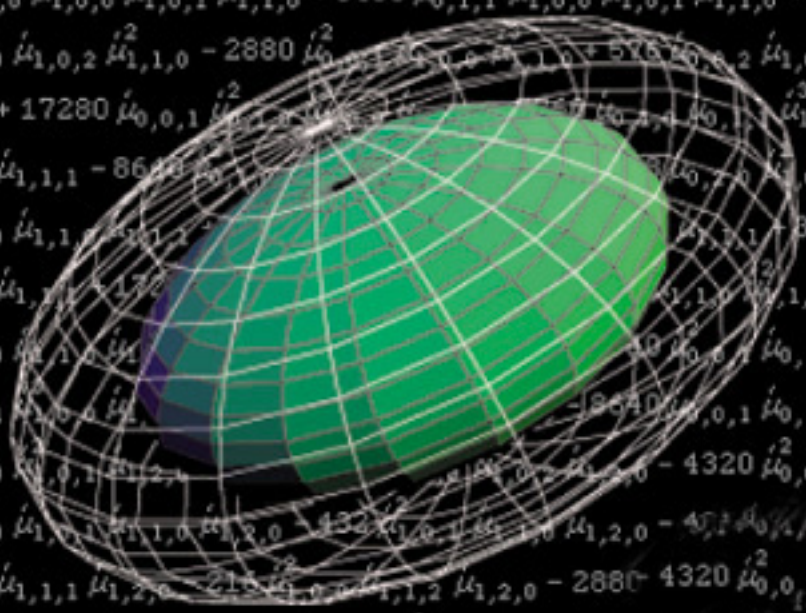
SPRINGER TEXTS IN STATISTICS

MATHEMATICAL STATISTICS

with *Mathematica*®



COLIN ROSE

MURRAY D. SMITH

Appendix

A.1 Is That the Right Answer, Dr Faustus?

○ ***Symbolic Accuracy***

Many people find the vagaries of integration to be a less than salubrious experience. Excellent statistical reference texts can make 'avoidance' a reasonable strategy, but one soon comes unstuck when one has to solve a non-textbook problem. With the advent of computer software like **mathStatica**, the Faustian joy of computerised problem solving is made ever more delectable. Indeed, over time, it seems likely that the art of manual integration will slowly wither away, much like long division has been put to rest by the pocket calculator. As we become increasingly reliant on the computer, we become more and more dependent on its accuracy. *Mathematica* and **mathStatica** are, of course, not always infallible, they are not panaceas for solving all problems, and it is possible (though rare) that they may get an integral or summation problem wrong. Lest this revelation send some readers running back to their reference texts, it should be stressed that those same reference texts suffer from exactly the same problem and for the same reason: mistakes usually occur because something has been 'typeset' incorrectly. In fact, after comparing **mathStatica**'s output with thousands of solutions in reference texts, it is apparent that even the most respected reference texts are peppered with surprisingly large numbers of errors. Usually, these are typographic errors which are all the more dangerous because they are hard to detect. A healthy scepticism for both the printed word and electronic output is certainly a valuable (though time-consuming) trait to develop.

One advantage of working with a computer is that it is usually possible to test almost any symbolic solution by using numerical methods. To illustrate, let us suppose that $X \sim$ Chi-squared(n) with pdf $f(x)$:

```
f = x^(n/2-1) e^(-x/2) / (2^(n/2) Γ[n/2]) ;      domain[f] = {x, 0, ∞} && {n > 0};
```

We wish to find the mean deviation $E\left[\,|\,X-\mu\,|\,\right]$, where μ denotes the mean:

```
μ = Expect[x, f]
```

```
n
```

Since *Mathematica* does not handle absolute values well, we shall enter $|x-\mu|$ as the expression `If[`$x < \mu$`, `$\mu - x$`, `$x - \mu$`]`. Then, the mean deviation is:

```
sol = Expect[If[x < μ, μ - x, x - μ], f]
```

$$\frac{4\,\text{Gamma}\left[1+\frac{n}{2},\ \frac{n}{2}\right] - 2\,n\,\text{Gamma}\left[\frac{n}{2},\ \frac{n}{2}\right]}{\Gamma\left[\frac{n}{2}\right]}$$

If, however, we refer to an excellent reference text like Johnson *et al.* (1994, p.420), the mean deviation is listed as:

```
JKBsol = e^(-n/2) n^(n/2) / (2^(n/2-1) Γ[n/2]);
```

First, we check if the two solutions are the same, by choosing a value for n, say $n = 6$:

```
{sol, JKBsol} /. n → 6.
```

```
{2.6885, 1.34425}
```

Clearly, at least one of the solutions is wrong! Generally, the best way to check an answer is to use a completely different methodology to derive it again. Since our original attempt was *symbolic*, we now use *numerical* methods to calculate the answer. This can be done using functions such as `NIntegrate` and `NSum`. Here is the mean deviation as a numerical integral when $n = 6$:

```
NIntegrate[ (Abs[x - μ] f) /. n → 6., {x, 0, ∞}]
```

```
— General::unfl : Underflow occurred in computation.

— General::unfl : Underflow occurred in computation.

— General::stop : Further output of
     General::unfl will be suppressed during this calculation.

— NIntegrate::ncvb :
   NIntegrate failed to converge to prescribed accuracy after 7
     recursive bisections in x near x = 5.918918918918919`.

  2.68852
```

The warning messages can be ignored, since a rough approximation serves our purpose here. The numerical answer shows that **mathStatica**'s symbolic solution is correct; further experimentation reveals that the solution given in Johnson *et al.* is out by a factor of two. This highlights how important it is to check all output, from both reference books and computers.

