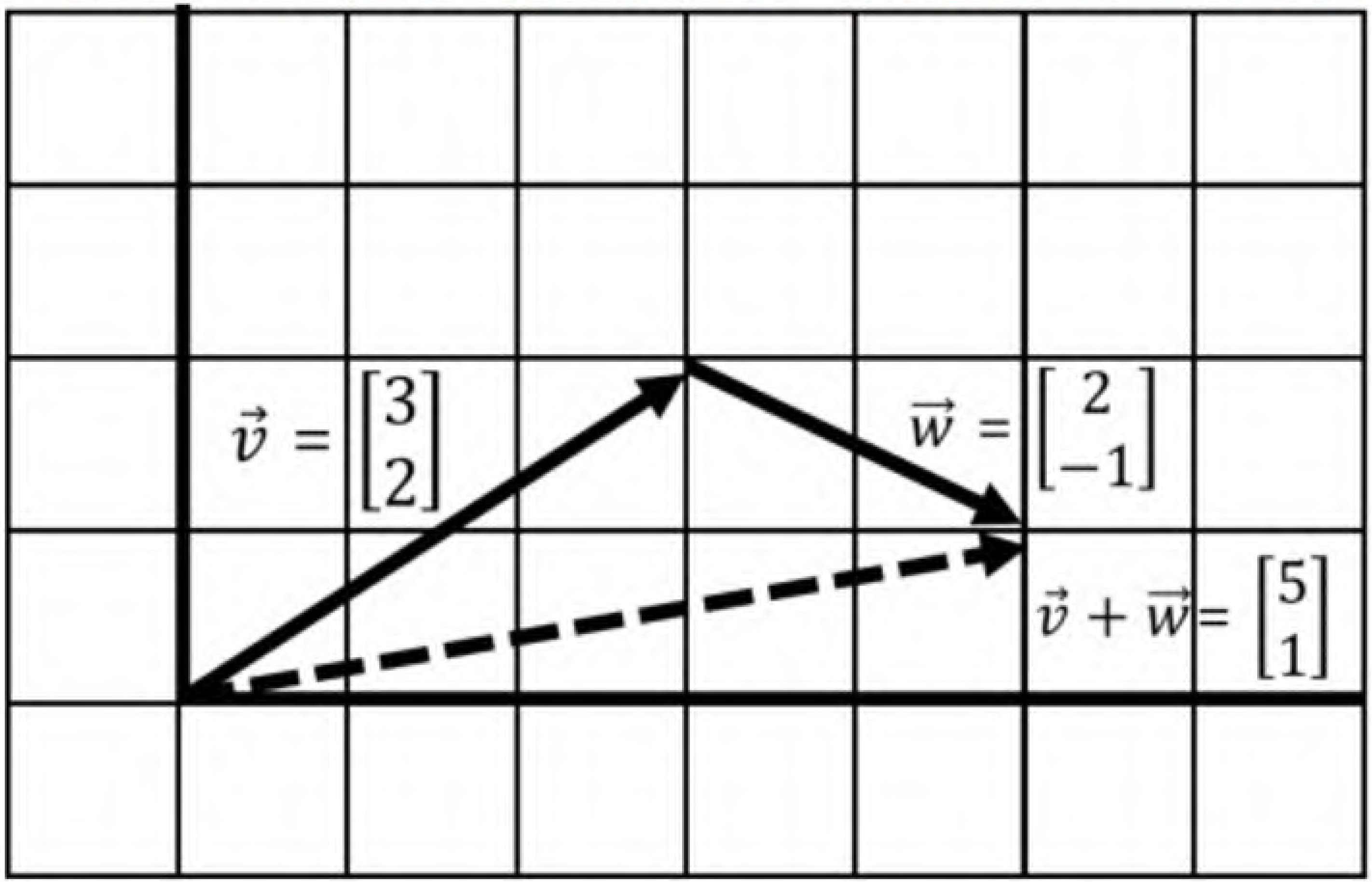
- 使用NumPy在Python中组合两个向量

```
from numpy import array
v = array([3,2])
w = array([2,-1])
# 向量（矢量）之和
v_plus_w = v + w
# 显示组合后的向量（打印结果啦）
print(v_plus_w) # [5, 1]
```

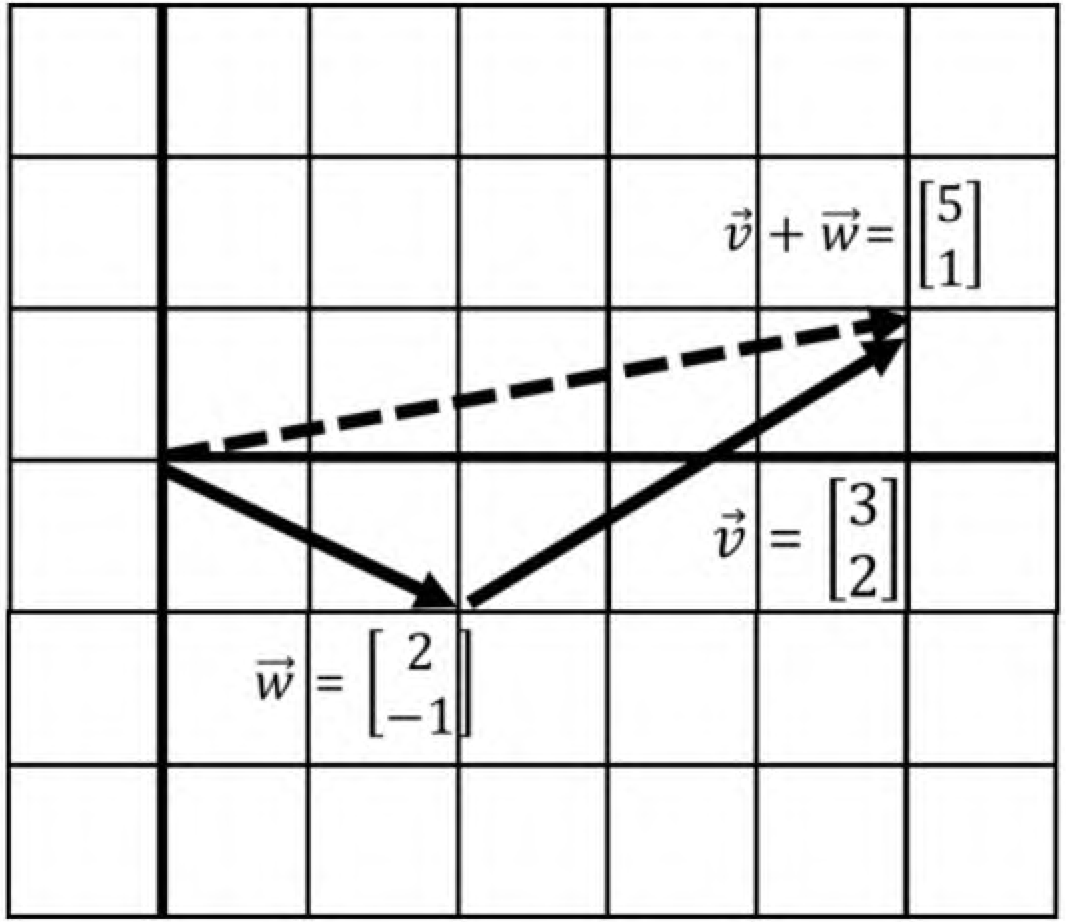
02添加和组合向量

- 在视觉上，直观地将这两个向量添加到一起，请把一个向量连接到另一个向量，然后走到最后一个向量的尖端。结束点是一个新向量，是两个向量之和的结果。

- 当我们走到最后一个向量$\vec{w}$的末尾时，我们最终得到一个新的向量[5,1]



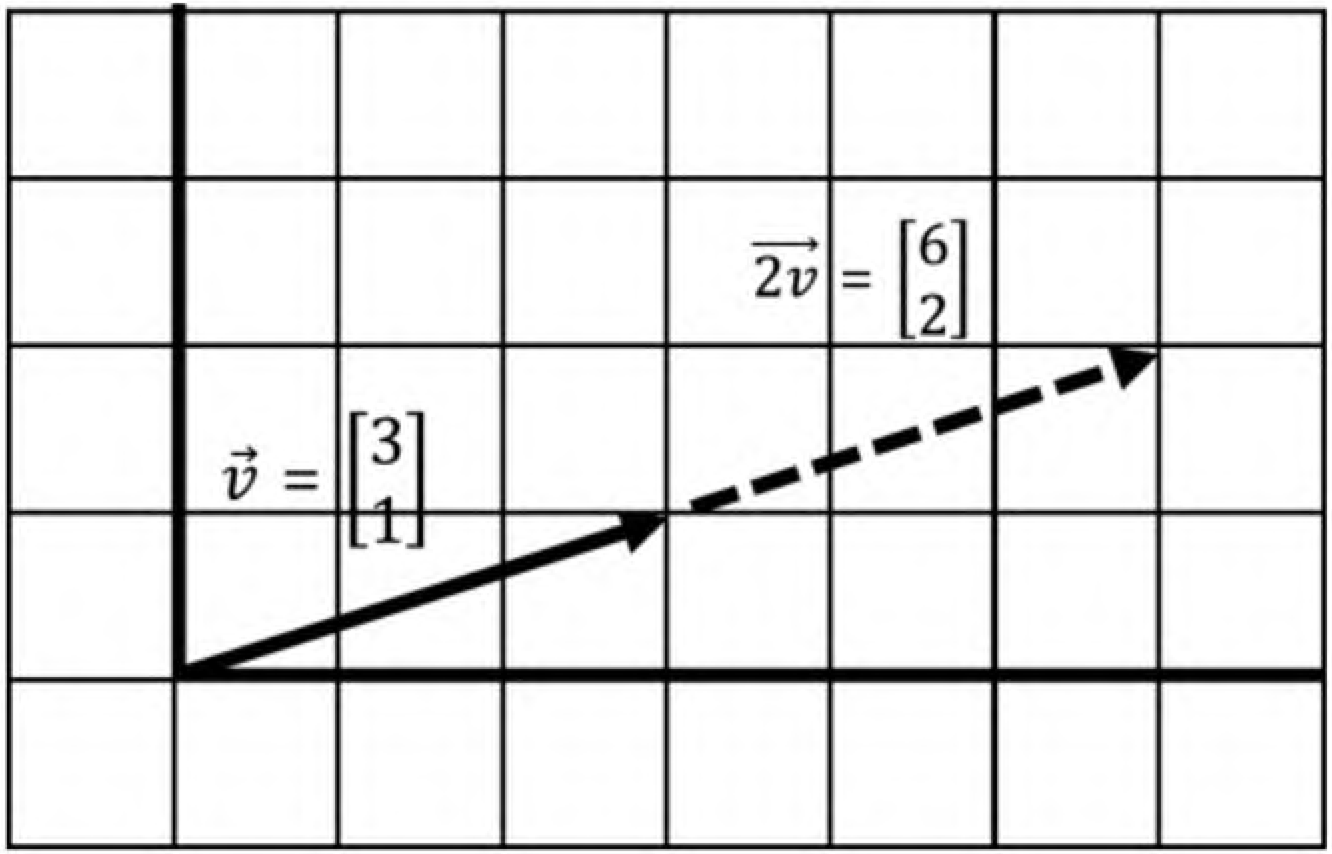
02添加和组合向量



- 新向量是v和w相加的结果。在实践中这可以简单地将数据相加在一起。
- 注意到：在$\vec{w}$之前添加$\vec{v}$或反之亦然，这意味着它是可交换的，操作顺序无关紧要。（比较一下上一张图和左图）
- 如果我们在$\vec{v}$之前走$\vec{w}$最终得到与上一张图所示的相同结果向量[5，1]。

