I wrote the first version of this reference during Spring 2007, as I learned R while teaching my Modeling & Simulation course at the University of Maine. The course covers population and epidemiological modeling, including deterministic and stochastic models in discrete and continuous time, along with spatial models; most of the exercises consist of writing various computer simulations. When I taught earlier versions of the course, it was based on MATLAB only. In Spring 2007, some biology graduate students in the class who had learned R in statistics courses asked if they could use R in my class as well, and I said yes. My colleague Bill Halteman was a great help as I frantically learned R to stay ahead of the class. As I went, every time I learned how to do something in R for the course, I added it to this reference, so that I wouldn’t forget it later. Some items took a huge amount of time searching for a simple way to do what I wanted, but I was pleasantly surprised that pretty much everything I do in MATLAB has an equivalent in R. I was also inspired to write this after seeing the “R for Octave Users” reference written by Robin Hankin. I’ve continued to add to the document, with many additions based on topics that came up while teaching courses on Advanced Linear Algebra and Numerical Analysis.

This reference is organized into general categories. There is also a MATLAB index and an R index at the end, which should make it easy to look up a command you know in one of the languages and learn how to do it in the other (or if you’re trying to read code in whichever language is unfamiliar to you, allow you to translate back to the one you are more familiar with). The index entries refer to the item numbers in the first column of the reference document, rather than page numbers.

Any corrections, suggested improvements, or even just notification that the reference has been useful are appreciated. I hope all the time I spent on this will prove useful for others in addition to myself and my students. Note that sometimes I don’t necessarily do things in what you may consider the “best” way in a particular language. I often tried to do things in a similar way in both languages, and where possible I’ve avoided the use of MATLAB toolboxes or R packages which are not part of the core distributions. But if you believe you have a “better” way (either simpler, or more computationally efficient) to do something, feel free to let me know.


Permission is granted to make and distribute verbatim copies of this manual provided this permission notice is preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Copyright ©2007–2011 David Hiebeler
## Contents

1 Help 3

2 Entering/building/indexing matrices 3  
   2.1 Cell arrays and lists 6  
   2.2 Structs and data frames 6

3 Computations 6  
   3.1 Basic computations 6  
   3.2 Complex numbers 7  
   3.3 Matrix/vector computations 8  
   3.4 Root-finding 14  
   3.5 Function optimization/minimization 14  
   3.6 Numerical integration / quadrature 15  
   3.7 Curve fitting 16

4 Conditionals, control structure, loops 17

5 Functions, ODEs 21

6 Probability and random values 23

7 Graphics 27  
   7.1 Various types of plotting 27  
   7.2 Printing/saving graphics 35  
   7.3 Animating cellular automata / lattice simulations 36

8 Working with files 37

9 Miscellaneous 38  
   9.1 Variables 38  
   9.2 Strings and Misc. 39

10 Spatial Modeling 43

Index of MATLAB commands and concepts 44

Index of R commands and concepts 49
## 1 Help

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Show help for a function (e.g. <code>sqrt</code>)</td>
<td><code>help sqrt</code>, or <code>helpwin sqrt</code> to see it in a separate window</td>
<td><code>help(sqrt)</code> or <code>?sqrt</code></td>
</tr>
<tr>
<td>2</td>
<td>Show help for a built-in keyword (e.g. <code>for</code>)</td>
<td><code>help for</code></td>
<td><code>help('for')</code> or <code>?for</code></td>
</tr>
<tr>
<td>3</td>
<td>General list of many help topics</td>
<td><code>help</code></td>
<td><code>library()</code> to see available libraries, or <code>library(help='base')</code> for very long list of stuff in base package which you can see help for</td>
</tr>
<tr>
<td>4</td>
<td>Explore main documentation in browser</td>
<td><code>doc</code> or <code>helpbrowser</code> (previously it was <code>helpdesk</code>, which is now being phased out)</td>
<td><code>help.start()</code></td>
</tr>
<tr>
<td>5</td>
<td>Search documentation for keyword or partial keyword (e.g. functions which refer to “binomial”)</td>
<td><code>lookfor binomial</code></td>
<td><code>help.search('binomial')</code></td>
</tr>
</tbody>
</table>

