

numerica-plus

Andrew Parsloe
(ajparsloe@gmail.com)

December 11, 2021

Abstract

The `numerica-plus` package defines commands to iterate and find fixed points of functions of a single variable, to find the zeros or extrema of such functions, and to calculate the terms of recurrence relations.

Note:

- This document applies to version 2.0.0 of `numerica-plus.def`.
- A version of `numerica` from or later than 2021/12/07 is required; (`numerica` requires `amsmath`, `mathtools` and the L^AT_EX3 bundles `l3kernel` and `l3packages`).
- I refer a number of times in this document to *Handbook of Mathematical Functions*, edited by Milton Abramowitz and Irene A. Segun, Dover, 1965. This is abbreviated to *HMF*.
- Version 2 of `numerica-plus`
 - is the first stand-alone version; (in v.1 of `numerica` the commands `\nmcIterate`, `\nmcSolve` and `\nmcRecur` were loaded with the `plus` package option);
 - makes some small code adjustments;
 - amends documentation.

Contents

1	Introduction	3
1.1	Example of use: the rotating disk	3
1.1.1	Circuits	6
1.2	Shared syntax of the new commands	8
1.2.1	Settings	9
1.2.2	Nesting	10
2	Iterating functions: <code>\nmcIterate</code>	11
2.1	Star (*) option: fixed points	13
2.1.1	Use with <code>\nmcInfo</code>	14
2.2	Settings option	15
2.2.1	Inherited settings	15
2.2.2	<code>\nmcIterate</code> -specific settings	15
2.2.3	Changing default values	18
2.3	Errors	19
3	Finding zeros and extrema: <code>\nmcSolve</code>	20
3.1	Extrema	21
3.1.1	The search strategy	21
3.2	Star (*) option	23
3.3	Settings option	24
3.3.1	Inherited settings	24
3.3.2	<code>\nmcSolve</code> -specific settings	25
3.3.3	Changing default values	28
4	Recurrence relations: <code>\nmcRecur</code>	29
4.1	Notational niceties	30
4.1.1	Vv-list and recurrence variable	30
4.1.2	Form of the recurrence relation	31
4.1.3	First order recurrences (iteration)	32
4.2	Star (*) option	32
4.3	Settings	33
4.3.1	Inherited settings	33
4.3.2	<code>\nmcRecur</code> -specific settings	34

4.3.3	Changing default values	36
4.3.4	Orthogonal polynomials	36
4.3.5	Nesting	37
5	Reference summary	39
5.1	Commands defined in <code>numerica-plus</code>	39
5.2	Settings for the three commands	39
5.2.1	Settings for <code>\nmcIterate</code>	39
5.2.2	Settings for <code>\nmcSolve</code>	40
5.2.3	Settings for <code>\nmcRecur</code>	40

Chapter 1

Introduction

Entering

```
\usepackage{numerica-plus}
```

in the preamble of your document makes available the commands

- `\nmcIterate`, a command to iterate a function (apply it repeatedly to itself), including finding fixed points (values x where $f(x) = x$);
- `\nmcSolve`, a command to find the zeros of functions of a single variable (values x for which $f(x) = 0$) or, failing that, local maxima or minima of such functions;
- `\nmcRecur`, a command to calculate the values of terms in recurrence relations in a single (recurrence) variable (like the terms of the Fibonacci sequence or Legendre polynomials).

`numerica-plus` requires a version of `numerica` from or later than 2021/11/26. If found, `numerica` is loaded automatically, making available the `\nmcEvaluate`, `\nmcInfo`, `\nmcMacros`, `\nmcConstants`, and `\nmcReuse` commands; see the `numerica` documentation for details on the use of these commands.

The commands of the present package all share the syntax of `\nmcEvaluate`. I will discuss them individually in later chapters but turn first to something more than a ‘toy’ example that illustrates their use and gives a sense of ‘what they are about’.

1.1 Example of use: the rotating disk

Consider a disk rotating uniformly with angular velocity ω in an anticlockwise sense in an inertial system in which the disk’s centre $\mathbf{0}$ is at rest. Three distinct points $\mathbf{1}$, $\mathbf{2}$, $\mathbf{3}$ are fixed in the disk and, in a co-rotating polar coordinate system centred at $\mathbf{0}$, have polar coordinates (r_i, θ_i) ($i, j = 1, 2, 3$). Choose $\mathbf{01}$ as initial line so that $\theta_1 = 0$.

The cosine rule for solving triangles tells us that the time t_{ij} in the underlying inertial system for a signal to pass from \mathbf{i} to \mathbf{j} satisfies the equation

$$t_{ij} = c^{-1} \sqrt{r_i^2 + r_j^2 - 2r_i r_j \cos(\theta_j - \theta_i + \omega t_{ij})} \equiv f(t_{ij}),$$

where c is the speed of light. (Equally, we could be describing an acoustic signal between points on a disk rotating uniformly in a still, uniform atmosphere – in which case c would be the speed of sound.) Although the equation doesn't solve algebraically for the time t_{ij} , it does tell us that $t = t_{ij}$ is a *fixed point* of the function $f(t)$. To calculate fixed points we use the command `\nmcIterate`, or its short-name form `\iter`, with the star option, `\iter*`. For `\iter` the star option means: continue iterating until a fixed point has been reached and, as with the `\eval` command, suppress all elements from the display save for the numerical result.

First, though, values need to be assigned to the various parameters. Suppose we use units in which $c = 30$, and $\omega = 0.2$ radians per second. To avoid having to write these values in the vv-list every time, I have put in the preamble to this document the statement

```
\constants{ c=30,\omega=0.2 }
```

For the polar coordinates of $\mathbf{1}$ and $\mathbf{3}$ I have chosen $r_1 = 10$, $r_3 = 20$ and $\theta_3 = 0.2$ radians (remember $\theta_1 = 0$). To find a fixed point t_{13} I give t an initial trial value 1 (plucked from the air). Its position as the rightmost item in the vv-list tells `\iter` that t is the iteration variable:

```
\iter*{ c^{-1}\sqrt{r_1^2+r_3^2-2r_1 r_3
\cos(\theta_3+\omega t)}
}[ r_1=10,r_3=20,\theta_3=0.2,t=1 ],
\quad\info{iter}.
```

\implies 0.356899, 5 iterations. The short-name form of the `\nmcInfo` command from `numerica` has been used to display the number of iterations required to attain the fixed-point value.

To six figures, only five iterations are needed, which seems rapid but we can check this by substituting $t = 0.356899$ back into the formula and `\eval`-uating it:

```
\eval*{ c^{-1}\sqrt{r_1^2+r_3^2-2r_1 r_3
\cos(\theta_3+\omega t)}
}[ r_1=10,r_3=20,\theta_3=0.2,t=0.356899 ]
```

\implies 0.356899, confirming that we have indeed calculated a fixed point. That it did indeed take only 5 iterations can be checked by omitting the asterisk from the `\iter` command and specifying the total number of iterations to perform. I choose `do=7` to show not just the 5th iteration but also the next two just to confirm that the result is stable. We shall view all 7: `see=7`. Because of the

length of the formula I have suppressed display of the vv-list by giving the key `vvd` an empty value:¹

```
\iter[do=7,see=7,vvd=]
  {\[ c^{-1}\sqrt{r_1^2+r_3^2-2r_1 r_3
    \cos(\theta_3+\omega t)} \]}
  [ r_1=10,r_3=20,\theta_3=0.2,t=1 ]
```

⇒

$$c^{-1}\sqrt{r_1^2 + r_3^2 - 2r_1r_3 \cos(\theta_3 + \omega t)} = 0.382355$$

↪ 0.357756
↪ 0.356928
↪ 0.3569
↪ 0.356899
↪ 0.356899
↪ 0.356899

The display makes clear that on the 5th iteration, the 6-figure value has been attained.

Alternatively, we could use the `\nmcRecur` command, or its short-name form `\recur`, to view the successive iterations, since an iteration is a first-order recurrence: $f_{n+1} = f(f_n)$:

```
\recur[do=8,see1=0,see2=5,vvd={,\(\vv)\},*]
  {\[ f_{n+1}=c^{-1}\sqrt{r_1^2+r_3^2-2r_1 r_3
    \cos(\theta_3+\omega f_n)} \]}
  [ r_1=10,r_3=20,\theta_3=0.2,f_{0}=1 ]
```

⇒

$$f_{n+1} = c^{-1}\sqrt{r_1^2 + r_3^2 - 2r_1r_3 \cos(\theta_3 + \omega f_n)},$$

$(r_1 = 10, r_3 = 20, \theta_3 = 0.2, f_0 = 1)$
→ 0.356928, 0.3569, 0.356899, 0.356899, 0.356899

I have specified `do=8` terms rather than 7 since the zero-th term ($f_0 = 1$) is included in the count. I've chosen to view the last 5 of them but none prior to those by writing `see1=0,see2=5`. Notice the `vvd` setting, pushing display of the vv-list and result to new lines and suppressing equation numbering with the `*` setting (which turns the `multline` environment into a `multline*` environment).

Another and perhaps more obvious way to find the value of t_{13} , is to look for a zero of the function $f(t) - t$. That means using the command `\nmcSolve` or its short-name form `\solve`. I shall do so with the star option `\solve*` which suppresses display of all but the numerical result. A trial value for t is required. I have chosen `t=0`:

¹See the associated document `numerica.pdf`, the chapter on settings.

```

\solve*{ c^{-1}\sqrt{r_1^2+r_3^2-2r_1 r_3}
\cos(\theta_3+\omega t)} - t }
[ r_1=10,r_3=20,\theta_3=0.2,t=0 ],
\quad\nmcInfo{solve}.

```

$\implies 0.356898, \quad 1+20$ steps.

Nearly the same answer as before is attained but this time many more steps have been required. This is to be expected. The `\solve` command uses the bisection method. Since $1/2^{10} \approx 1/10^3$, about 10 bisections are needed to determine 3 decimal places. Hence we can expect about 20 bisections for a 6-decimal-place answer. The particular form of the `\nmcInfo` command display, ‘1 + 20 steps’, indicates that it took 1 search step to find an interval in which the function vanished and, having found that interval, 20 bisections to narrow the position of the zero to 6-figures. I will discuss the discrepancy in the final figure in Chapter 3; see §3.3.2.3.