03缩放向量（Scaling Vectors）

- 缩放是增加或缩小向量的长度。把单个值做相乘或缩放，称为标量。如下图所示，向量$\vec{v}$ 按因子2缩放，使其加倍。



03缩放向量（Scaling Vectors）

- 在Python中执行此缩放操作就像将向量乘以标量一样简单。

$$\vec{v} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

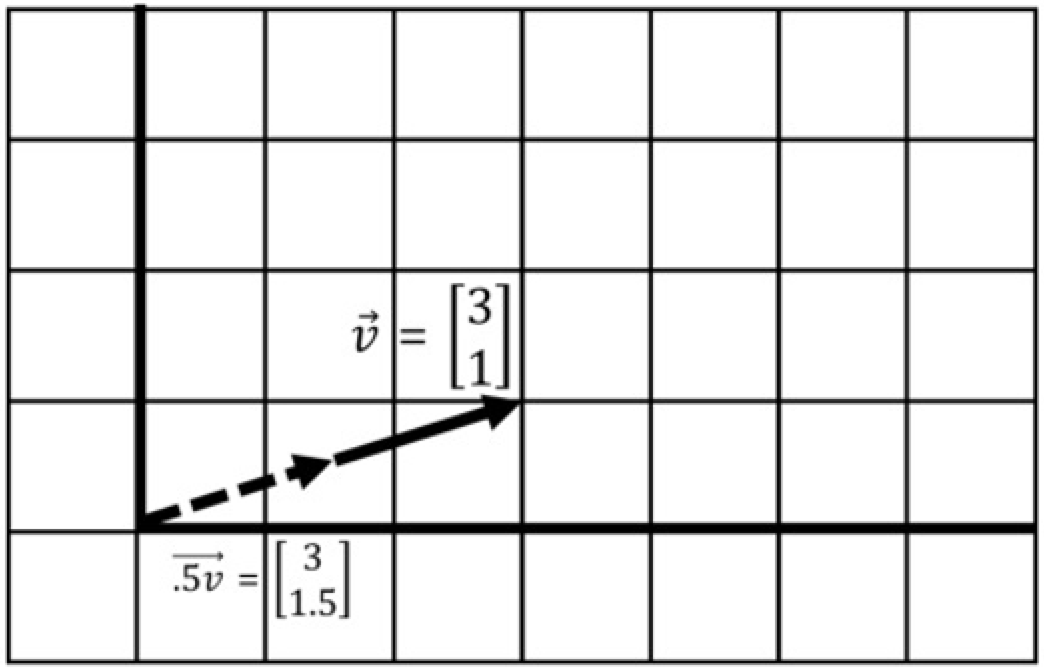
$$2\vec{v} = 2\begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \times 2 \\ 1 \times 2 \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \end{bmatrix}$$

- 使用NumPy在Python中缩放数字

```
from numpy import array
v = array([3,1])
# 缩放向量
scaled_v = 2.0 * v
# 显示缩放后的向量
print(scaled_v)
# 结果会是[6 2]
```

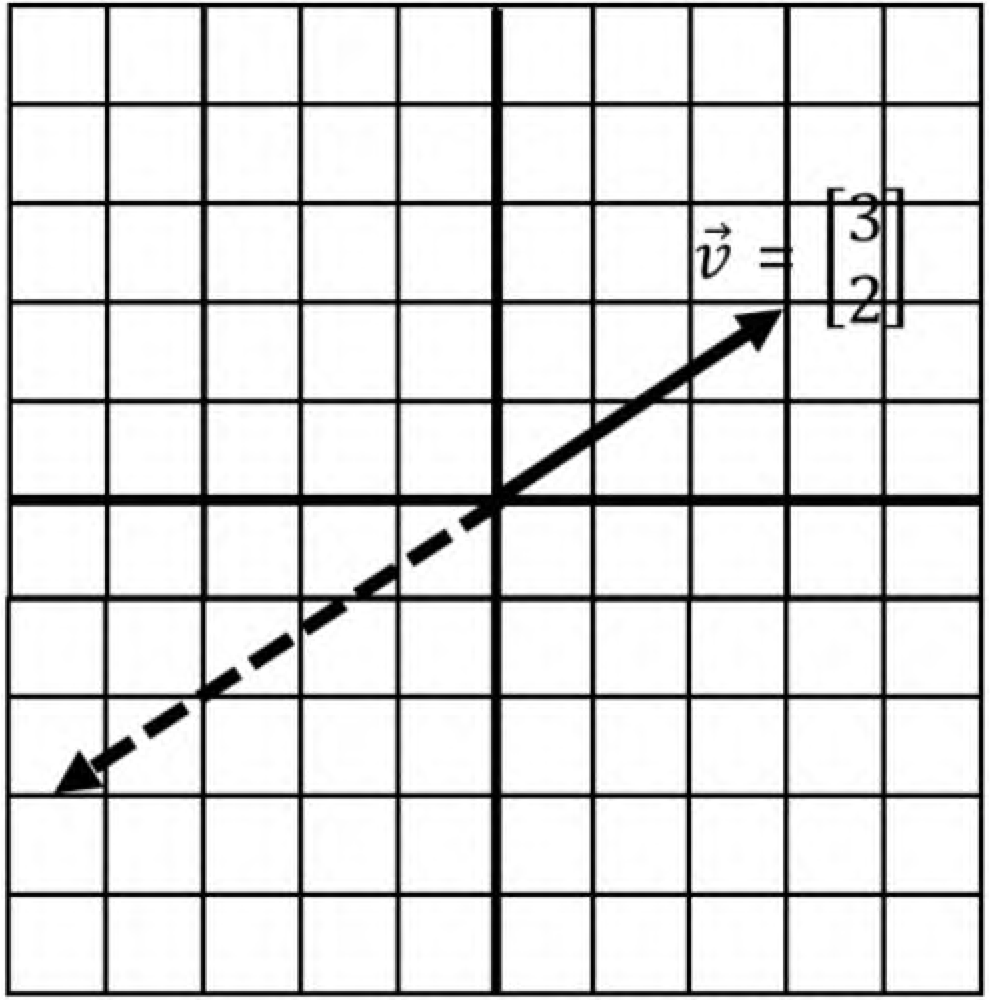
03缩放向量（Scaling Vectors）

- 同理，可把向量缩小。如图所示，被缩小了0.5倍，这是原来的一半。



- 每个数据操作都可以用向量来考虑，甚至是一些平均值。
- 以缩放为例。假设我们试图获得整个社区的平均房屋价价格和平均平方公尺。首先把向量相加，分别组合它们的价格值和平方尺，得到一个包含总值和总平方尺的巨大向量。然后，通过除以房屋数量N来缩小向量，这实际上是乘以1/N。这样就有了一个包含平均房屋价格和平均平方公尺的向量。

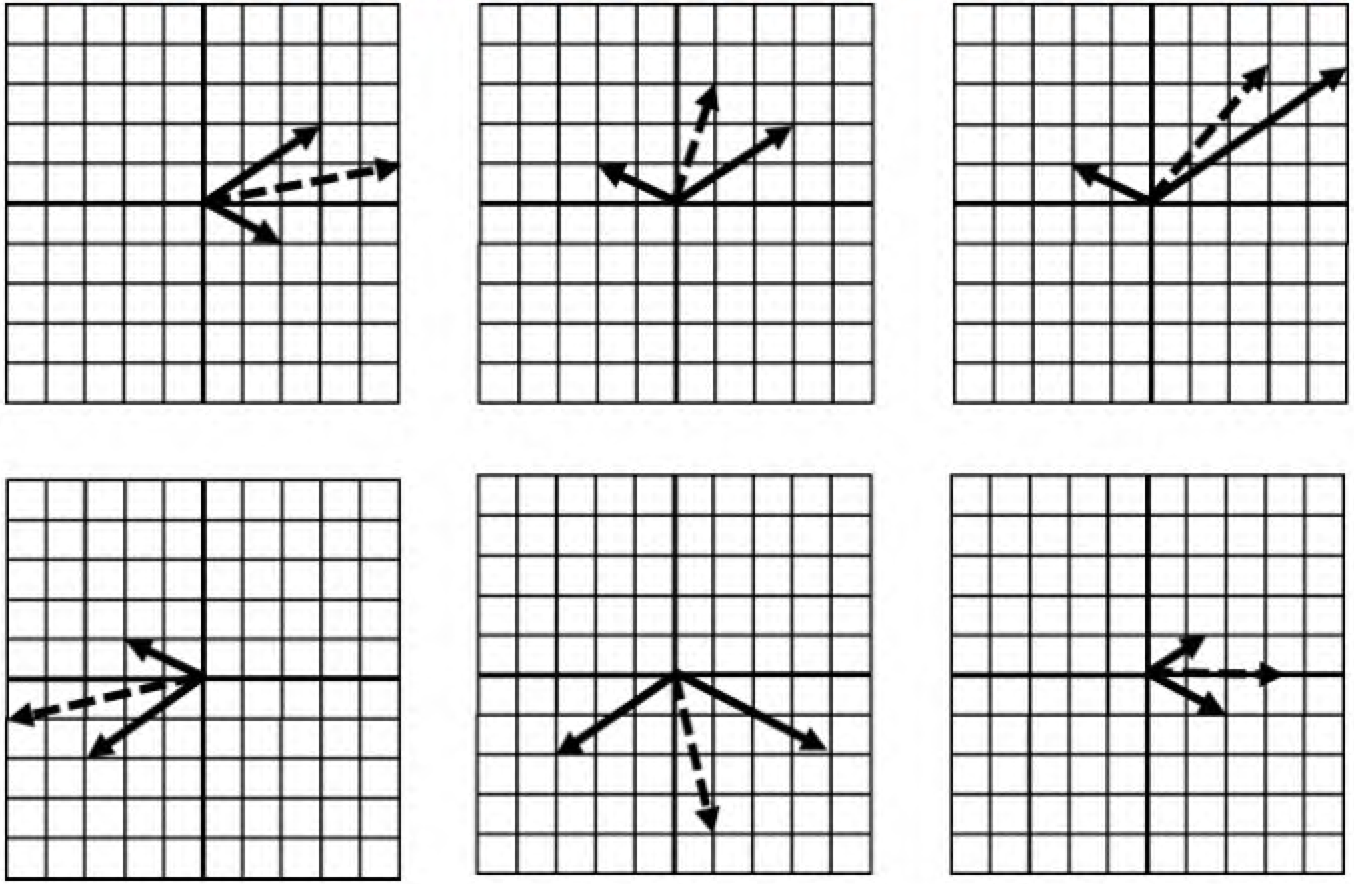
03缩放向量（Scaling Vectors）



- 请注意：缩放向量不会改变其方向，只会改变其大小。
- 再注意：该规则有一个轻微的例外。将向量乘以负数时，它会翻转向量的方向，如图所示。
- 不过，按负数缩放并没有真正改变方向，因为它仍然存在于同一条线上。这涉及到一个称为【线性相关性】的概念。