Finally, since μ is used frequently throughout the text, it is good housekeeping to:

```
Clear[μ]
```

… prior to leaving this example. ■

○ ***Numerical Accuracy***

> "A rapacious monster lurks within every computer,
> and it dines exclusively on accurate digits."
>
> McCullough (2000, p. 295)

Unfortunately, numerical accuracy is treated poorly in many common statistical packages, as McCullough (1998, 1999a, 1999b) has detailed.

> "Many textbooks convey the impression that all one has to do is use a computer to solve the problem, the implicit and unwarranted assumption being that the computer's solution is accurate and that one software package is as good as any other."
>
> McCullough and Vinod (1999, p. 635)

As a general 'philosophy', we try to avoid numerical difficulties altogether by treating problems symbolically (exactly), to the extent that this is possible. This means that we try to solve problems in the most general way possible, and that we also try to stop machine-precision numbers from sneaking into the calculation. For example, we can input one-and-a-half as $\frac{3}{2}$ (an exact symbolic entity), rather than as 1.5. In this way, *Mathematica* can solve many problems in an exact way, even though other packages would have to treat the same problem numerically. Of course, some problems can only be treated numerically. Fortunately, *Mathematica* provides two numerical environments for handling them:

(i) *Machine-precision numbers* (also known as floating-point): Almost all computers have optimised hardware for doing numerical calculations. These machine-precision calculations are very fast. However, using machine-precision forces all numbers to have a fixed precision, usually 16 digits of precision. This may not be enough to distinguish between two close numbers. For more detail, see Wolfram (1999, Section 3.1.6).

(ii) *Arbitrary-precision numbers:* These numbers can contain any number of digits, and *Mathematica* keeps track of the precision at all points of the calculation. Unfortunately, arbitrary-precision numerical calculations can be very slow, because they do not take advantage of a computer's hardware floating-point capabilities. For more detail, see Wolfram (1999, Section 3.1.5).

Therein lies the trade-off. If you use machine-precision numbers in *Mathematica*, the assumption is that you are primarily concerned with efficiency. If you use arbitrary-precision numbers, the assumption is that you are primarily concerned with accuracy. For more detail on numerical precision in *Mathematica*, see Sofroniou (1996). For a definitive discussion of *Mathematica*'s accuracy as a statistical package, see McCullough (2000). For *Mathematica*, the news is good:

> "By virtue of its variable precision arithmetic and symbolic power, *Mathematica*'s performance on these reliability tests far exceeds any finite-precision statistical package."
>
> McCullough (2000, p. 296)

⊕ ***Example 1:*** Machine-Precision and Arbitrary-Precision Numbers

Let $X \sim N(0, 1)$ with pdf $f(x)$:

```
f = e^(-x^2/2) / Sqrt[2 π];      domain[f] = {x, -∞, ∞};
```

The cdf, $P(X \le x)$, as a symbolic entity, is:

```
F = Prob[x, f]
```

$$\frac{1}{2}\left(1 + \text{Erf}\left[\frac{x}{\sqrt{2}}\right]\right)$$

This is the exact solution. McCullough (2000, p. 290) considers the point $x = -7.6$, way out in the left tail of the distribution. We shall enter -7.6 using exact integers:

```
sol = F /. x → -76/10
```

$$\frac{1}{2}\left(1 - \text{Erf}\left[\frac{19\sqrt{2}}{5}\right]\right)$$

… so this answer is exact too. Following McCullough, we now find the numerical value of `sol` using both machine-precision `N[sol]` and arbitrary-precision `N[sol,20]` numbers:

```
N[sol]
```

$$1.48215 \times 10^{-14}$$

```
N[sol, 20]
```

$$1.4806537490048047086 \times 10^{-14}$$

Both solutions are correct up to three significant digits, 0.0000000000000148, but they differ thereafter. In particular, the machine-precision number is incorrect at the fourth significant digit. By contrast, all twenty requested significant digits of the arbitrary-precision number 0.00000000000001480653749004804708 6 are correct, as we may verify with:

```
NIntegrate[f, {x, -∞, -76/10},
 WorkingPrecision → 30, PrecisionGoal → 20]
```

$$1.4806537490048047086 \times 10^{-14}$$

In the next input, we start off by using machine-precision, since −7.6 is entered with 2 digit precision, and we then ask *Mathematica* to render the result at 20-digit precision. Of course, this is meaningless — the extra added precision `N[·,20]` cannot eliminate the problem we have created:

```
N[ F /. x → -7.6, 20]
```

$$1.48215 \times 10^{-14}$$

If numerical accuracy is important, the moral is not to let machine-precision numbers sneak into one's workings. ■

A.2 Working with Packages

Packages contain programming code that expand *Mathematica*'s toolset in specialised fields. One can distinguish *Mathematica* packages from *Mathematica* notebooks, because they each have different file extensions, as Table 1 summarises.

file extension	*description*
`.m`	*Mathematica* package
`.nb`	*Mathematica* notebook

Table 1: Packages and notebooks

The following suggestions will help avoid problems when using packages:

(i) Always load a package in its own Input cell, separate from other calculations.

(ii) Prior to loading a package, it is often best to first quit the kernel (type `Quit` in the front end, or use Kernel Menu ▷ Quit Kernel). This avoids so-called 'context' problems. In particular, **mathStatica** should *always* be started from a fresh kernel.

(iii) The Wolfram packages are organised into families. The easiest way to load a specific Wolfram package is to simply load its family. For instance, to use any of the Wolfram statistics functions, simply load the statistics context with:

```
<< Statistics`
```

Note that the ` used in `<<Statistics`` is not a ', nor a ', but a `.