## 2 Entering/building/indexing matrices

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Enter a row vector $\vec{v} = \begin{bmatrix} 1 &amp; 2 &amp; 3 &amp; 4 \end{bmatrix}$</td>
<td><code>v=[1 2 3 4]</code></td>
<td>$v=c(1,2,3,4)$ or alternatively $v=\text{scan()}$ then enter “1 2 3 4” and press Enter twice (the blank line terminates input)</td>
</tr>
<tr>
<td>7</td>
<td>Enter a column vector $\begin{bmatrix} 1 \ 2 \ 3 \ 4 \end{bmatrix}$</td>
<td><code>[1; 2; 3; 4]</code></td>
<td><code>c(1,2,3,4)</code></td>
</tr>
<tr>
<td></td>
<td>(R does not distinguish between row and column vectors.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Enter a matrix $\begin{bmatrix} 1 &amp; 2 &amp; 3 \ 4 &amp; 5 &amp; 6 \end{bmatrix}$</td>
<td><code>[1 2 3 ; 4 5 6]</code></td>
<td>To enter values by row: <code>matrix(c(1,2,3,4,5,6), nrow=2, byrow=TRUE)</code>. To enter values by column: <code>matrix(c(1,4,2,5,3,6), nrow=2)</code></td>
</tr>
<tr>
<td>9</td>
<td>Access an element of vector v</td>
<td><code>v(3)</code></td>
<td><code>v[3]</code></td>
</tr>
<tr>
<td>10</td>
<td>Access an element of matrix A</td>
<td><code>A(2,3)</code></td>
<td><code>A[2,3]</code></td>
</tr>
<tr>
<td>11</td>
<td>Access an element of matrix A using a single index: indices count down the first column, then down the second column, etc.</td>
<td><code>A(5)</code></td>
<td><code>A[5]</code></td>
</tr>
<tr>
<td>12</td>
<td>Build the vector $[2 3 4 5 6 7]$</td>
<td><code>2:7</code></td>
<td><code>2:7</code></td>
</tr>
<tr>
<td>13</td>
<td>Build the vector $[7 6 5 4 3 2]$</td>
<td><code>7:-1:2</code></td>
<td><code>7:2</code></td>
</tr>
<tr>
<td>14</td>
<td>Build the vector $[2 5 8 11 14]$</td>
<td><code>2:3:14</code></td>
<td><code>seq(2,14,3)</code></td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>---</td>
</tr>
<tr>
<td>15</td>
<td>Build a vector containing ( n ) equally-spaced values between ( a ) and ( b ) inclusive</td>
<td><code>linspace(a,b,n)</code></td>
<td><code>seq(a,b,length.out=n)</code> or just <code>seq(a,b,len=n)</code></td>
</tr>
<tr>
<td>16</td>
<td>Build a vector containing ( n ) logarithmically equally-spaced values between ( 10^a ) and ( 10^b ) inclusive</td>
<td><code>logspace(a,b,n)</code></td>
<td><code>10^seq(a,b,len=n)</code></td>
</tr>
<tr>
<td>17</td>
<td>Build a vector of length ( k ) containing all zeros</td>
<td><code>zeros(k,1)</code> for a column vector or <code>zeros(1,k)</code> for a row vector</td>
<td><code>rep(0,k)</code></td>
</tr>
<tr>
<td>18</td>
<td>Build a vector of length ( k ) containing the value ( j ) in all positions</td>
<td><code>j*ones(k,1)</code> for a column vector or <code>j*ones(1,k)</code> for a row vector</td>
<td><code>rep(j,k)</code></td>
</tr>
<tr>
<td>19</td>
<td>Build an ( m \times n ) matrix of zeros</td>
<td><code>zeros(m,n)</code></td>
<td><code>matrix(0,nrow=m,ncol=n)</code> or just <code>matrix(0,m,n)</code></td>
</tr>
<tr>
<td>20</td>
<td>Build an ( m \times n ) matrix containing ( j ) in all positions</td>
<td><code>j*ones(m,n)</code></td>
<td><code>matrix(j,nrow=m,ncol=n)</code> or just <code>matrix(j,m,n)</code></td>
</tr>
<tr>
<td>21</td>
<td>( n \times n ) identity matrix ( I_n )</td>
<td><code>eye(n)</code></td>
<td><code>diag(n)</code></td>
</tr>
<tr>
<td>22</td>
<td>Build diagonal matrix ( A ) using elements of vector ( v ) as diagonal entries</td>
<td><code>diag(v)</code></td>
<td><code>diag(v,nrow=length(v))</code> (Note: if you are sure the length of vector ( v ) is 2 or more, you can simply say <code>diag(v)</code>.)</td>
</tr>
<tr>
<td>23</td>
<td>Extract diagonal elements of matrix ( A )</td>
<td><code>v=diag(A)</code></td>
<td><code>v=diag(A)</code></td>
</tr>
<tr>
<td>24</td>
<td>“Glue” two matrices ( a1 ) and ( a2 ) (with the same number of rows) side-by-side</td>
<td><code>[a1 a2]</code></td>
<td><code>cbind(a1,a2)</code></td>
</tr>
<tr>
<td>25</td>
<td>“Stack” two matrices ( a1 ) and ( a2 ) (with the same number of columns) on top of each other</td>
<td><code>[a1; a2]</code></td>
<td><code>rbind(a1,a2)</code></td>
</tr>
<tr>
<td>26</td>
<td>Given vectors ( x ) and ( y ) of lengths ( m ) and ( n ) respectively, build ( n \times m ) matrices ( X ) whose rows are copies of ( x ) and ( Y ) whose columns are copies of ( y )</td>
<td><code>[X,Y]=meshgrid(x,y)</code></td>
<td><code>m=length(x); n=length(y); X=matrix(rep(x,each=n),nrow=n); Y=matrix(rep(y,m),nrow=n)</code></td>
</tr>
<tr>
<td>27</td>
<td>Reverse the order of elements in vector ( v )</td>
<td><code>v(end:-1:1)</code></td>
<td><code>rev(v)</code></td>
</tr>
<tr>
<td>28</td>
<td>Column 2 of matrix ( A )</td>
<td><code>A(:,2)</code></td>
<td><code>A[,2]</code> Note: that gives the result as a vector. To make the result a ( m \times 1 ) matrix instead, do <code>A[,2,drop=FALSE]</code></td>
</tr>
<tr>
<td>29</td>
<td>Row 7 of matrix ( A )</td>
<td><code>A(7,:)</code></td>
<td><code>A[7,]</code> Note: that gives the result as a vector. To make the result a 1 ( \times n ) matrix instead, do <code>A[7,,drop=FALSE]</code></td>
</tr>
<tr>
<td>30</td>
<td>All elements of ( A ) as a vector, column-by-column</td>
<td><code>A(:)</code> (gives a column vector)</td>
<td><code>c(A)</code></td>
</tr>
<tr>
<td>31</td>
<td>Rows 2–4, columns 6–10 of ( A ) (this is a ( 3 \times 5 ) matrix)</td>
<td><code>A(2:4,6:10)</code></td>
<td><code>A[2:4,6:10]</code></td>
</tr>
<tr>
<td>32</td>
<td>A ( 3 \times 2 ) matrix consisting of rows 7, 7, and 6 and columns 2 and 1 of ( A ) (in that order)</td>
<td><code>[7 7 6], [2 1]</code></td>
<td><code>A[(1:m-s1-1)%m+1, (1:n-s2-1)%n+1]</code></td>
</tr>
<tr>
<td>33</td>
<td>Circularly shift the rows of matrix ( A ) down by ( s_1 ) elements, and right by ( s_2 ) elements</td>
<td><code>circshift(A, [s1 s2])</code></td>
<td>No simple way, but modulo arithmetic on indices will work: <code>m=dim(A)[1]; n=dim(A)[2]; A[(1:m-s1-1)%m+1, (1:n-s2-1)%n+1]</code></td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>---------</td>
<td>---</td>
</tr>
<tr>
<td>34</td>
<td>Flip the order of elements in each row of matrix A</td>
<td><code>fliplr(A)</code></td>
<td><code>t(apply(A,1,rev))</code> or <code>A[,ncol(A):1]</code></td>
</tr>
<tr>
<td>35</td>
<td>Flip the order of elements in each column of matrix A</td>
<td><code>flipud(A)</code></td>
<td><code>apply(A,2,rev)</code> or <code>A[nrow(A):1,]</code></td>
</tr>
</tbody>
</table>
| 36  | Given a single index `ind` into an $m \times n$ matrix A, compute the row r and column c of that position (also works if `ind` is a vector) | $[r,c] = \text{ind2sub}(\text{size}(A), \text{ind})$ | $r = ((\text{ind}-1) \mod m) + 1$
$c = \text{floor}((\text{ind}-1) / m) + 1$
| 37  | Given the row r and column c of an element of an $m \times n$ matrix A, compute the single index `ind` which can be used to access that element of A (also works if r and c are vectors) | `ind = \text{sub2ind}(\text{size}(A), r, c)` | $\text{ind} = (c-1) \times m + r$ |
| 38  | Given equal-sized vectors r and c (each of length k), set elements in rows (given by r) and columns (given by c) of matrix A equal to 12. That is, k elements of A will be modified. | `inds = \text{sub2ind}(\text{size}(A), r, c)`; $A(\text{inds}) = 12$; | `inds = \text{cbind}(r, c)`
$A[\text{inds}] = 12$ |
| 39  | Truncate vector v, keeping only the first 10 elements | `v = v(1:10)` | `v = v[1:10]`, or `\text{length}(v) = 10` also works |
| 40  | Extract elements of vector v from position a to the end | `v(a:end)` | `v[a:length(v)]` |
| 41  | All but the $k^{th}$ element of vector v | `v([1:(k-1) \ (k+1):\text{end}])` or `v([k]) = []` (but this will modify the original vector v) | `v[-k]` |
| 42  | All but the $j^{th}$ and $k^{th}$ elements of vector v | `v(~\text{ismember}(1:\text{length}(v),[j k]))` or `v([j k]) = []` (but this will modify the original vector v) | `v[c(-j,-k)]` |
| 43  | Reshape matrix A, making it an $m \times n$ matrix with elements taken columnwise from the original A (which must have $mn$ elements) | $A = \text{reshape}(A,m,n)$ | `\text{dim}(A) = c(m,n)` |
| 44  | Extract the lower-triangular portion of matrix A | $L = \text{tril}(A)$ | $L = A; L[\text{upper.tri}(L)] = 0$ |
| 45  | Extract the upper-triangular portion of matrix A | $U = \text{triu}(A)$ | $U = A; U[\text{lower.tri}(U)] = 0$ |
| 46  | Enter $n \times n$ Hilbert matrix H where $H_{ij} = 1/(i + j - 1)$ | `hilb(n)` | `\text{Hilbert}(n)`, but this is part of the Matrix package which you’ll need to install (see item 338 for how to install/load packages). |
| 47  | Enter an $n$-dimensional array, e.g. a $3 \times 4 \times 2$ array with the values 1 through 24 | `\text{reshape}(1:24, 3, 4, 2)` or `\text{reshape}(1:24, [3 4 2])` | `\text{array}(1:24, c(3,4,2))` (Note that a matrix is 2-D, i.e. rows and columns, while an array is more generally N-D) |
### 2.1 Cell arrays and lists

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>Build a vector ( v ) of length ( n ), capable of containing different data types in different elements (called a <strong>cell array</strong> in MATLAB, and a <strong>list</strong> in R)</td>
<td>( v = \text{cell}(1,n) ) In general, ( \text{cell}(m,n) ) makes an ( m \times n ) cell array. Then you can do e.g.: ( v(1) = 12 ) ( v(2) = 'hi there' ) ( v(3) = \text{rand}(3) )</td>
<td>( v = \text{vector}('list',n) ) Then you can do e.g.: ( v[[1]] = 12 ) ( v[[2]] = 'hi there' ) ( v[[3]] = \text{matrix}(	ext{runif}(9),3) )</td>
</tr>
<tr>
<td>49</td>
<td>Extract the ( i )th element of a cell/list vector ( v )</td>
<td>( w = v{i} ) If you use regular indexing, i.e. ( w = v(i) ), then ( w ) will be a ( 1 \times 1 ) cell matrix containing the contents of the ( i )th element of ( v ).</td>
<td>( w = v[[i]] ) If you use regular indexing, i.e. ( w = v[i] ), then ( w ) will be a list of length 1 containing the contents of the ( i )th element of ( v ).</td>
</tr>
<tr>
<td>50</td>
<td>Set the name of the ( i )th element in a list.</td>
<td>(MATLAB does not have names associated with elements of cell arrays.)</td>
<td>Use ( \text{names}(v) ) to see all names, and ( \text{names}(v)=\text{NULL} ) to clear all names.</td>
</tr>
</tbody>
</table>

### 2.2 Structs and data frames

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>Create a matrix-like object with different named columns (a <strong>struct</strong> in MATLAB, or a <strong>data frame</strong> in R)</td>
<td>( \text{avals}=2*\text{ones}(1,6); ) ( \text{yvals}=6:-1:1; ) ( v=[1 5 3 2 3 7]; ) ( d=\text{struct}('a',\text{avals}, 'yy', \text{yvals}, 'fac', v); )</td>
<td>( v=c(1,5,3,2,3,7); d=\text{data.frame(} ) ( \text{cbind}(a=2, yy=6:1), v) )</td>
</tr>
</tbody>
</table>

Note that I (surprisingly) don’t use R for statistics, and therefore have very little experience with data frames (and also very little with MATLAB structs). I will try to add more to this section later on.