1.1.1 Circuits

Okay, so we can calculate the time taken in the underlying inertial system for a signal to pass from one point of the rotating disk to another. How long does it take to traverse the circuit **1** to **2** to **3** and back to **1**? That means forming the sum $t_{12} + t_{23} + t_{31}$, hence calculating the separate t_{ij} and then using `\eval` to calculate their sum.

To simplify things, I assume a little symmetry. Let the (polar) coordinates of **1** be $(a, 0)$, of **2** be $(r, -\theta)$, and of **3** be (r, θ) : **2** and **3** are at the same radial distance from the centre **0** and at the same angular distance from the line **01** but on opposite sides of it, **3** ahead of the line, **2** behind it. The rotation is in the direction of positive θ . Rather than just calculate $t_{12} + t_{23} + t_{31}$ for the circuit **1231**, I also calculate the time $t_{13} + t_{32} + t_{21}$ for a signal to traverse the same circuit but in the opposite sense, **1321**, and compare them (form the difference).

Note that with **2** and **3** positioned as they are relative to **1**, a signal against the rotation from **3** to **1** takes the same time as a signal from **1** to **2** and, in the sense of rotation, a signal from **2** to **1** takes the same time as a signal from **1** to **3**. To see this, suppose the signal from **2** to **1** starts at time $t = 0$; it reaches **1** at a later time $t = t'$ when the disk has rotated an angle $\omega t'$. Viewed from the underlying inertial system, the signal path is a straight line from a point on a circle of radius r to a point on a concentric circle of radius a , the points subtending an angle at the centre **0** of $\theta + \omega t'$. But **3** at time t' and **1** at time $t = 0$ also subtend an angle $\theta + \omega t'$ at **0**, and also lie on circles of radii r and a respectively. In the underlying inertial system the line segments **1(0)3(t')** and **2(0)1(t')** are of equal length. Similarly, if a signal from **3** at time $t = 0$ reaches **1** at time $t = t''$ then **3(0)1(t'')** and **1(0)2(t'')** are of equal length. Hence the round trip times are $2t_{12} + t_{23}$ and $2t_{13} + t_{32}$.

1.1.1.1 Nesting commands

Analytically, both t_{21} and t_{13} are the same fixed point of the function

$$c^{-1}\sqrt{r^2 + a^2 - 2ra \cos(\theta + \omega t)}$$

and t_{31} and t_{12} are the same fixed point of the function

$$c^{-1}\sqrt{r^2 + a^2 - 2ra \cos(\theta - \omega t)}.$$

To calculate $2t_{12} + t_{23}$ therefore means calculating

$$\begin{aligned} & 2\text{\texttt{\textbackslash iter*}}\{ c^{-1}\sqrt{a^2+r^2-2ar} \\ & \quad \text{\texttt{\textbackslash cos}}(\text{\texttt{\textbackslash theta}}-\text{\texttt{\textbackslash omega t}})\} \\ & + \text{\texttt{\textbackslash iter*}}\{ c^{-1}\sqrt{2r^2-2r^2} \\ & \quad \text{\texttt{\textbackslash cos}}(2\text{\texttt{\textbackslash theta}}+\text{\texttt{\textbackslash omega t}})\} \end{aligned}$$

with the analogous expression for $2t_{13} + t_{32}$. But we can do the comparison of round trip times ‘in one go’ by nesting the `\iter*` commands inside an `\eval*` command:

```
\eval*{ % circuit 1231
  2\iter*[var=t]{ c^{-1}\sqrt{a^2+r^2-2ar
    \cos(\theta-\omega t)} }[8]
  + \iter*[var=t]{ c^{-1}\sqrt{2r^2-2r^2
    \cos(2\theta+\omega t)} }[8]
% circuit 1321
- 2\iter*[var=t]{ c^{-1}\sqrt{a^2+r^2-2ar
  \cos(\theta+\omega t)} }[8]
- \iter*[var=t]{ c^{-1}\sqrt{2r^2-2r^2
  \cos(2\theta-\omega t)} }[8]
}[ a=10,r=20,\theta=0.2,t=1 ]
```

$\implies 0.034746$.

By itself this result is of little interest beyond seeing that `numerica-plus` can handle the calculation. What *is* interesting is to find values of our parameters for which the time difference vanishes – say values of θ , given the other parameters, especially the value of r . Is there a circuit such that it takes a signal the same time to travel in opposite senses around the circuit, despite the rotation of the disk? Rather than nesting the `\iter*` commands inside an `\eval`, we need to nest them in a `\solve` command:

```
\solve[p=.,var=\theta,+=1,vvd=\\,*,+=1]
{\ [ % circuit 1231
  2\times\iter*[var=t,+=1]{ c^{-1}\sqrt{a^2+r^2-2ar
    \cos(\theta-\omega t)} }
  + \iter*[var=t,+=1]{ c^{-1}\sqrt{2r^2-2r^2
    \cos(2\theta+\omega t)} }
```

```

% circuit 1321
- 2\times\iter*[var=t,+=1]{ c^{-1}\sqrt{a^2+r^2-2ar}
    \cos(\theta+\omega t)} }
- \iter*[var=t,+=1]{ c^{-1}\sqrt{2r^2-2r^2}
    \cos(2\theta-\omega t)} }
\]][ a=10,r=20,\theta=0.1,t=1 ]

```

⇒

$$2 \times 0.537778 + 1.221266 - 2 \times 0.61442 - 1.067983 = 0.000001$$

$$\rightarrow \theta = 1.035741$$

One point to note here is the use of `\times` (in `2\times\iter*`). In this example the formula is displayed (`\solve` wraps around math delimiters). Without the `\times` the result would have been the same but the display of the formula would have juxtaposed the ‘2’s against the following decimals, making it look as if signal travel times were 20.537778 and 20.61442 (and no doubt causing perplexity). Also note the `vvd=\\` to place the result on a new line and suppress display of the `vv`-list.

So this expression gives a value of $\theta_{\Delta t=0}$ for one value of r . The obvious next step is to create a table of such values. I show how that is done in the document `numerica-tables.pdf` using the command `\nmcTabulate` defined in the associated package `numerica-tables`. But this is not a research paper on the rotating disk. I wished to show how the different commands of `numerica-plus` can be used to explore a meaningful problem. And although it looks as if a lot of typing is involved, once $c^{-1}\sqrt{r^2 + a^2 - 2ra \cos(\theta - \omega t)}$ has been formed in \LaTeX and values specified in the `vv`-list, much of the rest is copy-and-paste with minor editing.

1.2 Shared syntax of the new commands

`numerica-plus` offers three new commands for three processes: `\nmcIterate` (short-name form `\iter`) for iterating functions, `\nmcSolve` (short-name form `\solve`) for finding the zeros or (local) extrema of functions, and `\nmcRecur` (short-name form `\recur`) for calculating terms of recurrence relations.

All three commands share the syntax of the `\nmcEvaluate` (or `\eval`) command detailed in the associated document `numerica.pdf`. When all options are used the command looks like, for instance,

```
\nmcIterate*[settings]{expr.}[vv-list][num. format]
```

You can substitute `\nmcSolve`, or `\nmcRecur` for `\nmcIterate` here. The arguments are similar to those for `\nmcEvaluate`.

1. `*` optional switch; if present ensures a single number output with no formatting, or an appropriate error message if the single number cannot be produced;

2. `[settings]` optional comma-separated list of *key=value* settings for this particular command and calculation;
3. `{expr.}` the only mandatory argument; the mathematical expression in L^AT_EX form that is the object of interest;
4. `[vv-list]` optional comma-separated list of *variable=value* items; for `\iter` and `\solve` the *rightmost* (or innermost) variable in the *vv-list* may have special significance;
5. `[num. format]` optional format specification for presentation of the numerical result (rounding, padding with zeros, scientific notation); boolean output is suppressed for these commands.

Like `\nmcEvaluate`, for all three commands the way the result is displayed depends on whether the command wraps around math delimiters, or is used between math delimiters or in the absence of math delimiters. These distinctions are relevant *only if the optional star * is absent*.

- When the star option is used, the *result* is a number only, without any formatting or *vv-list* display, or an error message is displayed.
- When the star option is not used and one of the following is the case
 - the command wraps around math delimiters, e.g. `\iter{$ expr. $}`, then
 - * the result is displayed in the form *formula = result*, (*vv-list*) or the form *formula* \rightarrow *result*, (*vv-list*) as appropriate;
 - the command is used within math delimiters, e.g. `\[\iter...\]`, then
 - * the result is displayed in the form *result*, (*vv-list*) (without reference to the formula);
 - the command is used in the absence of delimiters, then
 - * the result is presented as if it had been used between `\[` and `\]`.

Looking at the various examples in the preceding section on the rotating disk you will see illustrations of all these situations.

1.2.1 Settings

Nearly all the settings available to the `\eval` command are available to these other commands. To save switching between documents I reproduce in Table 1.1 the options found in `numerica.pdf`, although for discussion of the options you will need to refer to that document. In addition, each of the present commands also has settings of its own, discussed at the relevant parts of the following chapters.

Table 1.1: Inherited settings options

key	type	meaning	default
dbg	int	debug ‘magic’ integer	0
^	char	exponent mark for sci. notation input	e
xx	int (0/1)	multi-token variable switch	1
()	int (0/1/2)	trig. function arg. parsing	0
o		degree switch for trig. funcions	
log	num	base of logarithms for <code>\log</code>	10
vvmode	int (0/1)	vv-list calculation mode	0
vvd	tokens	vv-list display-style spec.	{,}\mskip 12mu plus 6mu minus 9mu(vv)
vvi	token(s)	vv-list text-style spec.	{,}\mskip 36mu minus 24mu(vv)
*		suppress equation numbering if <code>\</code> in <code>vvd</code>	
p	char(s)	punctuation (esp. in display-style)	, (comma)
S+	int	extra rounding for stopping criterion for sums	2
S?	int ≥ 0	stopping criterion query terms for sums	0
P+	int	extra rounding for stopping criterion for products	2
P?	int ≥ 0	stopping criterion query terms for products	0

1.2.2 Nesting

In v.1 of `numerica`, for commands to be nested one within an another, it was necessary for the inner command to be starred (and thus produce a purely numerical result). With v.2 of `numerica` this is no longer the case. A nested command is detected as such and the star automatically set, whether the user has explicitly starred the command or not. Provided the starred form of a command actually does produce a numerical result and not an error message then it can be nested within the main argument of any one of the other commands, including itself. The example of use, §1.1 above, shows several examples of this. The starred form can also be used in the vv-list of any one of the commands, including itself. The associated document `numerica.pdf` shows examples of an `\eval*` command being used in the vv-list of an `\eval` command.

