Appendix I

Matrices

The purpose of this appendix is to briefly review matrices in sufficient detail to cover their use in other parts of the book such as in computer graphics, computational geometry, and Markov chains. This is very much *not* a comprehensive discussion of the subject. For that I suggest most textbooks on linear algebra or engineering maths, which will at least have a chapter or two about matrices; a good single text is Richard Bronson's *Schaum's Outline Of Theory And Problems Of Matrix Operations* [**Bro88**], which can be found in the Internet Archive at `https://archive.org/details/schaum-s-outlines-matrix-operations`.

A second aim is to introduce Mat.py, a simple class for manipulating matrices. I've chosen this approach rather than rely on Numpy matrices so that its easier to understand the algorithms used, especially for obtaining a determinant or a matrix inverse. Numpy is vastly superior in terms of functionality and efficiency, but these advantages are less important for the small matrices used here.

Mat.py is copiously documented, and comes with an extensive test-rig that shows off its features. To try it out, download the code, and run `python Mat.py`.

I.1 What is a Matrix?

A *matrix* is a set of real or complex numbers arranged in rows and columns to form a rectangular array. A matrix having m rows and n columns is called an $m \times n$ matrix and is referred to as having *order* $m \times n$.

A matrix is indicated by writing the array within brackets. For example:

$$\begin{pmatrix} 5 & 7 & 2 \\ 6 & 3 & 8 \end{pmatrix}$$

is a 2×3 matrix. Note that the number of rows is stated first and the number of columns second.

A whole matrix can be denoted by a single letter printed in bold type. For example, let

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}.$$

Similarly, the vextor $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$.

With `Mat.py`, you can create a matrix either by supplying a 2D list of values or by reading the data from a text file, such as `m1.txt`:

```
>>> from Mat import Mat
>>> a = Mat([[1,2,3],[4,5,6],[7,8,9]])
>>> print(a)
 1.00    2.00    3.00
 4.00    5.00    6.00
 7.00    8.00    9.00

>>> b = Mat.readSq('m1.txt')
Reading square: m1.txt
>>> print(b)
 0.80    0.19    0.01
 0.20    0.70    0.10
 0.10    0.20    0.70
```

I.2 Equality

Two matrices are said to be equal if corresponding elements throughout are equal. Thus, the two matrices must also be of the same order. So, if

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 4 & 6 & 5 \\ 2 & 3 & 7 \end{pmatrix}$$

then $a_{11} = 4; a_{12} = 6; a_{13} = 5; a_{21} = 2$; etc.

I.3 Addition and Subtraction

To be added or subtracted, two matrices must be of the same order. The sum or difference is then determined by adding or subtracting corresponding elements. For example:

$$\begin{pmatrix} 4 & 2 & 3 \\ 5 & 7 & 6 \end{pmatrix} + \begin{pmatrix} 1 & 8 & 9 \\ 3 & 5 & 4 \end{pmatrix} = \begin{pmatrix} 4+1 & 2+8 & 3+9 \\ 5+3 & 7+5 & 6+4 \end{pmatrix} = \begin{pmatrix} 5 & 10 & 12 \\ 8 & 12 & 10 \end{pmatrix}$$

Naturally, Mat.py offers these operations, both as methods called add() and subtract(), and through operator overloading:

```
>>> print(a + b)
 1.80    2.19    3.01
 4.20    5.70    6.10
 7.10    8.20    9.70

>>> print(a - b)
 0.20    1.81    2.99
 3.80    4.30    5.90
 6.90    7.80    8.30
```

I.4 Multiplication

I.4.1 Scalar multiplication. To multiply a matrix by a single number, each individual element of the matrix is multiplied by that value. For example,

$$4 \cdot \begin{pmatrix} 3 & 2 & 5 \\ 6 & 1 & 7 \end{pmatrix} = \begin{pmatrix} 12 & 8 & 20 \\ 24 & 4 & 28 \end{pmatrix}$$

Mat.py offers scalar multiplication through operator overloading:

```
>>> print(5*a)
 5.00    10.00   15.00
 20.00   25.00   30.00
 35.00   40.00   45.00

>>> print(b*2)
 1.60    0.38    0.02
 0.40    1.40    0.20
 0.20    0.40    1.40
```

