STATS 782

R Programming

Numerical Computation

Computer Arithmetic Is Not Exact

- The following R statement appears to give the correct answer.

```
> 0.3 - 0.2
[1] 0.1
```

Computer Arithmetic Is Not Exact

- The following R statement appears to give the correct answer.

```
> 0.3 - 0.2
[1] 0.1
```

- But all is not as it seems.

```
> 0.3 - 0.2 == 0.1
[1] FALSE
```

Computer Arithmetic Is Not Exact

- The following R statement appears to give the correct answer.

```
> 0.3 - 0.2
[1] 0.1
```

- But all is not as it seems.

```
> 0.3 - 0.2 == 0.1
[1] FALSE
```

- The difference between the values being compared is small, but important.

```
> (0.3 - 0.2) - 0.1
[1] -2.775558e-17
```

Floating-Point Arithmetic

- The results on the previous slide occur because R uses *floating-point arithmetic* to carry out its calculations.
- Floating-point arithmetic approximates real values by a finite set of approximating (binary) rationals.
- The use of an approximate representation produces two kinds of errors.
  - The errors inherent in the approximations.
  - The errors which occur during calculations.

Measuring Machine Precision

- A computer has only a finite number of values to approximate the complete set of real numbers.
- This means that there is a limit to the accuracy that can be obtained in approximating any value.
- This means that when x is small enough, the value $1+x$ must be approximated by the value 1.
- This gives a practical way of measuring machine accuracy.

The Relative Machine Precision

- The accuracy of a floating-point system is measured by the *relative machine precision* or *machine epsilon*.
- This is the smallest positive value which can be added to 1 to produce a value different from 1.
- A machine epsilon of 10^{-7} indicates that there are roughly 7 decimal digits of precision in the numeric values stored and manipulated by the computer.
- It is easy to write a program to determine the relative machine precision.

Computing the Relative Machine Precision

Here is a small function for computing the machine epsilon.

```
> macheps =
      function()
      {
          eps = 1
          while(1 + eps/2 != 1)
              eps = eps/2
          eps
      }
```

And here it is in action.

```
> macheps()
[1] 2.220446e-16
```

Floating-Point Precision

- The preceding program shows that there are roughly 16 decimal digits of precision to R arithmetic.
- It is possible to see the effects of this limited precision directly.

```
> a = 12345678901234567890
> print(a, digits=20)
[1] 12345678901234567168
```

- The effects of finite precision show up in the results of calculations.

IEEE 754 Arithmetic

- R uses the *IEEE 754 Arithmetic Standard* for floating-point arithmetic on virtually all platforms.
- This standard specifies how all arithmetic operations are carried out and many special functions are calculated.
- R uses double precision arithmetic for its floating-point calculations, which is carried out with 53 binary digits of precision.
- Note that $2^{-52} \approx 2.220446 \times 10^{-16}$, which explains the value of the relative machine precision.

Restrictions Imposed On Numbers

Numbers are accurate to about 16 significant digits.

The range of integers that can be represented exactly is from -2^{53} to 2^{53} (i.e., roughly -10^{17} to 10^{17}).

The maximum real value that can be represented is about

$$1.797693 \times 10^{308}.$$

The minimum positive real value that can be represented is about

$$2.225074 \times 10^{-308}.$$

Error in Floating-Point Computations

- Subtraction of positive values is one place where the finite precision of floating-point arithmetic is a potential problem.

```
> x = 1+1.234567890e-10
> print(x, digits = 20)
[1] 1.0000000001234568003

> y = x - 1
> print(y, digits = 20)
[1] 1.2345680033831740 73e-10
```

- There are 16 correct digits in `x`, but only 6 correct digits in `y`.

Cancellation Error

- The subtraction of nearly equal quantities is a major source of inaccuracy in numerical calculations.
- The inaccuracy occurs because the leading digits of the quantities cancel, revealing any accuracy in the representation of the values being operated on.
- Any calculations which involve cancellation of nearly equal quantities require special care.

An Example

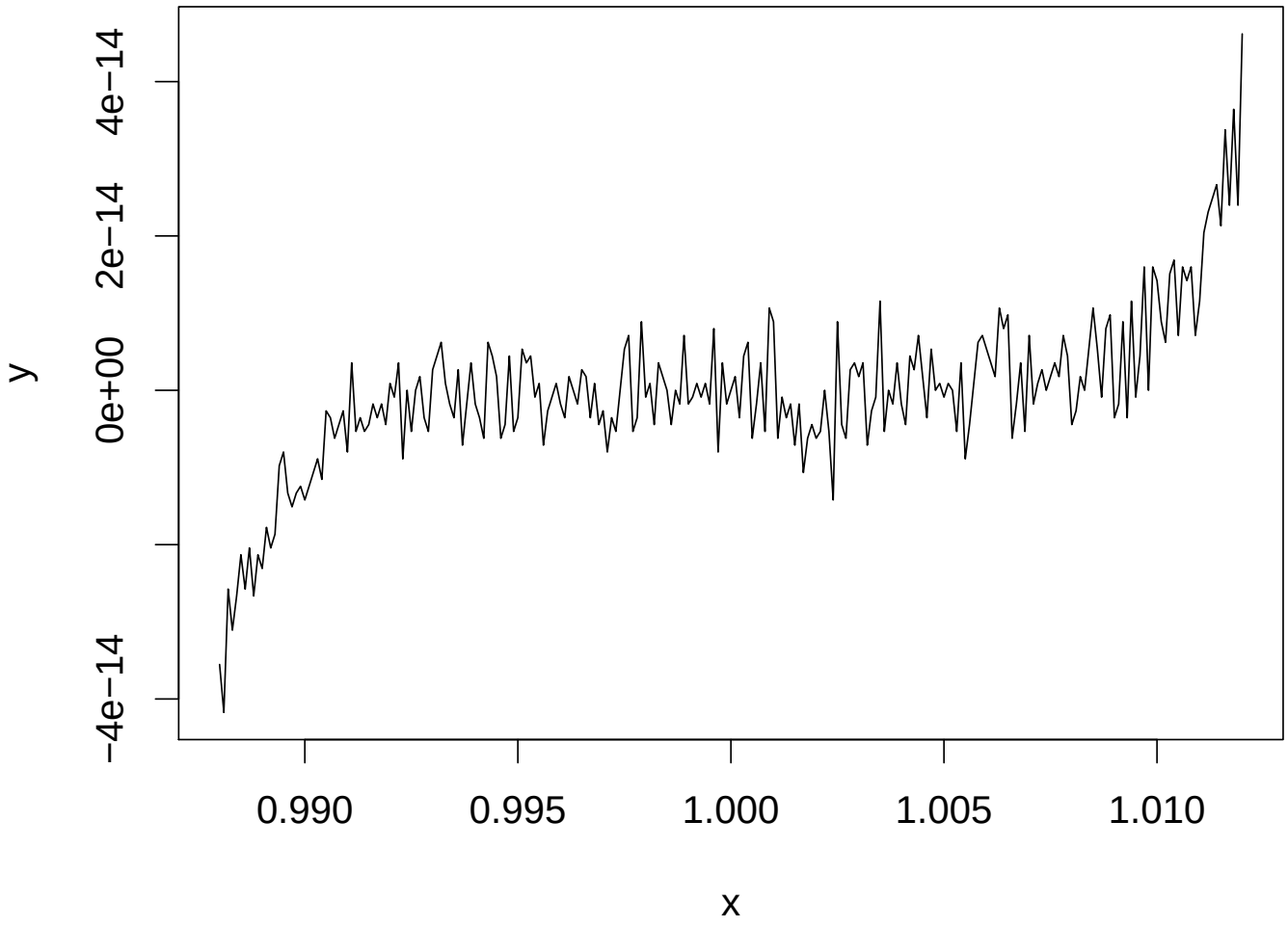
The function

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$

is a 7th degree polynomial, and its graph should appear very smooth.

To check this we can compute and graph the function over the range $[.988, 1.012]$.

```
> x = seq(.988, 1.012, by = 0.0001)
> y = x^7 - 7*x^6 + 21*x^5 - 35*x^4 +
      35*x^3 - 21*x^2 + 7*x - 1
> plot(x, y, type = "l")
```


y
−4e−14
0e+00
2e−14
4e−14
0.990
0.995
1.000
1.005
1.010
x

Problem Analysis

To see where the cancellation error comes from in this example, let's split the polynomial into individual terms and see what happens when we sum them.

```
> x = .99
> y = c(x^7, - 7*x^6, + 21*x^5, - 35*x^4, +
        35*x^3, - 21*x^2, + 7*x, - 1)
> cumsum(y)
[1]  9.320653e-01 -5.658296e+00  1.431250e+01
[4] -1.930837e+01  1.465210e+01 -5.930000e+00
[7]  1.000000e+00 -1.521006e-14
```

It is the last subtraction (of 1) which causes the catastrophic cancellation and loss of accuracy.

A Workaround

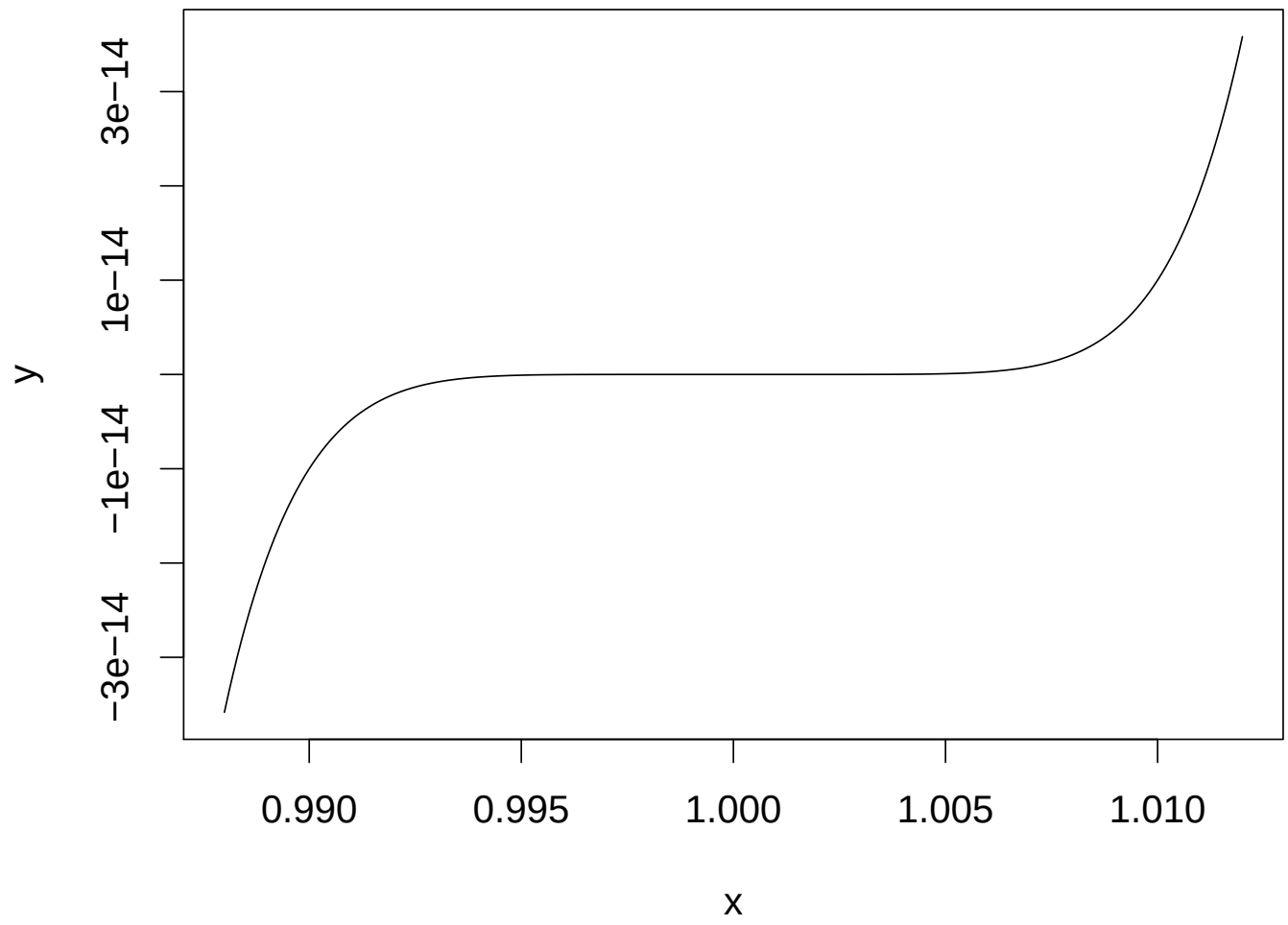
In this case we can reformulate the problem by noticing that

$$x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 = (x-1)^7.$$

Notice that although we are still getting cancellation, when 1 is subtracted from values close to 1, we are only losing a few digits of accuracy.

The difference is apparent in the plot.

```
> x = seq(.988, 1.012, by = 0.0001)
> y = (x - 1)^7
> plot(x, y, type = "l")
```


y
3e−14
1e−14
−1e−14
−3e−14
0.990
0.995
1.000
1.005
1.010
x

Example: The Sample Variance

While it is mathematically true that

$$\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2 = \frac{1}{n-1}\left[\sum_{i=1}^{n}x_i^2 - n\bar{x}^2\right],$$

the left-hand side of this equation provides a much better computational procedure for finding the sample variance than the right-hand side.

If the mean of x_i is far from 0, then $\sum x_i^2$ and $n\bar{x}^2$ will be large and nearly equal to each other. The relative error which results from applying the right-hand side formula can be very large.

There can, of course, be loss of accuracy using the formula on the left, but it is not nearly as catastrophic.

Example: Computing the Exponential Function

The exponential function is defined by the power series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!},$$

and this series can be used as the basis for an algorithm for computing e^x.

```
> expf =
      function(x, n = 10)
      1 + sum(x^(1:n) / cumprod(1:n))

> expf(1)
[1] 2.718282
```

The Exponential Function (Cont.)

By rearranging the function slightly, we can make the computation more efficient.

```
> expf =
      function(x, n = 10)
      1 + sum(cumprod(x / 1:n))

> expf(1)
[1] 2.718282
```

There is still the problem of choosing `n` so that the series is approximated accurately, but so that there are no extra computations.

Terminating Series Summation

Notice that the terms of the series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!},$$

eventually decrease in magnitude. This is because the nth term in the series is equal to x/n times the previous one. Eventually x/n must be less than 1 because, for any x value, the value of n must eventually be bigger than the value of x.

One strategy for summing the series is to accumulate terms of the series until the terms become so small that they do not change the sum.