Chapter 2

Iterating functions: `\nmcIterate`

Only in desperation would one try to evaluate a continued fraction by stacking fraction upon fraction upon fraction like so:

```
\eval{\[ 1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}}}}}}}}}}}}} \]}
```

⇒

$$1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}}}}}}}}}}}} = 1.618026$$

`numera-plus` provides a command for tackling problems like this sensibly. In such problems a function is repeatedly applied to itself (iterated). This is done through the command `\nmcIterate` or (short-name form) `\iter`. Thus to evaluate this continued fraction write (for instance),

```
\iter[do=15,see=5]{\[ 1+1/x \]}[x=1] ⇒
```

```
1 + 1/x = 2,    (x = 1)
... final 5 of 15:
↔ 1.618056
↔ 1.618026
↔ 1.618037
↔ 1.618033
↔ 1.618034
```

The `\iter` command evaluates $1 + 1/x$ when $x = 1$ and then uses this value as a new x -value to substitute into $1 + 1/x$, to once again evaluate and use as a new x -value, and so on. It looks as if the repeated iterations are approaching $\text{\eval{\$ \tfrac{\sqrt{5}+1}{2} \$}} \implies \frac{\sqrt{5}+1}{2} = 1.618034$. Increasing the number of iterations in the example from `do=15` to, say, `do=18`, shows that this is indeed the case.

```
\iter[do=18,see=5]{\[ 1+1/x \]}[x=1] ==>
      1 + 1/x = 2,      (x = 1)
      ... final 5 of 18:
      ↪ 1.618033
      ↪ 1.618034
      ↪ 1.618034
      ↪ 1.618034
      ↪ 1.618034
```

But iteration of functions is not limited to continued fractions. Particularly since the emergence of chaos theory, iteration has become an important study in its own right. Any function with range within its domain can be iterated – repeatedly applied to itself – like the cosine:

```
\iter[do=20,see=4]{\[ \cos x \]}[x=\pi/2] ==>
      cos x = 0,      (x = \pi/2)
      ... final 4 of 20:
      ↪ 0.738369
      ↪ 0.739567
      ↪ 0.73876
      ↪ 0.739304
```

which displays the first one and last four of 20 iterations of $\cos x$ when $x = \frac{\pi}{2}$. It looks as if the cosine is ‘cautiously’ approaching a limit, perhaps around 0.738 or 0.739. You need to nearly double the number of iterations (`do=40`) to confirm that this is so.

The logistic function $kx(1 - x)$ exhibits a variety of behaviours depending on the value of k . For instance, with $k = 3.5$ we get a period-4 cycle:

```
\iter[do=12,see=8]{\[ kx(1-x) \]}[k=3.5,x=0.5] ==>
      kx(1 - x) = 0.875,      (k = 3.5, x = 0.5)
      ... final 8 of 12:
      ↪ 0.874997
      ↪ 0.38282
      ↪ 0.826941
      ↪ 0.500884
      ↪ 0.874997
      ↪ 0.38282
      ↪ 0.826941
      ↪ 0.500884
```

and with $k = 3.1$ we get a period-2 cycle, although it takes many more iterations to stabilize there:

```
\iter[do=42,see=4]{\[ kx(1-x) \]}[k=3.1,x=0.5] ==>
      kx(1 - x) = 0.775,    (k = 3.1, x = 0.5)
      ... final 4 of 42:
      ↪ 0.764567
      ↪ 0.558014
      ↪ 0.764567
      ↪ 0.558014
```

2.1 Star (*) option: fixed points

In the first two of these examples, iteration eventually ended at a *fixed point*. This is a point x where $f(x) = x$. Appending a star (asterisk) to the `\iter` command is the signal for iteration to continue until a fixed point has been reached at the specified rounding value:

```
\iter*{ 1+a/x }[a=n(n+1),n=1,x=1] ==> 2
```

(with the default rounding value 6).¹ The star overrides any value for the number of iterations to perform (the `do` key) that may have been entered in the settings option. It also overrides any elements of the display other than the numerical result. With the star option math delimiters are irrelevant – other than displaying minus signs correctly when `\iter*` is between them.

A function may not approach a fixed point when iterated – see the examples with the logistic function above. To prevent an infinite loop `\iter*` counts the number of iterations performed and when that number reaches a certain cut-off value – the default is 100 – the loop terminates and a message is displayed:

```
\iter*{kx(1-x)}[k=3.5,x=0.5] ==>
!!! No fixed point attained after 100 iterations of: formula. !!!
```

In this case we *know* that a fixed point does not exist, but that may not always be the case. One response to a message like this is to change parameter values or starting value of the iteration variable. For instance, changing the parameter value to $k = 1.5$,

```
\iter*{kx(1-x)}[k=1.5,x=0.5] ==> 0.333334,
```

means a fixed point is now attained. It is easy to check that $1/3$ is indeed a fixed point (but that makes the 4 in the last decimal place a concern; see the extra rounding setting, §2.2.2.2).

But should a fixed point still not eventuate after ‘fiddling’ with parameter and start values, there are two general adjustments one might try: either

¹For your own interest try also putting $n = 2, 3, 4, \dots$ in the `vv`-list of this expression.

1. reduce the rounding value, from the default 6 (or the one specified), to a smaller value, or
2. increase the cut-off figure from the default 100 to some higher value.

The former is done via the trailing number format optional argument of the `\iter` command; the latter is done via the settings option, see §2.2, specifically §2.2.2.3.

2.1.1 Use with `\nmcInfo`

It is of interest to know how many iterations are required to reach a fixed point at a particular rounding value. That knowledge allows a good guess as to whether a fixed point will be attained at a greater rounding value. Thus when iterating the function

$$f(t_{ij}) = c^{-1} \sqrt{r_i^2 + r_j^2 - 2r_i r_j \cos(\theta_j - \theta_i + \omega t_{ij})}$$

in §1.1 only 5 iterations were required to attain 6-figure accuracy for the fixed point. That information came by following the `\iter*` command with `\nmcInfo` (or `\info`) with the argument `iter`. And generally, for any ‘infinite’ process, follow the command with an `\info` command if you want to know how many ‘steps’ – in the present case iterations – are required to achieve the result. So, if 5 iterations achieve 6-figure accuracy, presumably something like 10 iterations will achieve 12-figure accuracy:

```
\iter*{ c^{-1}\sqrt{r_i^2+r_j^2-2r_i r_j
  \cos(\theta_{ij}+\omega t)}
  }[ r_i=10,r_j=20,\theta_{ij}=0.2,t=1 ] [12]
,\quad\info{iter}.
```

⇒ 0.356899026113 , 9 iterations. (Remember, `numerica-plus` knows the values of c and ω from a `\constants` statement in the preamble.) And indeed only 9 iterations suffice to achieve 12-figure accuracy:

```
\iter[do =11,see=4]
{ c^{-1}\sqrt{r_i^2+r_j^2-2r_i r_j
  \cos(\theta_{ij}+\omega t)}
}[ r_i=10,r_j=20,\theta_{ij}=0.2,t=1 ] [12]
```

```
⇒
0.382354696292,    (r_i = 10, r_j = 20, θ_ij = 0.2, t = 1)
... final 4 of 11:
↔ 0.356899026114
↔ 0.356899026113
↔ 0.356899026113
↔ 0.356899026113
```

Or again, with another example from earlier,

`$ \iter*{\cos x}[x=\pi/2] $,\ \info{iter}. \implies 0.739085, 37 iterations.`

That suggests that around $2 \times 37 = 74$ iterations will give a $2 \times 6 = 12$ -figure answer, well within the cut-off figure of 100:

`$ \iter*{\cos x}[x=\pi/2][12] $,\ \info{iter}. \implies 0.739085133215, 72 iterations.`

2.2 Settings option

The settings option is a comma-separated list of items of the form *key = value*.

2.2.1 Inherited settings

Nearly all of the keys discussed in the settings option for `\nmcEvaluate` are available for `\nmcIterate`. Table 1.1 above lists these, repeating a table from `numerica.pdf`. Thus should a quantity in the *vv*-list depend on the iteration variable, forcing an implicit mode calculation, simply enter, as with `\eval`, `vv@=1` (alternatively, `vvmode=1`) in the settings option:

`\iter*[vv@=1]{f(x)}[f(x)=1+a/x,a=12,x=1] \implies 4.`

Implicit in the example is the default multi-token setting `xx=1` inherited from `\eval` and ensuring that the multi-token variable $f(x)$ is treated correctly.

Let's add `dbg=1` to the example:

`\iter*[dbg=1,vv@=1]{f(x)}[f(x)=1+a/x,a=12,x=1] \implies`

```
vv-list: \nmc_x =1+a/x, a=12, x=1
function: \nmc_x
stored: \nmc_x =3.999999832569298, a=12, x=4.000000223240948
fp-form: (3.999999832569298)
result: 4
```

The multi-token variable $f(x)$ has been changed to a single-token. The values shown under 'stored' and 'fp-form' are those of the *final* iteration.

2.2.2 \nmcIterate-specific settings

In addition to the inherited settings there are some specific to `\nmcIterate`. These are listed in Table 2.1.

Table 2.1: Settings for `\nmcIterate`

key	type	meaning	default
<code>var</code>	token(s)	iteration variable	
<code>+</code>	int	fixed point extra rounding	0
<code>max</code>	int > 0	max. iteration count (fixed points)	100
<code>do</code>	int > 0	number of iterations to perform	5
<code>see</code>	int > 0	number of final iterations to view	4
	int (0/1/2)	form of result saved with <code>\</code>	0

2.2.2.1 Iteration variable

In nearly all of the examples so far, the iteration variable has been the rightmost variable in the `vv`-list and has not needed to be otherwise specified. However it is sometimes not feasible to indicate the variable in this way. In that case, entering

```
var = <variable name>
```

in the `settings` option enables the variable to be specified, irrespective of what the rightmost variable in the `vv`-list is. Here, `<variable name>` will generally be a character like `x` or `t` or a token like `\alpha`, but it could also be a multi-token name like `x'` or `\beta_{ij}` (or even `Fred` if you so chose). Although the iteration variable can be independently specified like this, it must still be given an initial *value* in the `vv`-list – only it need not be the rightmost variable.