I.4.2 Multiplication of two matrices. Two matrices can be multiplied together only when the number of columns in the first is equal to the number of rows in the second. For example, if

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

$$\begin{aligned} \text{then } \mathbf{A} \cdot \mathbf{b} &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \\ &= \begin{pmatrix} a_{11}b_1 + a_{12}b_2 + a_{13}b_3 \\ a_{21}b_1 + a_{22}b_2 + a_{23}b_3 \end{pmatrix} \end{aligned}$$

Each element in the top *row* of **A** is multiplied by the corresponding element in the first *column* of **b** and the products added. Similarly, the second row of the product is found by multiplying each element in the second row of **A** by the corresponding element in the first column of **b**.

In general, if **A** is an $m \times n$ matrix and **B** is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

then the matrix product $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ is the $m \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

where

$$\begin{aligned} c_{ij} &= a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} \\ &= \textstyle\sum_{k=1}^{n} a_{ik}b_{kj}, \quad \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, p. \\ &= \textstyle\sum_{i=1}^{m} \sum_{j=1}^{p} \sum_{k=1}^{n} a_{ik}b_{kj} \end{aligned}$$

An example:

$$\begin{pmatrix} 4 & 7 & 6 \\ 2 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 8 \\ 5 \\ 9 \end{pmatrix} = \begin{pmatrix} 4 \times 8 + 7 \times 5 + 6 \times 9 \\ 2 \times 8 + 3 \times 5 + 1 \times 9 \end{pmatrix} = \begin{pmatrix} 32 + 45 + 54 \\ 16 + 15 + 9 \end{pmatrix} = \begin{pmatrix} 121 \\ 40 \end{pmatrix}$$

Multiplying this (2×3) matrix and a (3×1) matrix gives a product of order (2×1).

Mat.py has a multiply() method, and also overrides the ”*” and ”@” operators.

```
>>> print(a*b)
 1.50    2.19    2.31
 4.80    5.46    4.74
 8.10    8.73    7.17

>>> print(b@a)
 1.63    2.63    3.63
 3.70    4.70    5.70
 5.80    6.80    7.80

>>> c = Mat([[1,2],[3,4]])
>>> print(a*c)
ValueError: Wrong dimensions for multiplication.
```

The second example illustrates the non-commutativity of matrix multiplication, and the third example shows the error raised when the matrix sizes do not match.

The code that implements matrix multiplication is very similar to the maths:

```
def multiply(self, other):
  if self.nCols != other.nRows:
      raise ValueError("Wrong dimensions for multiplication.")

  # Initialize the result matrix with zeros
  result = [[0 for _ in range(other.nCols)]
                     for _ in range(self.nRows)]

  # Explicit loops for matrix multiplication
  for i in range(self.nRows):
    for j in range(other.nCols):
      sumVal = 0
      for k in range(self.nCols): # or other.nRows
        sumVal += self.data[i][k] * other.data[k][j]
      result[i][j] = sumVal
  return Mat(result)
```

There is a more 'pythonic' way to write this code, based around list comprehensions, which is commented out in Mat.py.

The three `for`-loops indicate that the running time is cubic and dependent on the sizes of the matrices. If the matrices are $n \times n$ square, then the time is $O(n^3)$. A lot of effort has gone into trying to reduce this due to multiplication's importance, and an algorithm due to Volker Strassen reduces it to $O(n^{\log_2 7})$, which is significant if the matrices are large. A good description can be found in Section 4.2 of Cormen *et. al*'s *Algorithms* [**CLRS22**], and there's a nice visualization of the technique on the Wikipedia page (`https://en.wikipedia.org/wiki/Strassen_algorithm`).

I.5 Transpose of a Matrix

If the rows and columns of a matrix are interchanged, such that:

- the first row becomes the first column,
- the second row becomes the second column,
- the third row becomes the third column, etc.,

then the new matrix is called the *transpose* of the original. If $\mathbf{A}$ is the original, its transpose is denoted by $\bar{\mathbf{A}}$ or $\mathbf{A}^T$. We shall use the latter. For example, if

$$\mathbf{A} = \begin{pmatrix} 4 & 6 \\ 7 & 9 \\ 2 & 5 \end{pmatrix}$$

then

$$\mathbf{A}^T = \begin{pmatrix} 4 & 7 & 2 \\ 6 & 9 & 5 \end{pmatrix}$$

I.6 Special Matrices

(a) A *square* matrix is a matrix of order $n \times n$. For example:

$$\begin{pmatrix} 1 & 2 & 5 \\ 6 & 8 & 9 \\ 1 & 7 & 4 \end{pmatrix}$$

is a 3×3 matrix.