Power Series – An Implementation

```
> expf =
      function(x)
      {
          n = 0
          term = 1
          oldsum = 0
          newsum = 1
          while(newsum != oldsum) {
              oldsum = newsum
              n = n + 1
              term = term * x / n
              newsum = newsum + term
          }
          newsum
      }
```

Power Series – Positive x

We can check the results of this function by comparing it with R's built-in exponential function. For positive x values, the results are good.

Indeed, the relative errors are close to the machine epsilon.

```
> (expf(1) - exp(1))/exp(1)
[1] 1.633713e-16

> (expf(20) - exp(20))/exp(20)
[1] -1.228543e-16
```

Power Series – Negative x

For negative x values, however, some of the relative errors can be very large.

```
> (expf(-1) - exp(-1))/exp(-1)
[1] 3.017899e-16

> (expf(-20) - exp(-20))/exp(-20)
[1] 1.727543
```

In the last case above, there are no correct digits in the computed value.

Why?

Power Series – Reasons

When the argument to `expf` is negative, the terms of the power series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!},$$

alternate in sign.

For large negative x values, the value returned by the function is small while at least some of the terms are large.

This means that, at some point, there has to be near-cancellation of the accumulated sum and the next term of the series. This is the source of the large relative error.

When the argument to `expf` is positive, all the terms of the series are positive and there is no cancellation.

Power Series – A Fix

Fortunately, there is a relatively simple way to correct the algorithm by recalling that

$$e^{-x} = 1/e^{x}.$$

This means that, when x is negative, it is better to compute the result as

$$e^{x} = 1/e^{-x}.$$

Power Series - A Better Algorithm

```
> expf =
      function(x)
      {
          if ((neg = (x < 0))) x = -x
          n = 0; term = 1
          oldsum = 0; newsum = 1
          while(newsum != oldsum) {
              oldsum = newsum
              n = n + 1
              term = term * x / n
              newsum = newsum + term
          }
          if (neg) 1/newsum else newsum
      }
```

A Vectorised Algorithm

The functions we have defined only work for scalar functions. It is not too hard to vectorise them.

Given an input vector of k values, $x_1, \ldots, x_k$, we keep a vector of sums, $s_1, \ldots, s_k$, and continue adding terms $t_1, \ldots, t_k$, updating the terms with

$$\begin{pmatrix} t_1 \\ \vdots \\ t_k \end{pmatrix} \leftarrow \frac{1}{n} \begin{pmatrix} t_1 \\ \vdots \\ t_k \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix}.$$

In the case of `expf` we continue adding terms to the sums until there is no change in any of the sums.

Vectorisation Function

```
> expf =
      function(x)
      {
          x = ifelse((neg = (x < 0)), -x, x)
          n = 0; term = 1
          oldsum = 0; newsum = 1
          while(any(newsum != oldsum)) {
              oldsum = newsum
              n = n + 1
              term = term * x / n
              newsum = newsum + term
          }
          ifelse(neg, 1/newsum, newsum)
      }
```

Vectorisation Results and Accuracy

```
> x = c(-20, -10, 0, 10, 20)

> expf(x)
[1] 2.061154e-09 4.539993e-05 1.000000e+00
[4] 2.202647e+04 4.851652e+08

> exp(x)
[1] 2.061154e-09 4.539993e-05 1.000000e+00
[4] 2.202647e+04 4.851652e+08

> (expf(x) - exp(x))/exp(x)
[1]  2.006596e-16  1.492571e-16  0.000000e+00
[4] -3.303280e-16 -1.228543e-16
```

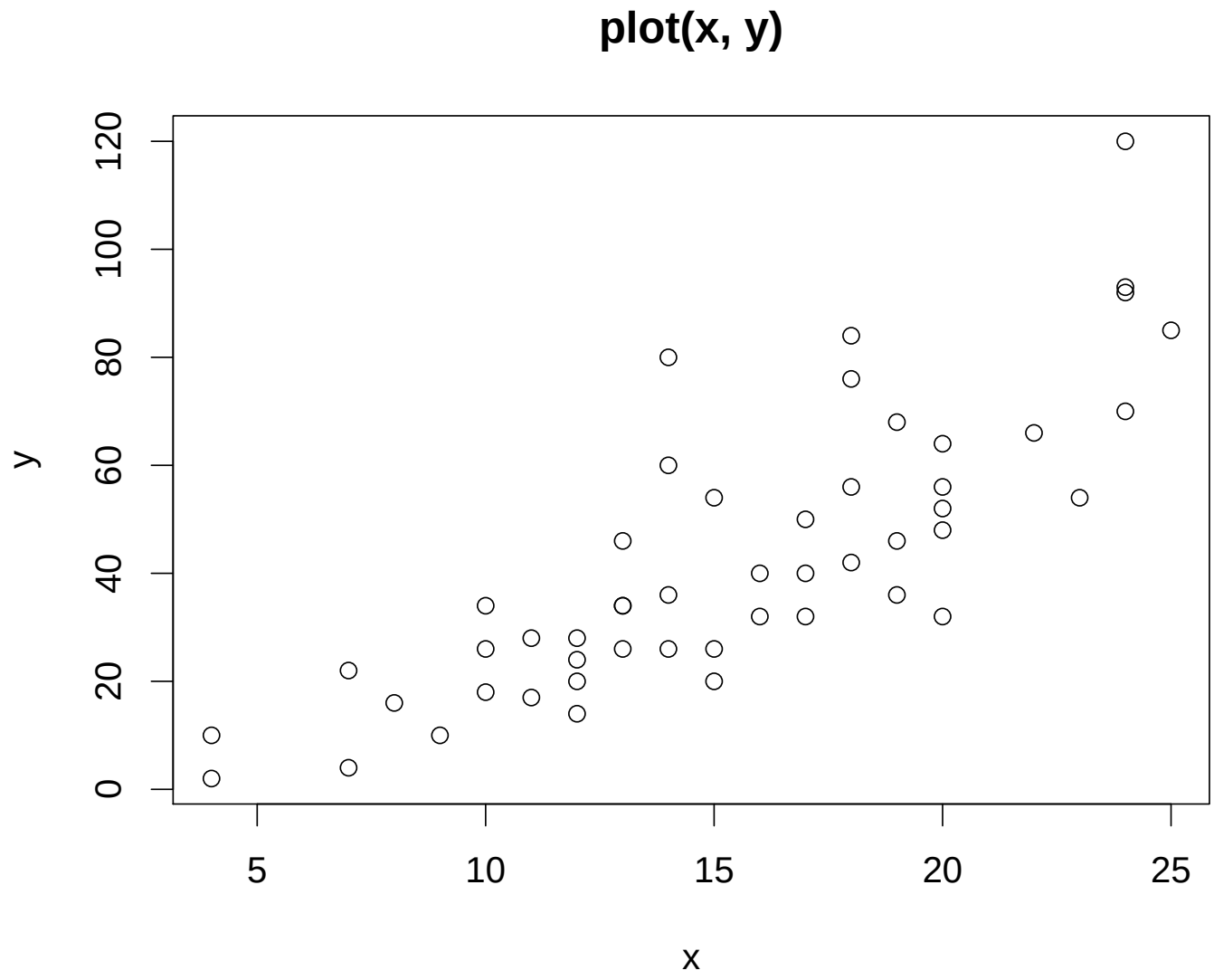
Simple Graphics

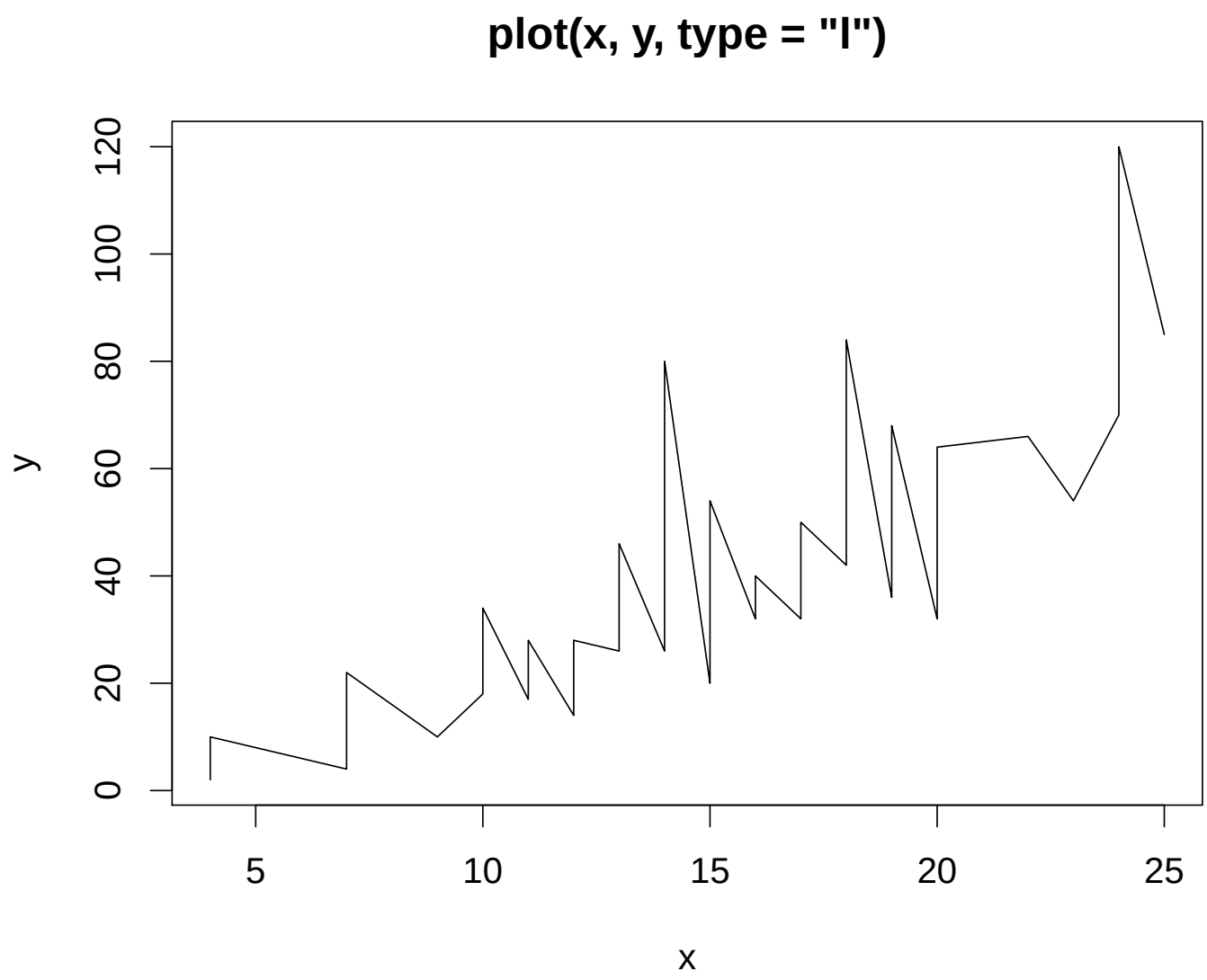
The function `plot` provides basic graphics facilities. The function call

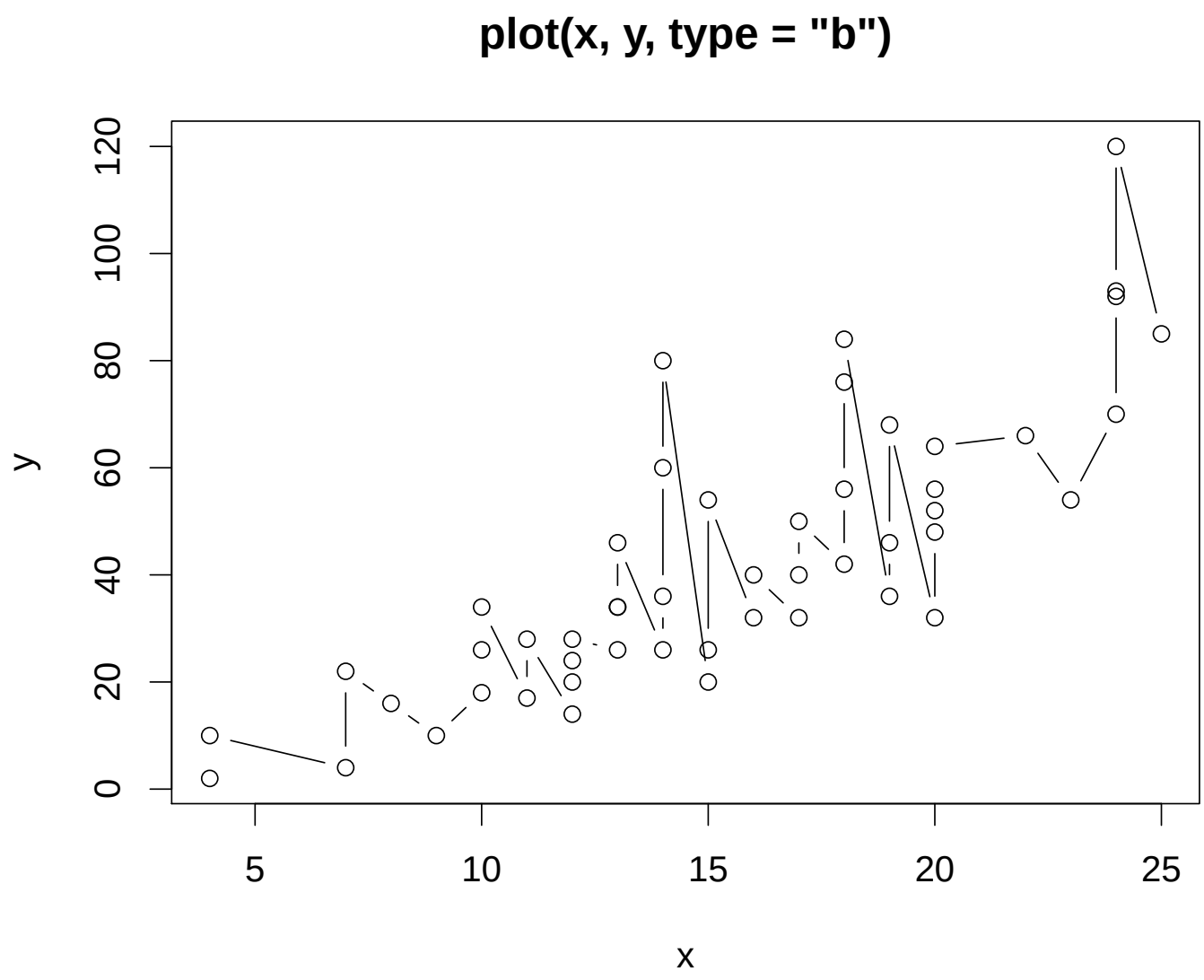
```
plot(x, y)
```

produces a scatter plot, with the points located at the coordinates in the variables `x`, `y`.