(iv) **mathStatica** is also a package, and we can load it using:

```
<< mathStatica.m
```

or

```
<< mathStatica`
```

A.3 Working with =, →, == and :=

```
ClearAll[x, y, z, q]
```

○ ***Comparing*** `Set ( = )` ***With*** `Rule ( → )`

Consider an expression such as:

```
y = 3 x^2
```

$3\,x^2$

We want to find the value of `y` when `x = 3`. Two standard approaches are: (i) `Set ( = )`, and (ii) `Rule ( → )`.

(i) `Set ( = )`: Here, we set `x` to be `3`:

```
x = 3; y
```

```
27
```

By entering `x = 3` in *Mathematica,* we lose the generality of our analysis — `x` is now just the number `3` (and not a general variable `x`). Thus, we can no longer find, for example, the derivative `D[y,x]`; nor can we `Plot[y,{x,1,2}]`. In order to return `y` to its former pristine state, we first have to clear `x` of its set value:

```
Clear[x]; y
```

$3\,x^2$

To prevent these sorts of problems, we tend to avoid using approach (i).

(ii) `Rule ( → )`: Instead of *setting* `x` to be `3`, we can simply *replace* `x` with `3` in just a single expression, by using a rule; see also Wolfram (1999, Section 2.4.1). For example, the following input reads, "Evaluate *y* when *x* takes the value of 3":

```
y /. x → 3
```

```
27
```

This time, we have not permanently changed `y` or `x`. Since everything is still general, we can still find, for example, the derivative of `y` with respect to `x`:

```
D[y, x]
```

```
6 x
```

○ ***Comparing*** `Set ( = )` ***With*** `Equal ( == )`

In some situations, both = and → are inappropriate. Suppose we want to solve the equation `z == Log[x]` in terms of `x`. If we input `Solve[z = Log[x], x]` (with one equal sign), we are actually asking *Mathematica* to `Solve[Log[x], x]`, which is not an equation. Consequently, the = sign should never be used with the `Solve` function. Instead, we use the == sign to represent a symbolic equation:

```
Solve[z == Log[x], x]
```

```
{{x → ℯ^z}}
```

If, by mistake, we enter `Solve[z = Log[x], x]`, then we must first `Clear[z]` before evaluating `Solve[z == Log[x], x]` again.

○ ***Comparing*** `Set ( = )` ***With*** `SetDelayed ( := )`

When defining functions, it is usually better to use `SetDelayed ( := )` than an *immediate* `Set ( = )`. When one uses `Set ( = )`, the right-hand side is immediately evaluated. For example:

```
F1[x_] = x + Random[]
```

```
0.733279 + x
```

So, if we call `F1` four times, the same pseudo-random number appears four times:

```
Table[F1[q], {4}]
```

```
{0.733279 + q, 0.733279 + q, 0.733279 + q, 0.733279 + q}
```

But, if we use `SetDelayed ( := )`, as follows:

```
F2[x_] := x + Random[]
```

then each time we call the function, we get a different pseudo-random number:

```
Table[F2[q], {4}]
```

```
{0.143576 + q, 0.77971 + q, 0.778795 + q, 0.618496 + q}
```

While this distinction may appear subtle at first, it becomes important when one starts writing *Mathematica* functions. Fortunately, it is quite easy to grasp after a few examples.

In similar vein, one can use `RuleDelayed` :→ instead of an immediate `Rule` →.

A.4 Working with Lists

Mathematica uses curly braces {} to denote lists, *not* parentheses ( ). Here, we enter the list $X = \{x_1, \ldots, x_6\}$:

```
X = {x1, x2, x3, x4, x5, x6};
```

The fourth element, or part, of list X is:

```
X[[4]]
```

```
x4
```

Sometimes, `X⟦4⟧` is used rather than `X[[4]]`. The fancy double bracket ⟦ is obtained by entering ESC [[ESC . We now add 5 to each element of the list:

```
X + 5
```

```
{5 + x1, 5 + x2, 5 + x3, 5 + x4, 5 + x5, 5 + x6}
```

Other common manipulations include:

```
Plus @@ X
```

```
x1 + x2 + x3 + x4 + x5 + x6
```

```
Times @@ X
```

```
x1 x2 x3 x4 x5 x6
```

```
Power @@ X
```

```
x1^(x2^(x3^(x4^(x5^x6))))
```

Here is a more sophisticated function that constructs an alternating sum:

```
Fold[(#2 - #1) &, 0, Reverse[X]]
```

```
x1 - x2 + x3 - x4 + x5 - x6
```

Next, we construct an Assumptions statement for the x_i, assuming they are all positive:

```
Thread[X > 0]
```

```
{x1 > 0, x2 > 0, x3 > 0, x4 > 0, x5 > 0, x6 > 0}
```

Here is a typical **mathStatica** 'domain' statement assuming $x_i \in (-\infty, 0)$:

```
Thread[{X, -∞, 0}]
```

```
{{x1, -∞, 0}, {x2, -∞, 0}, {x3, -∞, 0},
 {x4, -∞, 0}, {x5, -∞, 0}, {x6, -∞, 0}}
```

Finally, here is some data:

```
data = Table[Random[], {6}]
```

```
{0.530808, 0.164839, 0.340276,
 0.595038, 0.674885, 0.562323}
```

which we now attach to the elements of X using rules →, as follows:

```
Thread[X → data]
```

```
{x1 → 0.530808, x2 → 0.164839, x3 → 0.340276,
 x4 → 0.595038, x5 → 0.674885, x6 → 0.562323}
```

These tricks of the trade can sometimes be very useful indeed.

A.5 Working with Subscripts

In mathematical statistics, it is both common and natural to use subscripted notation such as $y_1, \ldots, y_n$. This section first discusses "The Wonders of Subscripts" in *Mathematica,* and then provides "Two Cautionary Tips".