A square matrix $\mathbf{A}$ is *symmetric* if $a_{ij} = a_{ji}$, e.g.

$$\begin{pmatrix} 1 & 2 & 5 \\ 2 & 8 & 9 \\ 5 & 9 & 4 \end{pmatrix}$$

is symmetrical about the leading diagonal. Note that $\mathbf{A} = \mathbf{A}^T$.

A square matrix $\mathbf{A}$ is *skew-symmetric* if $a_{ij} = -a_{ji}$, e.g.

$$\begin{pmatrix} 0 & 2 & 5 \\ -2 & 0 & 9 \\ -5 & -9 & 0 \end{pmatrix}$$

In that case, $\mathbf{A} = -\mathbf{A}^T$.

(b) A *diagonal* matrix is a square matrix with all elements zero except those on the leading diagonal, thus

$$\begin{pmatrix} 5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 7 \end{pmatrix}$$

(c) A *unit* (or *identity*) matrix is a diagonal matrix in which the elements on the leading diagonal are all unity, i.e.

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

The unit matrix is denoted by $\mathbf{I}_n$, where n is its size.

(d) A *null* matrix is one whose elements are all zeros, i.e.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

and is denoted by $\mathbf{0}_n$, where n is its size

I.7 Determinant of a Square Matrix

The determinant $\det \mathbf{A}$ of a $n \times n$ matrix $\mathbf{A}$ is calculated as follows:

(1) if $n = 1$ then $A = |a|$ and $\det A = a$

(2) if $n \geq 2$ then

$$\det \mathbf{A} = \sum_{i=1}^{n} a_{ki}(-1)^{i+k} \det(\mathbf{A}'_{ik})$$

where $\mathbf{A}'_{ik}$ is the *submatrix* of $\mathbf{A}$ with the ith row and kth column deleted. The choice of k doesn't matter because the result will be the same.

This approach, known as Laplace's method after Pierre-Simon Laplace, systematically reduces a determinant of order n to smaller determinants of order $n - 1$.

The determinant is often written as

$$\det \mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & & \end{vmatrix}$$

For a 2×2 matrix, the Laplace equation can be replaced by

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

and for a 3×3 matrix,

$$\begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix} = A \begin{vmatrix} E & F \\ H & I \end{vmatrix} - B \begin{vmatrix} D & F \\ G & I \end{vmatrix} + C \begin{vmatrix} D & E \\ G & H \end{vmatrix}$$

A square matrix $\mathbf{A}$ is said to be *singular* if det $\mathbf{A} = 0$, and non-singular if det $\mathbf{A} \neq 0$.

The determinant of the following 3×3 matrix is calculated with Laplace's method:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

I chose the first column, $k = 1$, to generate the three submatrices in Fig. I.1.

The result:

$$\begin{aligned} \det(\mathbf{A}) &= (-1)^{1+1}\, 1\, [(5 \cdot 9) - (6 \cdot 8)] && i = 1,\ k = 1 \\ &\quad + (-1)^{2+1}\, 4\, [(2 \cdot 9) - (3 \cdot 8)] && i = 2,\ k = 1 \\ &\quad + (-1)^{3+1}\, 7\, [(2 \cdot 6) - (3 \cdot 5)] && i = 3,\ k = 1 \\ &= 1(-3) - 4(-6) + 7(-3) \\ &= 0 \end{aligned}$$