- If only one argument is provided it is used as the `y` variable. The `x` variable is assumed to have the value `1:length(y)`.
- If an additional argument, `type="l"`, is provided in the call, the points are joined with straight lines.
- If an additional argument, `type="b"`, is provided in the call, both points and lines are drawn.


plot(x, y)
y
0
20
40
60
80
100
120
5
10
15
20
25
x


plot(x, y, type = "l")
y
0
20
40
60
80
100
120
5
10
15
20
25
x


plot(x, y, type = "b")
y
0
20
40
60
80
100
120
5
10
15
20
25
x

Color Control In R

There are a variety of ways to specify a colour in R.

- For compatibility with S, colours can be specified as small positive integers.
- Colors can be specified by name. A PDF colour chart is available on the class website.
- Colors can by specifying the amount of Red, Green and Blue which are mixed to produce the color.

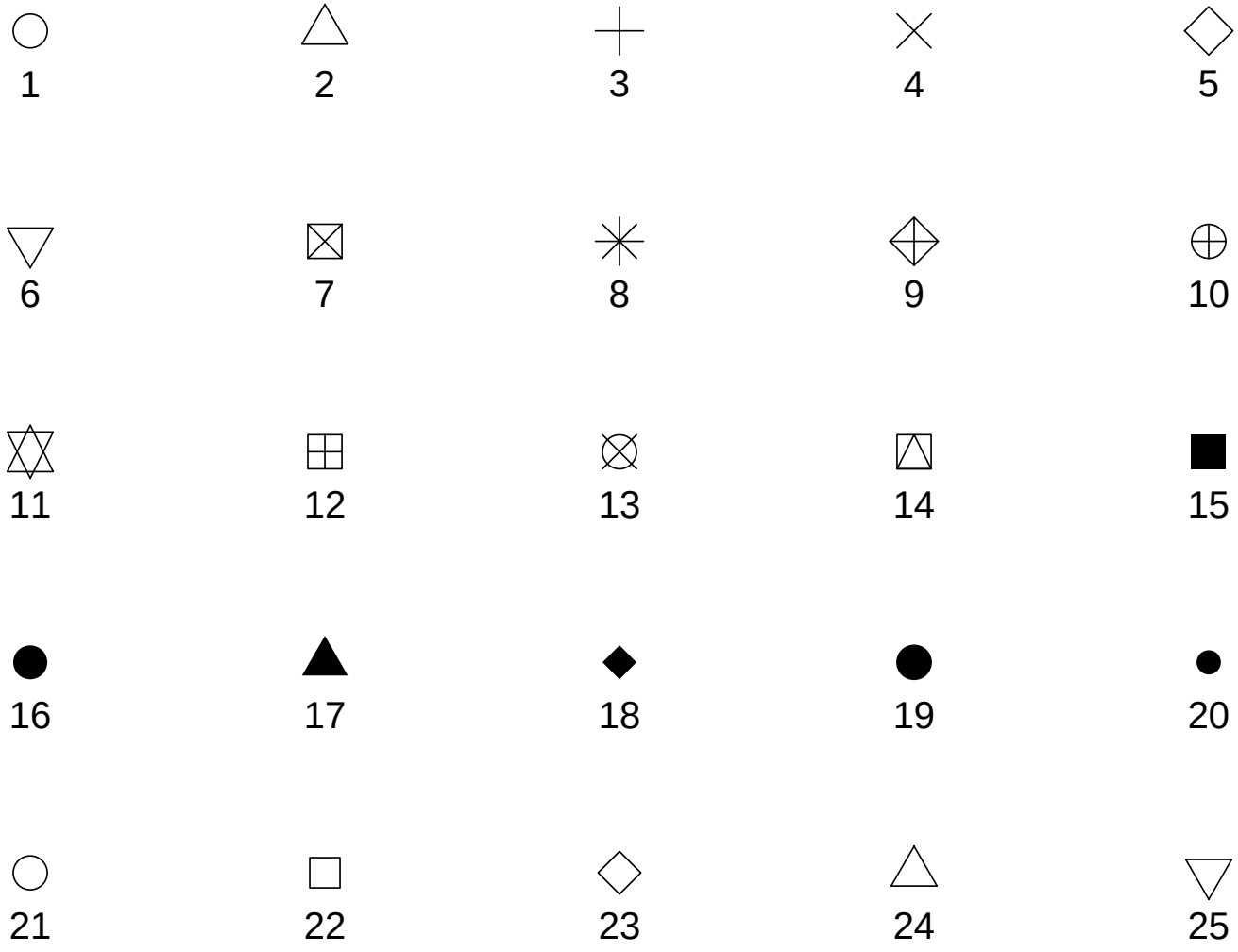
Some valid color specifications in R are:

`col=2`	this is Red
`col="purple"`	this Purple
`col="#FFFF00"`	this is Yellow
`col=rgb(1,0,1)`	this is Magenta
`col=hsv(1/12,1,1)`	this is Orange(ish)

Control of Plotting Symbols

- Plotting symbols are specified by setting the `pch=` argument to an integer in the range from 0 to 25.
- 0 corresponds to the “blank” or invisible symbol.
- The symbols corresponding to the values 1 through 14 are drawn with lines.
- The symbols corresponding to the values 15 through 20 are drawn in solid colors.
- The symbols corresponding to the values 21 through 25 can be filled with a solid color and have their borders drawn with a different color.

R Plotting Symbols



Line Customisation

The line *type* can be specified with an argument of the form `lty=`*type*. The line type can be specified by name:

> `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`,
> `"longdash"`, `"twodash"`,

or as a hexadecimal specification of the form `"DUDU"`, where `D` represents a distance drawn with "pen down" and `U` is a distance with "pen up."

`"1111"`	high density dots
`"1313"`	spaced-out dots
`"1333"`	dot-dash
`"3373"`	short-dash, longdash

The widths of lines can be specified with the parameter `lwd=`*wd*, with thicknesses specified relative to the default of 1.

Annotation

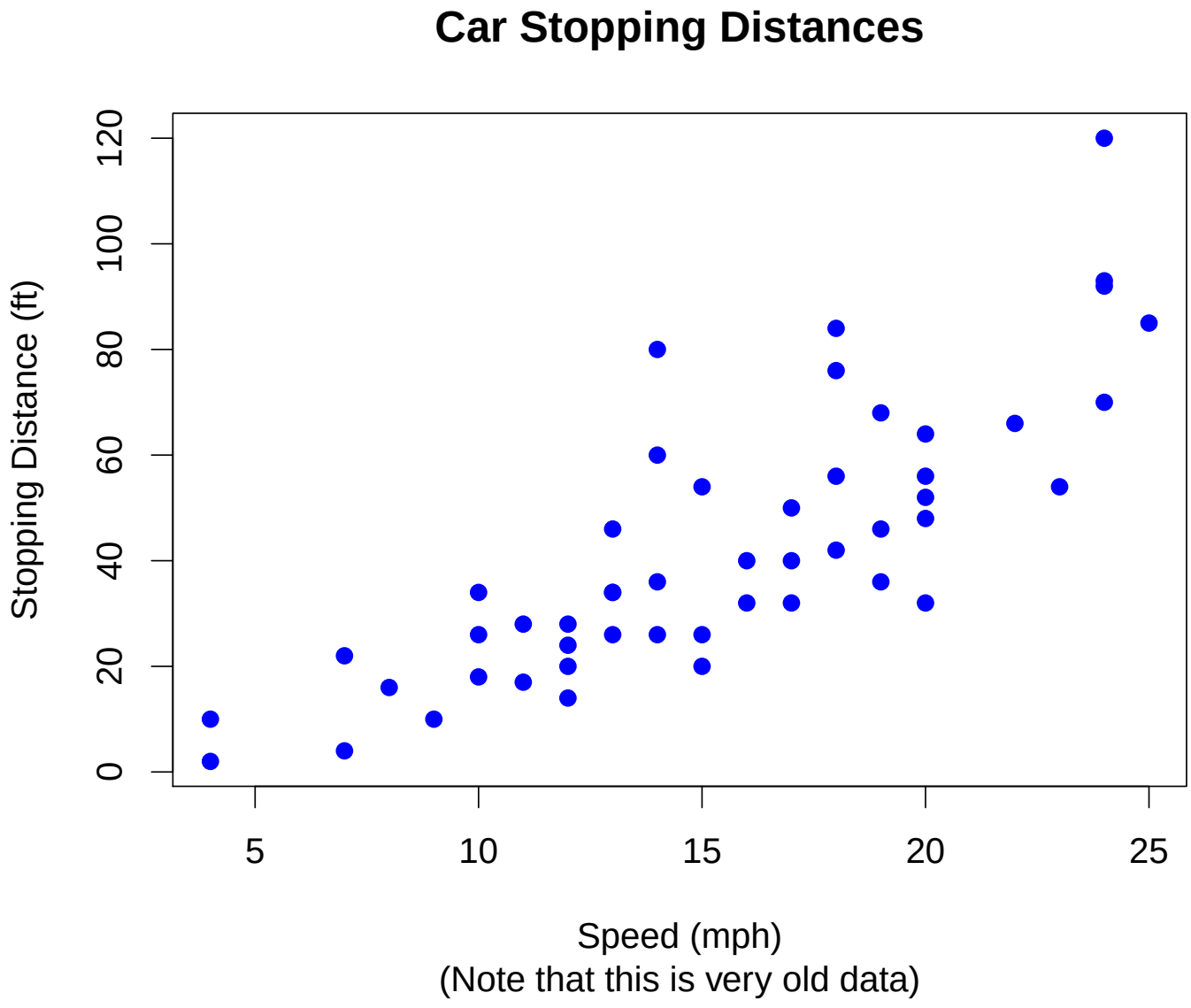
The `plot` function has arguments which can be used to customized the labelling on a plot.

`main=`	an overall title for the plot
`xlab=`	a label for the x-axis
`ylab=`	a label for the y-axis
`sub=`	a subtitle to be placed below the plot

Annotation can be added to an existing plot by calling the function `title` with the same arguments.

An Example

```
> plot(cars$speed, cars$dist,
       pch = 19, col = "blue",
       main = "Car Stopping Distances",
       xlab = "Speed (mph)",
       ylab = "Stopping Distance (ft)",
       sub = "(Note that this is very old data)")
```


Car Stopping Distances
Stopping Distance (ft)
0
20
40
60
80
100
120
5
10
15
20
25
Speed (mph)
(Note that this is very old data)

Customising Plot Axes

The following `plot` arguments make it possible to customise a plot's axes.

`xlim`	limits on the *x* axis
`ylim`	limits on the *y* axis
`log`	which axes are logarithmic (e.g. `"x"`, `"xy"`).
`asp`	ratio of *y* distances to *x* distances

The main use of `asp` is `asp=1` which means that equal data units on the *x* and *y* axes are represented by equal distances on screen.

Adding to Plots

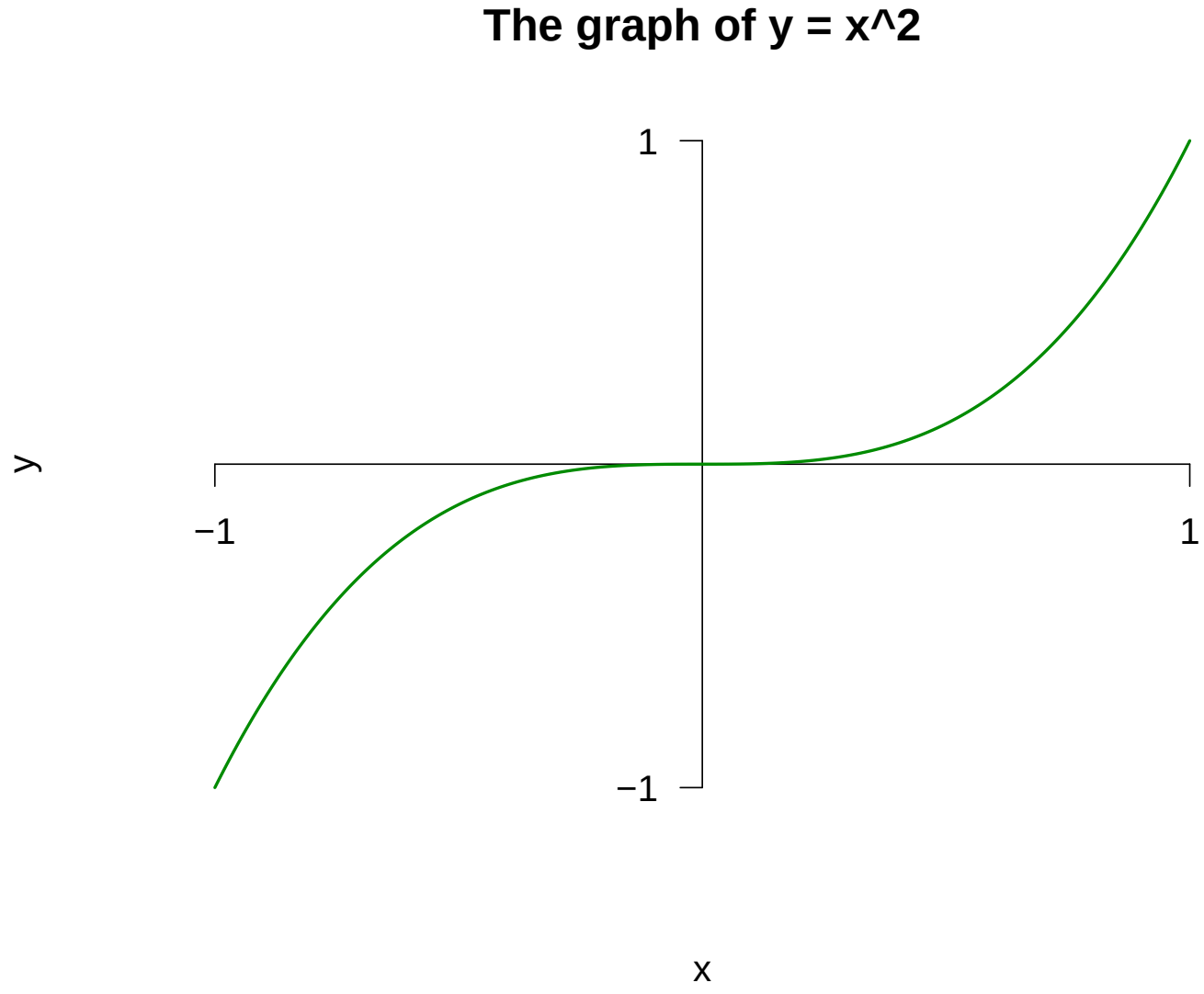
The following functions can be used to add to existing plots.