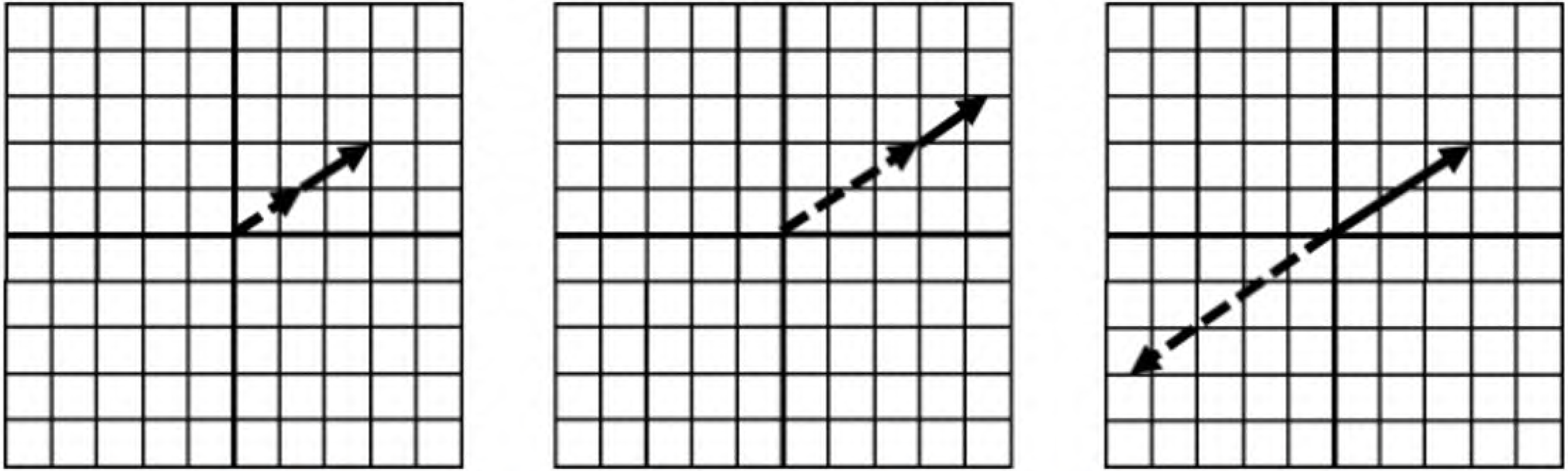
04跨度和线性相关性

- 上述两个操作：添加两个向量并缩放它们，产生了一个简单但是强大的想法。即：我们可以组合两个向量并缩放它们，以创建任何结果向量！



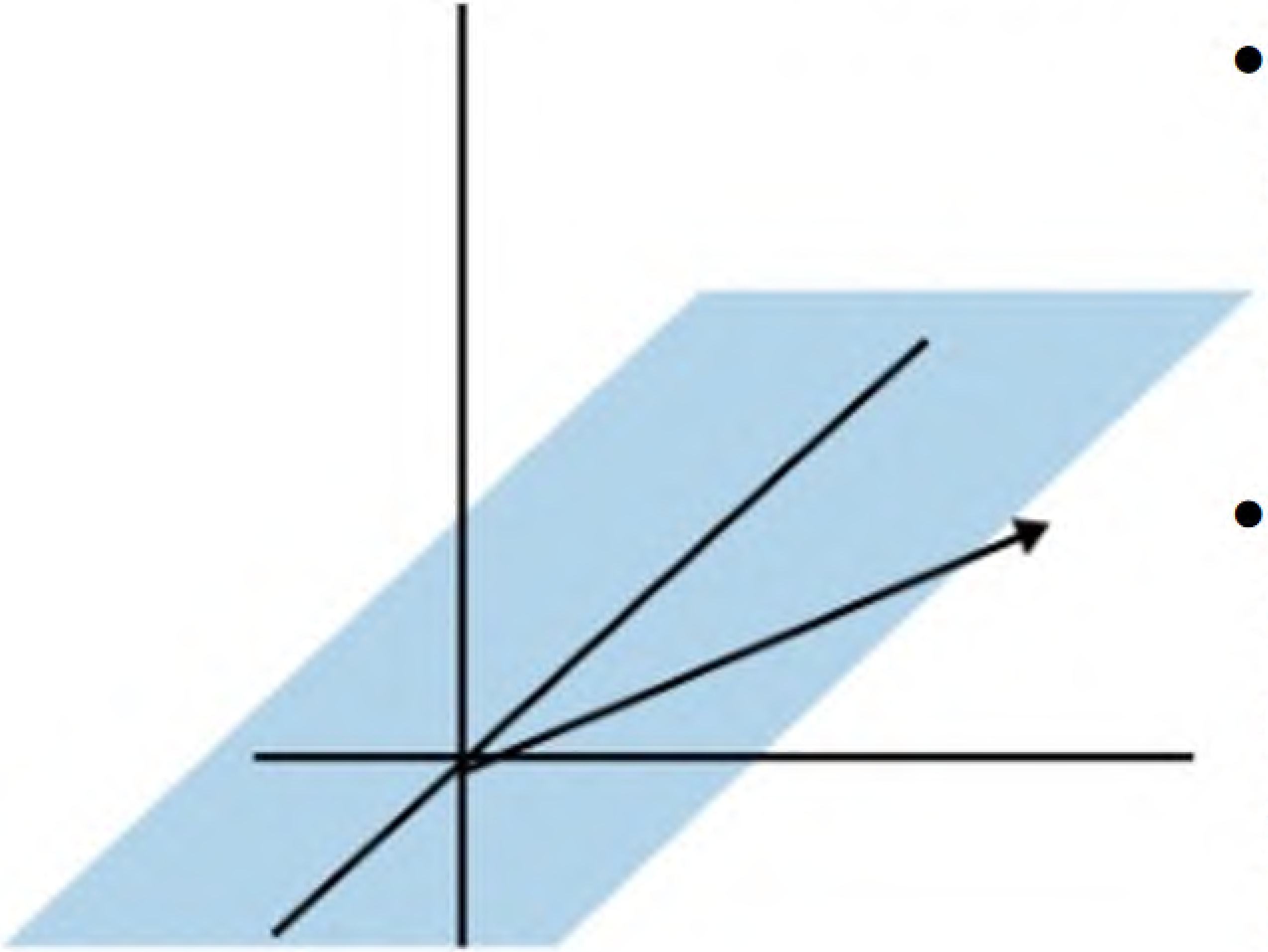
- $\vec{v}$和$\vec{w}$在方向上是固定的，除了用负标量翻转，还可以使用缩放来自由创建由$\overrightarrow{v+w}$组成的任何向量。
- 这个可能向量的整个空间称为【跨度】，在多数情况下，跨度通过缩放和求和，可从这两个向量创建无限向量。

04跨度和线性相关性



- 当我们在两个不同的方向上，有两个向量时，它们是【线性独立】的，并且具有这个无限跨度。
- 当两个向量存在于同一方向上或存在于同一线上时，这些向量的组合会卡在同一条线上，将我们的跨度【限制】在该线上。无论如何缩放它，生成的和向量也会卡在同一条线上。这使得它们线性相关，如上三图所示。

04跨度和线性相关性

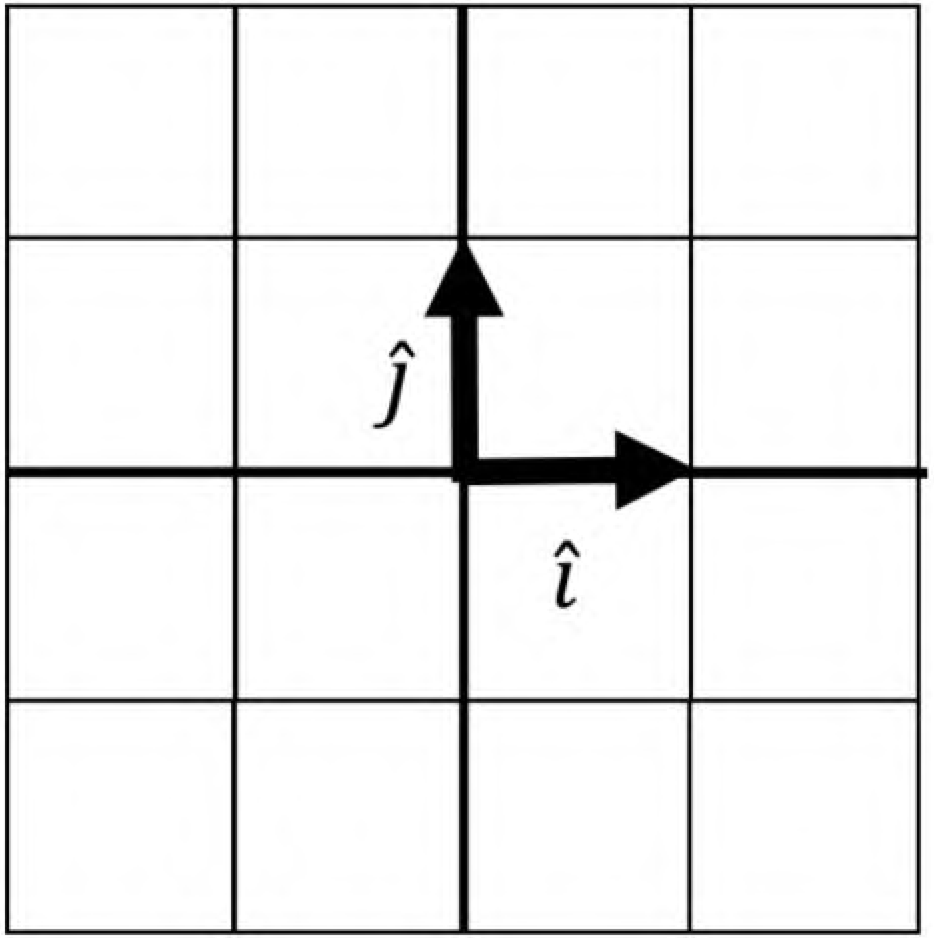


- 在三个或更多的维度中，当我们有一组线性相关的向量时，我们通常会在较少的维度中卡在平面上。如图所示，一个三维向量被卡在二维平面上。
- 许多问题在线性相关时，会变得困难或无法解决。例如，线性相关的方程组可能会导致变量消失。但如果您具有线性独立性，那么从两个或多个向量创建所需的任何向量的灵活性对于解决方案来说是无价的！