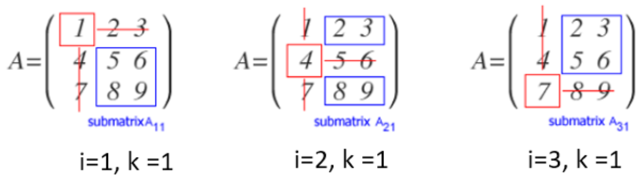

Figure I.1. The Laplace Method Applied to a 3×3 Matrix

I.7.1 Implementation. The implementation of the Laplace method is very close to the maths:

```
def determinant(self):
  if not self.isSquare():
    raise ValueError("Only square matrices have a determinant")

  n = self.nRows
  # base cases
  if n == 1:
    return self.data[0][0]
  if n == 2:
    return self.data[0][0] * self.data[1][1] - \
           self.data[0][1] * self.data[1][0]

  k = 0   # delete kth column
  det = 0
  for i in range(n):
    minor = self.submatrixExcept([i], [k])
    det += self.data[k][i] * (-1)**(i+k) * minor.determinant()
  return det
```

An example using this function:

```
>>> print(a)
 1.00    2.00    3.00
 4.00    5.00    6.00
 7.00    8.00    9.00

>>> print(a.determinant())
0
```

Its big drawback is its running time, $O(n!)$. Fortunately, approaches based on Gaussian elimination (see below) can reduce this to $O(n^3)$. Mat.py includes both versions, but the Gaussian-inspired code is commented out.

I.8 Inverse of a Square Matrix

An $n \times n$ matrix $\mathbf{A}$ is said to be *invertible* if there exists a matrix $\mathbf{A}^{-1}$, called the inverse of $\mathbf{A}$, such that

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n$$

One key result from linear algebra is:

$$\mathbf{A} \text{ is invertible } \Leftrightarrow \det(\mathbf{A}) \neq 0$$

We'll prove this by considering the contrapositive:

$$\text{if } \det(\mathbf{A}) = 0, \text{ then } \mathbf{A} \text{ is not invertible}$$

Assume:

$$\det(\mathbf{A}) = 0$$

Now suppose that $\mathbf{A}$ *is* invertible. Then, by definition, there exists $\mathbf{A}^{-1}$ such that:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}_n$$

Take the determinant of both sides:

$$\det(\mathbf{A}\mathbf{A}^{-1}) = \det(\mathbf{I}_n)$$

Using the property of determinants that $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B})$ (Binet's theorem), then

$$\det(\mathbf{A})\det(\mathbf{A}^{-1}) = \det(\mathbf{I}_n) = 1$$

But we assumed $\det(\mathbf{A}) = 0$, so:

$$0 \cdot \det(\mathbf{A}^{-1}) = 0 \neq 1$$

This is a contradiction. Therefore, our assumption that $\mathbf{A}$ is invertible must be false. Therefore, our contrapositive is true:

$$\text{if } \det(\mathbf{A}) = 0, \text{ then } \mathbf{A} \text{ is not invertible}$$

which means that the positive version is also true:

$$\mathbf{A} \text{ is invertible } \Leftrightarrow \det(\mathbf{A}) \neq 0$$

I.9 Solution of a Set of Linear Equations

Consider the set of linear equations:

$$\begin{array}{rcl}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n & = & b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n & = & b_2 \\
\vdots \qquad\qquad\qquad \vdots & = & \vdots \\
a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n & = & b_n
\end{array}$$

They can be written in matrix form:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

i.e. $\mathbf{Ax} = \mathbf{b}$ where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}; \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}; \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

If we multiply both sides of the matrix equation by the inverse of $\mathbf{A}$, we have:

$$\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b}$$

But

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n$$

so

$$\begin{aligned} \mathbf{I}_n\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \end{aligned}$$

Therefore, if we calculate the inverse of the matrix of coefficients and pre-multiply matrix $\mathbf{b}$ by it, we'll get a matrix of solutions in $\mathbf{x}$.

I.10 Gaussian Elimination

Gaussian elimination solves a linear system $\mathbf{Ax} = \mathbf{b}$ by transforming it into an equivalent system $\mathbf{Ux} = \mathbf{c}$ with an upper triangular matrix $\mathbf{U}$ (all entries in $\mathbf{U}$ below the diagonal are zero). This is done by applying three types of transformations to the *augmented* matrix $[\mathbf{A} \,|\, \mathbf{b}]$.

(1) Interchange two equations;

(2) Multiply an equation by some non-zero constant;

(3) Replace an equation with the sum of the same equation and a multiple of another equation.

Once the augmented matrix $[\mathbf{A} \,|\, \mathbf{b}]$ is transformed into $[\mathbf{U} \,|\, \mathbf{c}]$, we can solve $\mathbf{Ux} = \mathbf{c}$ using back-substitution.