`points`	draw points
`lines`	draw connected line segments
`abline`	add a straight line to a plot
`segments`	draw disconnected line segments
`arrows`	add arrows to a plot
`rect`	add rectangles to a plot
`polygon`	add polygons to a plot
`text`	add text to a plot

Building Plots

The following example shows how plots can be built up incrementally from component parts. The initial `plot` call simply establishes an empty plotting region and the other function calls add graphical elements to it.

```
> x = seq(-1, 1, length = 1001)
> y = x^3
> plot(x, y, type = "l", axes = FALSE, ann = FALSE)
> axis(1, at = c(-1, 1), pos = 0)
> axis(2, at = c(-1, 1), pos = 0, las = 1)
> lines(x, y, col = "green4", lwd = 2)
> title(main = "The graph of y = x^2")
> title(xlab = "x", ylab = "y")
```


The graph of y = x^2
1
y
−1
1
−1
x

Root Finding

- Finding the solutions of equations of the form $g(x) = c$ is a very important problem in applied mathematics (and statistics).
- By writing $f(x) = g(x) - c$, the above equation is equivalent to the equation $f(x) = 0$, and we can restrict our attention to finding the solutions of this type of equation.
- The solutions are called *roots* or *zeros* of the function $f(x)$.
- A function may have zero, one, or many roots.

The Roots of a General Polynomial

- R has a built-in routine, called `polyroot`, for finding *all* the roots of a polynomial.
- The coefficients of the polygon

$$a_n x^n + \cdots + a_1 x + a_0$$

  are passed in a vector $(a_0, a_1, \ldots, a_n)$ to `polyroot`.
- The computed roots are returned in a vector.
- Since, in general, the roots of polynomials are complex-valued, this vector has mode `"complex"`.

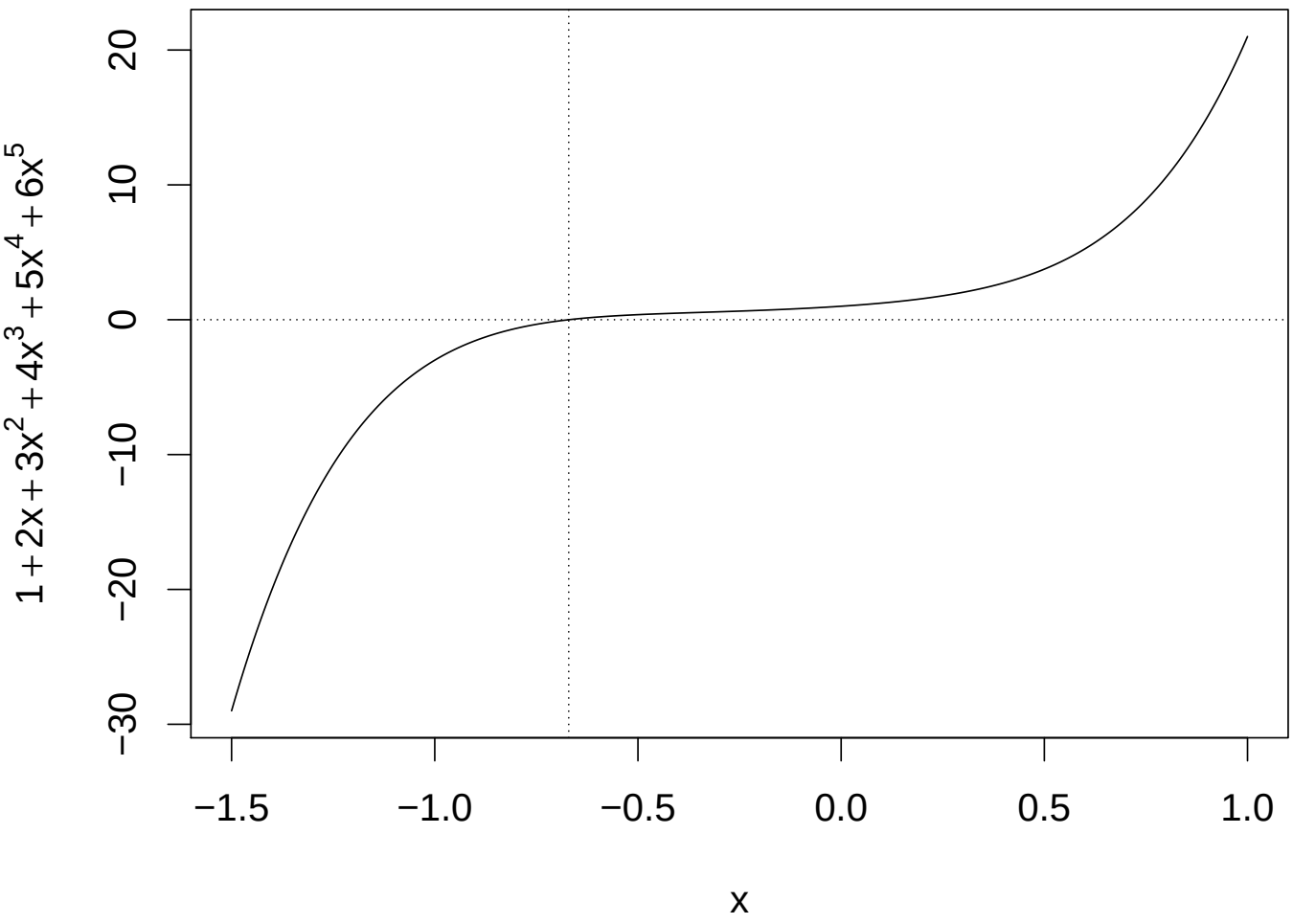
The Roots of a General Polynomial

We can find the roots of the two equations

$$x^2 + 2x + 1 = 0$$
$$6x^5 + 5x^4 + 4x^3 + 3x^2 + 2x + 1 = 0$$

as follows.

```
> polyroot(c(1, 2, 1))
[1] -1-0i -1+0i

> polyroot(1:6)
[1]  0.2941946+0.6683671i
[2] -0.3756952+0.5701752i
[3] -0.3756952-0.5701752i
[4]  0.2941946-0.6683671i
[5] -0.6703320+0.0000000i
```


$1+2x+3x^2+4x^3+5x^4+6x^5$
20
10
0
−10
−20
−30
−1.5
−1.0
−0.5
0.0
0.5
1.0
x

An R Function for General Root-Finding

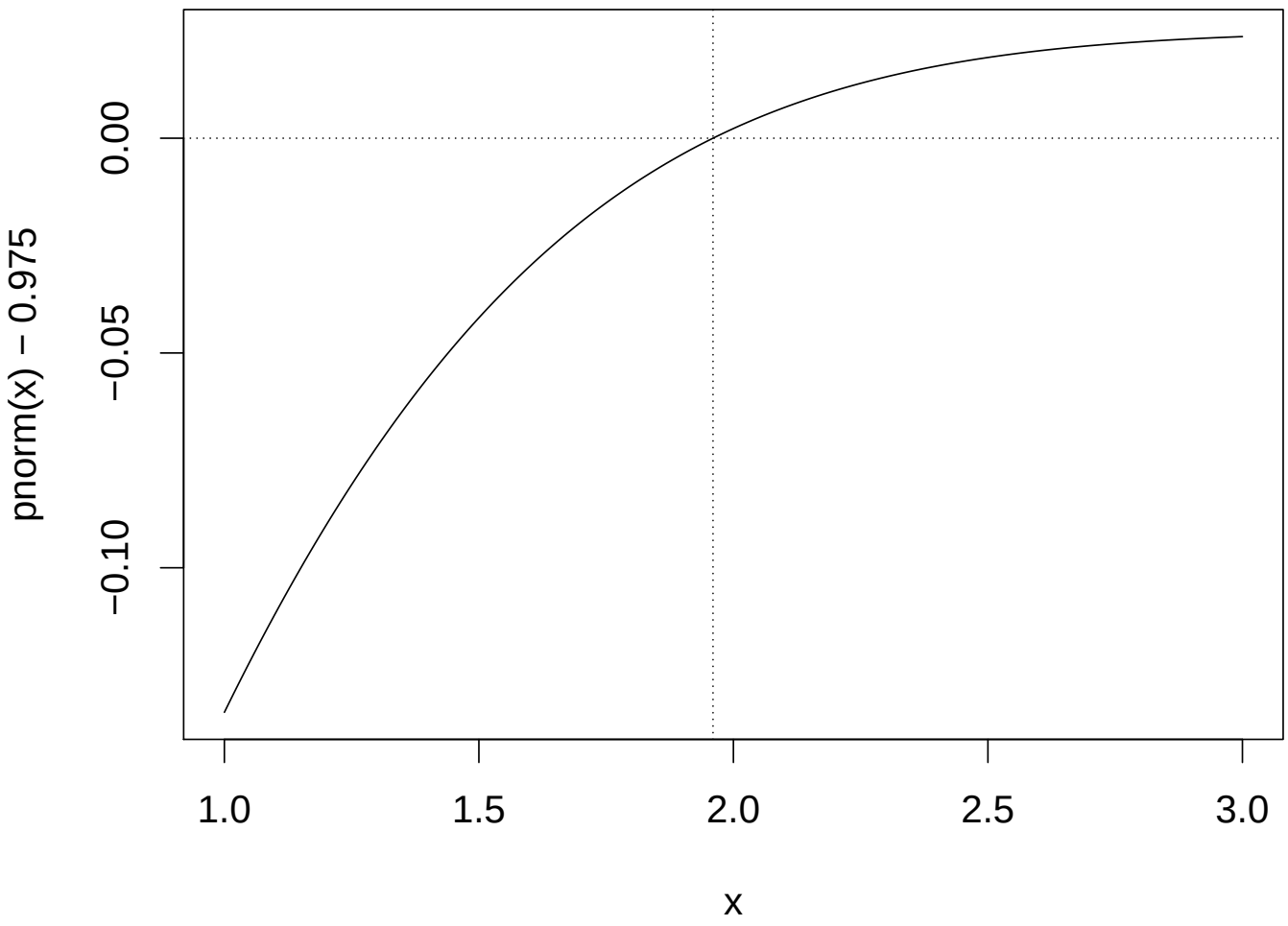
- The function `uniroot` in R locates a root of a function over an interval.
- The function values at the endpoints of the interval need to be of opposite sign, which thus ensures the existence of at least one root within the interval.
- The values of the interval endpoints must be provided. They can be passed to the function by either the argument `interval` as a vector of two values, or the arguments `lower` and `upper`.

Returned Value - Default Settings

```
> uniroot(function(x) pnorm(x) - .975,
          lower = 1, upper = 3)[-4]
$root
[1] 1.959966

$f.root
[1] 1.128129e-07

$iter
[1] 7

$estim.prec
[1] 6.103516e-05
```

Returned Value - More Accuracy

```
> uniroot(function(x) pnorm(x) - .975,
          lower = 1, upper = 3, tol = 1e-10)[-4]
$root
[1] 1.959964

$f.root
[1] 0

$iter
[1] 8

$estim.prec
[1] 0.0001787702
```


pnorm(x) − 0.975
0.00
−0.05
−0.10
1.0
1.5
2.0
2.5
3.0
x

Finding Minima and Maxima

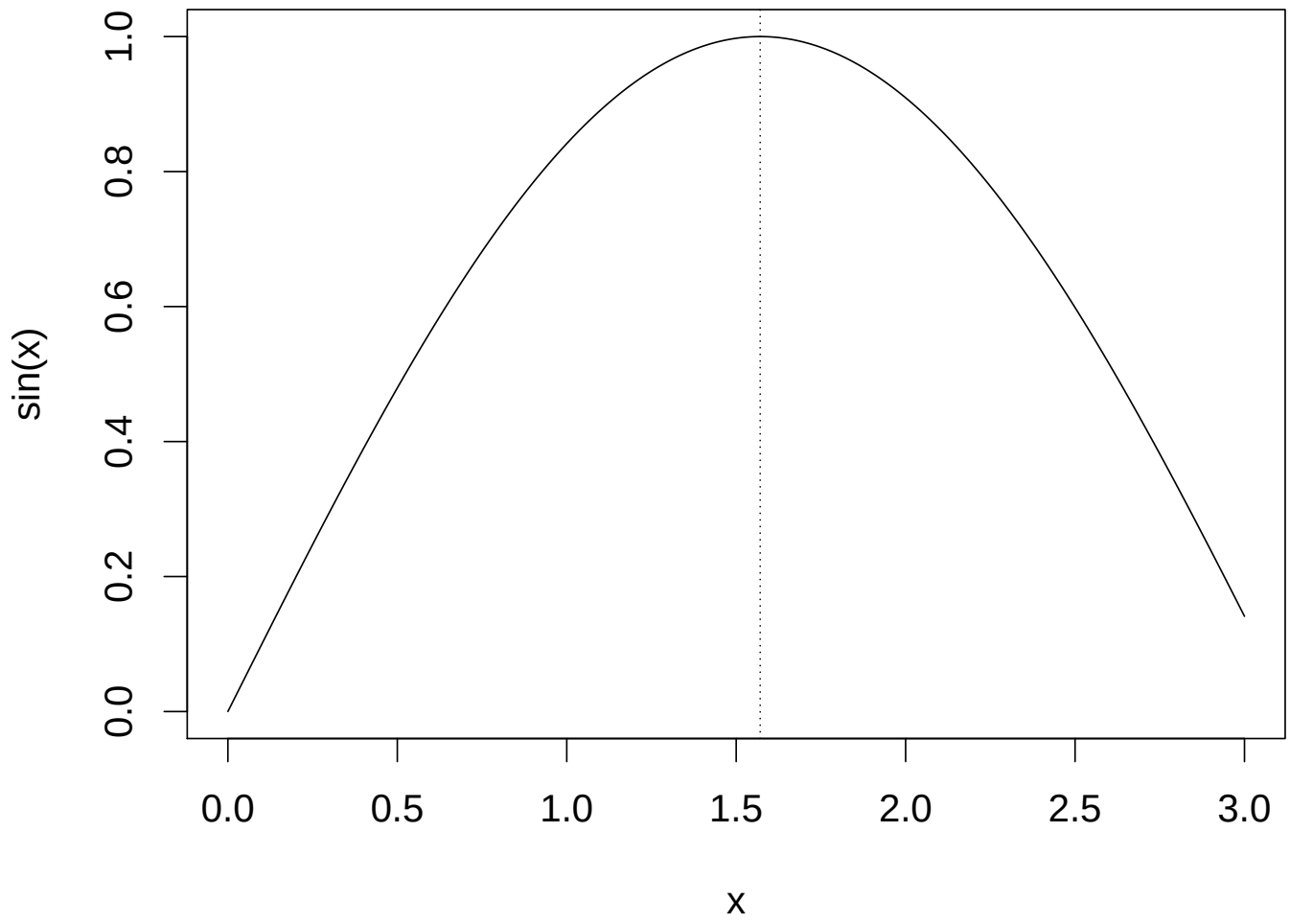
- The problem of finding the local minimum or maximum of a function is related to root finding.
- If f is differentiable, the extrema occur at the roots of the derivative of f'.
- When the derivative of f is unknown or difficult to calculate there are direct search methods which will locate extrema.