### 3 Computations

#### 3.1 Basic computations

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>( a + b, a - b, ab, a/b )</td>
<td>( a+b, a-b, a\cdot b, a/b )</td>
<td>( a+b, a-b, a\cdot b, a/b )</td>
</tr>
<tr>
<td>53</td>
<td>( \sqrt{a} )</td>
<td>( \text{sqrt}(a) )</td>
<td>( \text{sqrt}(a) )</td>
</tr>
<tr>
<td>54</td>
<td>( a^b )</td>
<td>( a^b )</td>
<td>( a^b )</td>
</tr>
<tr>
<td>55</td>
<td>(</td>
<td>a</td>
<td>) (note: for complex arguments, this computes the modulus)</td>
</tr>
<tr>
<td>56</td>
<td>( e^a )</td>
<td>( \text{exp}(a) )</td>
<td>( \text{exp}(a) )</td>
</tr>
<tr>
<td>57</td>
<td>( \ln(a) )</td>
<td>( \log(a) )</td>
<td>( \log(a) )</td>
</tr>
<tr>
<td>58</td>
<td>( \log_2(a), \log_{10}(a) )</td>
<td>( \log(2)(a), \log(10)(a) )</td>
<td>( \log(2)(a), \log(10)(a) )</td>
</tr>
<tr>
<td>59</td>
<td>( \sin(a), \cos(a), \tan(a) )</td>
<td>( \sin(a), \cos(a), \tan(a) )</td>
<td>( \sin(a), \cos(a), \tan(a) )</td>
</tr>
<tr>
<td>60</td>
<td>( \sin^{-1}(a), \cos^{-1}(a), \tan^{-1}(a) )</td>
<td>( \text{asin}(a), \text{acos}(a), \text{atan}(a) )</td>
<td>( \text{asin}(a), \text{acos}(a), \text{atan}(a) )</td>
</tr>
<tr>
<td>61</td>
<td>( \sinh(a), \cosh(a), \tanh(a) )</td>
<td>( \sinh(a), \cosh(a), \tanh(a) )</td>
<td>( \sinh(a), \cosh(a), \tanh(a) )</td>
</tr>
<tr>
<td>62</td>
<td>( \sinh^{-1}(a), \cosh^{-1}(a), \tanh^{-1}(a) )</td>
<td>( \text{asinh}(a), \text{acosh}(a), \text{atanh}(a) )</td>
<td>( \text{asinh}(a), \text{acosh}(a), \text{atanh}(a) )</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------------------------</td>
<td>------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>63</td>
<td>$n \text{ MOD } k$ (modulo arithmetic)</td>
<td>mod(n,k)</td>
<td>n %% k</td>
</tr>
<tr>
<td>64</td>
<td>Round to nearest integer</td>
<td>round(x)</td>
<td>round(x) (Note: R uses IEC 60559 standard, rounding 5 to the even digit — so e.g. round(0.5) gives 0, not 1.)</td>
</tr>
<tr>
<td>65</td>
<td>Round down to next lowest integer</td>
<td>floor(x)</td>
<td>floor(x)</td>
</tr>
<tr>
<td>66</td>
<td>Round up to next largest integer</td>
<td>ceil(x)</td>
<td>ceiling(x)</td>
</tr>
<tr>
<td>67</td>
<td>Sign of $x$ (+1, 0, or -1)</td>
<td>sign(x)</td>
<td>sign(x) (Does not work with complex values)</td>
</tr>
<tr>
<td>68</td>
<td>Error function $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$</td>
<td>erf(x)</td>
<td>2<em>pnorm(x</em>sqrt(2))-1</td>
</tr>
<tr>
<td>69</td>
<td>Complementary error function $\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt = 1-\text{erf}(x)$</td>
<td>erfc(x)</td>
<td>2<em>pnorm(x</em>sqrt(2),lower=FALSE)</td>
</tr>
<tr>
<td>70</td>
<td>Inverse error function $\text{erfinv}(x)$</td>
<td>erfinv(x)</td>
<td>qnorm((1+x)/2)/sqrt(2)</td>
</tr>
<tr>
<td>71</td>
<td>Inverse complementary error function $\text{erfcinv}(x)$</td>
<td>erfcinv(x)</td>
<td>qnorm(x/2,lower=FALSE)/sqrt(2)</td>
</tr>
<tr>
<td>72</td>
<td>Binomial coefficient $(\begin{pmatrix} n \ k \end{pmatrix}) = \frac{n!}{k!(n-k)!}$</td>
<td>nchoosek(n,k)</td>
<td>choose(n,k)</td>
</tr>
</tbody>
</table>

Note: the various functions above (logarithm, exponential, trig, abs, and rounding functions) all work with vectors and matrices, applying the function to each element, as well as with scalars.

### 3.2 Complex numbers

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>Enter a complex number</td>
<td>1+2i</td>
<td>1+2i</td>
</tr>
<tr>
<td>74</td>
<td>Modulus (magnitude)</td>
<td>abs(z)</td>
<td>abs(z) or Mod(z)</td>
</tr>
<tr>
<td>75</td>
<td>Argument (angle)</td>
<td>angle(z)</td>
<td>Arg(z)</td>
</tr>
<tr>
<td>76</td>
<td>Complex conjugate</td>
<td>conj(z)</td>
<td>Conj(z)</td>
</tr>
<tr>
<td>77</td>
<td>Real part of $z$</td>
<td>real(z)</td>
<td>Re(z)</td>
</tr>
<tr>
<td>78</td>
<td>Imaginary part of $z$</td>
<td>imag(z)</td>
<td>Im(z)</td>
</tr>
</tbody>
</table>
### 3.3 Matrix/vector computations