In the following example the rightmost variable is `a` which is clearly *not* the iteration variable:

```
\iter[var=x,do=40,see=5]{\$ 1+a/x \$}[x=a/6,a=6][*] ==>
1 + a/x = 7.000000,      (x = a/6, a = 6)
... final 5 of 40:
  ↪ 2.999998
  ↪ 3.000001
  ↪ 2.999999
  ↪ 3.000000
  ↪ 3.000000
```

2.2.2.2 Extra rounding for fixed-point calculations

`numera-plus` determines that a fixed point has been reached when the difference between successive iterations vanishes when rounded to the current rounding value. One might want reassurance that this really is the correct value by seeking a fixed point at a higher rounding value than that displayed. This extra rounding is achieved by entering

`+ = <integer>`

in the settings option. By default this extra rounding is set to zero.

We have seen before that $\cos x$ starting at $x = \frac{1}{2}\pi$ takes 37 iterations to reach a 6-figure fixed point 0.739085, about 6 iterations per decimal place. By entering `+=1` in the settings option the number of iterations is increased to 43, 6 more than 37 but, reassuringly, the 6-figure result that is displayed remains unchanged:

```
$ \iter*+=1{\cos x}[x=\pi/2] $,\ \info{iter}.  $\implies$  0.739085, 43
iterations.
```

2.2.2.3 Maximum iteration count for fixed-point searches

To prevent a fixed-point search from continuing indefinitely when no fixed point exists, there needs to be a maximum number of iterations specified after which point the search is called off. By default this number is 100. To change it enter

`max = <positive integer>`

in the settings option.

2.2.2.4 Number of iterations to perform

To specify the number of iterations to perform enter

`do = <positive integer>`

in the settings option. Note that if the `*` option is present this value will be ignored and iteration will continue until either a fixed point or the maximum iteration count is reached. By default `do` is set to 5. (Note that `do` can be set to a greater number than `max`; `max` applies only to `\iter*`.)

2.2.2.5 Number of iterations to show

To specify the number of final iterations to show enter

`see = <positive integer>`

in the settings option. By default `see` is set to 4. Always it is the *last* `see` iterations that are displayed. If `see` is set to a greater value than `do`, all iterations are shown. If the star option is used the `see` value is ignored.

2.2.2.6 Form of result saved by `\nmcReuse`

By entering

`reuse = <integer>`

in the settings option of the `\iter` command it is possible to specify the form of result that is saved when using `\nmcReuse`. (This setting has no effect when the star option is used with `\nmcIterate`. In that case only the numerical result of the fixed point calculation – if successful – is saved.) The possibilities are:

- `int=0` (or any integer $\neq 1, 2$) saves the display resulting from the `\iter` command (the default);
- `int=1` saves a comma-separated list of braced pairs of the form: `{k, value-of-k-th-iterate}`;
- `int=2` saves a comma-separated list of iterate values.

Note that the number and content of the items in the lists are those resulting from the `see` setting (the number of iterations to view).

```
\iter[reuse=1,do=12,see=4]
  {\[ kx(1-x) \]}[k=3.5,x=0.5]
\reuse{logistic}
```

⇒

$$kx(1-x) = 0.875, \quad (k = 3.5, x = 0.5)$$

... final 4 of 12:
 ↪ 0.874997
 ↪ 0.38282
 ↪ 0.826941
 ↪ 0.500884

whence `\logistic` ⇒ 9,0.874997,10,0.38282,11,0.826941,12,0.500884. As you can see the control sequence `\logistic` displays as a comma-separated list of numbers, alternating between the iterate ordinal and the iterate value. That these are stored as braced pairs can be seen by using TeX's `\meaning` command:

```
\meaning \logistic
```

⇒ macro:->{9,0.874997},{10,0.38282},{11,0.826941},{12,0.500884}

2.2.3 Changing default values

If you wish to change the default values of the various settings for `\nmcIterate` this can be done by entering new values in a configuration file `numerica-plus.cfg` as described in the chapter on settings in the associated document `numerica.pdf`. The relevant keys are listed in Table 2.2, corresponding to the `+`, `max`, `do`, `see` and `reuse` settings of the `\iter` command. (Obviously it makes no sense

Table 2.2: Defaults for `\nmcIterate`

key	default
<code>iter-extra-rounding</code>	0
<code>iter-max-iterations</code>	100
<code>iter-do</code>	5
<code>iter-see-last</code>	4
<code>iter-reuse</code>	0

to have a default setting for the iteration variable. That will change from case to case.)

2.3 Errors

By errors I refer to `numerica-plus` errors rather than `LATEX` errors. We have already met one in the discussion of fixed points:

```
\iter*{kx(1-x)}[k=3.5,x=0.5] ==>
!!! No fixed point attained after 100 iterations of: formula. !!!
```

For a function to be iterated indefinitely, its range must lie within or be equal to its domain. If even part of the range of a function lies outside its domain, then on repeated iteration there is a chance that a value will eventually be calculated which lies in this ‘outside’ region. Iteration cannot continue beyond this point and an error message is generated. As an example consider the inverse cosine, `\arccos`. This can be iterated only so far as the iterated values lie between ± 1 inclusive. If we try to iterate `\arccos` at 0 for example, since $\cos \frac{1}{2}\pi = 0$, $\arccos 0 = 1.5708$ (which is $\frac{1}{2}\pi$) so only a first iterate is possible. But we could choose an initial value more carefully; 37 iterations of the cosine at $\frac{1}{2}\pi$ led to a fixed point 0.739085, so let’s choose 0.739085 as initial point and perform 37 iterations:

```
\iter[do=37,see=4]{\[\ \arccos x \]}[x=0.739085] ==>
arccos x = 0.739085,    (x = 0.739085)
... final 4 of 37:
↔ 0.644659
↔ 0.870219
↔ 0.515149
↔ 1.029615
```

The result of the 37th iteration is greater than 1. Thus increasing the number of iterations to 38 should generate an error message:

```
\iter[do=38,see=4]{\[\ \arccos x \]}[x=0.739085]
==>!!! 13fp error ‘Invalid operation’ in: formula. !!!
```

`13fp` objects when asked to find the inverse cosine of a number greater than 1.

Chapter 3

Finding zeros and extrema: `\nmcSolve`

`numerica-plus` provides a command, `\nmcSolve` (short-name form `\solve`), for finding a zero of a function, should it have one. In the following example,

$$\begin{aligned} & \text{\solve[p]{\[e^{-ax}-bx^2 \]}[a=2,b=3,{x}=0] \implies \\ & e^{ax} - bx^2 = -0.000002, \quad (a = 2, b = 3) \rightarrow x = -0.390647 \end{aligned}$$

I have sought and found a solution x to the equation $e^{ax/2} - bx^2 = 0$ when $a = 2$ and $b = 3$, starting with a trial value $x = 0$, entered as the *rightmost* variable in the vv-list (and em-braced since I don't want this trial value displaying in the presentation of the result). Although x has been found to the default six-figure accuracy, it is evident that the function vanishes only to five figures. Let's check:

$$\begin{aligned} & \text{\eval{\$ bx^2 \$}[b=3,x=x=-0.390647] \implies \\ & \quad bx^2 = 0.457815, \quad (b = 3, x = -0.390647), \\ & \text{\eval{\$ e^{-ax} \$}[a=2,x=-0.390647] \implies \\ & \quad e^{ax} = 0.457813, \quad (a = 2, x = -0.390647); \end{aligned}$$

the values agree save in the final digit.

This discrepancy in the final decimal place or places is a general feature of solutions found by `\solve`. It is the value of x , not the value of $f(x)$, that is being found (in this case) to six figures. If the graph of a function crosses the x -axis steeply then the x value (the zero) may be located to a higher precision than the function value. Conversely, if the graph of a function crosses the x -axis gently (at a shallow angle) then the function value will vanish to a greater number of decimal places than the zero (the x value) is found to.

A second example, which we can check against values tabulated in *HMF*, is to find a value of x that satisfies $\tan x = \lambda x$. In other words, find a zero of $\tan x - \lambda x$. In the example λ is negative, so a trial value for x greater than $\pi/2$ seems like a good idea. I've chosen $x = 2$.

```
\solve{\tan x - \lambda x }[\lambda=-1/0.8,{x}=2] [5] ==>
\tan x - \lambda x = -0.00002, (\lambda = -1/0.8) \to x = 1.95857.
```

Table 4.19 of *HMF* lists values of x against λ and this is the value tabulated there.

3.1 Extrema

A function may not have a zero; or, for the given initial trial value and initial step in the search for a zero, there may be a local extremum in the way. In that case `numerica-plus` may well locate the local extremum (maximum or minimum but not a saddle point). For example for the quadratic $(2x - 1)^2 + 3x + 1$ the `\solve` command gives the result

```
\solve[vvi=]{(2x-1)^2+3x+1 }[x=2]
==>(2x - 1)^2 + 3x + 1 = 1.9375 \to x = 0.124999.
```

Since $(2x - 1)^2 + 3x + 1 \neq 0$ for any (real number) x , we deduce that the quadratic takes a minimum value 1.9375 at $x = 0.125$ – easily confirmed analytically. This particular minimum is a global minimum but in general any extremum found is only *local*. The function may well take larger or smaller values (or vanish for that matter) further afield.

It is also worth noting in this example the `vvi=` in the settings option which suppresses display of the `vv-list`. (The only member of the `vv-list` is the trial value `x=2` which we do not want to display.)

Note that the function for which a zero is being sought is *not* equated to zero when entered in the `\solve` command. It is `\solve{ f(x) }`, not `\solve{ f(x)=0 }`. This is precisely because it may be an extremum that is found rather than a zero (if extremum or zero is found at all – think e^x). The display of the result makes clear which is which, equating $f(x)$ to its value, zero or extremum depending on what has been found, as you can see in the preceding examples.