The `optimize` Function

- The R function `optimize` can be used to locate extrema of functions.
- It can be used to search for a local maximum or local minimum of a function in a given interval.
- The `tol` controls the accuracy with which the search is carried out.

Example: Finding a Function Maximum

```
> f = function(x) sin(x)

> optimize(f, lower = 0, upper = 3,
           maximum = TRUE, tol = 1e-10)
$maximum
[1] 1.570796

$objective
[1] 1

> pi/2
[1] 1.570796
```


sin(x)
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x

Interpolation

- To *Interpolate* is to seek a function $f(x)$ that passes exactly through a given set of points (x_i, y_i), $i = 1, \ldots, n$.
- Without loss of generality, we assume that $x_1 < x_2 < \cdots < x_n$.
- Interpolation provides a way of visualising the appearance of an entire function when only a few values are known.
- Interpolation can also be used to provide an economical way of approximating functions. A small table of values is kept and interpolation used to obtain other values.

Spline Interpolation

- Spline interpolation produces results using a polynomial (often of low degree) to interpolate between the values.
- Adjacent polynomials are made to agree at their common endpoint (because they interpolate) and it is possible to make the joins smooth (in the sense that the derivatives are continuous).

Cubic Splines

- Cubic splines are the most commonly used splines in practice. A cubic spline needs to possess the following properties.
  - Interpolation: $s(x_i) = y_i$, $i = 0, \ldots, n$.
  - Continuity: $s_{i-1}(x_i) = s_i(x_i)$, $i = 1, \ldots, n-1$.
  - Smoothness: $s'_{i-1}(x_i) = s'_i(x_i)$ and $s''_{i-1}(x_i) = s''_i(x_i)$, $i = 1, \ldots, n-1$.
- The properties above do not determine the splines completely; an additional constraint is still required at the two endpoints x_0 and x_n.

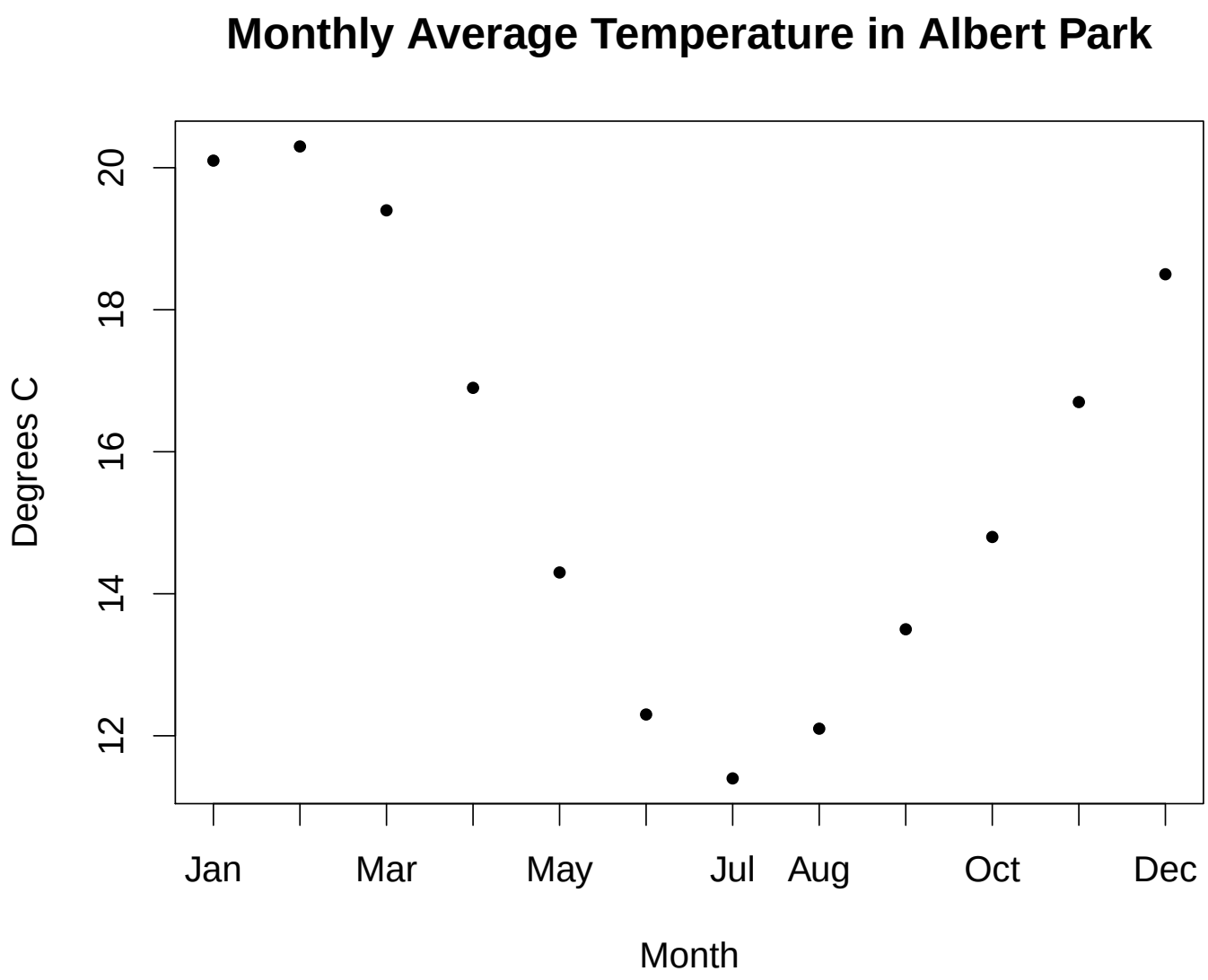
Endpoint Rules

- *Forsythe, Malcolm and Moler*: A quadratic curve is interpolated through the first three and last three points and the slope of these curves at the endpoints is used as the value of $s'(x_0)$ and $s'(x_n)$.
- *Natural Splines*: It is assumed that $s''(x_0) = s''(x_n) = 0$.
- *Periodic Splines*: It is assumed that function can be extended periodically beyond the end of the interval $[x_0, x_n]$, which means that $s(x_0) = s(x_n)$, $s'(x_0) = s'(x_n)$ and $s''(x_0) = s''(x_n)$.

An Example: Temperatures in Albert Park

The following table gives the monthly average temperatures, in °C, observed at the meteorology station in Albert Park, which is adjacent to the University of Auckland.

Jan	*Feb*	*Mar*	*Apr*	*May*	*Jun*
20.1	20.3	19.4	16.9	14.3	12.3
Jul	*Aug*	*Sep*	*Oct*	*Nov*	*Dec*
11.4	12.1	13.5	14.8	16.7	18.5


Monthly Average Temperature in Albert Park
Degrees C
12
14
16
18
20
Jan
Mar
May
Jul
Aug
Oct
Dec
Month

Interpolating Temperatures

- These monthly values can be regarded as the output of a moving average of (approximately) 30 consecutive daily average temperatures, computed at the middle of each month.
- To get a more complete idea about how temperature varies throughout the year we could examine the result of smoothly *interpolating* these values.
- Given a set of points to be interpolated, the R function `splinefun` computes the interpolating spline function.

Interpolating Temperatures

- In the case of the temperature values, it is most sensible to use a *periodic spline* because the temperatures wrap around from December to January.

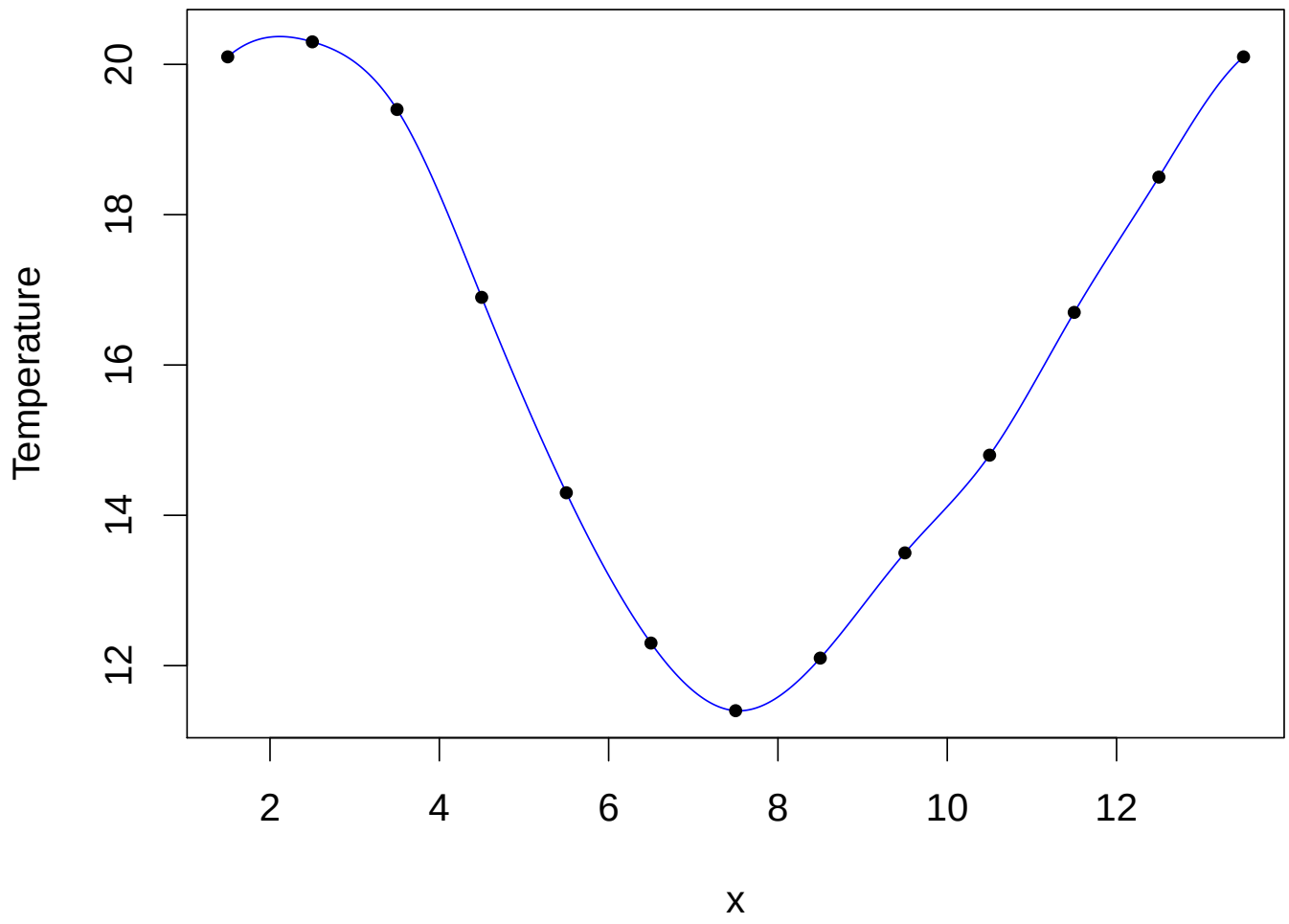
```
> xm = 1:13 + .5
> ym = c(20.1, 20.3, 19.4, 16.9, 14.3,
         12.3, 11.4, 12.1, 13.5, 14.8,
         16.7, 18.5, 20.1)
> f = splinefun(xm, ym, method = "periodic")
```

- The result returned by `splinefun` is a *function* which can be used to interpolate the specified values.

Interpolating Temperatures

- The interpolating function for the temperatures can be plotted by evaluating it at a finely spaced grid of points from 1 (January) to 13 (the next January).

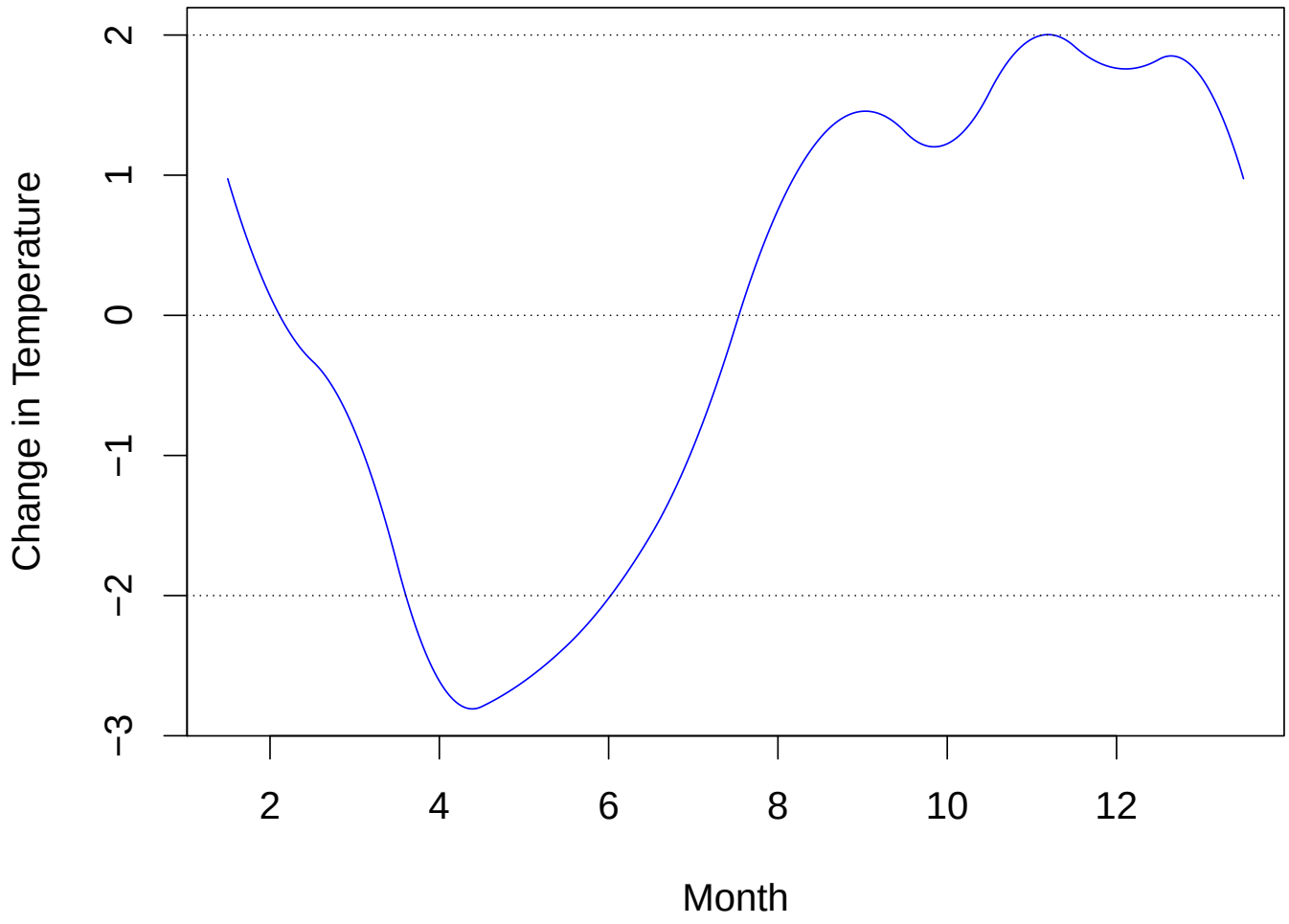
```
> x = seq(1.5, 13.5, length = 2000)
> plot(x, f(x), type = "l", col = "blue",
       ylab = "Temperature")
> points(xm, ym, pch=20)
```


Temperature
12
14
16
18
20
2
4
6
8
10
12
x

Heating and Cooling Rates

- The plot suggests that cooling in Autumn occurs at a faster rate than the warming in spring.
- We can check this by differentiating the interpolated function.

```
> plot(x, f(x, deriv = 1),
       type = "l", col = "blue",
       ylab = "Change in Temperature",
       xlab = "Month")

> abline(h = c(-2, 0, 2), lty = "dotted")
```


Change in Temperature
2
1
0
−1
−2
−3
2
4
6
8
10
12
Month

The Hottest Day

- We can search for the maximum temperature as follows.

```
> imax = optimize(f, lower = 1, upper = 3,
                  maximum = TRUE)
> imax
$maximum
[1] 2.113256

$objective
[1] 20.37027
```

- The maximum average temperature is about 20.37 degrees and occurs just after the start of February.