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>Vector dot product $\vec{x} \cdot \vec{y} = \vec{x}^T \vec{y}$</td>
<td><code>dot(x,y)</code></td>
<td><code>sum(x*y)</code></td>
</tr>
<tr>
<td>80</td>
<td>Vector cross product $\vec{x} \times \vec{y}$</td>
<td><code>cross(x,y)</code></td>
<td>Not in base R, but e.g. the <code>xprod</code> function from the RSEIS package will do it (see item 338 for how to install/load packages)</td>
</tr>
<tr>
<td>81</td>
<td>Matrix multiplication $AB$</td>
<td><code>A * B</code></td>
<td><code>-</code></td>
</tr>
<tr>
<td>82</td>
<td>Element-by-element multiplication of $A$ and $B$</td>
<td><code>A .* B</code></td>
<td><code>A * B</code></td>
</tr>
<tr>
<td>83</td>
<td>Transpose of a matrix, $A^T$</td>
<td><code>t(A)</code> (This is actually the complex conjugate (i.e. Hermitian) transpose; use <code>A.'</code> for the non-conjugate transpose if you like; they are equivalent for real matrices.)</td>
<td><code>t(A)</code> for transpose, or <code>Conj(t(A))</code> for conjugate (Hermitian) transpose</td>
</tr>
<tr>
<td>84</td>
<td>Solve $A\vec{x} = \vec{b}$</td>
<td><code>A\b</code> Warning: if there is no solution, MATLAB gives you a least-squares “best fit.” If there are many solutions, MATLAB just gives you one of them.</td>
<td><code>solve(A,b)</code> Warning: this only works with square invertible matrices.</td>
</tr>
<tr>
<td>85</td>
<td>Reduced echelon form of $A$</td>
<td><code>rref(A)</code></td>
<td>R does not have a function to do this</td>
</tr>
<tr>
<td>86</td>
<td>Determinant of $A$</td>
<td><code>det(A)</code></td>
<td><code>det(A)</code></td>
</tr>
<tr>
<td>87</td>
<td>Inverse of $A$</td>
<td><code>inv(A)</code></td>
<td><code>solve(A)</code></td>
</tr>
<tr>
<td>88</td>
<td>Trace of $A$</td>
<td><code>trace(A)</code></td>
<td><code>sum(diag(A))</code></td>
</tr>
<tr>
<td>89</td>
<td>$AB^{-1}$</td>
<td><code>A/B</code></td>
<td><code>A %*% solve(B)</code></td>
</tr>
<tr>
<td>90</td>
<td>Element-by-element division of $A$ and $B$</td>
<td><code>A ./ B</code></td>
<td><code>A / B</code></td>
</tr>
<tr>
<td>91</td>
<td>$A^{-1}B$</td>
<td><code>A\B</code></td>
<td><code>solve(A,B)</code></td>
</tr>
<tr>
<td>92</td>
<td>Square the matrix $A$</td>
<td><code>A^2</code></td>
<td><code>A %*% A</code></td>
</tr>
<tr>
<td>93</td>
<td>Raise matrix $A$ to the $k^{th}$ power</td>
<td><code>A^k</code> (No easy way to do this in R other than repeated multiplication $A %<em>% A %</em>% A ...$)</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>Raise each element of $A$ to the $k^{th}$ power</td>
<td><code>A.^k</code></td>
<td><code>A^k</code></td>
</tr>
<tr>
<td>95</td>
<td>Rank of matrix $A$</td>
<td><code>rank(A)</code></td>
<td><code>qr(A)$rank</code></td>
</tr>
<tr>
<td>96</td>
<td>Set $w$ to be a vector of eigenvalues of $A$, and $V$ a matrix containing the corresponding eigenvectors</td>
<td><code>[V,D]=eig(A)</code> and then <code>w=diag(D)</code> since MATLAB returns the eigenvalues on the diagonal of $D$</td>
<td><code>tmp=eigen(A); w=tmp$values; V=tmp$vectors</code></td>
</tr>
<tr>
<td>97</td>
<td>Permed $LU$ factorization of a matrix</td>
<td><code>[L,U,P]=lu(A)</code> then the matrices satisfy $PA = LU$. Note that this works even with non-square matrices</td>
<td><code>tmp=expand(lu(Matrix(A))); L=tmp$L; U=tmp$U; P=tmp$P</code> then the matrices satisfy $A = PLU$, i.e. $P^{-1}A = LU$. Note that the <code>lu</code> and <code>expand</code> functions are part of the Matrix package (see item 338 for how to install/load packages). Also note that this doesn’t seem to work correctly with non-square matrices. $L$, $U$, and $P$ will be of class Matrix rather than class matrix; to make them the latter, instead do $L=as.matrix(tmp$L), U=as.matrix(tmp$U), and $P=as.matrix(tmp$P) above.</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>----</td>
</tr>
<tr>
<td>98</td>
<td>Singular-value decomposition: given ( m \times n ) matrix ( A ) with rank ( r ), find ( m \times r ) matrix ( P ) with orthonormal columns, diagonal ( r \times r ) matrix ( S ), and ( r \times n ) matrix ( Q^T ) with orthonormal rows so that ( PSA^T = A )</td>
<td>([P,S,Q]=svd(A, \text{ 'econ' }))</td>
<td>tmp=svd(A); U=tmp$u; V=tmp$v; S=diag(tmp$d)</td>
</tr>
<tr>
<td>99</td>
<td>Schur decomposition of square matrix, ( A = QTQ^H = QTQ^{-1} ) where ( Q ) is unitary (i.e. ( Q^H Q = I )) and ( T ) is upper triangular; ( Q^H = QT ) is the Hermitian (conjugate) transpose</td>
<td>([Q,T]=schur(A))</td>
<td>tmp=Schur(Matrix(A)); T=tmp@T; Q=tmp@Q Note that Schur is part of ( ) the Matrix package (see item 338 for how to install/load ( ) packages). ( T ) and ( Q ) will be of class Matrix rather than ( ) class matrix; to make them the latter, instead do ( T=as.matrix(tmp@T) ) and ( Q=as.matrix(tmp@Q) ) above.</td>
</tr>
<tr>
<td>100</td>
<td>Cholesky factorization of a square, symmetric, positive definite matrix ( A = R^T R ), where ( R ) is upper-triangular</td>
<td>( R = \text{chol}(A) )</td>
<td>( R = \text{chol}(A) ) Note that chol is part of ( ) the Matrix package (see item 338 for how to install/load packages).</td>
</tr>
<tr>
<td>101</td>
<td>( QR ) factorization of matrix ( A ), where ( Q ) is orthogonal (satisfying ( QQ^T = I )) and ( R ) is upper-triangular</td>
<td>([Q,R]=[qr(A) \text{ satisfying } QR = A, \text{ or } [Q,R,E]=[qr(A) \text{ to do permuted } QR \text{ factorization satisfying } AE = QR] )</td>
<td>( z=qr(A); Q=qr.Q(z); R=qr.R(z); E=diag(n)[,z$pivot] ) (where ( n ) is the number of columns in ( A )) gives ( ) permuted ( QR ) factorization satisfying ( AE = QR )</td>
</tr>
<tr>
<td>102</td>
<td>Vector norms</td>
<td>( \text{norm(v,1)} ) for 1-norm ( |v|_1 ), ( \text{norm(v,2)} ) for Euclidean norm ( |v|<em>2 ), ( \text{norm(v,inf)} ) for infinity-norm ( |v|</em>\infty ), and ( \text{norm(v,p)} ) for ( p )-norm ( |v|_p = (\sum</td>
<td>v_i</td>
</tr>
<tr>
<td>103</td>
<td>Matrix norms</td>
<td>( \text{norm(A,1)} ) for 1-norm ( |A|<em>1 ), ( \text{norm(A,2)} ) for 2-norm ( |A|<em>2 ), ( \text{norm(A,inf)} ) for infinity-norm ( |A|</em>\infty ), and ( \text{norm(A,'fro')} ) for Frobenius norm ( (\sum_i (A^T A)</em>{ii})^{1/2} )</td>
<td>( \text{norm(A,'1')} ) for 1-norm ( |A|<em>1 ), ( \text{max(svd(A)$d)} ) for 2-norm ( |A|<em>2 ), ( \text{norm(A,'1')} ) for infinity-norm ( |A|</em>\infty ), and ( \text{norm(A,'fro')} ) for Frobenius norm ( (\sum_i (A^T A)</em>{ii})^{1/2} )</td>
</tr>
<tr>
<td>104</td>
<td>Condition number ( \text{cond}(A) = |A|_1 |A^{-1}|_1 ) of ( A ), using 1-norm</td>
<td>( \text{cond}(A,1) ) (Note: MATLAB also has a function ( \text{rcond}(A) ) which computes reciprocal condition estimator using the 1-norm)</td>
<td>( 1/r\text{cond}(A,'\text{1'}) )</td>
</tr>
<tr>
<td>105</td>
<td>Condition number ( \text{cond}(A) = |A|_2 |A^{-1}|_2 ) of ( A ), using 2-norm</td>
<td>( \text{cond}(A,2) )</td>
<td>( \text{kappa(A, exact=TRUE)} ) (leave out the “exact=TRUE” for an estimate)</td>
</tr>
<tr>
<td>106</td>
<td>Condition number ( \text{cond}(A) = |A|<em>\infty |A^{-1}|</em>\infty ) of ( A ), using infinity-norm</td>
<td>( \text{cond}(A,\text{inf}) )</td>
<td>( 1/r\text{cond}(A,'\text{T'}) )</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>---------</td>
<td>---</td>
</tr>
<tr>
<td>107</td>
<td>Mean of all elements in vector or matrix</td>
<td><code>mean(v)</code> for vectors, <code>mean(A(:))</code> for matrices</td>
<td><code>mean(v) or mean(A)</code></td>
</tr>
<tr>
<td>108</td>
<td>Means of columns of a matrix</td>
<td><code>mean(A)</code></td>
<td><code>colMeans(A)</code></td>
</tr>
<tr>
<td>109</td>
<td>Means of rows of a matrix</td>
<td><code>mean(A,2)</code></td>
<td><code>rowMeans(A)</code></td>
</tr>
<tr>
<td>110</td>
<td>Standard deviation of all elements in vector or matrix</td>
<td><code>std(v)</code> for vectors, <code>std(A(:))</code> for matrices. This normalizes by $n - 1$. Use <code>std(v,1)</code> to normalize by $n$.</td>
<td><code>sd(v)</code> for vectors, <code>sd(c(A))</code> for matrices. This normalizes by $n - 1$.</td>
</tr>
<tr>
<td>111</td>
<td>Standard deviations of columns of a matrix</td>