05线性变换

- 将两个方向固定的向量相加，但缩放它们以获得不同的组合向量。
- 这个组合向量，除了线性相关的情况外，可以指向任何方向，并且具有我们选择的任何长度。
- 这为线性变换提供了一种直觉，其中我们使用一个向量以类似函数的方式变换另一个向量。

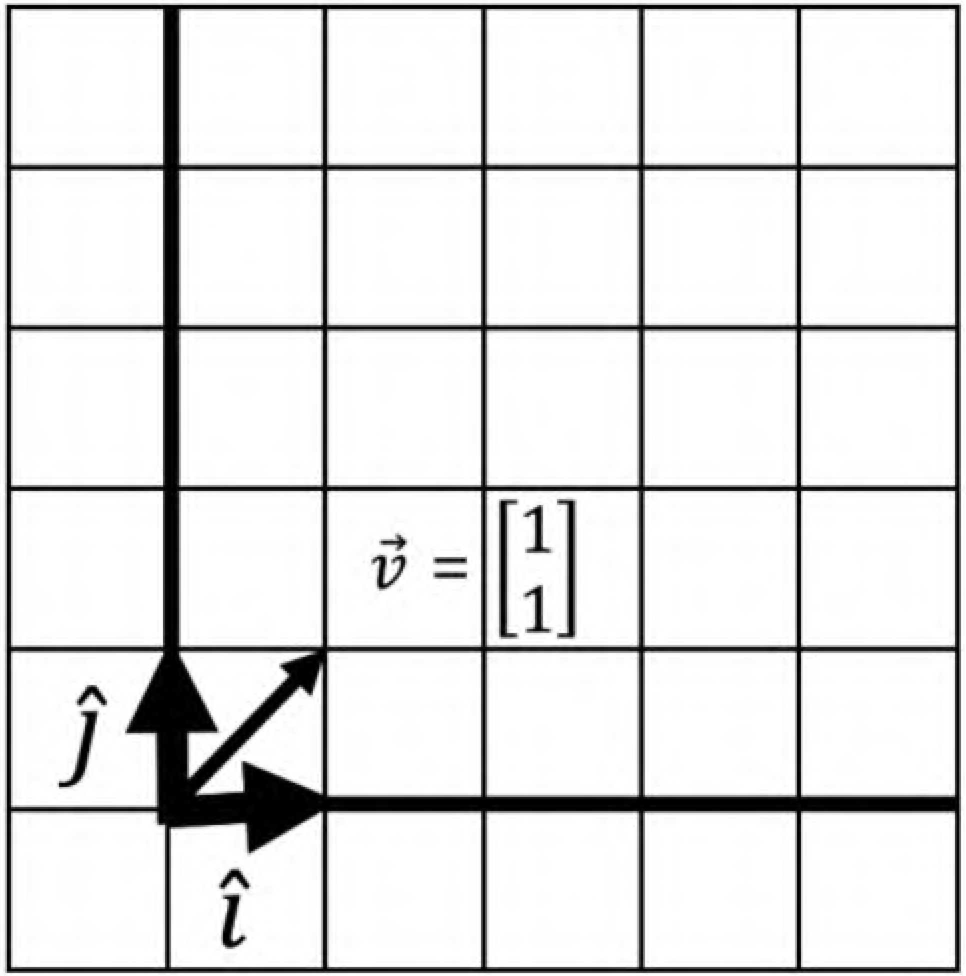
06基本向量（Basis Vectors）



- 假设有两个简单的向量$\hat{i}$和$\hat{j}$（“i-hat”和“j-hat”）。这些被称为基向量，用于描述其他向量上的变换。它们通常具有1的长度，并指向垂直的正方向，如左图所示。

$$\hat{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \hat{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \text{basis} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

06基本向量（Basis Vectors）

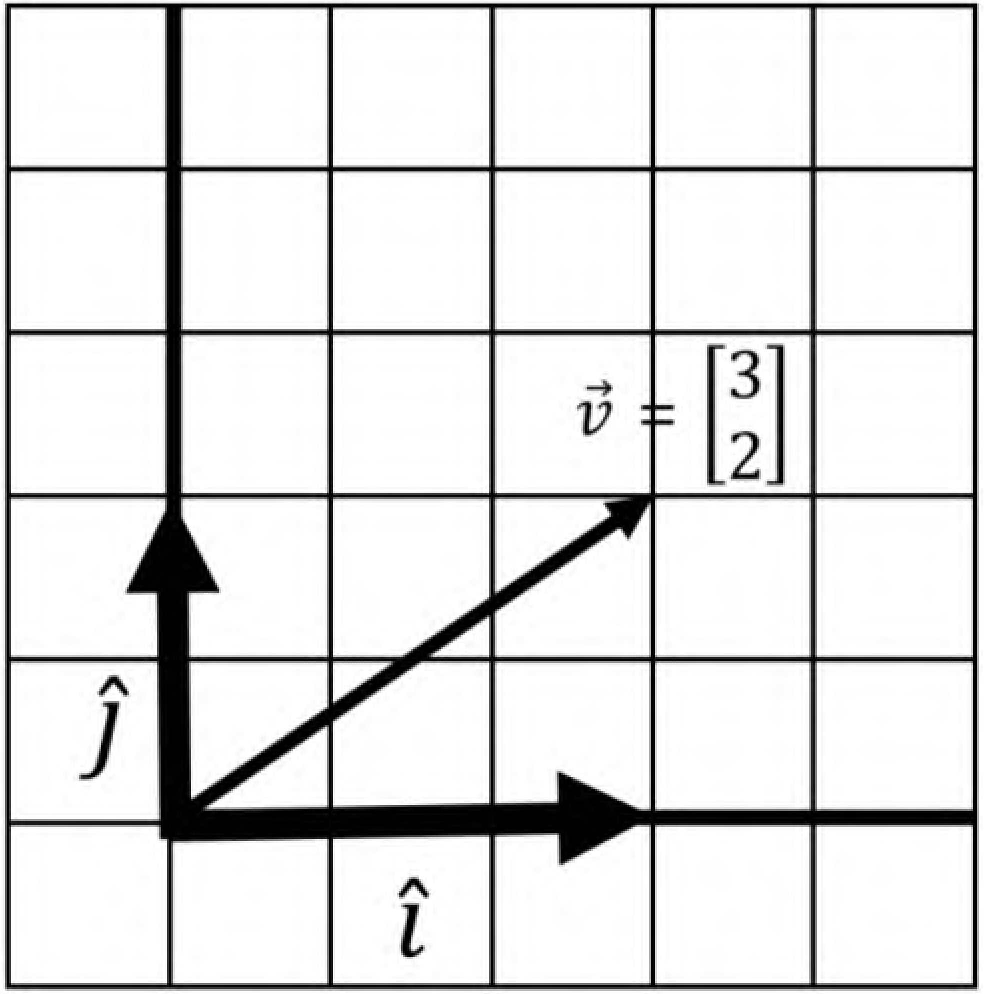


- 矩阵是向量（例如$\hat{\imath}$和$\hat{J}$的集合，该向量可以具有多行和列，是打包数据的方便方法。我们可以使用$\hat{\imath}$和$\hat{\jmath}$来创建任何通过缩放和相加得到的向量。
- 如果想要向量$\vec{v}$落在[3,2]，先把$\hat{\imath}$拉伸3倍，把$\hat{\jmath}$拉伸2倍。

$$3\hat{i} = 3\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}3\\0\end{bmatrix}$$

$$2\hat{j} = 2\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\2\end{bmatrix}$$

06基本向量（Basis Vectors）

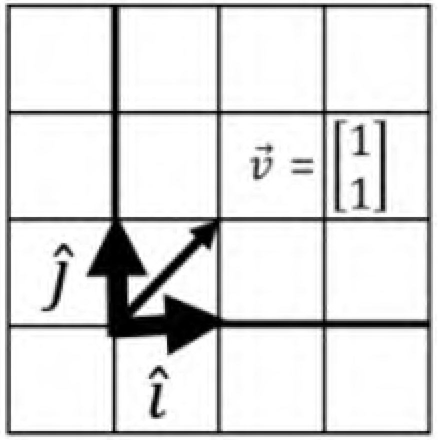


- 基本向量的变换，具有：拉伸、挤压、剪切或旋转的向量运动。
- 如图所示，缩放Î和Ĵ沿着向量$\vec{v}$拉伸了空间。这有被称为线性变换。
- 向量$\vec{v}$由Î和Ĵ相加组成。因此，我们只需将拉伸的Î和Ĵ相加，就可以看到向量$\vec{v}$落在何处。

$$\vec{v}_{new} = \begin{bmatrix} 3 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