Hottest Day

- Using the fact that February has 28 days, we can convert the offset into days as follows.

```
> 28 * (imax$maximum - 2)
[1] 3.171179
```

- The maximum (interpolated) average temperature occurs at about the 3rd of February.

Coldest Day

- We can use `optimize` to locate the minimum temperature in a very similar fashion.

```
> imin = optimize(f, lower = 6, upper = 8)
> imin
$minimum
[1] 7.538566

$objective
[1] 11.39856
```

- The minimum temperature of 11.40 occurs at about the 17th of July.

Optimisation

Optimisation is the process of finding a minimum or maximum for a function.

Given a function $f(x)$ we want to determine either the point, $x_{\min}$, where $f(x)$ attains its minimum value

$$f(x_{\min}) = \min_x f(x),$$

or the point $x_{\max}$ where $f(x)$ attains its maximum value

$$f(x_{\max}) = \max_x f(x).$$

Optimisation of Smooth Functions

When the function to be optimised is smooth (i.e. differentiable) then the optima can be found by locating the roots of the equation

$$f'(x) = 0.$$

Sometimes the functions to be minimised are smooth, and it may even be possible to carry out the process explicitly.

Example: Given the values:

0.06	0.36	0.59	0.73	0.96
0.22	0.85	0.05	0.09	

find the point θ such that the sum of the squared distances of the points from θ is a minimum.

Mathematical Analysis

In general, if the points are $x_1, \ldots, x_n$, then we are seeking to minimise the value of:

$$Q(\theta) = \sum_{i=1}^{n} (x_i - \theta)^2.$$

We can find the minimum by solving the equation $Q'(\theta) = 0$.

$$0 = \sum_{i=1}^{n} 2(x_i - \theta) = \sum_{i=1}^{n} 2x_i - \sum_{i=1}^{n} 2\theta$$

Which means

$$n\theta = \sum_{i=1}^{n} x_i \quad \text{or} \quad \theta = \bar{x}.$$

Optimisation of General Functions

If $f(x)$ is not smooth, we can't use calculus to find the optima.

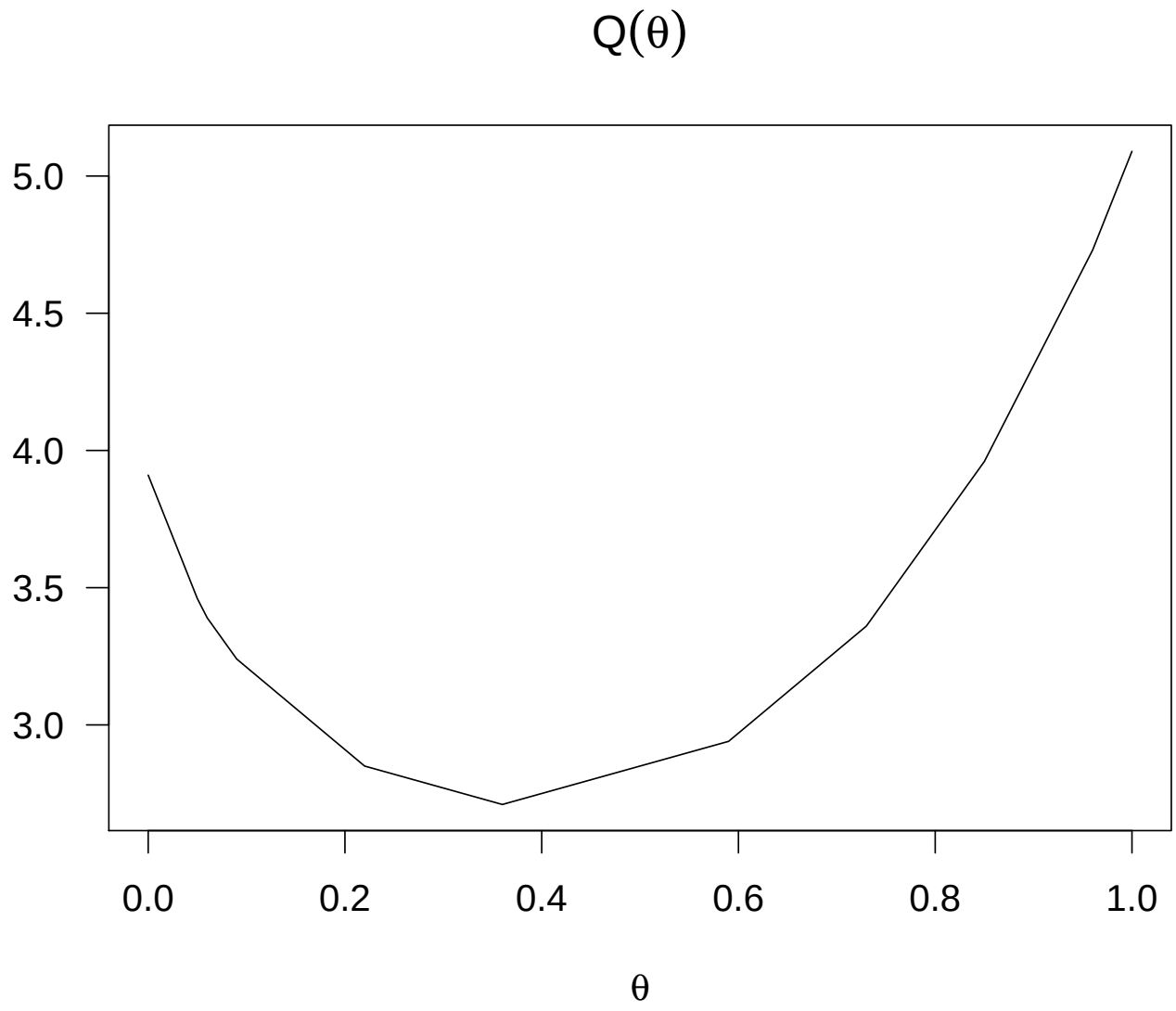
Example: Given the values:

0.06	0.36	0.59	0.73	0.96
0.22	0.85	0.05	0.09	

find the point θ such that the sum of the absolute distances of the points from θ is a minimum.

In this case we are looking for the minimum of

$$Q(\theta) = \sum_{i=1}^{n} |x_i - \theta|.$$


Q(θ)
5.0
4.5
4.0
3.5
3.0
0.0
0.2
0.4
0.6
0.8
1.0
θ

Optimisation in R

R provides a function, called `optimise`, for numerically optimising functions of one variable (smooth or not). The function looks for either a maximum or a minimum of a given function in a specified interval.

The optimum value found by `optimise` will be either a local minimum or local maximum of the given function. It may not be the global minimum or maximum.

By default a minimum is located. If a maximum is required, the optional argument, `maximum = TRUE`, should be specified.

Example: Least-Squares

Lets look for the value which minimises the sum of squared deviations of the values 0.06, 0.36, 0.59, 0.73, 0.96, 0.22, 0.85, 0.05, 0.09 around a given point.

```
> y = c(0.06, 0.36, 0.59, 0.73, 0.96,
        0.22, 0.85, 0.05, 0.09)
> Q = function(theta) sum((y - theta)^2)

> optimize(Q, c(0, 1), tol = 1e-12)
$minimum
[1] 0.4344444

$objective
[1] 1.018622
```

Example: Least-Squares (Cont.)

The value computed by `optimize` is 0.4344444. We know that the minimising value is the mean so let's check it.

```
> mean(y)
[1] 0.4344444
```

The lesson here is that mean is the solution to a particular optimisation problem.

It turns out that many statistical procedures are actually the solutions to optimisation problems. This means that optimisation is a very important methodology for statisticians.

Example: Least Absolute Deviations

Let's look for the solution of the least absolute deviations problem.

```
> Q = function(theta) sum(abs(y - theta))

> optimise(Q, c(0, 1), tol = 1e-12)$minimum
[1] 0.36
```

Example: Least Absolute Deviations

Let's look for the solution of the least absolute deviations problem.

```
> Q = function(theta) sum(abs(y - theta))
```

```
> optimise(Q, c(0, 1), tol = 1e-12)$minimum
[1] 0.36
```

This turns out to be a very familiar value.

```
> median(y)
[1] 0.36
```

(A short geometric argument shows that this is always the case.)

Optimising Functions of Two or More Variables

The `optimize` function only works for functions of one variable. Many problems require that we maximise or minimise functions of two or more variables.

When two or more variables are involved, a function called `optim` must be used instead.

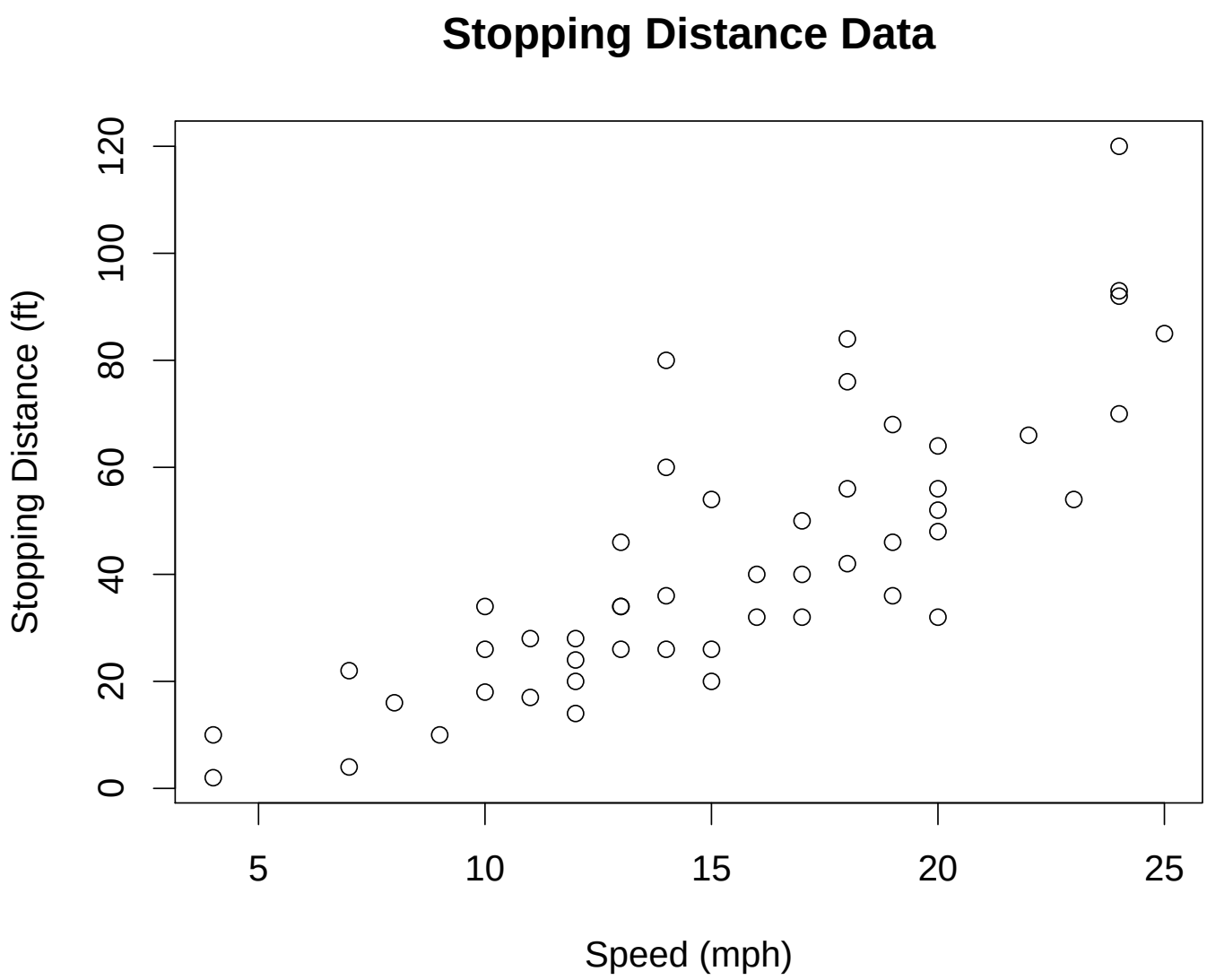
`optim` will work on both smooth and non-smooth functions, but if the function is smooth, much better performance can be obtained by specifying that `optim` should use methods tuned for smooth functions.

Example: Least Absolute Deviations

To illustrate the process, we'll use the `cars` data set.

This data set contains observations on stopping distances (in feet) and speeds (in miles per hour) for a set of cars.

We'll fit the stopping distance as a function of speed using least absolute deviations.