I.10.1 Example. Suppose that we want to solve

$$\begin{pmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 8 \\ 10 \end{pmatrix}.$$

To keep track of the operations, we'll write them in the following form, $R_2 = R_2 - 2 * R_1$, which means that a new row 2 is computed by subtracting 2 times row 1 from the current row 2.

$$\left(\begin{array}{ccc|c} 2 & 4 & -2 & 2 \\ 4 & 9 & -3 & 8 \\ -2 & -3 & 7 & 10 \end{array}\right) \to \left(\begin{array}{ccc|c} 2 & 4 & -2 & 2 \\ 0 & 1 & 1 & 4 \\ 0 & 1 & 5 & 12 \end{array}\right) \quad \begin{array}{l} R_2 = R_2 - 2 * R_1 \\ R_3 = R_3 + R_1 \end{array}$$

$$\to \left(\begin{array}{ccc|c} 2 & 4 & -2 & 2 \\ 0 & 1 & 1 & 4 \\ 0 & 0 & 4 & 8 \end{array}\right) \quad R_3 = R_3 - R_2$$

In other words, the original system is equivalent to:

$$\begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 8 \end{pmatrix}.$$

This can be solved by back-substitution, producing

$$\begin{aligned} x_3 &= \tfrac{8}{4} = 2 \\ x_2 &= \tfrac{4-1*2}{1} = 2 \\ x_1 &= \tfrac{2-4*2-(-2)*2}{2} = -1 \end{aligned}$$

I.11 Matrix Inversion Using Gaussian Elimination

An $n \times n$ matrix $\mathbf{A}$ can have an inverse only if Gaussian elimination produces a matrix with all diagonal elements equal to 1. For $n = 3$ this looks like:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \xrightarrow{\text{Gaussian elimination}} \begin{bmatrix} 1 & ? & ? \\ 0 & 1 & ? \\ 0 & 0 & 1 \end{bmatrix}$$

Now we can keep going by applying elementary row operations to the triangle in the upper half until we get $\mathbf{I}_n$:

$$\begin{bmatrix} 1 & ? & ? \\ 0 & 1 & ? \\ 0 & 0 & 1 \end{bmatrix} \xrightarrow{\text{Row operations}} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We carry out this version of Gaussian elimination by forming an $n \times 2n$ matrix, $\mathbf{C}$, from $[\mathbf{A} \mid \mathbf{I}_n]$:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right]$$

If the left half of the transformed matrix has a upper-triangular form with a zero on its diagonal, then $\mathbf{A}$ is not invertible. Otherwise keep going until the left half becomes $\mathbf{I}_n$. For $n = 3$ the result will look like:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & b_{11} & b_{12} & b_{13} \\ 0 & 1 & 0 & b_{21} & b_{22} & b_{23} \\ 0 & 0 & 1 & b_{31} & b_{32} & b_{33} \end{array}\right]$$

I.11.1 Example. Find the inverse of

$$\mathbf{A} = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0.5 \\ 0 & 0.5 & 0.5 \end{bmatrix}$$

Form a 3 ×6 matrix **C** and perform Gaussian elimination on it.

$$\left[\begin{array}{ccc|ccc} 0.5 & 0.5 & 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 1 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 & 1 \end{array}\right] \xrightarrow{R2=R2-R1} \left[\begin{array}{ccc|ccc} 0.5 & 0.5 & 0 & 1 & 0 & 0 \\ 0 & -0.5 & 0.5 & -1 & 1 & 0 \\ 0 & 0.5 & 0.5 & 0 & 0 & 1 \end{array}\right]$$

$$\xrightarrow{R3=R3+R2} \left[\begin{array}{ccc|ccc} 0.5 & 0.5 & 0 & 1 & 0 & 0 \\ 0 & -0.5 & 0.5 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

$$\xrightarrow{R1=R1*2} \left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 2 & 0 & 0 \\ 0 & -0.5 & 0.5 & -1 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

$$\xrightarrow{R2=R2*-2} \left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 2 & 0 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

The left half is in upper-triangular form, but we still need to get rid of its nonzero off-diagonal elements:

$$\left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 2 & 0 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right] \xrightarrow{R2=R2+R3} \left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

$$\xrightarrow{R1=R1-R2} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & -1 \\ 0 & 1 & 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 & 1 & 1 \end{array}\right]$$

You should check that

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

is indeed $\mathbf{A}^{-1}$.