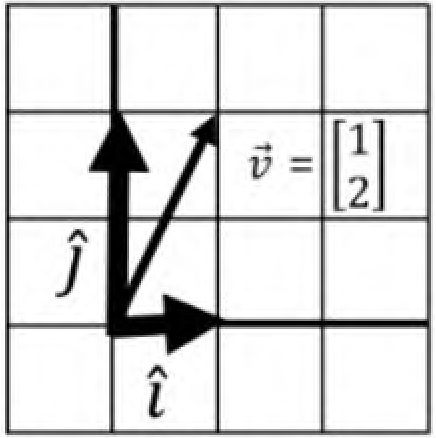
06基本向量（Basis Vectors）

- 通过线性变换，可以实现四种运动，如图所示。

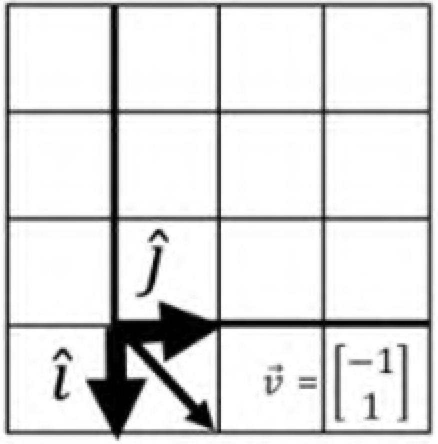


BASIS

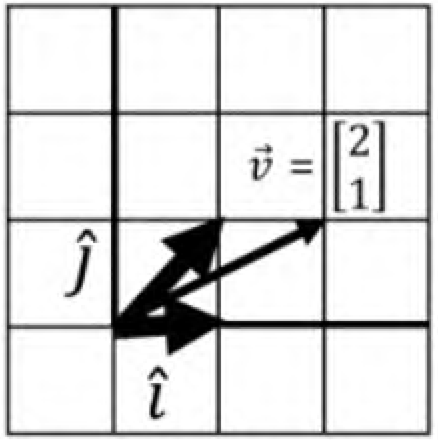
- 这四个线性变换是线性代数的核心部分。缩放矢量将拉伸或挤压它。旋转将改变矢量空间，反转将翻转向量空间，以便Î和Ĵ交换各自的位置。



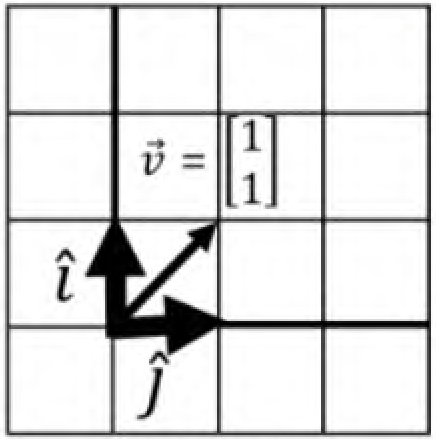
SCALE



ROTATE



SHEAR



INVERSION

- 注意到：不能有非线性的变换，导致不再遵守直线的曲线或曲线变换。

07矩阵与向量相乘

- 变换后的Î和Ĵ非常重要，因为它不仅允许我们创建向量，还允许我们变换现有向量。
- 创建向量和转换向量实际上是一回事，考虑到基础向量是变换前后的起点，这一切都是相对的。
- 将向量$\vec{v}$转换为给定封装为矩阵的基本向量Î和Ĵ的公式为：

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dy & cf + dh \end{bmatrix}$$

- Î是第一列[a，c]，Ĵ是列[b，d]。把这两个基本向量打包为矩阵，该矩阵也表示为二维或多维数网格的向量的集合。
- 通过应用基本向量，对向量进行的这种变换，称为矩阵-向量乘法。

07矩阵与向量相乘

- 要使用NumPy在Python中执行此转换，我们需要声明基本向量作为矩阵，然后使用运算符dot（ ）将其应用于向量$\vec{v}$。这就是所谓的【点积】。

```
from numpy import array
basis = array(
    [[3, 0],
    [0, 2]]
  )
v = array([1,1])
new_v = basis.dot(v)
print(new_v)
```

07矩阵与向量相乘

- 分解基本向量，然后组合成一个矩阵。注意到：此时需要换位或交换列和行。这是因为NumPy的array（ ）函数将会执行相反方向，把每个向量填充为行而不是列。

```
from numpy import array
i_hat = array([2, 0])
j_hat = array([0, 3])
basis = array([i_hat, j_hat]).transpose()
v = array([1,1])
new_v = basis.dot(v)
print(new_v)
```

07矩阵与向量相乘

- 向量$\vec{v}$从[2,1]开始，而$\hat{\imath}$和$\hat{\jmath}$分别从[1,0]和[0,1]开始。然后将$\hat{\imath}$和$\hat{\jmath}$转换为[2,0]和[0,3]。
- 用数学方法手工计算得出：

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} (2)(2) + (0)(1) \\ (2)(0) + (3)(1) \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

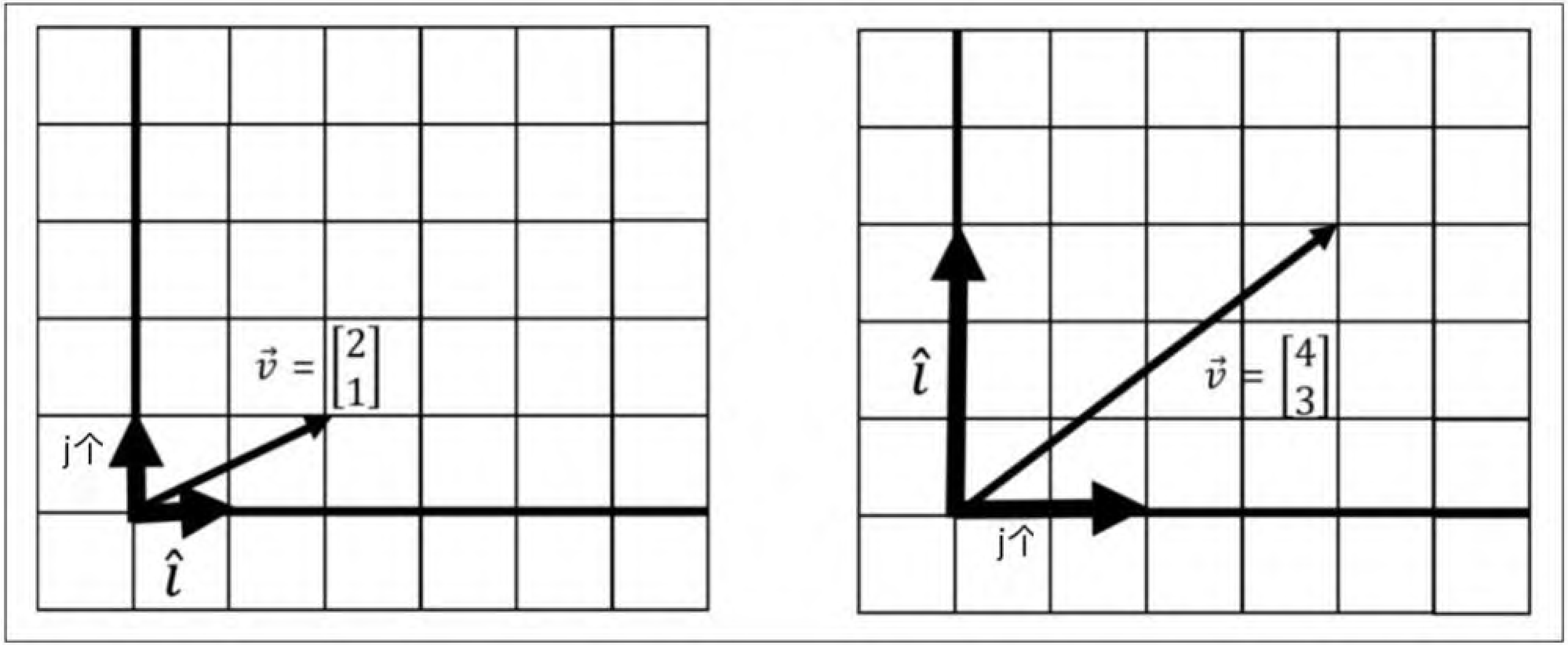
07矩阵与向量相乘

- 使用NumPy变换向量：

```
from numpy import array
i_hat = array([2, 0])
j_hat = array([0, 3])
#使用i-hat和j-hat组合基本矩阵还需要将行转置为列。
basis = array([i_hat, j_hat]).transpose()
v = array([2,1])
new_v = basis.dot(v)
print(new_v)
```

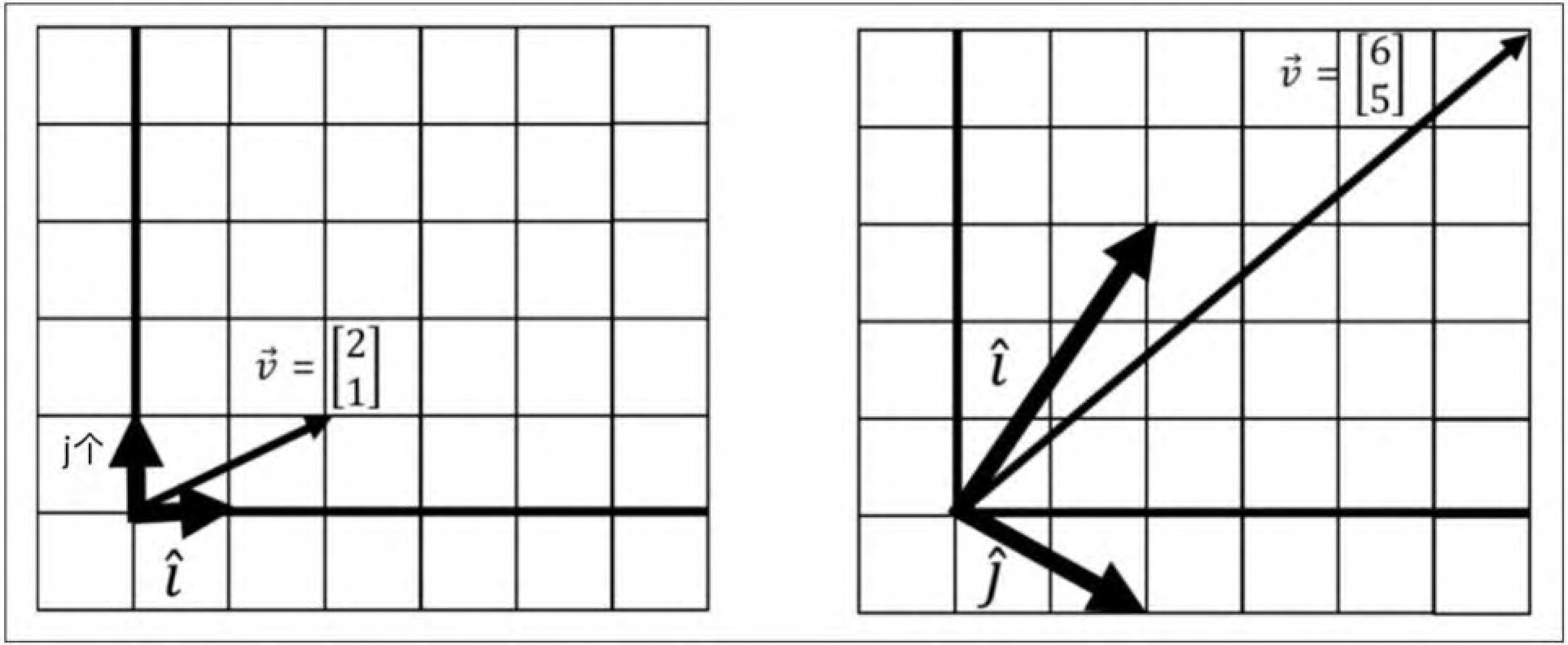
07矩阵与向量相乘

- 向量$\vec{v}$现在位于[4，3]。拉伸线性变换：



07矩阵与向量相乘

- 取向量$\vec{v}$的值[2,1]。Î和Ĵ从[1,0]和[0,1]开始，但随后被转换并降落在[2,3]和[2，-1]。对空间进行旋转、剪切和平移的线性变换：