Stopping Distance Data
Stopping Distance (ft)
0
20
40
60
80
100
120
5
10
15
20
25
Speed (mph)

Example: Least Absolute Deviations

We start by defining the function to be minimised. This needs to be a function of one variable; the parameters stored as a vector.

```
> Q = function(theta)
      sum(abs(cars$dist - theta[1] -
              theta[2] * cars$speed))
```

`optim` works by improving an initial estimate of the optimum. We can use least-squares as a starting point.

```
> theta = coef(lm(dist~speed, data = cars))
> theta
(Intercept)       speed
 -17.579095    3.932409
```

Example: Least Absolute Deviations

Now we call `optim` with the given starting point and function.

```
> z = optim(theta, Q)
```

The `optim` function returns a list containing information about the optimisation process.

The first thing to examine is the component named `convergence`. If this is zero, then process converged normally and we can move on to examine the results.

```
> z$convergence
[1] 0
```

Example: Least Absolute Deviations

The computed estimates are stored in the component named `par`.

```
> z$par
(Intercept)       speed
 -11.600009    3.400001
```

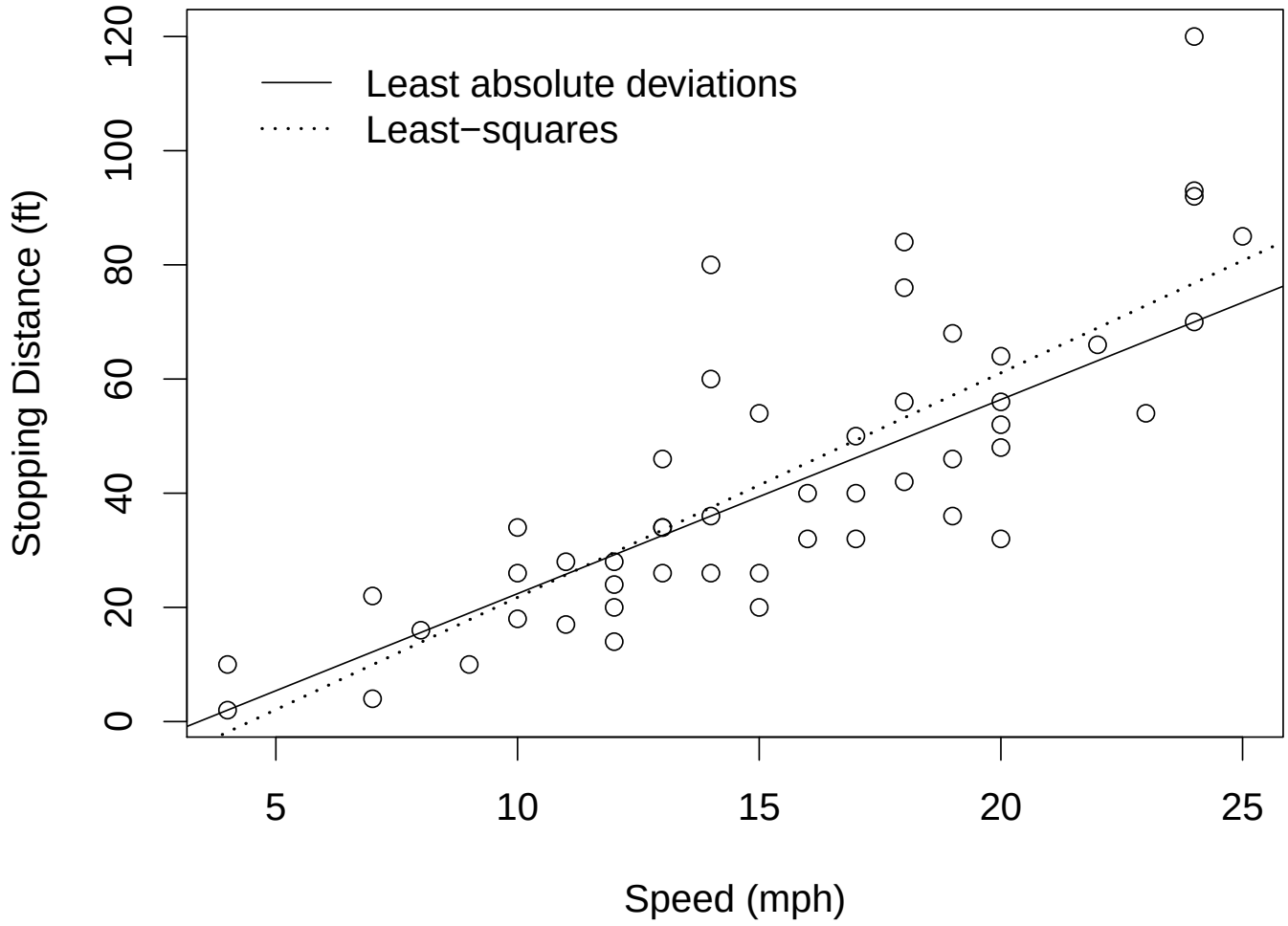
These differ from the least-squares parameter estimates.

```
> theta
(Intercept)       speed
 -17.579095    3.932409
```

We can see the difference between the two fits with a plot.

Example: Least Absolute Deviations

```
> plot(dist ~ speed, data = cars,
       xlab = "Speed (mph)",
       ylab = "Stopping Distance (ft)")
> abline(z$par)
> abline(theta, lty = "dotted", lwd = 2)
> legend(4, 120,
         legend = c("Least absolute deviations",
           "Least-squares"),
         lty = c("solid", "dotted"),
         lwd = c(1, 2),
         bty = "n")
```


Least absolute deviations
Least−squares
Stopping Distance (ft)
0
20
40
60
80
100
120
5
10
15
20
25
Speed (mph)

Example: Women's Fertility

In studying the fertility of groups of women, it is common to collect data on the length of time it takes a woman to become pregnant after she becomes susceptible to conception (usually after marriage).

As a concrete example, consider the data in the following table, which provides the times to conception for a group of 342 women from the Hutterite sect in the United States.

Month	Count	Month	Count	Month	Count
0 – 1	103	6 – 7	12	12 – 15	9
1 – 2	53	7 – 8	9	15 – 18	7
2 – 3	43	8 – 9	6	18 – 24	7
3 – 4	27	9 – 10	8	24+	4
4 – 5	30	10 – 11	10		
5 – 6	9	11 – 12	5		

Example: A Model

A simple probability model for how conception takes place leads to the following simple formula for the distribution function of conception time:

$$F(t) = P(T \leq t) = 1 - \frac{h^a}{(h+t)^a}.$$

Here h and a are (unknown) parameters of the distribution function.

- h is a scale parameter which affects the spread of the distribution; and
- a determines the shape of the distribution.

Example: Probabilities

It is very easy to define this function in R.

```
> F = function(t, a, h) 1 - h^a / (h + t)^a
```

For given values of h and a, the distribution function F allows us to compute the probability that T lies in any interval, $(u, v]$.

$$P[u < T \leq v] = F(v) - F(u).$$

Assuming that $a = 5$ and $h = 10$, we can compute the probability that the time to conception lies in each of the bins in the table as follows

```
> breaks = c(0:12, 15, 18, 24, Inf)
> probs = diff(F(breaks, 5, 10))
```

Example: Expected Cell Counts

Given these probabilities we can compute the expected number of observations in each bin for a sample of 342 women.

We'll round the values to make them easier to read.

```
> round(342 * probs, 1)
 [1] 129.6  74.9  45.3  28.5  18.6  12.4
 [7]   8.5   6.0   4.3   3.1   2.3   1.7
[13]   3.1   1.5   1.2   0.8
```

Example: Chi-Square Statistic

Now that we have expected cell counts, we can carry out a chi-square goodness-of-fit test to see how well this model fits the observed conception times.

We can create a function to compute the formula

$$X^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

as follows

```
> chi2 =
      function(Obs, Exp)
      sum((Obs - Exp)^2/Exp)
```

Example: Chi-Square Statistic Value

We can now apply this function to the Hutterite data and our computed expected values.

```
> Hutterites = c(103, 53, 43, 27, 30, 9, 12,
                 9, 6, 8, 10, 5, 9, 7, 7, 4)
> breaks = c(0:12, 15, 18, 24, Inf)
> chi2(Hutterites, 342 * diff(F(breaks, 5, 10)))
[1] 134.75
```

Example: p-Value

The p-value associated with this statistic can then be computed as follows.

```
> X2 = chi2(Hutterites,
            342 * diff(F(breaks, 5, 10)))
> df = length(Hutterites) - 1
> 1 - pchisq(X2, df)
[1] 0
```

Clearly the statistic is highly significant and we must conclude that the simple model with $a = 5$ and $h = 10$ does not provide a good description of the Hutterite data.

Example: An Optimisation Problem

Although the particular values of a and h we used didn't provide a good model for the data, it may be possible to find values of a and h which do better.

One way to look for such values is to search for the values of a and h which make the chi-square statistic as small as possible.

This is now a simple nonlinear optimisation problem.

To solve the problem with R, we need to define a function to be minimised. Using fragments of the code above, this is a simple task.

```
> Q = function(theta)
      chi2(Hutterites,
           342 * diff(F(breaks,
                        theta[1], theta[2])))
```

Example: Using `optim`

We can now set about optimising `Q` using the R function `optim`.

In this case the function being minimised is a smooth one and we can use a more efficient algorithm. We do this by telling `optim` to use the BFGS method.

We'll start the optimisation process with a starting value of $\theta = (5, 10)$.

```
> z = optim(c(5, 10), Q, method = "BFGS")
```

Example: The Optimised Solution

Examining `z$convergence` reveals that the process converged in a satisfactory manner.

The best fitting parameters and minimum value for the chi-square statistics can be obtained as follows.

```
> z$par
[1] 3.126338 9.582163
> z$value
[1] 16.76438
```

Example: Degrees of Freedom

Because we used 16 cells to compute the chi-square statistic, it might be seen that it is appropriate to use 15 degrees of freedom.

In fact this is not the case because we have used our data to estimate the best fitting parameter values.

In cases like this, it is appropriate to use the number of cells minus 1 minus the number of estimated parameters.

This case this means using $16 - 3 = 13$ degrees of freedom.

Example: p-Value

Using 13 degrees of freedom, the p-value for the chi-square statistic is

```
> 1 - pchisq(z$value, 13)
[1] 0.2103005
```

It appears the the model with the estimated values of a and h does provide a reasonable description of the data values.

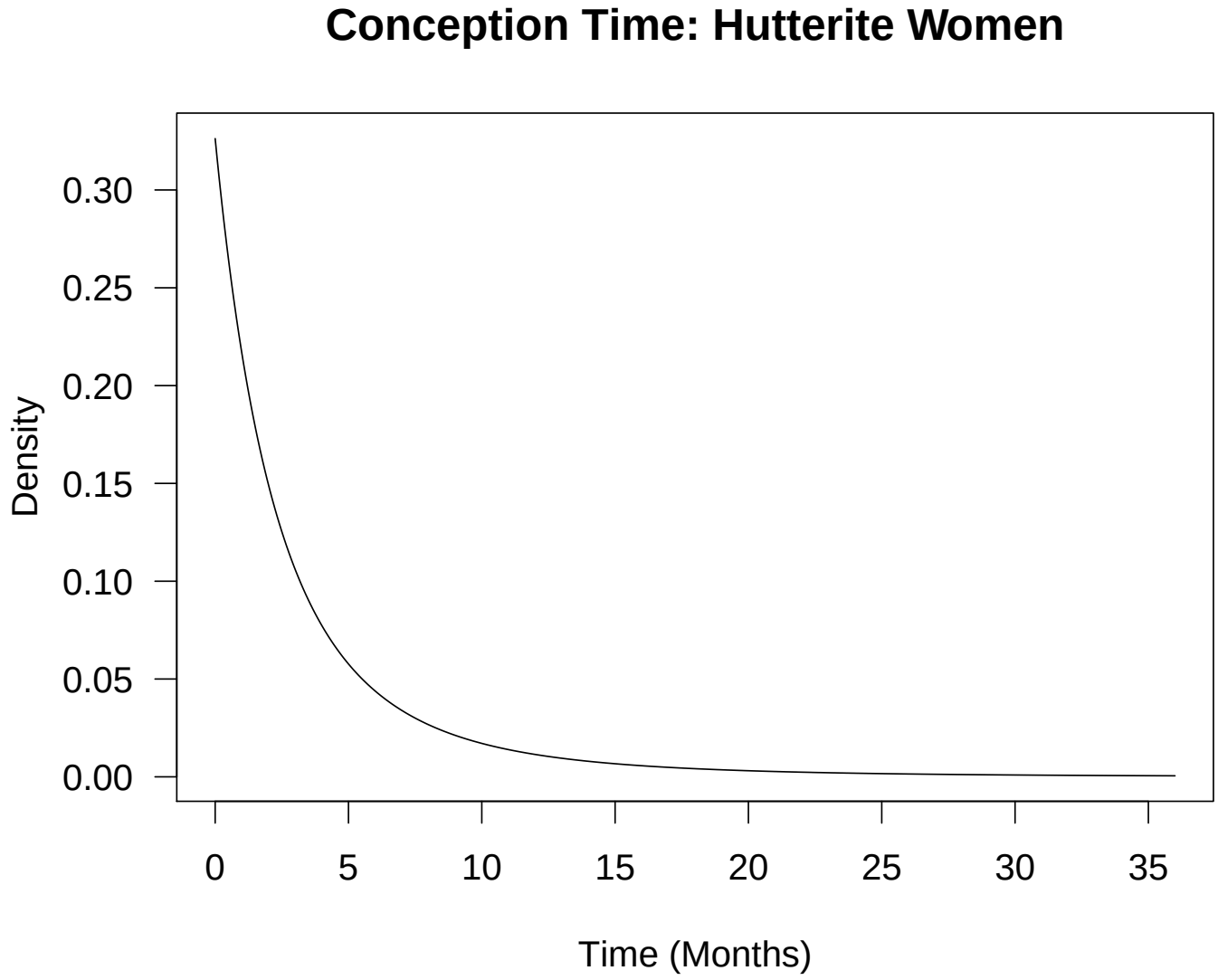
Example: Using the Fitted Distribution

Now that we have a fitted distribution, we can compute the times for which there is .5, .75, .9 chance of conception by. This means using `uniroot`.

```
> f = function(t)
      F(t, z$par[1], z$par[2]) - .5
> uniroot(f, c(0, 24), tol = 1e-12)$root
[1] 2.378409

> f = function(t)
      F(t, z$par[1], z$par[2]) - .9
> uniroot(f, c(0, 24), tol = 1e-12)$root
[1] 10.4315
```

Example: Graphing the Fitted Distribution

```
> f = function(t, a, h)
      a * h^a / (h + t)^(a + 1)

> x = seq(0, 36, length = 1001)

> plot(x, f(x, z$par[1], z$par[2]),
       type = "l", las = 1,
       main = "Conception Time: Hutterite Women",
       xlab = "Time (Months)", ylab = "Density")
```


Conception Time: Hutterite Women
Density
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25
30
35
Time (Months)

Maximum Likelihood

One of the most common uses of optimisation in statistics is parameter estimation by *maximum likelihood*.