○ *The Wonders of Subscripts*

```
Clear[μ]
```

Subscript notation $\mu_1, \mu_2, \ldots, \mu_8$ offers many advantages over 'dead' notation such as $\mu 1, \mu 2, \ldots, \mu 8$. For instance, let:

```
r = Range[8]
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Then, to create the list $z = \{\mu_1, \mu_2, \ldots, \mu_7, \mu_8\}$, we enter:

```
z = Thread[μr]
```

```
{μ1, μ2, μ3, μ4, μ5, μ6, μ7, μ8}
```

We can now take advantage of *Mathematica*'s advanced pattern matching technology to convert from subscripts to, say, powers:

```
z /. μx_ → s^x
```

$$\{s, s^2, s^3, s^4, s^5, s^6, s^7, s^8\}$$

and back again:

```
% /. s^x_. → μx
```

$$\{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6, \mu_7, \mu_8\}$$

Next, we convert the μ_i into functional notation $\mu[i]$:

```
z /. μx_ → μ[x]
```

$$\{\mu[1], \mu[2], \mu[3], \mu[4], \mu[5], \mu[6], \mu[7], \mu[8]\}$$

Now, suppose that μ_t (t = 1, …, 8) denotes μ at time t. Then, we can go 'back' one period in time:

```
z /. μt_ → μt-1
```

$$\{\mu_0, \mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6, \mu_7\}$$

Or, try something like:

```
z /. μt_ → μt / μ9-t^t
```

$$\left\{\frac{\mu_1}{\mu_8}, \frac{\mu_2}{\mu_7^2}, \frac{\mu_3}{\mu_6^3}, \frac{\mu_4}{\mu_5^4}, \frac{\mu_5}{\mu_4^5}, \frac{\mu_6}{\mu_3^6}, \frac{\mu_7}{\mu_2^7}, \frac{\mu_8}{\mu_1^8}\right\}$$

Because the index t is 'live', quite sophisticated pattern matching is possible. Here, for instance, we replace the even-numbered subscripted elements with $\not\Delta$:

```
z /. μt_ :> If[EvenQ[t], Δt , μt]
```

$$\{\mu_1, \not\Delta_2, \mu_3, \not\Delta_4, \mu_5, \not\Delta_6, \mu_7, \not\Delta_8\}$$

Now suppose that a random sample of size n = 8, say:

```
data = {0, 1, 3, 0, 1, 2, 0, 2};
```

is collected from a random variable $X \sim$ Poisson(λ) with pmf $f(x)$:

```
f = e^-λ λ^x / x!;  domain[f] = {x, 0, ∞} && {λ > 0} && {Discrete};
```

Then, using subscript notation, the *symbolic* likelihood can be entered as:

```
L = Product[(f /. x → x_i), {i, 1, n}]
```

$$\prod_{i=1}^{n} \frac{e^{-\lambda}\,\lambda^{x_i}}{x_i\,!}$$

while the *observed* likelihood is obtained via:

```
L /. {n → 8, x_i_ :> data[[i]]}
```

$$\frac{1}{24}\, e^{-8\lambda}\, \lambda^{9}$$

○ ***Two Cautionary Tips***

Caution 1: While subscript notation has many advantages in *Mathematica*, its use also requires some care. This is because the internal representation in *Mathematica* of the subscript expression y_1 is quite different to the `Symbol` `y`. Technically, this is because `Head[y] == Symbol`, while `Head[`y_1`] == Subscript`. That is, *Mathematica* thinks of `y` as a `Symbol`, while it thinks of y_1 as `Subscript[y,1]`; see also Appendix A.8. Because of this difference, the following is important.

Suppose we set `y = 3`. To clear `y`, we would then enter `Clear[y]`:

```
y = 3; Clear[y]; y
```

```
y
```

For this to work, `y` must be a `Symbol`. It will not work for y_1, because the internal representation of y_1 is `Subscript[y,1]`, which is not a `Symbol`. The same goes for $\bar{y}$, $\hat{y}$, y^*, and other notational variants of `y`. For instance:

```
y₁ = 3; Clear[y₁]; y₁
```

```
— Clear::ssym : y₁ is not a symbol or a string.
```

```
3
```

Instead, to clear y_1, one must use either y_1 `= .`, as in:

```
y₁ = 3;  y₁ = .;  y₁
```

```
y₁
```

or the more savage `Clear[Subscript]`:

```
y₁ = 3;  Clear[Subscript];  y₁
```

```
y₁
```

Note that `Clear[Subscript]` will clear *all* subscripted variables. This can be used as a nifty trick to clear all of $\{y_1, y_2, \ldots, y_n\}$ simultaneously!

Caution 2: In *Mathematica* Version 4.0, there are still a few functions that do not handle subscripted variables properly (though this seems to be mostly fixed as of Version 4.1). This problem can usually be overcome by wrapping `Evaluate` around the relevant expression. For instance, under Version 4.0, the following generates error messages:

```
f = Exp[x1]; NIntegrate[f, {x1, -∞, 2}]
```

```
– Function::flpar :
   Parameter specification {x1} in Function[{x1}, {f}]
     should be a symbol or a list of symbols.
– General::stop : Further output of Function::flpar will
     be suppressed during this calculation.
– NIntegrate::inum :
   Integrand {4.} is not numerical at {x1} = {1.}.
```

```
NIntegrate[f, {x1, -∞, 2}]
```

Wrapping `Evaluate` around `f` overcomes this 'bug' by forcing *Mathematica* to evaluate `f` prior to starting the numerical integration:

```
f = Exp[x1]; NIntegrate[Evaluate[f], {x1, -∞, 2}]
```

```
7.38906
```

Alternatively, the following also works fine:

```
NIntegrate[ Exp[x1], {x1, -∞, 2}]
```

```
7.38906
```

Similarly, the following produces copious error messages under Version 4.0:

```
f = x1 + x2; Plot3D[f, {x1, 0, 1}, {x2, 0, 1}]
```

but if we wrap `Evaluate` around `f`, the desired plot is generated:

```
f = x1 + x2; Plot3D[Evaluate[f], {x1, 0, 1}, {x2, 0, 1}]
```

As a different example, the following works fine:

```
D[x1^3 x, x1]
```

```
3 x x1^2
```

but the next input does not work as we might expect, because `x1 = Subscript[x, 1]` is interpreted by *Mathematica* as a function of `x`:

```
D[x1^3 x, x]
```

```
x1^3 + 3 x x1^2 Subscript^(1,0) [x, 1]
```

A.6 Working with Matrices

This appendix gives a brief overview of matrices in *Mathematica*. A good starting point is also Wolfram (1999, Sections 3.7.1 – 3.7.11). Two standard *Mathematica* add-on packages may also be of interest, namely `LinearAlgebra`MatrixManipulation`` and `Statistics`DataManipulation`.

○ *Constructing Matrices*

In *Mathematica*, a matrix is represented by a list of lists. For example, the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

can be entered into *Mathematica* as follows:

```
A = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}}
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

If **mathStatica** is loaded, this output will appear on screen as a fancy formatted matrix. If **mathStatica** is not loaded, the output will appear as a `List` (just like the input). If you do not like the fancy matrix format, you can switch it off with the **mathStatica** function `FancyMatrix` — see Appendix A.8.

Keyboard entry: Table 2 describes how to enter fancy matrices directly from the keyboard. This entry mechanism is quite neat, and it is easily mastered.

short cut	*description*
CTRL ,	add a column
CTRL RET	add a row

Table 2: Creating fancy matrices using the keyboard

For example, to enter the matrix $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$, type the following keystrokes in an Input cell:

(1 CTRL , 2 CTRL , 3 CTRL RET 4 TAB 5 TAB 6 →)

While this may appear as the fancy matrix $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$, the internal representation in *Mathematica* is still `{{1, 2, 3}, {4, 5, 6}}`.

A number of *Mathematica* functions are helpful in constructing matrices, as the following examples illustrate. Here is an identity matrix:

```
IdentityMatrix[5]
```

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

… a diagonal matrix:

```
DiagonalMatrix[{a, b, c, d}]
```

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & d \end{pmatrix}$$

… a more general matrix created with `Table`:

```
Table[a[i, j], {i, 2}, {j, 4}]
```

$$\begin{pmatrix} a[1,1] & a[1,2] & a[1,3] & a[1,4] \\ a[2,1] & a[2,2] & a[2,3] & a[2,4] \end{pmatrix}$$

… an example using subscript notation:

```
Table[a_{i,j}, {i, 2}, {j, 4}]
```

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \end{pmatrix}$$

… an upper-triangular matrix:

```
Table[ If[i ≤ j, ☺, 0], {i, 5}, {j, 5}]
```

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

… and a Hilbert matrix:

```
Table[1 / (i + j - 1), {i, 3}, {j, 3}]
```

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{pmatrix}$$

○ ***Operating on Matrices***

Consider the matrices:

```
M = {{a, b}, {c, d}};      B = {{1, 2}, {3, 4}};
```

For detail on getting pieces of matrices, see Wolfram (1999, Section 3.7.2). In particular, here is the first row of M:

```
M[[1]]
```

```
{a, b}
```

An easy way to grab, say, the second column of M is to select it with the mouse, copy, and paste it into a new Input cell. If desired, this can then be converted into InputForm (Cell Menu ▷ ConvertTo ▷ InputForm). Alternatively, we can obtain the second column with:

```
M[[All, 2]]
```

```
{a, c}
```

The dimension (2×2) of matrix M is obtained with:

```
Dimensions[M]
```

```
{2, 2}
```

The transpose of M is:

```
Transpose[M]
```

$$\begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

The determinant of M is given by:

```
Det[M]
```

```
-b c + a d
```

The inverse of M is:

```
Inverse[M]
```

$$\begin{pmatrix} \frac{d}{-b\,c+a\,d} & -\frac{b}{-b\,c+a\,d} \\ -\frac{c}{-b\,c+a\,d} & \frac{a}{-b\,c+a\,d} \end{pmatrix}$$

The trace is the sum of the elements on the main diagonal:

```
Tr[M]
```

```
a + d
```

Here are the eigenvalues of M:

```
Eigenvalues[M]
```

$$\left\{\frac{1}{2}\left(a+d-\sqrt{a^2+4\,b\,c-2\,a\,d+d^2}\right),\ \frac{1}{2}\left(a+d+\sqrt{a^2+4\,b\,c-2\,a\,d+d^2}\right)\right\}$$

To illustrate matrix addition, consider $B + M$:

```
B + M
```

$$\begin{pmatrix} 1+a & 2+b \\ 3+c & 4+d \end{pmatrix}$$

To illustrate matrix multiplication, consider $B\,M$:

```
B.M
```

$$\begin{pmatrix} a+2\,c & b+2\,d \\ 3\,a+4\,c & 3\,b+4\,d \end{pmatrix}$$

… which is generally not equal to $M\,B$:

```
M.B
```

$$\begin{pmatrix} a+3\,b & 2\,a+4\,b \\ c+3\,d & 2\,c+4\,d \end{pmatrix}$$

Similarly, here is the product $B\,M\,B$:

```
B.M.B
```

$$\begin{pmatrix} a+2\,c+3\,(b+2\,d) & 2\,(a+2\,c)+4\,(b+2\,d) \\ 3\,a+4\,c+3\,(3\,b+4\,d) & 2\,(3\,a+4\,c)+4\,(3\,b+4\,d) \end{pmatrix}$$