3.1.1 The search strategy

If you have some sense of where a function has a zero, then choose a trial value in that vicinity. `\solve` uses a bisection method to home in on the zero. It therefore needs *two* initial values. For the first it uses the trial value you specify, call it a and for the second, by default, it uses $a + 1$. (The default value 1 for the initial step from the trial value can be changed in the settings option; see §3.3.) If $f(a)$ and $f(a + 1)$ have opposite signs then that is good. Bisection of the interval $[a, a + 1]$ can begin immediately in order to home in on the precise point where f vanishes. Write $b = a + 1$.

- Let $c = \frac{1}{2}(a + b)$; if $f(c) = 0$ the zero is found; otherwise either $f(a), f(c)$ are of opposite signs or $f(c), f(b)$ are of opposite signs. In the former case write $a_1 = a, b_1 = c$; in the latter case write $a_1 = c, b_1 = b$ and then redefine $c = \frac{1}{2}(a_1 + b_1)$. Continue the bisection process, either until an exact zero c of f is reached ($f(c) = 0$) or a value c is reached where the difference between a_{n+1} and b_{n+1} is zero at the specified rounding value. (But note, $f(c)$ may not vanish at that rounding value – the zero might be elsewhere in the interval and f might cross the axis at a steep slope.)

However $f(a)$ and $f(b) = f(a + 1)$ may not have opposite signs. If we graph the function $y = f(x)$ and suppose $f(a), f(b)$ are distinct but of the same sign, then the line through the points $(a, f(a)), (b, f(b))$ will intersect the x -axis to the left of a or the right of b depending on its slope. We search always *towards the x -axis* in steps of $b - a$ ($= 1$ with default values).

- If the line intersects the axis to the left of a then $c = a - (b - a)$ and we set $a_1 = c, b_1 = a$; if the line intersects the axis to the right of b then $c = b + (b - a)$ and we set $b_1 = c, a_1 = b$. The hope is that by always taking steps in the direction towards the x -axis that eventually $f(c)$ will be found to lie on the *opposite* side of the axis from $f(a_n)$ or $f(b_n)$, at which point the bisection process begins.
- Of course this may not happen. At some point c may lie to the left of a_n but $|f(c)| > |f(a_n)|$, or c may lie to the right of b_n but $|f(c)| > |f(b_n)|$. The slope has reversed. In that case we halve the step value to $\frac{1}{2}(b - a)$ and try again in the same direction as before from the same point as before (a_n or b_n as the case may be).
- Should we find at some point that $f(a_n) = f(b_n)$ then the previous strategy does not apply. In this case we choose a_{n+1} and b_{n+1} at the quarter and three-quarter marks between a_n and b_n . Either $f(a_{n+1})$ and $f(b_{n+1})$ will differ and the previous search strategy can start again or we are on the way to finding an extremum of f .

As already noted it is also possible that our function has neither zeros nor extrema. To prevent the search continuing indefinitely, `numerica` uses a cut-off value for the maximum number of steps pursued – by default set at 100.

3.1.1.1 Elusive extrema

The strategy ‘search always towards the x -axis’ has a consequence: it means that a local maximum above the x -axis will almost certainly not be found, since ‘towards the x -axis’ pulls the search away from the maximum. Similarly a local minimum below the x -axis will also not be found since ‘towards the x -axis’ pulls the search away from the minimum.

One way of countering this elusiveness is to add a constant value (possibly negative) to the function whose zeros and extrema are being sought. The zeros of the function will change but the abscissae (x values) of the extrema remain

unchanged. If the constant is big enough it will push a local minimum above the axis where it can be found or, for a negative constant, push a local maximum below the axis where it can be found.

For example $f(x) = x^3 - x$ has roots at $-1, 0, 1$, a local maximum at $-\frac{1}{\sqrt{3}}$ and a local minimum at $\frac{1}{\sqrt{3}}$. To locate the minimum, I have added an unnecessarily large constant k to $f(x)$. ($k = 1$ would have sufficed, but note, $k = 0$ fails.)

$$\backslash\text{solve}\{\$ x^3-x+k \$\}[k=5,\{x\}=0.5] \implies \\ x^3 - x + k = 4.6151, \quad (k = 5) \rightarrow \quad x = 0.577351.$$

Checking, $\backslash\text{eval}\{\$\tfrac{1}{\sqrt{3}}\} \implies \frac{1}{\sqrt{3}} = 0.57735$. There is a discrepancy in the 6th decimal place which can be eliminated by using the extra rounding setting; see §3.3.2.3.

3.1.1.2 False extrema

A function which ‘has an infinity’ at a particular value can result in a false extremum being found:

$$\backslash\text{solve}\{\$ 1/x \$\}[x=-1/3] \implies \\ 1/x = -3145728.00033, \quad (x = -1/3) \rightarrow \quad x = 0.$$

One needs to look for extrema with some awareness, a general sense of how the function behaves. ‘Searching blind’ may lead to nonsense results. In this particular example, changing the rounding value will show the supposed extremum jumping from one large value to another and not settling at a particular value.

3.2 Star (*) option

A starred form of the `\nmcSolve` command suppresses all elements of display of the result apart from the numerical value. In v.1 of `numerica`, when the commands of `numerica-plus` were invoked with a package option, nesting a `\solve*` command within another command was the form to use. Now that `numerica-plus` is a separate package (but labelled v.2), the star is no longer necessary. `numerica` and associated packages understand a nested command to be the starred form, whether the star is explicitly present or not.

With the ‘elusive’ extremum example above, we can find the actual value of the minimum by nesting `\solve*` or `\solve` within the vv-list of an `\eval` command:

$$\backslash\text{eval}\{\$ x^3-x \$\}[x=\{\backslash\text{solve}\{y^3-y+k\}[k=5,y=0.5]\}] \implies \\ x^3 - x = -0.3849, \quad (x = 0.577351).$$

(Note the braces around the `\solve` and its vv-list to hide *its* square-brackets from the parsing of the vv-list of the `\eval` command.) The result is to be compared with $\backslash\text{eval}\{*\$ x^3-x \$\}[x=\tfrac{1}{\sqrt{3}}] \implies -0.3849$.

3.3 Settings option

The settings option is a comma-separated list of items of the form *key = value*.

3.3.1 Inherited settings

The keys discussed in the settings option for `\nmcEvaluate` are also available for `\nmcSolve`. The very first example in this chapter used the punctuation option `p` (`\solve[p]{\ [...]}`) inherited from the `\eval` command to ensure a comma after the display-style presentation of the result. We also saw in the quadratic example illustrating extrema the use of `vv` with no value to suppress display of the `vv`-list: `\solve[vv=]{\$... }`

Putting `dbg=1` produces a familiar kind of display. Using the function

$$ct - \sqrt{a^2 + b^2 - 2ab \cos(\beta + \omega t)}$$

from the rotating disk problem,

```
\solve[dbg=1,var=t]
  {\$ ct-\sqrt{a^2+b^2}-2ab\cos(\beta+\omega t)}
  \$}[a=10,b=20,\beta=1,{t}=0][4]
```

⇒

```
vv-list: a=10, b=20, \beta =1, t=0
function: ct-\sqrt {a^2+b^2}-2ab\cos (\beta +\omega t)}
stored: a=10, b=20, \beta =1, t=0.601715087890625
fp-form: (30)(0.601715087890625)-sqrt((10)^2+(20)^2)-
          2(10)(20)cos(((1)+(0.2)(0.601715087890625))))
result: 0.6017
```

3.3.1.1 Multi-line display of the result

By default the result is presented on a single line. Unless the star option is being used, this can be of the form *function = function value, (vv-list) → result*. It takes only a slightly complicated formula and only a few variables in the `vv`-list before this becomes a crowded line, likely to exceed the line width and extend into the margin. To split the display over two lines choose a `vvd` specification in the `vv`-list like, for instance, `vvd={,}\(vv)`. The `\` is a trigger for `numerica` to replace whatever environment the `\eval` command is wrapped around with a `multiline` environment. An asterisk in the `vv`-list replaces `multiline` with `multiline*` so that no equation number is used:

```
\solve[p=.,vvd={,}\(vv),*]
  {\$ ct-\sqrt{a^2+b^2}-2ab\cos(\beta+\omega t)}
  \$}[a=10,b=20,\beta=1,{t}=0][4]
```

Table 3.1: Settings for `\nmcSolve`

key	type	meaning	default
<code>var</code>	token(s)	equation variable	
<code>dvar</code>	real $\neq 0$	initial step size	1
<code>+</code>	int	extra rounding	0
<code>max</code>	int > 0	max. number of steps before cut off	100
<code>reuse</code>	int (0/1)	form of result saved with <code>\reuse</code>	0

\implies

$$ct - \sqrt{a^2 + b^2 - 2ab \cos(\beta + \omega t)} = -0.0007,$$

$$(a = 10, b = 20, \beta = 1) \rightarrow t = 0.6017$$

You could introduce a third line if you wished to display the result on a line of its own by using the spec. `vvd={,}\(vv)\`.

The function evaluates to -0.0007 . Is this a zero that has been found or an extremum? To find out, the calculation needs to be carried out to a higher rounding value which is the reason why `\nmcSolve` has an extra rounding setting; see §3.3.2.3 below.

3.3.2 `\nmcSolve`-specific settings

In addition there are some settings specific to `\nmcSolve`. These are listed in Table 3.1.

3.3.2.1 Equation variable

By default the equation variable is the *rightmost* variable in the `vv`-list. This may not always be convenient. A different equation variable can be specified by entering

```
var = <variable name>
```

in the `vv`-list. `<variable name>` will generally be a single character or token – x, t, α, ω – but is not necessarily of this kind. Multi-token names are perfectly acceptable (with the default multi-token setting; see the associated document `numerica.pdf` about this).

3.3.2.2 Initial step size

The `vv`-list contains the equation variable set to a trial value. But `\solve` needs *two* initial values to begin its search for a zero or extremum; see §3.1.1. Ideally,

these values will straddle a zero of the function being investigated. By default, the second trial value is 1 more than the first: if the equation variable is set to a trial value a then the second value defaults to $a + 1$. The ‘+1’ here can be changed by entering in the settings option

`dvar = <non-zero real number>`

For instance, `dvar=-1`, or `dvar=\pi` are two valid specifications of initial step size. The notation is prompted by the use of expressions like x and $x + dx$ for two nearby points in calculus.