<td><code>std(A)</code></td>
<td><code>apply(A,1,sd)</code></td>
</tr>
<tr>
<td>112</td>
<td>Standard deviations of rows of a matrix</td>
<td><code>std(A,0,2)</code> to normalize by $n - 1$, <code>std(A,1,2)</code> to normalize by $n`</td>
<td><code>apply(A,1,var)</code></td>
</tr>
<tr>
<td>113</td>
<td>Variance of all elements in vector or matrix</td>
<td><code>var(v)</code> for vectors, <code>var(A(:))</code> for matrices. This normalizes by $n - 1$. Use <code>var(v,1)</code> to normalize by $n$.</td>
<td><code>var(v)</code> for vectors, <code>var(c(A))</code> for matrices. This normalizes by $n - 1$.</td>
</tr>
<tr>
<td>114</td>
<td>Variance of columns of a matrix</td>
<td><code>var(A)</code></td>
<td><code>apply(A,2,var)</code></td>
</tr>
<tr>
<td>115</td>
<td>Variance of rows of a matrix</td>
<td><code>var(A,0,2)</code> to normalize by $n - 1$, <code>var(A,1,2)</code> to normalize by $n`</td>
<td><code>apply(A,1,var)</code></td>
</tr>
<tr>
<td>116</td>
<td>Mode of values in vector $v$</td>
<td><code>mode(v)</code> (chooses smallest value in case of a tie), or <code>[m,f,c]=mode(v)</code>; <code>c{1}</code> (gives list of all tied values)</td>
<td><code>as.numeric(names(sort(-table(v)))[1])</code> (chooses smallest value in case of a tie), or <code>as.numeric(names(table(v))[table(v)==max(sort(table(v)))]</code> (gives list of all tied values)</td>
</tr>
<tr>
<td>117</td>
<td>Median of values in vector $v$</td>
<td><code>median(v)</code></td>
<td><code>median(v)</code></td>
</tr>
<tr>
<td>118</td>
<td>Covariance for two vectors of observations</td>
<td><code>cov(v,w)</code> computes the $2 \times 2$ covariance matrix; the off-diagonal elements give the desired covariance</td>
<td><code>cov(v,w)</code></td>
</tr>
<tr>
<td>119</td>
<td>Covariance matrix, giving covariances between columns of matrix $A$</td>
<td><code>cov(A)</code></td>
<td><code>var(A)</code> or <code>cov(A)</code></td>
</tr>
<tr>
<td>120</td>
<td>Given matrices $A$ and $B$, build covariance matrix $C$ where $c_{ij}$ is the covariance between column $i$ of $A$ and column $j$ of $B$</td>
<td>I don't know of a direct way to do this in Matlab. But one way is <code>[Y,X]=meshgrid(std(B),std(A)); X.*Y.*corr(A,B)</code></td>
<td><code>cov(A,B)</code></td>
</tr>
<tr>
<td>121</td>
<td>Pearson's linear correlation coefficient between elements of vectors $v$ and $w$</td>
<td><code>corr(v,w)</code> Note: $v$ and $w$ must be column vectors. Or <code>corr(v(:),w(:))</code> will work for both row and column vectors.</td>
<td><code>cor(v,w)</code></td>
</tr>
<tr>
<td>122</td>
<td>Kendall's tau correlation statistic for vectors $v$ and $w$</td>
<td><code>corr(v,w,'type','kendall')</code></td>
<td><code>cor(v,w,method='kendall')</code></td>
</tr>
<tr>
<td>123</td>
<td>Spearman's rho correlation statistic for vectors $v$ and $w$</td>
<td><code>corr(v,w,'type','spearman')</code></td>
<td><code>cor(v,w,method='spearman')</code></td>
</tr>
<tr>
<td>124</td>
<td>Pairwise Pearson's correlation coefficient between columns of matrix $A$</td>
<td><code>corr(A)</code> The 'type' argument may also be used as in the previous two items</td>
<td><code>cor(A)</code> The method argument may also be used as in the previous two items</td>
</tr>
<tr>
<td>125</td>
<td>Matrix $C$ of pairwise Pearson's correlation coefficients between each pair of columns of matrices $A$ and $B$, i.e. $c_{ij}$ is correlation between column $i$ of $A$ and column $j$ of $B$</td>
<td><code>corr(A,B)</code> The 'type' argument may also be used as just above</td>
<td><code>cor(A,B)</code> The method argument may also be used as just above</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>---------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>126</td>
<td>Sum of all elements in vector or matrix</td>
<td><code>sum(v)</code> for vectors, <code>sum(A(:))</code> for matrices</td>
<td><code>sum(v)</code> or <code>sum(A)</code></td>
</tr>
<tr>
<td>127</td>
<td>Sums of columns of matrix</td>
<td><code>sum(A)</code></td>
<td><code>colSums(A)</code></td>
</tr>
<tr>
<td>128</td>
<td>Sums of rows of matrix</td>
<td><code>sum(A,2)</code></td>
<td><code>rowSums(A)</code></td>
</tr>
<tr>
<td>129</td>
<td>Product of all elements in vector or matrix</td>
<td><code>prod(v)</code> for vectors, <code>prod(A(:))</code> for matrices</td>
<td><code>prod(v)</code> or <code>prod(A)</code></td>
</tr>
<tr>
<td>130</td>
<td>Products of columns of matrix</td>
<td><code>prod(A)</code></td>
<td><code>apply(A,2,prod)</code></td>
</tr>
<tr>
<td>131</td>
<td>Products of rows of matrix</td>
<td><code>prod(A,2)</code></td>
<td><code>apply(A,1,prod)</code></td>
</tr>
<tr>
<td>132</td>
<td>Matrix exponential $e^A = \sum_{k=0}^\infty A^k/k!$</td>
<td><code>expm(A)</code></td>
<td><code>expm(Matrix(A))</code>, but this is part of the Matrix package which you’ll need to install (see item 338 for how to install/load packages).</td>
</tr>
<tr>
<td>133</td>
<td>Cumulative sum of values in vector</td>
<td><code>cumsum(v)</code></td>
<td><code>cumsum(v)</code></td>
</tr>
<tr>
<td>134</td>
<td>Cumulative sums of columns of matrix</td>
<td><code>cumsum(A)</code></td>
<td><code>apply(A,2,cumsum)</code></td>
</tr>
<tr>
<td>135</td>
<td>Cumulative sums of rows of matrix</td>
<td><code>cumsum(A,2)</code></td>
<td><code>t(apply(A,1,cumsum))</code></td>
</tr>
<tr>
<td>136</td>
<td>Cumulative sum of all elements of matrix (column-by-column)</td>
<td><code>cumsum(A(:))</code></td>
<td><code>cumsum(A)</code></td>
</tr>
<tr>
<td>137</td>
<td>Cumulative product of elements in vector v</td>
<td><code>cumprod(v)</code> (Can also be used in the various ways <code>cumsum</code> can)</td>
<td><code>cumprod(v)</code> (Can also be used in the various ways <code>cumsum</code> can)</td>
</tr>
<tr>
<td>138</td>
<td>Cumulative minimum or maximum of elements in vector v</td>
<td>I don’t know of an easy way to do this in MATLAB</td>
<td><code>cummin(v)</code> or <code>cummax(v)</code></td>
</tr>
<tr>
<td>139</td>
<td>Differences between consecutive elements of vector v. Result is a vector w 1 element shorter than v, where element i of w is element i + 1 of v minus element i of v</td>
<td><code>diff(v)</code></td>
<td><code>diff(v)</code></td>
</tr>
<tr>
<td>140</td>
<td>Make a vector y the same size as vector x, which equals 4 everywhere that x is greater than 5, and equals 3 everywhere else (done via a vectorized computation).</td>
<td><code>z = [3 4]; y = z((x &gt; 5)+1)</code> Or this will also work: <code>y=3*ones(size(x)); y(x&gt;5)=4</code></td>
<td><code>y = ifelse(x &gt; 5, 4, 3)</code></td>
</tr>
<tr>
<td>141</td>
<td>Minimum of values in vector v</td>
<td><code>min(v)</code></td>
<td><code>min(v)</code></td>
</tr>
<tr>
<td>142</td>
<td>Minimum of all values in matrix A</td>
<td><code>min(A(:))</code></td>
<td><code>min(A)</code></td>
</tr>
<tr>
<td>143</td>
<td>Minimum value of each column of matrix A</td>
<td><code>min(A)</code> (returns a row vector)</td>
<td><code>apply(A,2,min)</code> (returns a vector)</td>
</tr>
<tr>
<td>144</td>
<td>Minimum value of each row of matrix A</td>
<td><code>min(A, [ ], 2)</code> (returns a column vector)</td>
<td><code>apply(A,1,min)</code> (returns a vector)</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------------------------------------------------------</td>
<td>-------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>145</td>
<td>Given matrices $A$ and $B$, compute a matrix where each element is the minimum of the corresponding elements of $A$ and $B$</td>
<td>min($A, B$)</td>
<td>pmin($A, B$)</td>
</tr>
<tr>
<td>146</td>
<td>Given matrix $A$ and scalar $c$, compute a matrix where each element is the minimum of $c$ and the corresponding element of $A$</td>
<td>min($A, c$)</td>
<td>pmin($A, c$)</td>
</tr>
<tr>
<td>147</td>
<td>Find minimum among all values in matrices $A$ and $B$</td>
<td>min([$A(:,); B(:)]$)</td>
<td>min($A, B$)</td>
</tr>
<tr>
<td>148</td>
<td>Find index of the first time $\min(v)$ appears in $v$, and store that index in ind</td>
<td>$[y, \text{ind}] = \min(v)$</td>
<td>ind = which.min($v$)</td>
</tr>
</tbody>
</table>