… which is generally not equal to $B^{\mathrm{T}}\,M\,B$:

```
Transpose[B].M.B
```

$$\begin{pmatrix} a+3\,c+3\,(b+3\,d) & 2\,(a+3\,c)+4\,(b+3\,d) \\ 2\,a+4\,c+3\,(2\,b+4\,d) & 2\,(2\,a+4\,c)+4\,(2\,b+4\,d) \end{pmatrix}$$

Powers of a matrix, such as $B^3 = B\,B\,B$, can either be entered as:

```
MatrixPower[B, 3]
```

$$\begin{pmatrix} 37 & 54 \\ 81 & 118 \end{pmatrix}$$

or as:

```
B.B.B
```

$$\begin{pmatrix} 37 & 54 \\ 81 & 118 \end{pmatrix}$$

but *not* as:

```
B^3
```

$$\begin{pmatrix} 1 & 8 \\ 27 & 64 \end{pmatrix}$$

Mathematica does not provide a function for doing Kronecker products, so here is one we put together for this Appendix:

```
Kronecker[A_, B_] :=
  Partition[
     Flatten[
              Map[ Transpose, Outer[Times, A, B] ]
            ], Dimensions[A][[2]] Dimensions[B][[2]]  ]
```

For example, here is the Kronecker product $B \otimes M$:

```
Kronecker[B, M]
```

$$\begin{pmatrix} a & b & 2a & 2b \\ c & d & 2c & 2d \\ 3a & 3b & 4a & 4b \\ 3c & 3d & 4c & 4d \end{pmatrix}$$

and here is the Kronecker product $M \otimes B$:

```
Kronecker[M, B]
```

$$\begin{pmatrix} a & 2a & b & 2b \\ 3a & 4a & 3b & 4b \\ c & 2c & d & 2d \\ 3c & 4c & 3d & 4d \end{pmatrix}$$

A.7 Working with Vectors

There are two completely different ways to enter a vector in *Mathematica*:

(i) *The* `List` *Approach:* This is the standard *Mathematica* method. It does not distinguish between column and row vectors. Thus, `Transpose` cannot be used on these vectors.

(ii) *The* `Matrix` *Approach:* Here, a vector is entered as a special case of a matrix. This does distinguish between column and row vectors, so `Transpose` can be used with these vectors. Entering the vector this way takes more effort, but it can be less confusing and more 'natural' than the `List` approach.

In this book, we use approach (i). Mixing the two approaches is not recommended, as this may cause error and confusion.

○ *Vectors as Lists*

The standard *Mathematica* way to represent a vector is as a `List` {…}, not a matrix {{…}}. Consider, for example:

```
vec = {15, -3, 5}
```

```
{15, -3, 5}
```

Mathematica thinks `vec` is a vector:

```
VectorQ[vec]
```

```
True
```

Is `vec` a column vector or a row vector? The answer is *neither*. Importantly, when the `List` approach is used, *Mathematica* makes no distinction between column and row vectors. Instead, *Mathematica* carries out whatever operation is possible. This can be confusing and disorienting. To illustrate, suppose we are interested in the (3×1) column vector $\vec{v}$ and the (1×3) row vector $\vec{u}$, given by

$$\vec{v} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \quad \text{and} \quad \vec{u} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}.$$

Using the `List` approach, we enter both of them into *Mathematica* in the same way:

```
v = {a, b, c}
u = {1, 2, 3}
```

```
{a, b, c}
{1, 2, 3}
```

Although we can find the `Transpose` of a matrix, there is no such thing as a `Transpose` of a *Mathematica* `Vector`:

```
Transpose[v]
```

```
– Transpose::nmtx : The first two levels of the
    one-dimensional list {a, b, c} cannot be transposed.

Transpose[{a, b, c}]
```

Once again, this arises because *Mathematica* does not distinguish between column vectors and row vectors. To stress the point, this means that the *Mathematica* input for $\vec{v}$ and $\vec{v}^{\mathrm{T}}$ is exactly the same.

When the `Dot` operator is applied to two vectors, it returns a scalar. Thus, `v.v` is equivalent to $\vec{v}^{\mathrm{T}}\vec{v}\,(1\times 1)$:

```
v.v
```

$$a^2 + b^2 + c^2$$

while `u.u` is equivalent to $\vec{u}\,\vec{u}^{\mathrm{T}}\,(1\times 1)$:

```
u.u
```

```
14
```

In order to obtain $\vec{v}\,\vec{v}^{\mathrm{T}}\,(3\times 3)$ and $\vec{u}^{\mathrm{T}}\vec{u}\;(3\times 3)$, we have to derive the outer product using the rather cumbersome expression:

```
Outer[Times, v, v]
```

$$\begin{pmatrix} a^2 & a\,b & a\,c \\ a\,b & b^2 & b\,c \\ a\,c & b\,c & c^2 \end{pmatrix}$$

```
Outer[Times, u, u]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix}$$

Next, suppose:

```
M = {{1, 0, 0}, {4, 5, 6}, {0, 0, 9}};
```

Then, $\vec{v}^{\mathrm{T}} M \vec{v}$ (1×1) is evaluated with:

```
v.M.v
```

```
5 b^2 + a (a + 4 b) + c (6 b + 9 c)
```

and $\vec{u} M \vec{u}^{\mathrm{T}}$ (1×1) is evaluated with:

```
u.M.u
```

```
146
```

Once again, we stress that we do not use `u.M.Transpose[u]` here, because one cannot find the `Transpose` of a *Mathematica* `Vector`.

The **mathStatica** function `Grad[`f`,` $\vec{x}$`]` calculates the gradient of scalar f with respect to $\vec{x} = \{x_1, \ldots, x_n\}$, namely

$$\left\{ \frac{\partial f}{\partial x_1} \;, \ldots, \; \frac{\partial f}{\partial x_n} \right\}.$$

Here, then, is the gradient of $f = a\, b^2$ with respect to $\vec{v}$:

```
f = a b^2;   Grad[f, v]
```

```
{b^2, 2 a b, 0}
```

The derivative of a vector with respect to a vector yields a matrix. If $\vec{f}$ is an m-dimensional vector, and $\vec{x}$ is an n-dimensional vector, then `Grad[`$\vec{f}$`,` $\vec{x}$`]` calculates the $(m\times n)$ matrix:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

This is also known as the Jacobian matrix. Here is an example:

```
f = {a b^2, a, b, c^2, 1};   Grad[f, v]
```

```
{{b^2, 2 a b, 0}, {1, 0, 0}, {0, 1, 0}, {0, 0, 2 c}, {0, 0, 0}}
```

○ ***Vectors as Matrices***

Column vectors ($m\times1$) and row vectors ($1\times n$) are, of course, just special cases of an ($m\times n$) matrix. In this vein, one can force *Mathematica* to distinguish between a column vector and a row vector by entering them both as matrices {{…}}, rather than as a single `List` {…}. To illustrate, suppose we are interested again in the (3×1) column vector $\vec{v}$ and the (1×3) row vector $\vec{u}$, given by

$$\vec{v} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \quad \text{and} \quad \vec{u} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}.$$

This time, we shall enter both $\vec{v}$ and $\vec{u}$ into *Mathematica* as if they were matrices. So, we enter the column vector $\vec{v}$ as:

```
V = {{a}, {b}, {c}}
```

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

As far as *Mathematica* is concerned, this is *not* a `Vector`:

```
VectorQ[V]
```

```
False
```

Rather, *Mathematica* thinks it is a `Matrix`:

```
MatrixQ[V]
```

```
True
```

Similarly, we enter the row vector $\vec{u}$ as if it is the first row of a matrix:

```
U = {{1, 2, 3}}        (* not {1,2,3} *)
```

```
{{1, 2, 3}}
```

```
VectorQ[U]
```

```
False
```

```
MatrixQ[U]
```

```
True
```

Because V and U are *Mathematica* matrices, Transpose now works:

```
Transpose[V]
```

```
{{a, b, c}}
```

```
Transpose[U]
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

We can now use standard notation to find $\vec{v}^{\mathrm{T}}\vec{v}$ (1×1):

```
Transpose[V].V
```

$$\{\{a^2 + b^2 + c^2\}\}$$

and $\vec{u}\,\vec{u}^{\mathrm{T}}$ (1×1):

```
U.Transpose[U]
```

```
{{14}}
```

To obtain $\vec{v}\,\vec{v}^{\mathrm{T}}$ (3×3) and $\vec{u}^{\mathrm{T}}\vec{u}$ (3×3), we no longer have to use Outer products. Again, the answer is obtained using standard notation. Here is $\vec{v}\,\vec{v}^{\mathrm{T}}$:

```
V.Transpose[V]
```

$$\begin{pmatrix} a^2 & a\,b & a\,c \\ a\,b & b^2 & b\,c \\ a\,c & b\,c & c^2 \end{pmatrix}$$

and $\vec{u}^{\mathrm{T}}\vec{u}$:

```
Transpose[U].U
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix}$$

Next, suppose:

$$\mathbf{M} = \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{4} & \mathbf{5} & \mathbf{6} \\ \mathbf{0} & \mathbf{0} & \mathbf{9} \end{pmatrix};$$

Then, $\vec{v}^{\mathrm{T}} M\ \vec{v}$ (1×1) is evaluated with:

```
Transpose[V].M.V
```

```
{{5 b^2 + a (a + 4 b) + c (6 b + 9 c)}}
```

and $\vec{u}\, M\, \vec{u}^{\mathrm{T}}$ (1×1) is evaluated with:

```
U.M.Transpose[U]
```

```
{{146}}
```

… not with `U.M.U`.

The `Matrix` approach to vectors has the advantage that it allows one to distinguish between column and row vectors, which seems more natural. However, on the downside, many *Mathematica* functions (including `Grad`) have been designed to operate on a single `List` (`Vector`), not on a matrix; these functions will often *not* work with vectors that have been entered using the `Matrix` approach.

A.8 Changes to Default Behaviour

mathStatica makes a number of changes to default *Mathematica* behaviour. These changes only take effect after you load **mathStatica**, and they only remain active while **mathStatica** is running. This section lists three 'visual' changes.

Case 1: `Γ[x]`

If **mathStatica** is not loaded, the expression `Γ[x]` has no meaning to *Mathematica*. If **mathStatica** is loaded, the expression `Γ[x]` is interpreted as the *Mathematica* function `Gamma[x]`:

```
Γ[x] == Gamma[x]
```

```
True
```

Case 2: Subscript and Related Notation in Input Cells

```
Quit
```

If **mathStatica** is *not* loaded, it is best to avoid mixing `x` with its variants $\{x_1, \hat{x}, \ldots\}$ in Input cells. To see why, let us suppose we set `x = 3`:

```
x = 3
```

```
3
```

and then evaluate:

```
{x_1, x^*, x̄, x⃗, x̃, x̂, ẋ, x̀}
```

```
{3_1, 3^*, 3̄, 3⃗, 3̃, 3̂, 3̇, 3̀}
```

This output is not the desired behaviour in standard notational systems.