An example where the default step value is too big and a smaller one needs to be specified is provided by Planck’s radiation function (*HMF* Table 27.2),

$$f(x) = \frac{1}{x^5(e^{1/x} - 1)}.$$

From the (somewhat coarse-grained) table in *HMF* it is clear that there is a maximum of about 21.2 when x is a little more than 0.2. This is a maximum above the x -axis and hence ‘elusive’. To find it, subtract 100 (say) from the formula and again use the ability to nest commands to display the result. In the example, I find in the `vv`-list of the `\eval` command the value of x which maximizes the Planck radiation function, then calculate the maximum in the main argument of the `\eval` command. Note the `dvar=0.1` in the settings option of the `\solve*` command:

```
\eval [p=.]{\[\ \frac{1}{x^5(e^{1/x}-1)} \]}
  [ x={ \solve*[dvar=0.1]
    { \frac{1}{y^5(e^{1/y}-1)}-100 } [y=0.1]
  } ]
```

\Rightarrow

$$\frac{1}{x^5(e^{1/x} - 1)} = 21.201436, \quad (x = 0.201405).$$

The maximum is indeed a little over 21.2 and the x value a little more than 0.2.

The default `dvar=1` is too big for this problem. From the table in *HMF*, $f(0.1) = 4.540$ and $f(1.1) = 0.419$. Thus for $f(x) - 100$ the ‘towards the x -axis’ search strategy would lead to negative values of x with the default `dvar` setting.

3.3.2.3 Extra rounding

`\solve` determines that a zero or an extremum has been reached when the difference between two successive bisection values vanishes at the specified rounding value (the value in the final trailing optional argument of the `\solve` command; 6 by default). If our function is $f(x)$ then $|x_{n+1} - x_n| = 0$ to the specified rounding value and $f(x_n)$, $f(x_{n+1})$ have opposite signs or at least one vanishes. Then (assuming $x_{n+1} > x_n$ and continuity) there must be a critical value $x_c \in [x_n, x_{n+1}]$ such that $f(x_c) = 0$ exactly. But in general the critical value x_c will not coincide with x_n or x_{n+1} . If $f(x)$ crosses the x -axis at a steep angle it

may well be that although $f(x_c)$ vanishes to all 16 figures, $f(x_n)$ and $f(x_{n+1})$ do not, not even at the (generally smaller) specified rounding value. For instance, suppose $f(x) = 1000x - 3000$ and that our trial value is $x = e$:

$$\begin{aligned} \text{\solve[vvi=]\{\$ 1000x-3000 \$}[x=e] [4*]} &\implies \\ 1000x - 3000 = -0.0409 &\rightarrow x = 3.0000. \end{aligned}$$

Although the difference between successive x values vanishes to 4 places of decimals, $f(x)$ does not, not even to 2 places. If we want the function to vanish at the specified rounding value – 4 in the example – then we will need to locate the zero more precisely than that.

This is the purpose of the extra rounding key in the settings option. Enter

`+ = <integer>`

in the settings option of the `\solve` command to add `<integer>` to the rounding value determining the conclusion of the calculation. By default, `+=0`.

With this option available it is easy to check that `+=3` suffices in the example to ensure that both x and $f(x)$ vanish to 4 places of decimals,

$$\begin{aligned} \text{\solve[+=3]\{\$ 1000x-3000 \$}[x=e] [4*]} &\implies \\ 1000x - 3000 = 0.0000, (x = e) &\rightarrow x = 3.0000, \end{aligned}$$

and that `+=2` does not, i.e., we need to locate the zero to $4 + 3 = 7$ figures to ensure the function vanishes to 4 figures.

There is no need for the `<integer>` to be positive. In fact negative values can illuminate what is going on. In the first of the following, the display is to 10 places but `+=-4` the calculation is only to $10 - 4 = 6$ places. In the second, the display is again to 10 places, but `+=-3` the calculation is to $10 - 3 = 7$ places.

$$\begin{aligned} \text{\solve[+=-4]\{\$ 1000x-3000 \$}[x=e] [10*]} &\implies \\ 1000x - 3000 = -0.0008711259, (x = e) &\rightarrow x = 2.9999991289, \\ \text{\solve[+=-3]\{\$ 1000x-3000 \$}[x=e] [10*]} &\implies \\ 1000x - 3000 = -0.0000366609, (x = e) &\rightarrow x = 2.9999999633. \end{aligned}$$

Only in the second does $f(x) = 1000x - 3000$ vanish when rounded to 4 figures.

Returning to an earlier example (§3.3.1.1) in which it was not entirely clear whether a zero or an extremum had been found, we can now resolve the confusion. Use the extra rounding setting (and pad with zeros to emphasize the 4-figure display by adding an asterisk in the trailing optional argument):

$$\begin{aligned} &\text{\solve[+=2,vvd=\{,\}\(\vv),*]} \\ &\quad \{\$ ct-\sqrt{a^2+b^2}-2ab\cos(\beta+\omega t)\} \\ &\quad \{\$ [a=10,b=20,\beta=1,\{t\}=0] [4*]} \\ \implies & \\ & ct - \sqrt{a^2 + b^2 - 2ab \cos(\beta + \omega t)} = 0.0000, \\ & (a = 10, b = 20, \beta = 1) \rightarrow t = 0.6017 \end{aligned}$$

3.3.2.4 Maximum number of steps before cut-off

Once two function values have been found of different sign, bisection is guaranteed to arrive at a result. The problem is the *search* for two such values. This may not terminate – think of a function like e^x which lacks both zeros and extrema. To prevent an infinite loop, `\solve` cuts off the search after 100 steps. This cut-off value can be changed for a calculation by entering

```
max = <positive integer>
```

in the settings option.

To illustrate, we know that $1/x$ has neither zero nor extremum, but we do not get an infinite loop; we get an error message if we attempt to ‘solve’ $1/x$:

```
\solve{ 1/x }[x=1] ==>
!!! No zero/extremum found after 100 steps for function: 1/x. !!!
```

3.3.2.5 Form of result saved by `\nmcReuse`

As with `\eval` and `\iter` it is possible to specify to some extent what is saved to file when using `\reuse` after a `\solve` command. The form of entry in the settings option is

```
reuse = <integer>
```

If the star option is used with the `\solve` command the numerical result is the only thing saved, but in the absence of the star option,

- `reuse=0` saves *the form that is displayed*. For example, if the display is of the form *function = function value, (vv-list) → result* then that is what is saved; this is the default behaviour;
- `reuse=1` (or any non-zero integer) saves only the numerical result.

3.3.3 Changing default values

If you wish to change the default values of the various settings for `\nmcSolve` this can be done by entering new values in a configuration file `numerica-plus.cfg` as described in the chapter on settings in the associated document `numeric.pdf`. The relevant keys are listed in Table 3.2, corresponding to the `dvar`, `+`, `max` and `reuse` settings of the `\solve` command. (Obviously it makes no sense to have a default setting for the solution variable. That will change from case to case.)

Table 3.2: Defaults for `\nmcSolve`

key	default
<code>solve-first-step</code>	1
<code>solve-extra-rounding</code>	0
<code>solve-max-steps</code>	100
<code>solve-reuse</code>	0

Chapter 4

Recurrence relations:

`\nmcRecur`

One of the simplest recurrence relations is that determining the Fibonacci numbers, $f_{n+2} = f_{n+1} + f_n$, with initial values $f_0 = f_1 = 1$. The command `\nmcRecur`, short-name form `\recur`, allows calculation of the terms of this sequence:

```
$ \nmcRecur[do=8,see1=8,...]
{ f_{n+2}=f_{n+1}+f_{n} }
[f_{1}=1,f_{0}=1] $
```

$\implies 1, 1, 2, 3, 5, 8, 13, 21, \dots$

The recurrence relation is entered in the main argument (between braces), the initial values in the vv-list trailing the main argument, and the display specification is placed in the settings option: `do=8` terms to be calculated, all 8 to be viewed (`see1=8`), and the display to be concluded by an ellipsis to indicate that the sequence continues (those are three dots/periods/full stops in the settings option, not an ellipsis glyph).

A more complicated recurrence relation determines the Legendre polynomials:

$$(n+2)P_{n+2}(x) - (2n+3)xP_{n+1}(x) + (n+1)P_n(x) = 0.$$

For the purposes of `\recur` we need P_{n+2} expressed in terms of the lower order terms:

$$P_{n+2}(x) = \frac{1}{n+2} ((2n+3)xP_{n+1}(x) - (n+1)P_n(x)).$$

It is this standard form – the term to be calculated on the left, equated to an expression involving a fixed number of lower-order terms on the right – that `numerica-plus` works with. For $P_0(x) = 1$, $P_1(x) = x$ and $x = 0.5$, the terms are calculated thus:

```
\recur[p,do=11,see1=4,see2=2,vvd={,}\(vv)\,*,]
```

```

{\[ P_{n+2}(x)=\frac{1}{n+2}
\Bigl((2n+3)xP_{n+1}(x)-(n+1)P_n(x)\Bigr)
\]}[P_{1}(x)=x,P_{0}(x)=1,x=0.5]

```

⇒

$$P_{n+2}(x) = \frac{1}{n+2} \left((2n+3)xP_{n+1}(x) - (n+1)P_n(x) \right),$$

$$(P_1(x) = x, P_0(x) = 1, x = 0.5)$$

$$\rightarrow 1, 0.5, -0.125, -0.4375, \dots, -0.267899, -0.188229$$

where $P_9(0.5)$ and $P_{10}(0.5)$ are the last two displayed values (and to 6-figures are the values listed in *HMF* Table 8.1).

These examples also illustrate a common behaviour of the commands in `numerica` and associated packages: when wrapped around math delimiters: the display is of the *expression=result* form, and when placed between math delimiters the display is simply of the *result*. When used without math delimiters, `numerica-plus` treats the command as if it had been placed between `\[\]`.

4.1 Notational niceties

More than the other commands in `numerica` and associated packages, `\nmcRecur` depends on getting the notation into a standard form.