Notes:
- MATLAB and R both have a `max` function (and R has `pmax` and `which.max` as well) which behaves in the same ways as `min` but to compute maxima rather than minima.
- Functions like `exp, sin, sqrt` etc. will operate on arrays in both MATLAB and R, doing the computations for each element of the matrix.

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>149</td>
<td>Number of rows in $A$</td>
<td>size($A,1$)</td>
<td>nrow($A$)</td>
</tr>
<tr>
<td>150</td>
<td>Number of columns in $A$</td>
<td>size($A,2$)</td>
<td>ncol($A$)</td>
</tr>
<tr>
<td>151</td>
<td>Dimensions of $A$, listed in a vector</td>
<td>size($A$)</td>
<td>dim($A$)</td>
</tr>
<tr>
<td>152</td>
<td>Number of elements in vector $v$</td>
<td>length($v$)</td>
<td>length($v$)</td>
</tr>
<tr>
<td>153</td>
<td>Total number of elements in matrix $A$</td>
<td>numel($A$)</td>
<td>length($A$)</td>
</tr>
<tr>
<td>154</td>
<td>Max. dimension of $A$</td>
<td>length($A$)</td>
<td>max(dim($A$))</td>
</tr>
<tr>
<td>155</td>
<td>Sort values in vector $v$</td>
<td>sort($v$)</td>
<td>sort($v$)</td>
</tr>
<tr>
<td>156</td>
<td>Sort values in $v$, putting sorted values in $s$, and indices in idx, in the sense that $s[k] = x[idx[k]]$</td>
<td>$[s, \text{idx}] = \text{sort}(v)$</td>
<td>tmp=sort($v$,index.return=TRUE); s=tmp$x; idx=tmp$ix</td>
</tr>
<tr>
<td>157</td>
<td>Sort the order of the rows of matrix $m$</td>
<td>sortrows($m$)</td>
<td>m[order($m[,1]$),]</td>
</tr>
<tr>
<td></td>
<td>This sorts according to the first column, then uses column 2 to break ties, then column 3 for remaining ties, etc. Complex numbers are sorted by $\text{abs}(x)$, and ties are then broken by $\text{angle}(x)$.</td>
<td></td>
<td>This only sorts according to the first column. To use column 2 to break ties, and then column 3 to break further ties, do $m[\text{order}(m[,1], m[,2], m[,3])$,]</td>
</tr>
<tr>
<td>158</td>
<td>Sort order of rows of matrix $m$, specifying to use columns $c1$, $c2$, $c3$ as the sorting “keys”</td>
<td>sortrows($m$, [c1 c2 c3])</td>
<td>m[order($m[,c1], m[,c2], m[,c3]$),]</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>---</td>
</tr>
<tr>
<td>159</td>
<td>Same as previous item, but sort in decreasing order for columns ( c_1 ) and ( c_2 )</td>
<td><code>sortrows(m, [-c1 -c2 c2])</code></td>
<td><code>m[order(-m[,c1], -m[,c2], m[,c3]),]</code></td>
</tr>
<tr>
<td>160</td>
<td>Sort order of rows of matrix ( m ), and keep indices used for sorting</td>
<td><code>[y,i] = sortrows(m)</code></td>
<td><code>i=order(m[,1]); y=m[,i]</code></td>
</tr>
<tr>
<td>161</td>
<td>To count how many values in the vector ( v ) are between 4 and 7 (inclusive on the upper end)</td>
<td><code>sum((v &gt; 4) &amp; (v &lt;= 7))</code></td>
<td><code>sum((v &gt; 4) &amp; (v &lt;= 7))</code></td>
</tr>
<tr>
<td>162</td>
<td>Given vector ( v ), return list of indices of elements of ( v ) which are greater than 5</td>
<td><code>find(v &gt; 5)</code></td>
<td><code>which(v &gt; 5)</code></td>
</tr>
<tr>
<td>163</td>
<td>Given matrix ( A ), return list of indices of elements of ( A ) which are greater than 5, using single-indexing</td>
<td><code>find(A &gt; 5)</code></td>
<td><code>which(A &gt; 5)</code></td>
</tr>
<tr>
<td>164</td>
<td>Given matrix ( A ), generate vectors ( r ) and ( c ) giving rows and columns of elements of ( A ) which are greater than 5</td>
<td><code>[r,c] = find(A &gt; 5)</code></td>
<td><code>w = which(A &gt; 5, arr.ind=TRUE); r=w[,1]; c=w[,2]</code></td>
</tr>
<tr>
<td>165</td>
<td>Given vector ( x ) (of presumably discrete values), build a vector ( v ) listing unique values in ( x ), and corresponding vector ( c ) indicating how many times those values appear in ( x )</td>
<td><code>v = unique(x); c = hist(x,v);</code></td>
<td><code>w=table(x); c=as.numeric(w); v=as.numeric(names(w))</code></td>
</tr>
<tr>
<td>166</td>
<td>Given vector ( x ) (of presumably continuous values), divide the range of values into ( k ) equally-sized bins, and build a vector ( m ) containing the midpoints of the bins and a corresponding vector ( c ) containing the counts of values in the bins</td>
<td><code>[c,m] = hist(x,k)</code></td>
<td><code>w=hist(x,seq(min(x),max(x), length.out=k+1), plot=FALSE); m=w$mids; c=w$counts</code></td>
</tr>
<tr>
<td>167</td>
<td>Convolution / polynomial multiplication (given vectors ( x ) and ( y ) containing polynomial coefficients, their convolution is a vector containing coefficients of the product of the two polynomials)</td>
<td><code>conv(x,y)</code></td>
<td><code>convolve(x,rev(y),type='open')</code> Note: the accuracy of this is not as good as MATLAB; e.g. doing ( v=c(1,-1); ) for ( i ) in 2:20 ( v=convolve(v,c(-i,1), type='open') ) to generate the 20(^{th})-degree Wilkinson polynomial ( W(x) = \prod_{i=1}^{20} (x-i) ) gives a coefficient of ( \approx -780.19 ) for ( x^{19} ), rather than the correct value -210.</td>
</tr>
</tbody>
</table>
### 3.4 Root-finding

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>168</td>
<td>Find roots of polynomial whose coefficients are stored in vector ( v ) (coefficients in ( v ) are highest-order first)</td>
<td><code>roots(v)</code></td>
<td><code>polyroot(rev(v))</code> (This function really wants the vector to have the constant coefficient first in ( v ); ( rev ) reverses their order to achieve this.)</td>
</tr>
<tr>
<td>169</td>
<td>Find zero (root) of a function ( f(x) ) of one variable</td>
<td>Define function ( f(x) ), then do <code>fzero(f,x0)</code> to search for a root near ( x_0 ), or <code>fzero(f,[a b])</code> to find a root between ( a ) and ( b ), assuming the sign of ( f(x) ) differs at ( x = a ) and ( x = b ). Default forward error tolerance (i.e. error in ( x )) is machine epsilon ( \epsilon_{\text{mach}} ).</td>
<td>Define function ( f(x) ), then do <code>uniroot(f, c(a,b))</code> to find a root between ( a ) and ( b ), assuming the sign of ( f(x) ) differs at ( x = a ) and ( x = b ). Default forward error tolerance (i.e. error in ( x )) is fourth root of machine epsilon, ( (\epsilon_{\text{mach}})^{0.25} ). To specify e.g. a tolerance of ( 2^{-52} ), do <code>uniroot(f, c(a,b), tol=2^-52)</code></td>
</tr>
</tbody>
</table>

### 3.5 Function optimization/minimization

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>Find value ( m ) which minimizes a function ( f(x) ) of one variable within the interval from ( a ) to ( b )</td>
<td>Define function ( f(x) ), then do ( m = \text{fminbnd}(f, a, b) )</td>
<td>Define function ( f(x) ), then do ( m = \text{optimize}(f,c(a,b))$minimum )</td>
</tr>
<tr>
<td>171</td>
<td>Find value ( m ) which minimizes a function ( f(x,p1,p2) ) with given extra parameters (but minimization is only occurring over the first argument), in the interval from ( a ) to ( b ).</td>
<td>Define function ( f(x,p1,p2) ), then use an “anonymous function”: ( % \text{first define values for } p1 % \text{ and } p2 ), and then do: ( m=\text{fminbnd}(@x \ f(x,p1,p2),a,b) )</td>
<td>Define function ( f(x,p1,p2) ), then: # first define values for ( p1 # \text{ and } p2 ), and then do: ( m = \text{optimize}(f, c(a,b), p1=p1, p2=p2)$minimum )</td>
</tr>
<tr>
<td>172</td>
<td>Find values of ( x,y,z ) which minimize function ( f(x,y,z) ), using a starting guess of ( x = 1 ), ( y = 2.2 ), and ( z = 3.4 ).</td>
<td>First write function ( f(v) ) which accepts a vector argument ( v ) containing values of ( x ), ( y ), and ( z ), and returns the scalar value ( f(x,y,z) ), then do: ( \text{fminsearch}(\text{@f},[1 2.2 3.4]) )</td>
<td>First write function ( f(v) ) which accepts a vector argument ( v ) containing values of ( x ), ( y ), and ( z ), and returns the scalar value ( f(x,y,z) ), then do: ( \text{optim}(c(1,2.2,3.4),f)$par )</td>
</tr>
<tr>
<td>173</td>
<td>Find values of ( x,y,z ) which minimize function ( f(x,y,z,p1,p2) ), using a starting guess of ( x = 1 ), ( y = 2.2 ), and ( z = 3.4 ), where the function takes some extra parameters (useful e.g. for doing things like nonlinear least-squares optimization where you pass in some data vectors as extra parameters).</td>
<td>First write function ( f(v,p1,p2) ) which accepts a vector argument ( v ) containing values of ( x ), ( y ), and ( z ), along with the extra parameters, and returns the scalar value ( f(x,y,z,p1,p2) ), then do: ( \text{fminsearch}(\text{@f},[1 2.2 3.4], \ldots [ ], p1, p2) ) Or use an anonymous function: ( \text{fminsearch}(\text{@}(x) \ f(x,p1,p2), \ldots [1 2.2 3.4]) )</td>
<td>First write function ( f(v,p1,p2) ) which accepts a vector argument ( v ) containing values of ( x ), ( y ), and ( z ), along with the extra parameters, and returns the scalar value ( f(x,y,z,p1,p2) ), then do: ( \text{optim}(c(1,2.2,3.4), f, p1=p1, p2=p2)$par )</td>
</tr>
</tbody>
</table>
### 3.6 Numerical integration / quadrature