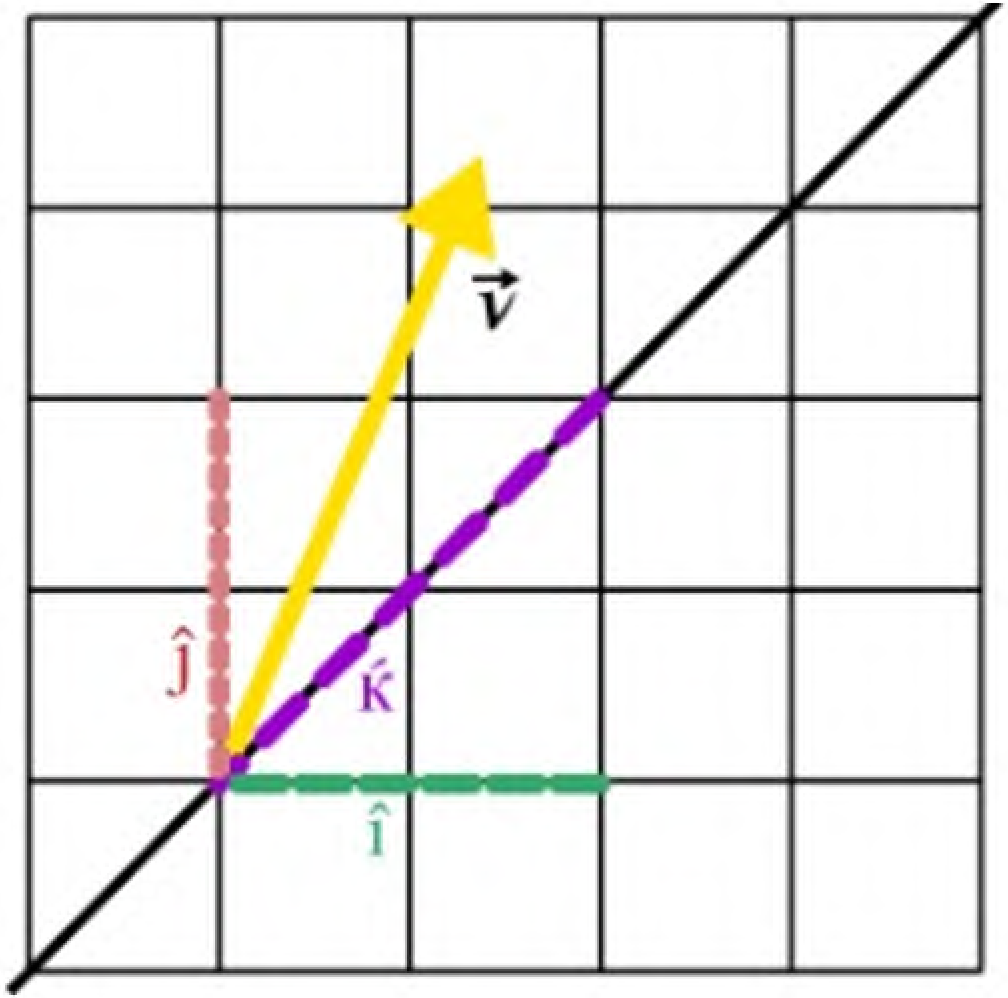


07矩阵与向量相乘

- 更复杂的转换：不仅缩放了$\hat{\imath}$和$\hat{\jmath}$还拉长了向量$\vec{v}$。实际上也剪切、旋转和翻转空间。

```
from numpy import array
i_hat = array([2, 3])
j_hat = array([2, -1])
basis = array([i_hat, j_hat]).transpose()
v = array([2,1])
new_v = basis.dot(v)
print(new_v)
```

08矩阵乘法



- 三维或者更高维度的基本向量：如果有三维向量空间，那么基本向量î，ĵ和ḱ只是不断从字母表中为每个新维度添加更多的字母。

- 一些线性变换可以将向量空间转换为较少或较多的维数，这正是非方矩阵的作用（其中，行数和列数不相等）。

08矩阵乘法

- 矩阵乘法可以想象为将多个变换应用于向量空间。每个转换都像一个函数，我们首先应用最内部的转换，然后向外应用每个后续的转换。

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- 通过使用下列公式来合并上述两个转换。采取“上下！上下！”模式把第一个矩阵的每一行相乘并相加到第二个矩阵的每个相应列。

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dy & cf + dh \end{bmatrix}$$

08矩阵乘法

- 因此，把两个单独的变换（旋转和剪切）合并为单个变换：

$$\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

$$=\begin{bmatrix}(1)(0)+(1)(1) & (-1)(1)+(1)(0)\\(0)(0)+(1)(1) & (0)(-1)+(1)(0)\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

$$=\begin{bmatrix}1 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

08矩阵乘法

- **合并两个转换**

```
from numpy import array
i_hat1 = array([0, 1])
j_hat1 = array([-1, 0])
transform1 = array([i_hat1, j_hat1]).transpose()
i_hat2 = array([1, 0])
j_hat2 = array([1, 1])
transform2 = array([i_hat2, j_hat2]).transpose()
combined = transform2 @ transform1
print("COMBINED MATRIX:\n {}".format(combined))
v = array([1, 2])
print(combined.dot(v))
```

08矩阵乘法

- 反向应用变换：把每个转换视为一个函数，把它们从最里面应用到最外面，就像嵌套函数调用一样。

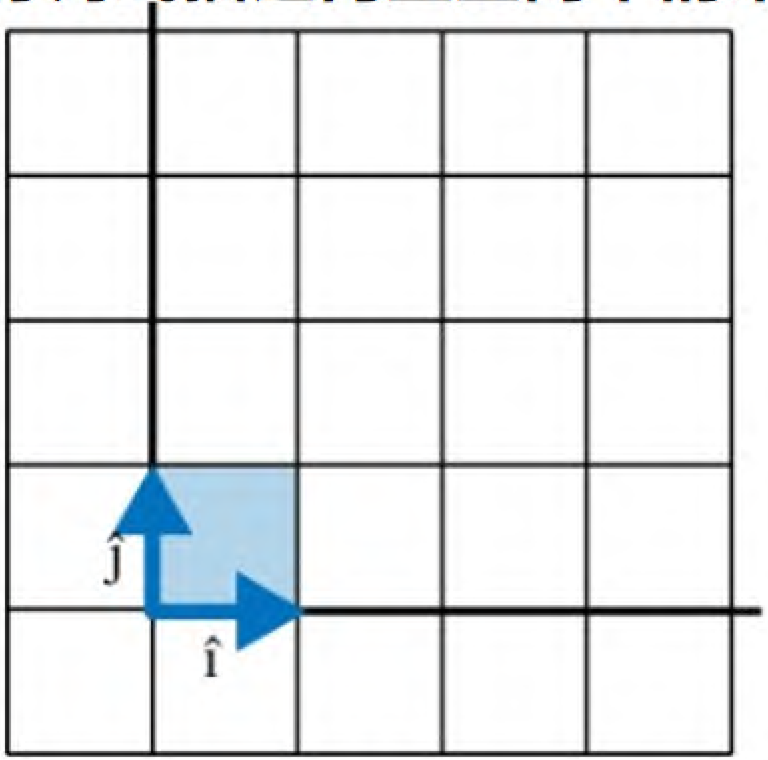
```
from numpy import array
i_hat1 = array([0, 1])
j_hat1 = array([-1, 0])
transform1 = array([i_hat1, j_hat1]).transpose()
i_hat2 = array([1, 0])
j_hat2 = array([1, 1])
transform2 = array([i_hat2, j_hat2]).transpose()
combined = transform1 @ transform2
print("COMBINED MATRIX:\n {}".format(combined))
v = array([1, 2])
print(combined.dot(v))
```

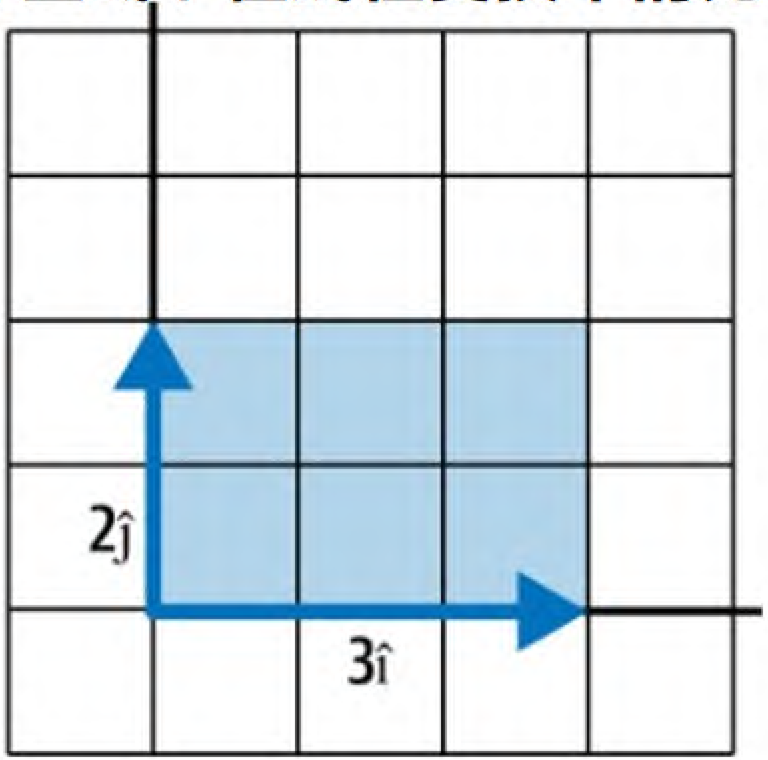
08矩阵乘法

- 线性变换和矩阵与数据科学有什么关系？
- 从导入数据到使用线性回归、逻辑回归和神经网络进行数值运算，线性变换是数学处理数据的核心。
- 然而，在实践中，我们很少花时间将数据以几何方式可视化为向量空间和线性变换。
- 不过，为了理解这些做作的数值运算的作用，就应该注意几何解释！
- 我们总不能记忆数值运算模式吧！