- The terms of the recurrence must be *subscripted*: f_n , $P_n(x)$ are examples.
- The recurrence relation is placed in the main (mandatory) argument of `\nmcRecur` in the form: *high-order term=function of lower-order terms*.
- The initial-value terms in the `vv`-list must occur left-to-right in the order *high to low* order.
- The recurrence variable changes by 1 between successive terms.

The example for Legendre polynomials in particular shows what is required. The Fibonacci example is simpler, since the recurrence variable does not occur independently in the recurrence relation as it does with the Legendre polynomials. In both cases though the recurrence variable is absent from the `vv`-list.

4.1.1 Vv-list and recurrence variable

The recurrence variable is required in the `vv`-list only when an implicit mode calculation is undertaken. Suppose we write A and B for the coefficients $2n+3$ and $n+1$ respectively in the Legendre recurrence. A and B will now need entries in the `vv`-list which means the recurrence variable will need a value assigned to it there too, and we will need to add `vv@=1` (or `vvmode=1`) to the settings option.


```
\recur [p, vvmode=1, do=11, see1=4, see2=2, vvd={,}\(vv)\,*,
  {\ [ P_{n+2}(x)=\frac{1}{n+2}
    \Bigl(AxP_{n+1}(x)-BP_n(x)\Bigl)
  \]} [P_{1}(x)=x, P_{0}(x)=1, x=0.5, A=2n+3, B=n+1, n=0]
```

⇒

$$P_{n+2}(x) = \frac{1}{n+2} \left(AxP_{n+1}(x) - BP_n(x) \right),$$

$$(P_1(x) = x, P_0(x) = 1, x = 0.5, A = 2n + 3, B = n + 1, n = 0)$$

$$\rightarrow 1, 0.5, -0.125, -0.4375, \dots, -0.267899, -0.188229$$

Since the `vv`-list is evaluated from the right, the left-to-right high-to-low ordering of the initial-value terms means the value of the lowest order term is read first. Although `numerica-plus` depends on this order of occurrence of the terms, they do not need to be *consecutive* as in the examples so far (although it is natural to enter them in this way). `numerica-plus` reads the value of the subscript of only the right-most term (the lowest order term), increments it by 1 when reading the next recurrence term to the left, and so on. The reading of the subscript of the lowest order term in the `vv`-list provides the initial value of the recurrence variable.

In the following example I have placed other items between $P_1(x)$ and $P_0(x)$ in the `vv`-list (but maintained their left-to-right order) and given the recurrence variable n a ridiculous initial value $\pi^2/12$. (Because of the order in which things get done ‘behind the scenes’, *some* value is necessary so that the n in ‘ $B = n + 1$ ’ does not generate an ‘unknown token’ message.) The result is unchanged.

```
\recur [p, vvmode=1, do=11, see1=4, see2=2, vvd={,}\(vv)\,*,
  {\ [ P_{n+2}(x)=\frac{1}{n+2}
    \Bigl(AxP_{n+1}(x)-BP_n(x)\Bigl)
  \]} [A=2n+3, P_{1}(x)=x, B=n+1, n=\pi^2/12, P_{0}(x)=1, x=0.5]
```

⇒

$$P_{n+2}(x) = \frac{1}{n+2} \left(AxP_{n+1}(x) - BP_n(x) \right),$$

$$(A = 2n + 3, P_1(x) = x, B = n + 1, n = \pi^2/12, P_0(x) = 1, x = 0.5)$$

$$\rightarrow 1, 0.5, -0.125, -0.4375, \dots, -0.267899, -0.188229$$

4.1.2 Form of the recurrence relation

As noted earlier, the form of the recurrence must be entered in the main argument in the form: *highest order term = function of consecutive lower order terms*. The number of lower order terms is the order of the recurrence. The Fibonacci and Legendre polynomial recurrences are both second order and presented in the form: *n + 2-th term = function of n + 1-th term and n-th term*. We could equally have done

```
\nmcRecur[p,do=8,see1=8,...]
{\$ f_{n}=f_{n-1}+f_{n-2} \$}
[f_{1}=1,f_{0}=1]
```

$\Rightarrow f_n = f_{n-1} + f_{n-2}$, ($f_1 = 1, f_0 = 1$) \rightarrow 1, 1, 2, 3, 5, 8, 13, 21, ... where now the recurrence is of the form *n-th term = function of n - 1-th term and n - 2-th term*, or (adjusting the coefficients as well as the recurrence terms),

```
\recur[p=.,do=10,see1=4,see2=2,vvd={,}\(\vv)\,*,*\]{\[
P_{n+1}(x)=\frac{1}{n+1}
\Bigl((2n+1)xP_n(x)-nP_{n-1}(x)\Bigr)
\]}[P_2(x)=-0.125,P_1(x)=x,x=0.5]
```

\Rightarrow

$$P_{n+1}(x) = \frac{1}{n+1} \left((2n+1)xP_n(x) - nP_{n-1}(x) \right),$$

$$(P_2(x) = -0.125, P_1(x) = x, x = 0.5)$$

$$\rightarrow 0.5, -0.125, -0.4375, -0.289062, \dots, -0.267899, -0.188229$$

The recurrence here is of the form *n + 1-th term = function of n-th term and n - 1-th term*. This last example has one further ‘wrinkle’. I’ve made $P_1(x)$ the lowest order term and decreased the number of terms to calculate by 1 accordingly.

4.1.3 First order recurrences (iteration)

The recurrence relations for both the Fibonacci sequence and Legendre polynomials are second order. There is no reason why the recurrence should not be of third or higher order or, indeed, lower. A first order recurrence provides an alternative means of iterating functions. `\recur` therefore provides a means to display the results of an iteration in a different form from `\iter`.

Iterating $1 + a/x$ in this way, 16 terms gives the sequence

```
\recur[do=16,see1=0,see2=3,...]{\$
x_{n+1}=1+a/x_n
\$}[x_0=1,a=1]
```

$\Rightarrow x_{n+1} = 1 + a/x_n$, ($x_0 = 1, a = 1$) \rightarrow 1.618037, 1.618033, 1.618034, ... to be compared with the example near the start of Chapter 2. (*That* effected 15 iterations; *this* uses 16 terms because of the extra $x_0 = 1$ term.)

4.2 Star (*) option

When the star option is used with the `\nmcRecur` command, only a single term, the *last*, is presented as the result. Repeating the last calculation, but with the star option produces

```
\recur*[p=.,do=10]{\[
P_{n+1}(x)=\frac{1}{n+1}
\Bigl((2n+1)xP_n(x)-nP_{n-1}(x)\Bigr)
\]}[P_2(x)=-0.125,P_1(x)=x,x=0.5]
```

⇒ -0.188229

Although punctuation (a full stop) was specified in the settings, it has been ignored in the display of the result. Other settings would also have been ignored with the exception of the `do` key which is required to know exactly which term to calculate. The star option produces a purely numerical answer without any trimmings.

This seems something of a waste of the star option since it gives much the same result as choosing `do=10,see1=0,see2=1`. Not *exactly* the same, since math delimiters are involved now, but sufficiently similar to make me wonder if I should change the starred form to apply only to those recurrences which approach a limit. The starred form would then produce the limiting value as its result (like `\iter*`). This is a possible change for future versions of `numera-plus` and should be borne in mind if using `\recur*`.

4.3 Settings

The settings option is a comma-separated list of items of the form *key = value*.

4.3.1 Inherited settings

Because recurrence terms are necessarily multi-token, the multi-token key is hard-coded in `\recur` to `xx=1`.

4.3.1.1 Multi-line formatting of result

When the `\recur` command wraps around math delimiters, the `vvd` setting is available to split display of the result over two or more lines. For example, `vvd={,}\(vv)` pushes the `vv`-list and sequence of calculated values to a second line; or, `vvd={,}\qqquad(vv)\(` pushes only the sequence of calculated values to a second line; or `vvd={,}\(vv)\(` pushes the `vv`-list, centred, to a second line and the sequence of values, right aligned, to a third line. The `*` setting is available to suppress equation numbering (by substituting `multline*` for `multline`).

```
\nmcRecur[do=8,see1=8,...,vvd={,}\qqquad(vv)\(,*]
{\$ f_{n+2}=f_{n+1}+f_n \$}
[f_1=1,f_0=1]
```

⇒

$$f_{n+2} = f_{n+1} + f_n, \quad (f_1 = 1, f_0 = 1) \quad \rightarrow \quad 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

4.3.2 `\nmcRecur`-specific settings

In addition to the inherited settings there are some specific to `\nmcRecur`. These are listed in Table 4.1 below.

4.3.2.1 Number of terms to calculate

By entering

```
do = <integer>
```

in the settings option you can specify how many terms of a recurrence to calculate. The default is set to 7 (largely to show a sufficient number of terms of the Fibonacci series to begin to be interesting). Note that `<integer>` will generally *not* correspond to the subscript on the last term calculated since that also depends on the value of the subscript of the lowest order term in the `vv`-list.

4.3.2.2 Number of terms to display

By entering

```
see1 = <integer1>, see2=<integer2>
```

in the settings option, you can specify how many initial terms of the recurrence and how many of the final terms calculated you want to view. If the sum of these settings is less than the `do` setting, then the terms are displayed with an intervening ellipsis. If the sum is greater than the `do` setting, then the values are adjusted so that their sum equals the `do` setting and all terms are displayed.

The adjustment is preferentially to `see1`. Suppose `do=7`, `see1=5`, `see2=4`. Then `see2` is left unchanged but `see1` is reduced to `7-4=3`. If, say, `do=7`, `see1=5`,

Table 4.1: Settings for `\nmcRecur`

key	type	meaning	default
<code>do</code>	<code>int</code> ≥ 0	number of terms to calculate	7
<code>see1</code>	<code>int</code> ≥ 0	number of initial terms to display	3
<code>see2</code>	<code>int</code> ≥ 0	number of final terms to display	2
<code>...</code>	chars	follow display of values with an ellipsis	
<code>reuse</code>	<code>int</code> (0/1/2)	form of result saved with <code>\reuse</code>	0

`see2=8`, then `see2` is reduced to 7 and `see1` to -1 (rather than zero, for technical reasons). The reason for preserving `see2` over `see1` is for the functioning of the `reuse` setting (see below).

The default value for `see1` is 3; the default value for `see2` is 2.