I.11.2 Why Does Gaussian Elimination Work?

The initial augmented matrix $[\mathbf{A} \mid \mathbf{I}_n]$ equals the matrix $\mathbf{C}$. For example, for $n = 3$:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right] = [\mathbf{A} \mid \mathbf{I}_n] = \mathbf{C}$$

The end result of the Gaussian elimination is the augmented matrix $[\mathbf{I}_n \mid \mathbf{B}]$:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & b_{11} & b_{12} & b_{13} \\ 0 & 1 & 0 & b_{21} & b_{22} & b_{23} \\ 0 & 0 & 1 & b_{31} & b_{32} & b_{33} \end{array}\right] = [\mathbf{I}_n \mid \mathbf{B}]$$

The procedure successively applies elementary row operations to the matrix $\mathbf{C}$. These can be implemented by successively multiplying $\mathbf{C}$ on the left by the elementary matrices $\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3, ..., \mathbf{E}_k$:

$$\mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\, \mathbf{C} \;=\; \mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\, [\mathbf{A} \mid \mathbf{I}_n]$$

These operations work independently on the two matrices, $\mathbf{A}$ and $\mathbf{I}_n$, resulting in:

$$[\mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\mathbf{A} \mid \mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\mathbf{I}_n] = [\mathbf{I}_n \mid \mathbf{B}]$$

The second argument shows that:

$$\mathbf{B} = \mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\mathbf{I}_n = \mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1.$$

The first argument shows that:

$$(\mathbf{E}_k \cdots \mathbf{E}_3\mathbf{E}_2\mathbf{E}_1)\,\mathbf{A} = \mathbf{B}\mathbf{A} = \mathbf{I}_n.$$

It follows that $\mathbf{B} = \mathbf{A}^{-1}$.

I.11.3 Implementation. The inverse() function in Mat.py starts by creating an augmented matrix with the object's data on the left and an identity matrix on the right. It then enters a loop which considers every column of the data looking for a row with the largest absolute value in column i. This approach is slightly different from the steps outlined above, but makes it less likely for the code to fall foul of rounding errors when using a small pivot. The chosen row is normalized and used to change the other rows, including setting their column i to 0. Once the loop ends, the inverse is extracted from the right half of the augmented matrix and returned.

```
def inverse(self):
  if not self.isSquare():
    raise ValueError("Only square matrices can be inverted")

  n = self.nRows
  # Create augmented matrix with self.data on the left
  # and the identity matrix on the right.
  aug = [self.data[i][:] + \
        [1 if i == j else 0 for j in range(n)]
                                    for i in range(n)]
  for i in range(n):
    # pivot row with largest absolute value in column i.
    pivotRowIdx = i
    maxVal = abs(aug[i][i])
    for r in range(i + 1, n):
      if abs(aug[r][i]) > maxVal:
        maxVal = abs(aug[r][i])
        pivotRowIdx = r

    if maxVal < EPS:
      raise ValueError("Pivot too small")

    if pivotRowIdx != i:
      aug[i], aug[pivotRowIdx] = aug[pivotRowIdx],aug[i]  # Swap rows

    pivot = aug[i][i]
    # Normalize the pivot row.
    for j in range(2 * n):
      aug[i][j] /= pivot

    # Eliminate the current column in all the other rows.
    for r in range(n):
      if r != i:
        factor = aug[r][i]
        for j in range(2 * n):
          aug[r][j] -= factor * aug[i][j]
  # Extract the right half as the inverse matrix.
  invData = [row[n:] for row in aug]
  return Mat(invData)
```

An example using this function:

```
>>> a = Mat([[0.5, 0.5, 0],[0.5,0,0.5],[0,0.5,0.5]])
>>> print(a)
 0.50    0.50    0.00
 0.50    0.00    0.50
 0.00    0.50    0.50

>>> b = a.inverse()
>>> print(b)
 1.00    1.00   -1.00
 1.00   -1.00    1.00
-1.00    1.00    1.00

>>> print(a*b)
 1.00    0.00    0.00
 0.00    1.00    0.00
 0.00    0.00    1.00
```