09决定因素

- 当我们执行线性变换时，我们有时会“扩展”或“压扁”空间。
- 如下两图所示，行列式测量线性变换如何缩放了6.0倍面积。
- 行列式描述向量空间中的采样区域，在线性变换中的比例变化量。





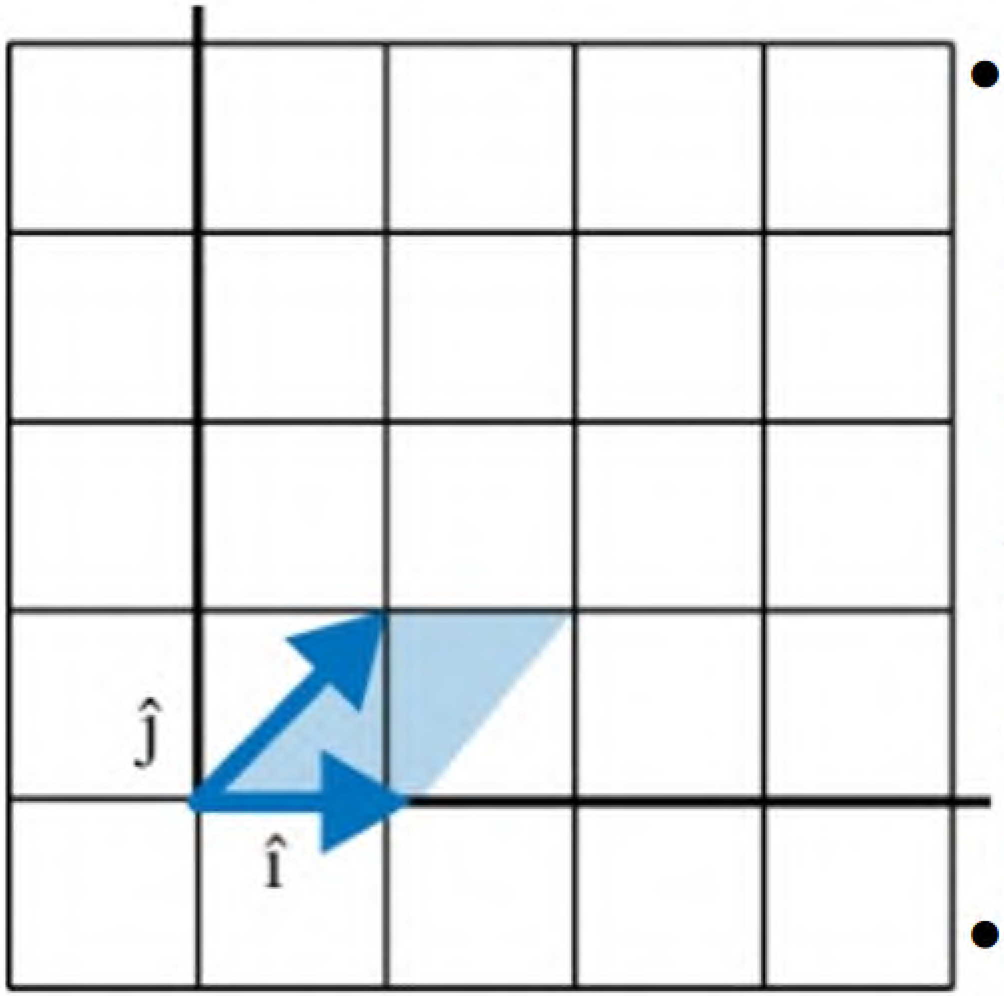
09决定因素

- 计算行列式：

```
from numpy.linalg import det
from numpy import array
i_hat = array([3, 0])
j_hat = array([0, 2])
basis = array([i_hat, j_hat]).transpose()
determinant = det(basis)
print(determinant)
```

- 简单的剪切和旋转不应影响行列式，因为面积不会改变。

09决定因素

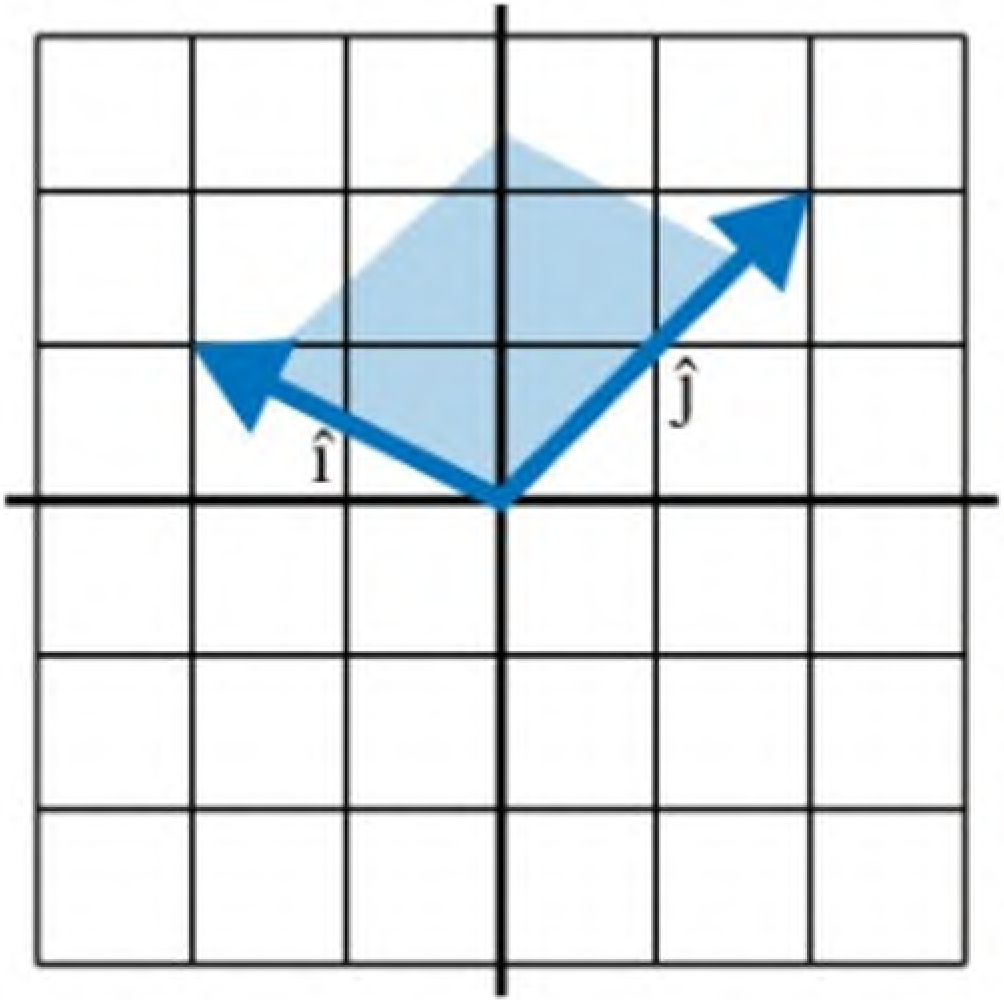


- 剪切的行列式

```
from numpy.linalg import det
from numpy import array
i_hat = array([1, 0])
j_hat = array([1, 1])
basis = array([i_hat, j_hat]).transpose()
determinant = det(basis)
print(determinant)
```

- 简单的剪切不会改变行列式

09决定因素

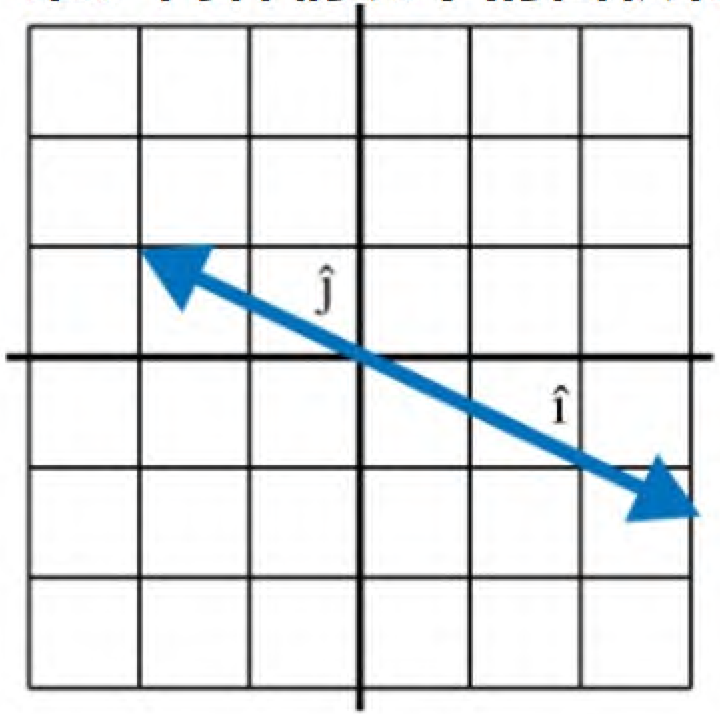


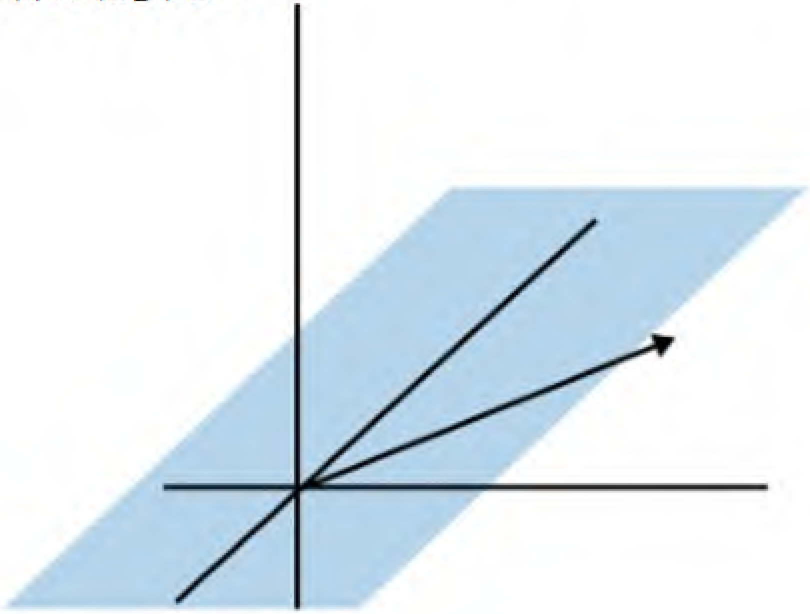
- 缩放将增加或减少行列式，因为这将增加/减少采样面积。
- 当方向翻转（î，ĵ交换顺时针位置）时，行列式将为负。

```
from numpy.linalg import det
from numpy import array
i_hat = array([-2, 1])
j_hat = array([1, 2])
basis = array([i_hat, j_hat]).transpose()
determinant = det(basis)
print(determinant)
```