4.3.2.3 Ellipsis

Including three dots in the settings option

...

ensures that a (proper) ellipsis is inserted after the final term is displayed. An example is provided by the display of the Fibonacci sequence at the start of this chapter. By default this option is turned off.

4.3.2.4 Form of result saved by `\nmcReuse`

By entering

```
reuse = <integer>
```

it is possible to specify the form of result that is saved when using `\nmcReuse`. (This setting has no effect when the star option is used with `\nmcRecur`. In that case only the numerical result of the final term calculated is saved.) There are three different outputs possible:

- `int=0` (or any integer $\neq 1, 2$) saves the full display (the default);
- `int=1` saves a comma-separated list of braced pairs of the form: `{k, value-of-term-k}` for the last `see2` terms calculated;
- `int=2` saves a comma-separated list of the values of the last `see2` terms calculated.

As an example, using `reuse=1`,

```
\recur[reuse=1,p=.,vemode=1,do=11,see1=4,see2=2,
vvd={,}\(vv)\(,*]
{\[ P_{n+2}(x)=\frac{1}{n+2}
\Bigl(kxP_{n+1}(x)-(n+1)P_n(x)\Bigl)
\]}[k=2n+3,n=1,P_{1}(x)=x,P_{0}(x)=1,x=0.5]
\reuse[legendre]
```

⇒

$$P_{n+2}(x) = \frac{1}{n+2} \left(kxP_{n+1}(x) - (n+1)P_n(x) \right),$$

$$(k = 2n + 3, n = 0, P_1(x) = x, P_0(x) = 1, x = 0.5)$$

$$\rightarrow 1, 0.5, -0.125, -0.4375, \dots, -0.267899, -0.188229$$

Now check to see what has been saved:

`\legendre$` $\implies 9, -0.267899, 10, -0.188229$.

As you can see, the final two (because of `see2=2`) of the 11 Legendre polynomials calculated ($P_0(x)$ is the first) have been saved, each value preceded by its index value. If the setting had been `reuse=2`, only the two values without the index values would have been saved. To see that they are saved as braced pairs, use TeX's `\meaning` command:

```
\meaning \legendre
 $\implies$  macro:->\{9,-0.267899\},\{10,-0.188229\}
```

4.3.3 Changing default values

If you wish to change the default values of the various settings for `\nmcRecur` this can be done by entering new values in a configuration file `numerica-plus.cfg` as described in the chapter on settings in the associated document `numerica.pdf`. The relevant keys are listed in Table 4.2, corresponding to the `do`, `see1`, `see2` and `reuse` settings of the `\recur` command.

Table 4.2: Defaults for `\nmcRecur`

key	default
<code>recur-do</code>	7
<code>recur-see-first</code>	3
<code>recur-see-last</code>	2
<code>recur-reuse</code>	0

4.3.4 Orthogonal polynomials

I've used Legendre polynomials in examples above, but orthogonal polynomials generally lend themselves to the `\recur` treatment. Quoting from *HMF* 22.7, orthogonal polynomials f_n satisfy recurrence relations of the form

$$a_{1n}f_{n+1}(x) = (a_{2n} + a_{3n}x)f_n(x) - a_{4n}f_{n-1}(x),$$

or in the standard form required by `\recur`,

$$f_{n+1}(x) = \frac{a_{2n} + a_{3n}x}{a_{1n}}f_n(x) - \frac{a_{4n}}{a_{1n}}f_{n-1}(x).$$

HMF 22.7 provides a listing of the coefficients a_{in} for the polynomials of Jacobi, Chebyshev, Legendre, Laguerre, Hermite and others, and tables for these polynomials.

For example, Laguerre polynomials satisfy the recurrence

$$L_{n+1}(x) = \frac{2n+1-x}{n+1}L_n(x) - \frac{n}{n+1}L_{n-1}(x).$$

with initial values $L_0(x) = 1$ and $L_1(x) = 1 - x$. So let's calculate the first 13 Laguerre polynomials for, say, $x = 0.5$:

```
\recur[do=13,see1=4,see2=2,vvd={,}\(vv)\,*,*]{\[
L_{n+1}(x)=\frac{2n+1-x}{n+1}L_n(x)-
\frac{n}{n+1}L_{n-1}(x)
\]}[L_{1}(x)=1-x,L_{0}(x)=1,x=0.5]
```

⇒

$$L_{n+1}(x) = \frac{2n+1-x}{n+1}L_n(x) - \frac{n}{n+1}L_{n-1}(x),$$

$$(L_1(x) = 1-x, L_0(x) = 1, x = 0.5)$$

$$\rightarrow 1, 0.5, 0.125, -0.145833, \dots, -0.313907, -0.23165$$

and for $x = 5$:

```
\recur[p=.,do=13,see1=4,see2=2,vvd={,}\(vv)\,*,*]{\[
L_{n+1}(x)=\frac{2n+1-x}{n+1}L_n(x)-
\frac{n}{n+1}L_{n-1}(x)
\]}[L_{1}(x)=1-x,L_{0}(x)=1,x=5]
```

⇒

$$L_{n+1}(x) = \frac{2n+1-x}{n+1}L_n(x) - \frac{n}{n+1}L_{n-1}(x),$$

$$(L_1(x) = 1-x, L_0(x) = 1, x = 5)$$

$$\rightarrow 1, -4, 3.5, 2.666667, \dots, 0.107544, -1.448604$$

The results (reassuringly) coincide with those provided in *HMF* Table 22.11.

4.3.5 Nesting

It is possible to use the `\recur` command (with star in v.1 of `numerica`, with or without star in v.2 of `numerica-plus`) within an `\eval`, `\iter`, or `\solve` command, and indeed in `\recur` itself, but with this caveat: if `\recur` is nested within another command, the initial terms of the recurrence – e.g., $f_1 = 1, f_0 = 1$, for the Fibonacci series, or $L_1(x) = 1-x, L_0(x) = 1$ for the Laguerre polynomials – *must be located in the vv-list of that inner \recur command*. Other shared variables can often be shifted to the vv-list of the outer command, but not these initial terms.

In the following example I multiply together (rather futilely) the third and fourth members of the sequence of Laguerre polynomials for $x = 5$ (the answer expected is `\eval{3.5\times2.666667}` $\implies 9.333334$). Note that although it is tempting to shift the shared vv-lists of the inner `\recur*` commands to the vv-list of the outer `\eval` command, in fact only the `x=5` entry has been transferred:

```
\eval[p=.]{$
\recur*[do=3]
```

```

{ L_{n+1}(x)=\frac{2n+1-x}{n+1}L_n(x)-
  \frac{n}{n+1}L_{n-1}(x)}
[L_{1}(x)=1-x,L_{0}(x)=1]
\times
\recur*[do=4]
{ L_{n+1}(x)=\frac{2n+1-x}{n+1}L_n(x)-
  \frac{n}{n+1}L_{n-1}(x)}
[L_{1}(x)=1-x,L_{0}(x)=1]
$}[x=5]

```

$$\implies 3.5 \times 2.666667 = 9.333334.$$

The terms of a recurrence relation are multi-token variables but `numerica` requires single tokens for its calculations. The problem for `\recur` is that the terms in the recurrence relation in the main (mandatory) argument differ from the terms in the vv-list: for instance f_n in the main argument, f_0 in the vv-list. If left like that, when `numerica` does its conversion from multi-token to single token variables, f_n would not be found since it differs from f_0 . Hence a crucial first step for `\recur` is to reconcile the different forms, which it does by converting the forms in the vv-list to the forms in the recurrence in the main argument. To be available for this form change, they must reside in the *inner* vv-list. In the outer vv-list they would be inaccessible to the inner command.

This suggests an alternative way of proceeding: write the initial values of the recurrence terms in the *same* form in which they occur in the recurrence relation, together with an initial value for the recurrence variable, e.g., $f_{n+1} = 1, f_n = 1, n = 0$. This is not how mathematicians write the initial values in recurrence relations, which is why I did not pursue it, but it neatly sidesteps what is otherwise an initial awkwardness.

Chapter 5

Reference summary

5.1 Commands defined in `numerica-plus`

1. `\nmcIterate`, `\iter`
2. `\nmcSolve`, `\solve`
3. `\nmcRecur`, `\recur`

5.2 Settings for the three commands

5.2.1 Settings for `\nmcIterate`

Settings option of `\nmcIterate`:

key	type	meaning	default
<code>var</code>	token(s)	iteration variable	
<code>+</code>	int	fixed point extra rounding	0
<code>max</code>	int > 0	max. iteration count (fixed points)	100
<code>do</code>	int > 0	number of iterations to perform	5
<code>see</code>	int > 0	number of final iterations to view	4
<code>reuse</code>	int (0/1/2)	form of result saved with <code>\reuse</code>	0

Configuration settings for `\nmcIterate`:

key	default
<code>iter-extra-rounding</code>	0
<code>iter-max-iterations</code>	100
<code>iter-do</code>	5
<code>iter-see-last</code>	4
<code>iter-reuse</code>	0

5.2.2 Settings for `\nmcSolve`

Settings option of `\nmcSolve`:

key	type	meaning	default
<code>var</code>	token(s)	equation variable	
<code>dvar</code>	real $\neq 0$	initial step size	1
<code>+</code>	int	extra rounding	0
<code>max</code>	int > 0	max. number of steps before cut off	100
<code>reuse</code>	int (0/1)	form of result saved with <code>\reuse</code>	0

Configuration settings for `\nmcSolve`:

key	default
<code>solve-first-step</code>	1
<code>solve-extra-rounding</code>	0
<code>solve-max-steps</code>	100
<code>solve-reuse</code>	0

5.2.3 Settings for `\nmcRecur`

Settings option of `\nmcRecur`:

key	type	meaning	default
<code>do</code>	int ≥ 0	number of terms to calculate	7
<code>see1</code>	int ≥ 0	number of initial terms to display	3
<code>see2</code>	int ≥ 0	number of final terms to display	2
<code>...</code>	chars	follow display of values with an ellipsis	
<code>reuse</code>	int (0/1/2)	form of result saved with <code>\reuse</code>	0

Configuration settings for `\nmcRecur`:

key	default
<code>recur-do</code>	7
<code>recur-see-first</code>	3
<code>recur-see-last</code>	2
<code>recur-reuse</code>	0