If $x_1, \ldots, x_n$ is a sample of independent observations from a distribution with density $f(x;\theta)$ then the likelihood for the sample is

$$L(\theta) = \prod_{i=1}^{n} f(x_i;\theta).$$

The principle of *maximum likelihood* says that good estimates of the unknown parameter(s) θ can be obtained by determining the value $\widehat{\theta}$ which maximises the likelihood function.

The estimates obtained in this way have all kinds of wonderful properties (e.g. consistency, efficiency, asymptotic normality) and, as a result, maximum likelihood has become very widely used.

Log Likelihood

Density functions tend to vary widely in magnitude and when multiplied together, as in the likelihood function, this can result in numerical computation problems.

Because of this (and also for theoretical reasons) it is common practice to work with the log of the likelihood function

$$l(\theta) = \log L(\theta).$$

It is also common practice to negate the sign of the log-likelihood to turn the problem into a minimisation problem rather than a maximisation problem.

$$Q(\theta) = -l(\theta)$$

An Example: The Normal Distribution

We'll look at the problem of obtaining the maximum likelihood for the normal distribution. We'll use 10 observations from a normal distribution with mean 20 and standard deviation 5.

```
> x = rnorm(10, 20, 5)
```

The negative log-likelihood is easy to implement.

```
> Q =
        function(theta)
        -sum(log(dnorm(x, theta[1], theta[2])))
```

This function can be minimised using `optim`.

Optimisation

Let's pretend that we don't know what the parameters of the distribution are and start with an arbitrary starting point.

```
> z = optim(c(1, 1), Q)
> z$convergence
[1] 0
> z$par
[1] 20.346887  4.518694
```

The usual estimates of the mean and standard deviation are

```
> mean(x)
[1] 20.37313
> sd(x)
[1] 4.76892
```

The maximum-likelihood estimate of the mean looks pretty close to the sample mean, but the estimate of the standard deviation looks a little off.

ML Estimates can be "Non-Standard"

The difference between the standard deviation estimates is predicted by theory.

The maximum-likelihood estimate of the standard deviation is

$$\widehat{\sigma} = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(x_i - \overline{x})^2}.$$

Compensating for this shows that the estimates are close.

```
> z$par[2]
[1] 4.518694
> sd(x) * sqrt(9/10)
[1] 4.524195
```

Choice of Optimisation Method

The default optimisation method used by `optim` is slow but robust.

When the likelihood is a smooth function of the parameters it can be useful to use a more efficient method.

The method of choice is the Broyden, Fletcher, Goldfarb and Shanno method which is Newton-like method.

Beware that the method can be fragile!

```
> z = optim(c(1, 1), Q, method="BFGS")
> z$convergence
[1] 1
> z$par
[1]  194.7203 3948.6246
```

Using the BFGS Method

It is important to use a good starting point when using the BFGS method with `optim`.

Starting closer to the minimum we get the following results.

```
> z = optim(c(19, 6), Q, method="BFGS")
> z$convergence
[1] 0
> z$par
[1] 20.373127  4.524178
```

Note that the exact results are

```
> c(mean(x), sd(x) * sqrt(9/10))
[1] 20.373128  4.524195
```

so that BFGS has done a better job than the default method.

Graphing the Likelihood Surface

We are now going to talk about producing graphs of the (log-) likelihood surface.

To help do this it will be helpful to have a version of the log-likelihood which is a vectorised function of two arguments (mean and standard deviation).

This is relatively straightforward and shows the standard recipe for vectorising a non-vectorised function.

Vectorising the Log-Likelihood

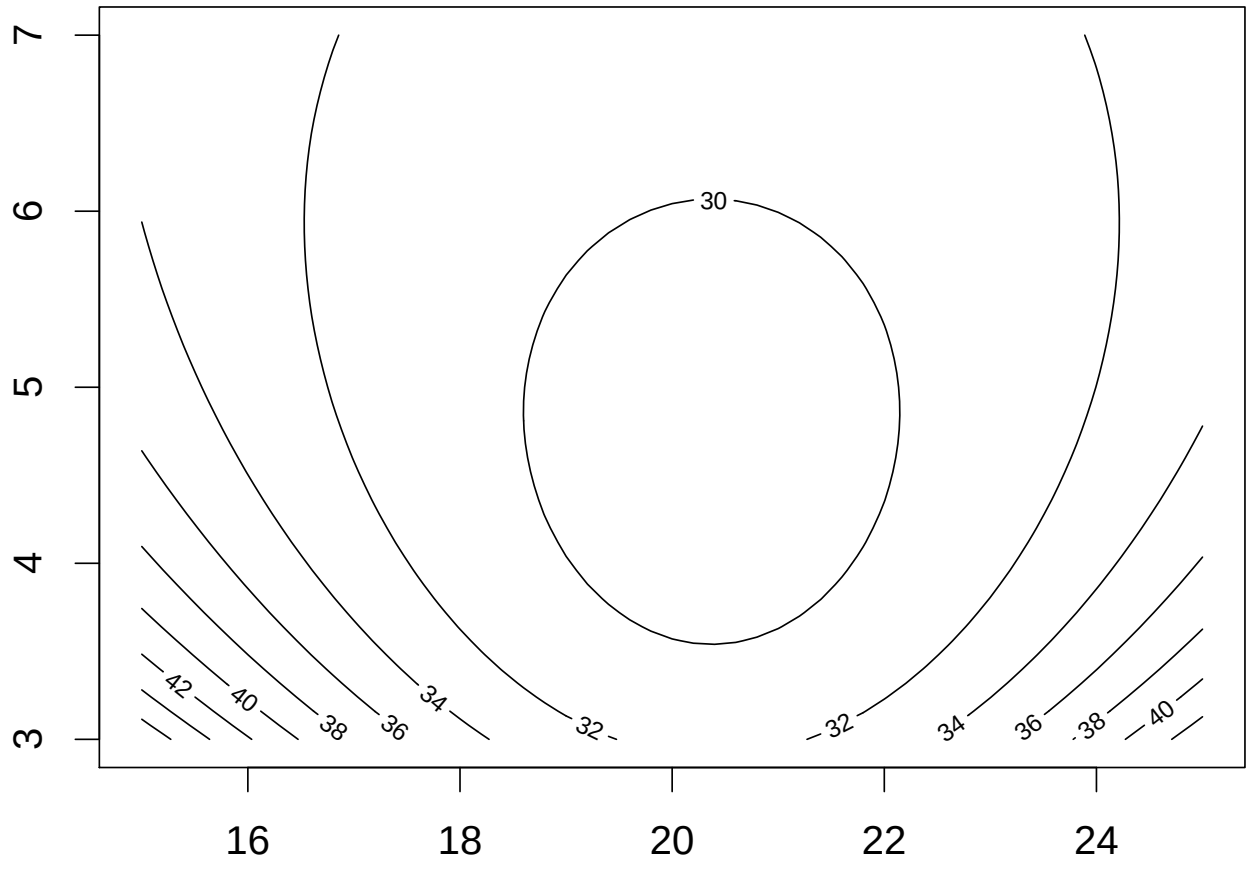
```
> Q2 =
      function(theta1, theta2) {
          ## Recycle arguments
          if (length(theta1) < length(theta2))
              theta1 = rep(theta1,
                  length = length(theta2))
          if (length(theta2) < length(theta1))
              theta2 = rep(theta2,
                  length = length(theta1))
          ## Vector calculation
          ans = numeric(length(theta1))
          for(i in 1:length(ans))
              ans[i] = -sum(log(dnorm(x, theta1[i],
                                      theta2[i])))
          ans
      }
```

Plotting the Log-Likelihood

Locating good starting points for `optim` can be helped by looking at graphs.

In two dimensions the log-likelihood can be inspected using a contour plot.

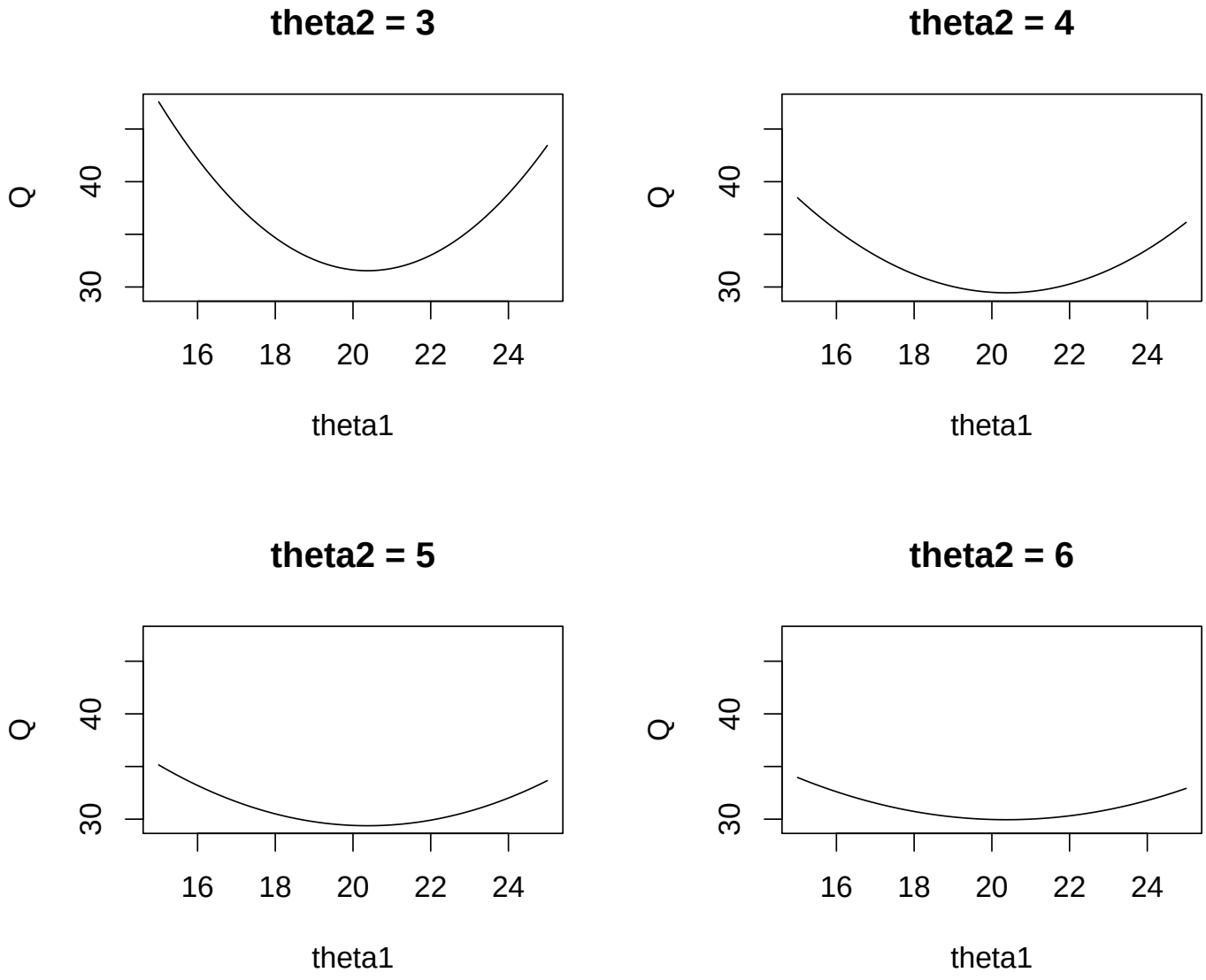
```
> theta1 = seq(15, 25, length = 51)
> theta2 = seq(3, 7, length = 51)
> z = outer(theta1, theta2, Q2)
> contour(theta1, theta2, z)
```


7
6
5
4
3
16
18
20
22
24
30
32
34
36
38
40
42

Likelihood Profiles

In higher dimensions it can be useful to obtain likelihood profiles (slices through the likelihood).

```
> theta1 = seq(15, 25, length = 1001)
> theta2 = c(3,4,5,6)
> z = outer(theta1, theta2, Q2)
> par(mfrow = c(2, 2))
> r = range(z)
> for(j in 1:length(theta2))
      plot(theta1, z[,j], ylim = r, type = "l",
           main = paste("theta2 =", theta2[j]),
           xlab = "theta1", ylab = "Q")
```


theta2 = 3
Q
30
40
16
18
20
22
24
theta1
theta2 = 4
Q
30
40
16
18
20
22
24
theta1
theta2 = 5
Q
30
40
16
18
20
22
24
theta1
theta2 = 6
Q
30
40
16
18
20
22
24
theta1

Grid Searches for Good Starting Values

The process of finding good starting points for the optimisation by carrying out a grid search.

```
> theta1 = 15:25
> theta2 = 1:10
> z = outer(theta1, theta2, Q2)
> which(z == min(z), TRUE)
     row col
[1,]   6   5
> i = which(z == min(z), TRUE)
> theta = c(theta1[i[1]], theta2[i[2]])
> theta
[1] 20  5
```

Using Loops for a Grid Search

Using `outer` only works for two-dimensional problems. In general the search must be carried out with nested loops.

```
> theta1 = 15:25
> theta2 = 1:10
> Qmin = Inf
> for(t1 in theta1)
      for(t2 in theta2) {
          Qtheta = Q(c(t1, t2))
          if (Qtheta < Qmin) {
              Qmin = Qtheta
              start = c(t1, t2)
          }
      }
> start
[1] 20  5
```

Standard Errors

Once estimates of the unknown parameters have been found it is important to have standard errors to go along with the estimates.

Standard errors can be obtained from the Hessian (or matrix of second derivatives of the negative log-likelihood).

The inverse of the Hessian provides an estimated variance-covariance matrix for the parameter estimates. The standard errors can be obtained as the square roots of its diagonal elements.

An approximate Hessian can be obtained from `optim` by supplying the argument `hessian=TRUE`.

Standard Error Computation

Here is how standard errors can be obtained.

```
> z = optim(c(20, 5), Q,
            method = "BFGS", hessian = TRUE)
> z$convergence
[1] 0
> z$par
[1] 20.373073  4.524198
> sqrt(diag(solve(z$hessian)))
[1] 1.430677 1.011642
```

We know that the standard error of the mean is $\sqrt{s^2/n}$ so we can compare with this.

```
> sd(x)/sqrt(length(x))
[1] 1.508065
```

Explaining the Different Standard Errors

The Hessian based estimate of the standard error looks different from the "usual" one. However, remember that the "usual" estimate of σ has been adjusted to be unbiased.

If we instead use the MLE for σ then the results are very close.

```
> sqrt(9/10) * sd(x)/sqrt(length(x))
[1] 1.430676

> sqrt(diag(solve(z$hessian)))
[1] 1.430677 1.011642
```