09决定因素

- 目前为止，行列式最关键的信息是转换是否线性相关。如果有一个行列式0，这意味着所有的空间都被压缩到一个较小的维度。
- 如图所示，两个线性相关的变换，其中二维空间压缩为一维，三维空间压缩为二维。两种情况下的面积和体积分别为0！





09决定因素

- 零的行列式：

```
from numpy.linalg import det
from numpy import array
i_hat = array([-2, 1])
j_hat = array([3, -1.5])
basis = array([i_hat, j_hat]).transpose()
determinant = det(basis)
print(determinant)
```

- 换言之，测试0行列式，对于确定变换是否具有线性相关性非常有用。当遇到这种情况时，可能意味着有一个困难的或无法解决的问题。

10矩阵的特殊类型

- 方形矩阵（Square Matrix）是具有相等行数和列数的矩阵：它们主要用于表示线性变换，并且是许多操作（如特征分解）的要求。

$$\begin{bmatrix} 4 & 2 & 7 \\ 5 & 1 & 9 \\ 4 & 0 & 1 \end{bmatrix}$$

- 单位矩阵（Identity Matrix）：当有一个单位矩阵时，基本上撤销了一个变换，可以找到起始的基础向量了。这对方程组求解中发挥重要作用。

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

10矩阵的特殊类型

- 逆矩阵（Inverse Matrix）是撤消另一个矩阵的变换的矩阵。矩阵A的逆矩阵称为A^{-1}。

$$A = \begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{bmatrix}$$

- 在A^{-1}和A之间执行矩阵乘法时，我们最终得到一个单位矩阵。

$$\begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

10矩阵的特殊类型

- 与单位矩阵类似的是对角线矩阵（Diagonal Matrix），它具有非零值的对角线，而其余值为0。在某些计算中是可取的，因为它们表示应用于向量空间的简单标量。它出现在一些线性代数运算中。

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

- 与对角矩阵类似的是三角形矩阵（Triangular Matrix），它在三角形之前具有非零值的对角线，而其余值为0。三角矩阵在许多数值分析任务中是可取的，因为它们通常更容易在方程组中求解。它们还出现在某些分解任务中，如LU分解。

$$\begin{bmatrix} 4 & 2 & 9 \\ 0 & 1 & 6 \\ 0 & 0 & 5 \end{bmatrix}$$

10矩阵的特殊类型

- 稀疏矩阵（Sparse Matrix）：大多数为零且具有很少非零元素的矩阵。
  - 在数学上，没有什么帮助。在计算上，它们提供了创造效率的机会。如果矩阵大部分为0，则稀疏矩阵将不会浪费存储一堆0的空间，而只保留非零单元的痕迹。
  - 当有大型稀疏矩阵时，可以使用稀疏函数来创建矩阵。

$$\text{sparse:}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

11方程组和逆矩阵

- 线性代数的基本用例之一是求解方程组。假设以下等式，您需要求解x、y和z：

$$
\begin{aligned}
4x + 2y + 4z &= 44 \\
5x + 3y + 7z &= 56 \\
9x + 3y + 6z &= 72
\end{aligned}
$$

- 我们可以尝试手动尝试不同的代数运算来隔离这三个变量，但如果希望计算机解决它，则需要用矩阵来表示这个问题。将系数提取到矩阵A中，将方程右侧的值提取到矩阵B中，将未知变量提取到矩阵X中。

$$
A = \begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 44 \\ 56 \\ 72 \end{bmatrix} \qquad X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}
$$

11方程组和逆矩阵

- 线性方程组的函数为AX=B。
- 我们需要将矩阵a与其他一些矩阵X进行转换，这将导致矩阵B：

$$AX = B$$

$$\begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 44 \\ 56 \\ 72 \end{bmatrix}$$

- 我们需要“撤消”A以便隔离X并获得X、y和z的值。撤消A是取A−1表示的A的倒数，并通过矩阵乘法将其应用于A。用代数方法表示：

$$AX = B$$
$$A^{-1}AX = A^{-1}B$$
$$X = A^{-1}B$$

11方程组和逆矩阵

- 为了计算矩阵A的逆矩阵，我们使用计算机，而不是手动求解。
- 注意到：如果把A^{-1}与A相乘，它将创建一个单位矩阵，即对角线中除1s外的所有零的矩阵。单位矩阵是与1相乘的线性代数等价物，这意味着它本质上没有影响，并且将有效地消除x、y和z的值：

$$A^{-1} = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{bmatrix} \qquad A = \begin{bmatrix} 4 & 2 & 4 \\ 5 & 3 & 7 \\ 9 & 3 & 6 \end{bmatrix} \qquad A^{-1}A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

11方程组和逆矩阵

- 利用SymPy研究逆矩阵和单位矩阵

```
from sympy import *
# 4x + 2y + 4z = 44
# 5x + 3y + 7z = 56
# 9x + 3y + 6z = 72
A = Matrix([
 [4, 2, 4],
 [5, 3, 7],
 [9, 3, 6]
])
```

```
inverse = A.inv()
identity = inverse * A
print("INVERSE: {}".format(inverse))
print("IDENTITY: {}".format(identity))
```

- 结果是 x = 2, y = 34, and z = –8.

11方程组和逆矩阵

● **使用NumPy求解方程组**

```
from sympy import *
# 4x + 2y + 4z = 44
# 5x + 3y + 7z = 56
# 9x + 3y + 6z = 72
A = Matrix([
 [4, 2, 4],
 [5, 3, 7],
 [9, 3, 6]
])
B = Matrix([
 44,
 56,
 72
])
X = A.inv() * B
print(X) # Matrix([[2], [34], [-8]])
```

11方程组和逆矩阵

- 数学符号的解，如右边所示，通过【高斯消去】法求得：
- 这种求解方程组的方法，也用于线性规划，其中不等式定义约束，目标最小化/最大化。
- 在实践中，很少有必要手工计算逆矩阵，可以让计算机做这件事。

$$A^{-1}B = X$$

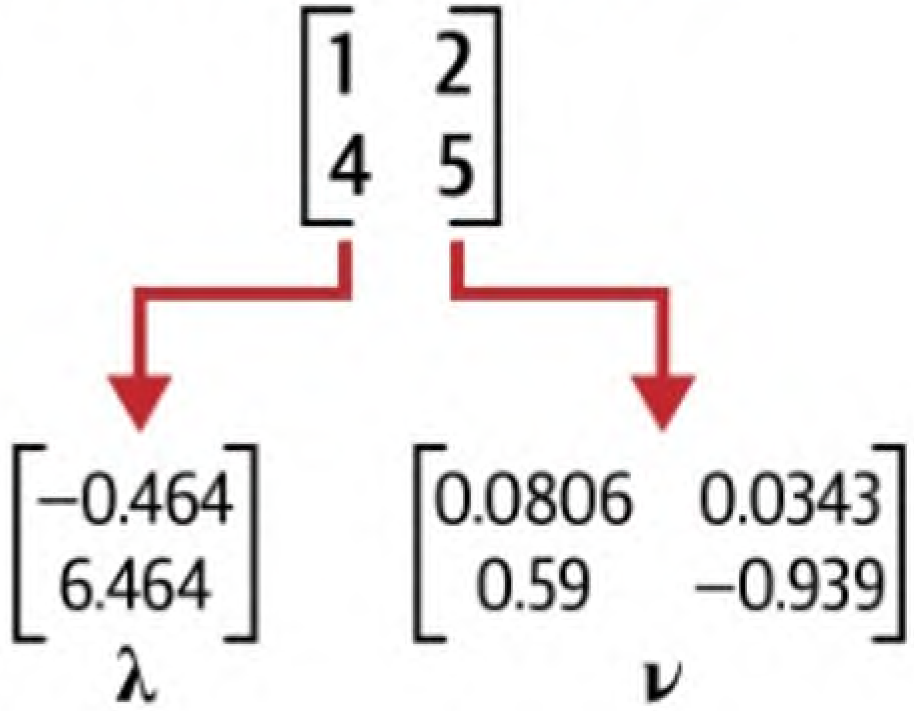
$$\begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{3} \\ 5.5 & -2 & \frac{4}{3} \\ -2 & 1 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 44 \\ 56 \\ 72 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 34 \\ -8 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

12特征向量和特征值

- 矩阵分解是将矩阵分解为其基本组成部分，类似于分解数（例如，10可分解为2×5）。
- 矩阵分解有助于诸如求逆矩阵和计算行列式以及线性回归等任务。
- 特征分解有助于将矩阵分解为组件，这些组件在不同的机器学习任务中更容易使用。注意到：它仅适用于平方矩阵。

- 在特征分解中，有两个分量：由λ表示的特征值和由v表示的特征向量，如图所示。



12特征向量和特征值

- 如果有一个平方矩阵A，它具有以下特征值方程：$Av = \lambda v$

➢ 如果A是原始矩阵，则它由特征向量v和特征值λ组成。父矩阵的每个维数都有一个特征向量和特征值，并且不是所有的矩阵都可以分解为一个特征向量和特征值。有时甚至会产生复（虚）数。

- 我们对公式进行一些调整以重建A：$A = Q\Lambda Q^{-1}$

➢ 在这个新公式中，Q是特征向量，∧是对角形式的特征值，Q^{-1}是Q的逆矩阵。对角形式是指将向量填充到零的矩阵中，并以与单位矩阵类似的模式占据对角线。

12特征向量和特征值

- 在NumPy中执行特征分解。

```
from numpy import array, diag
from numpy.linalg import eig, inv
A = array([
 [1, 2],
 [4, 5]
])
eigenvals, eigenvecs = eig(A)
print("EIGENVALUES")        # [-0.46410162 6.46410162]
print(eigenvals)
print("\nEIGENVECTORS")     # [[-0.80689822 -0.34372377]
print(eigenvecs)            #  [ 0.59069049 -0.9390708 ]]
```

12特征向量和特征值

- 在NumPy中分解和重新组合矩阵：从分解矩阵开始，然后重新编译它。

```
from numpy import array, diag
from numpy.linalg import eig, inv
A = array([
 [1, 2],
 [4, 5]
])
eigenvals, eigenvecs = eig(A)
print("EIGENVALUES")
print(eigenvals)
print("\nEIGENVECTORS")
print(eigenvecs)
print("\nREBUILD MATRIX")
Q = eigenvecs
R = inv(Q)
L = diag(eigenvals)
B = Q @ L @ R
print(B)
```

谢谢！

gulp@mail.las.ac.cn