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>174</td>
<td>Numerically integrate function $f(x)$ over interval from $a$ to $b$</td>
<td><code>quad(f,a,b)</code> uses adaptive Simpson’s quadrature, with a default absolute tolerance of $10^{-6}$. To specify absolute tolerance, use <code>quad(f,a,b,tol)</code></td>
<td><code>integrate(f,a,b)</code> uses adaptive quadrature with default absolute and relative error tolerances being the fourth root of machine epsilon, $(\epsilon_{\text{mach}})^{0.25} \approx 1.22 \times 10^{-4}$. Tolerances can be specified by using <code>integrate(f,a,b, rel.tol=tol1, abs.tol=tol2)</code>. Note that the function $f$ must be written to work even when given a vector of $x$ values as its argument.</td>
</tr>
<tr>
<td>175</td>
<td>Simple trapezoidal numerical integration using $(x,y)$ values in vectors $x$ and $y$</td>
<td><code>trapz(x,y)</code></td>
<td><code>sum(diff(x)*(y[-length(y)]+y[-1])/2)</code></td>
</tr>
</tbody>
</table>
## 3.7 Curve fitting

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>Fit the line $y = c_1 x + c_0$ to data in vectors $x$ and $y$.</td>
<td>$p = \text{polyfit}(x, y, 1)$&lt;br&gt;The return vector $p$ has the coefficients in descending order, i.e. $p(1)$ is $c_1$, and $p(2)$ is $c_0$.</td>
<td>$p = \text{coef}(\text{lm}(y \sim x))$&lt;br&gt;The return vector $p$ has the coefficients in ascending order, i.e. $p[1]$ is $c_0$, and $p[2]$ is $c_1$.</td>
</tr>
<tr>
<td>177</td>
<td>Fit the quadratic polynomial $y = c_2 x^2 + c_1 x + c_0$ to data in vectors $x$ and $y$.</td>
<td>$p = \text{polyfit}(x, y, 2)$&lt;br&gt;The return vector $p$ has the coefficients in descending order, i.e. $p(1)$ is $c_2$, $p(2)$ is $c_1$, and $p(3)$ is $c_0$.</td>
<td>$p = \text{coef}(\text{lm}(y \sim x^2 + I(x^2)))$&lt;br&gt;The return vector $p$ has the coefficients in ascending order, i.e. $p[1]$ is $c_0$, $p[2]$ is $c_1$, and $p[3]$ is $c_2$.</td>
</tr>
<tr>
<td>178</td>
<td>Fit $n^{th}$ degree polynomial $y = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$ to data in vectors $x$ and $y$.</td>
<td>$p = \text{polyfit}(x, y, n)$&lt;br&gt;The return vector $p$ has the coefficients in descending order, $p(1)$ is $c_n$, $p(2)$ is $c_{n-1}$, etc.</td>
<td>No simple built-in way. But this will work: $\text{coef}(\text{lm}((\text{as.formula(\text{paste('y~', paste('I(x^', 1:n, ')', sep=',' , collapse= '+')))}}))$&lt;br&gt;This more concise “lower-level” method will also work: $\text{coef}(\text{lm.fit}(\text{outer}(x, 0:n, '^'), y))$&lt;br&gt;Note that both of the above return the coefficients in ascending order. Also see the $\text{polyreg}$ function in the $\text{mda}$ package (see item 338 for how to install/load packages).</td>
</tr>
<tr>
<td>179</td>
<td>Fit the quadratic polynomial with zero intercept, $y = c_2 x^2 + c_1 x$ to data in vectors $x$ and $y$.</td>
<td>(I don’t know a simple way do this in MATLAB, other than to write a function which computes the sum of squared residuals and use $\text{fminsearch}$ on that function. There is likely an easy way to do it in the Statistics Toolbox.)</td>
<td>$p=\text{coef}(\text{lm}(y \sim -1 + x + I(x^2)))$&lt;br&gt;The return vector $p$ has the coefficients in ascending order, i.e. $p[1]$ is $c_0$, and $p[2]$ is $c_2$.</td>
</tr>
<tr>
<td>180</td>
<td>Fit natural cubic spline ($S''(x) = 0$ at both endpoints) to points $(x_i, y_i)$ whose coordinates are in vectors $x$ and $y$; evaluate at points whose $x$ coordinates are in vector $xx$, storing corresponding $y$’s in $yy$</td>
<td>$pp=\text{csape}(x, y, '\text{variational}'); yy=\text{ppval}(pp, xx)$ but note that $\text{csape}$ is in MATLAB’s Spline Toolbox</td>
<td>$\text{tmp=spline}(x, y, \text{method='natural'}, xout=xx); yy=\text{tmp}$y</td>
</tr>
<tr>
<td>181</td>
<td>Fit cubic spline using Forsythe, Malcolm and Moler method (third derivatives at endpoints match third derivatives of exact cubics through the four points at each end) to points $(x_i, y_i)$ whose coordinates are in vectors $x$ and $y$; evaluate at points whose $x$ coordinates are in vector $xx$, storing corresponding $y$’s in $yy$</td>
<td>I’m not aware of a function to do this in MATLAB</td>
<td>$\text{tmp=spline}(x, y, xout=xx); yy=\text{tmp}$y</td>
</tr>
</tbody>
</table>
### 4 Conditionals, control structure, loops

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
</table>
| 185 | “for” loops over values in a vector `v` (the vector `v` is often constructed via `a:b`) | `for i=v  
 command1  
 command2  
end` | If only one command inside the loop:  
`for (i in v)  
 command  
or  
for (i in v) command`  
If multiple commands inside the loop:  
`for (i in v) {  
 command1  
 command2  
}` |
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
</table>
| 186 | “if” statements with no else clause | if cond  
    command1  
    command2  
  end | If only one command inside the clause:  
    if (cond)  
    command  
    or  
    if (cond) command  
  If multiple commands:  
    if (cond) {  
    command1  
    command2  
  } |
| 187 | “if/else” statement | if cond  
    command1  
    command2  
  else  
    command3  
    command4  
  end | If one command in clauses:  
    if (cond)  
    command1 else  
    command2  
    or  
    if (cond) cmd1 else cmd2  
  If multiple commands:  
    if (cond) {  
    command1  
    command2  
  } else if (cond2) {  
    command2  
  } else if (cond3) {  
    command3  
    command4  
  } else {  
    command4  
  } |

Note: MATLAB also has an “elseif” statement, e.g.:  
if cond1  
  commands1  
elseif cond2  
  commands2  
elseif cond3  
  commands3  
else  
  commands4  
end

Warning: the “else” must be on the same line as command1 or the “{” (when typed interactively at the command prompt), otherwise R thinks the “if” statement was finished and gives an error.

R does not have an “elseif” statement (though see item 140 for something related), but you can do this:  
if (cond1) {  
  commands1  
} else if (cond2) {  
  commands2  
} else if (cond3) {  
  commands3  
} else {  
  commands4  
}
Logical comparisons which can be used on scalars in “if” statements, or which operate element-by-element on vectors/matrices:

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>R</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt; a</td>
<td>x &lt; a</td>
<td>True if x is less than a</td>
</tr>
<tr>
<td>x &gt; a</td>
<td>x &gt; a</td>
<td>True if x is greater than a</td>
</tr>
<tr>
<td>x &lt;= a</td>
<td>x &lt;= a</td>
<td>True if x is less than or equal to a</td>
</tr>
<tr>
<td>x &gt;= a</td>
<td>x &gt;= a</td>
<td>True if x is greater than or equal to a</td>
</tr>
<tr>
<td>x == a</td>
<td>x == a</td>
<td>True if x is equal to a</td>
</tr>
<tr>
<td>x != a</td>
<td>x != a</td>
<td>True if x is not equal to a</td>
</tr>
</tbody>
</table>

Scalar logical operators:

<table>
<thead>
<tr>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a AND b</td>
<td>a &amp;&amp; b</td>
<td>a &amp;&amp; b</td>
</tr>
<tr>
<td>a OR b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a XOR b</td>
<td>xor(a,b)</td>
<td>xor(a,b)</td>
</tr>
<tr>
<td>NOT a</td>
<td>~a</td>
<td>!a</td>
</tr>
</tbody>
</table>

The && and || operators are short-circuiting, i.e. && stops as soon as any of its terms are FALSE, and || stops as soon as any of its terms are TRUE.