```
Quit
```

However, if **mathStatica** is loaded, we can work with x and its variants {x_1, $\hat{x}$, …} at the same time without any 'problems':

```
<< mathStatica.m
```

```
x = 3
```

```
3
```

This time, *Mathematica* treats the variants {x_1, $\hat{x}$, ...} in the way we want it to:

```
{x_1, x^*, x̄, x⃗, x̃, x̂, ẋ, x̀}
```

```
{x_1, x^*, x̄, x⃗, x̃, x̂, ẋ, x̀}
```

This change is implemented in **mathStatica** simply by adding the attribute `HoldFirst` to the following list of functions:

```
lis = {Subscript, SuperStar, OverBar, OverVector,
   OverTilde, OverHat, OverDot, Overscript,
   Superscript, Subsuperscript, Underscript,
   Underoverscript, SubPlus, SubMinus, SubStar,
   SuperPlus, SuperMinus, SuperDagger, UnderBar};
```

This idea was suggested by Carl Woll. In our experience, it works brilliantly, without any undesirable side effects, and without the need for the Notation package which can interfere with the subscript manipulations used by **mathStatica**. If, for some reason, you do not like this feature, you can return to *Mathematica*'s default behaviour by entering:

```
ClearAttributes[Evaluate[lis], HoldFirst]
```

Of course, if you do this, some Input cells in this book may no longer work as intended.

Case 3: Matrix Output

If **mathStatica** is not loaded, matrices appear as lists. For example:

```
Quit
```

```
m = Table[i - j, {i, 4}, {j, 5}]
```

```
{{0, -1, -2, -3, -4}, {1, 0, -1, -2, -3},
 {2, 1, 0, -1, -2}, {3, 2, 1, 0, -1}}
```

If, however, **mathStatica** is loaded, matrices automatically appear nicely formatted as matrices. For example:

```
Quit
```

```
<< mathStatica.m
```

```
m = Table[i - j, {i, 4}, {j, 5}]
```

$$\begin{pmatrix} 0 & -1 & -2 & -3 & -4 \\ 1 & 0 & -1 & -2 & -3 \\ 2 & 1 & 0 & -1 & -2 \\ 3 & 2 & 1 & 0 & -1 \end{pmatrix}$$

Standard matrix operations still operate flawlessly:

```
m[[1]]
```

```
{0, -1, -2, -3, -4}
```

```
m + 2
```

$$\begin{pmatrix} 2 & 1 & 0 & -1 & -2 \\ 3 & 2 & 1 & 0 & -1 \\ 4 & 3 & 2 & 1 & 0 \\ 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

Moreover, it is extremely easy to extract a column (or two): simply select the desired column with the mouse, copy, and paste it into a new Input cell. If desired, you can then convert into InputForm (Cell Menu ▷ ConvertTo ▷ InputForm).

This trick essentially eliminates the need to use the awkward `MatrixForm` command. If, for some reason, you do not like this fancy formatted output (*e.g.* if you work with very large matrices), you can return to *Mathematica*'s default behaviour by simply evaluating:

```
FancyMatrix[Off]
```

```
– FancyMatrix is now Off.
```

Then:

```
m
```

```
{{0, -1, -2, -3, -4}, {1, 0, -1, -2, -3},
 {2, 1, 0, -1, -2}, {3, 2, 1, 0, -1}}
```

You can switch it on again with `FancyMatrix[On]`.

A.9 Building Your Own mathStatica Function

The building blocks of mathematical statistics include the expectations operator, variance, probability, transformations, and so on. A lot of effort and code has gone into creating these functions in **mathStatica**. The more adventurous reader can create powerful custom functions by combining these building blocks in different ways — much like a LEGO® set. To illustrate, suppose we want to write our own function to automate kurtosis calculations for an arbitrary univariate density function f. We recall that kurtosis is defined by

$$\beta_2 = \mu_4 / \mu_2^2$$

where $\mu_r = E[(X - \mu)^r]$. How many arguments should our `Kurtosis` function have? In other words, should it be `Kurtosis[x, μ, f]`, or `Kurtosis[x, f]`, or just `Kurtosis[f]`? If our function is smart, we will not need the '`x`', since this information can be derived from `domain[f]`; nor do we need the 'μ', because this can also be calculated from density `f`. So, the neat solution is simply `Kurtosis[f]`. Then, we might proceed as follows:

```
Kurtosis[f_] := Module[{xx,mean,var,sol,b=domain[f]},
    xx   =  If[ Head[b] === And, b[[1,1]], b[[1]]];
  mean   =  Expect[xx, f];
   var   =     Var[xx, f];
   sol   =  Expect[(xx - mean)^4, f] / var^2;
   Simplify[sol]       ]
```

In the above, the term `xx` picks out the random variable `x` from any given `domain[f]` statement. We also need to set the `Attributes` of our `Kurtosis` function:

```
SetAttributes[Kurtosis, HoldFirst]
```

What does this do? The `HoldFirst` expression forces the `Kurtosis` function to hold the `f` as an '`f`', rather than immediately evaluating it as, say, `f` = $e^{-\lambda}\,\lambda^x / x!$. By holding the `f`, the function can then find out what `domain[f]` has been set to, as opposed to `domain[`$e^{-\lambda}\,\lambda^x / x!$`]`. Similarly, it can evaluate `Expect[x, f]` or `Var[x, f]`. More generally, if we wrote a function `MyFunc[n_, f_]`, where `f` is the second argument (rather than the first), we would use `SetAttributes[MyFunc, HoldRest]`, so that the `f` is still held. To illustrate our new function, suppose $X \sim$ Poisson(λ) with pmf $f(x)$:

```
f = e^-λ λ^x / x!;   domain[f] = {x, 0, ∞} && {λ > 0} && {Discrete};
```

Then, the kurtosis of the distribution is:

```
Kurtosis[f]
```

$$3 + \frac{1}{\lambda}$$