Matrix logical operators (they operate element-by-element):

<table>
<thead>
<tr>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a AND b</td>
<td>a &amp; b</td>
<td>a &amp; b</td>
</tr>
<tr>
<td>a OR b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a XOR b</td>
<td>xor(a,b)</td>
<td>xor(a,b)</td>
</tr>
<tr>
<td>NOT a</td>
<td>~a</td>
<td>!a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>188</td>
<td>To test whether a scalar value x is between 4 and 7 (inclusive on the upper end)</td>
<td>if ((x &gt; 4) &amp;&amp; (x &lt;= 7))</td>
<td>if ((x &gt; 4) &amp;&amp; (x &lt;= 7))</td>
</tr>
<tr>
<td>189</td>
<td>To count how many values in the vector x are between 4 and 7 (inclusive on the upper end)</td>
<td>sum((x &gt; 4) &amp; (x &lt;= 7))</td>
<td>sum((x &gt; 4) &amp; (x &lt;= 7))</td>
</tr>
<tr>
<td>190</td>
<td>Test whether all values in a logical/boolean vector are TRUE</td>
<td>all(v)</td>
<td>all(v)</td>
</tr>
<tr>
<td>191</td>
<td>Test whether any values in a logical/boolean vector are TRUE</td>
<td>any(v)</td>
<td>any(v)</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>192</td>
<td>“while” statements to do iteration (useful when you don’t know ahead of time how many iterations you’ll need). E.g. to add uniform random numbers between 0 and 1 (and their squares) until their sum is greater than 20:</td>
<td>mysum = 0; mysumsqr = 0; while (mysum &lt; 20) r = rand; mysum = mysum + r; mysumsqr = mysumsqr + r^2; end</td>
<td>mysum = 0 mysumsqr = 0 while (mysum &lt; 20) { r = runif(1) mysum = mysum + r mysumsqr = mysumsqr + r^2 } (As with “if” statements and “for” loops, the curly brackets are not necessary if there’s only one statement inside the “while” loop.)</td>
</tr>
<tr>
<td>193</td>
<td>More flow control: these commands exit or move on to the next iteration of the innermost <code>while</code> or <code>for</code> loop, respectively.</td>
<td>break and continue</td>
<td>break and next</td>
</tr>
<tr>
<td>194</td>
<td>“Switch” statements for integers</td>
<td>switch (x)</td>
<td>R doesn’t have a <code>switch</code> statement capable of doing this. It has a function which is fairly limited for integers, but can which do string matching. See <code>?switch</code> for more. But a basic example of what it can do for integers is below, showing that you can use it to return different expressions based on whether a value is 1, 2,....</td>
</tr>
<tr>
<td></td>
<td></td>
<td>case 10 disp('ten')</td>
<td>mystr = switch(x, 'one', 'two', 'three'); print(myst)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>case {12,13} disp('dozen (bakers?)')</td>
<td>Note that <code>switch</code> returns NULL if x is larger than 3 in the above case. Also, continuous values of x will be truncated to integers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>otherwise disp('unrecognized')</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
## 5 Functions, ODEs

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
</table>
| 195 | Implement a function \(\text{add}(x,y)\) | Put the following in add.m:  
\[
\text{function retval=\text{add}(x,y)} \\
\text{retval = \text{x+y};}
\]  
Then you can do e.g. \text{add}(2,3) | Enter the following, or put it in a file and \text{source} that file:  
\[
\text{add = function(x,y) \{} \\
\text{\hspace{1cm} return(x+y)} \\
\text{\}}
\]  
Then you can do e.g. \text{add}(2,3). Note, the curly brackets aren’t needed if your function only has one line. Also, the \text{return} keyword is optional in the above example, as the value of the last expression in a function gets returned, so just \text{x+y} would work too. |
| 196 | Implement a function \(f(x,y,z)\) which returns multiple values, and store those return values in variables \(u\) and \(v\) | Write function as follows:  
\[
\text{function [a,b] = f(x,y,z)} \\
\text{\hspace{1cm} a = x\cdot y + z; \hspace{1cm} b=2\cdot \sin(x-z)};
\]  
Then call the function by doing:  
\[
[u,v] = f(2,8,12)
\] | Write function as follows:  
\[
\text{f = function(x,y,z) \{} \\
\text{\hspace{1cm} a = x\cdot y + z; \hspace{1cm} b=2\cdot \sin(x-z) } \\
\text{\hspace{1cm} return(list(a,b)) \}}
\]  
Then call the function by doing:  
\[
\text{tmp=f(2,8,12); u=tmp[[1]]; v=tmp[[2]]}. \text{ The above is most general, and will work even when } u \text{ and } v \text{ are different types of data. If they are both scalars, the function could simply return them packed in a vector, i.e. } \text{return(c(a,b))}. \text{ If they are vectors of the same size, the function could return them packed together into the columns of a matrix, i.e. } \text{return(cbind(a,b))}.
\] |
No. | Description | MATLAB | R |
--- | --- | --- | --- |
197 | Numerically solve ODE \( \frac{dx}{dt} = 5x \) from \( t = 3 \) to \( t = 12 \) with initial condition \( x(3) = 7 \) | First implement function 
```matlab
function retval=f(t,x)
    retval = 5*x;
end
```
Then do `ode45(@f,[3,12],7)` to plot solution, or `[t,x]=ode45(@f,[3,12],7)` to get back vector \( t \) containing time values and vector \( x \) containing corresponding function values. If you want function values at specific times, e.g. \( 3, 3.1, 3.2, \ldots, 11.9, 12 \), you can do `ode45(@f,3:0.1:12,7)`.

First implement function 
```r
f = function(t,x,parms) {
    w = x[1]; z = x[2];
    return(list(5*w, 3*w+7*z))
}
```
Then do `y=lsoda(7, seq(3,12, 0.1), myfunc,NA)` to obtain solution values at times \( 3, 3.1, 3.2, \ldots, 11.9, 12 \). The first column of \( y \), namely \( y[,1] \) contains the time values; the second column \( y[,2] \) contains the corresponding function values. Note: `lsoda` is part of the `deSolve` package (see item 338 for how to install/load packages).

198 | Numerically solve system of ODEs \( \frac{dw}{dt} = 5w, \frac{dz}{dt} = 3w+7z \) from \( t = 3 \) to \( t = 12 \) with initial conditions \( w(3) = 7, z(3) = 8.2 \) | First implement function 
```matlab
function retval=myfunc(t,x)
    w = x(1); z = x(2);
    retval = zeros(2,1);
    retval(1) = 5*w;
    retval(2) = 3*w + 7*z;
end
```
Then do `ode45(@myfunc,[3,12],[7;8.2])` to plot solution, or `ode45(@myfunc,[3,12],[7;8.2])` to get back vector \( t \) containing time values and matrix \( x \), whose first column containing \( w(t) \) values and second column containing \( z(t) \) values. If you want function values at specific times, e.g. \( 3.1, 3.2, \ldots, 11.9, 12 \), you can do `ode45(@myfunc,3:0.1:12,[7;8.2])`.

First implement function 
```r
myfunc = function(t,x,parms) {
    w = x[1]; z = x[2];
    return(list(c(5*w, 3*w+7*z)))
}
```
Then do `y=lsoda(c(7,8.2), seq(3,12, 0.1), myfunc,NA)` to obtain solution values at times \( 3, 3.1, 3.2, \ldots, 11.9, 12 \). The first column of \( y \), namely \( y[,1] \) contains the time values; the second column \( y[,2] \) contains the corresponding values of \( w(t) \); and the third column contains \( z(t) \). Note: `lsoda` is part of the `deSolve` package (see item 338 for how to install/load packages).

199 | Pass parameters such as \( r = 1.3 \) and \( K = 50 \) to an ODE function from the command line, solving \( \frac{dx}{dt} = rx(1-x/K) \) from \( t = 0 \) to \( t = 20 \) with initial condition \( x(0) = 2.5 \). | First implement function 
```matlab
function retval=func2(t,x,r,K)
    retval = r*x*(1-x/K);
end
```
Then do `ode45(@func2,[0 20],2.5, [ ], 1.3, 50)`.

First implement function 
```r
func2=function(t,x,parms) {
    r=parms[1]; K=parms[2]
    return(list(r*x*(1-x/K)))
}
```
Then do `y=lsoda(2.5,seq(0,20,0.1), func2,c(1.3,50))`.

Note: `lsoda` is part of the `deSolve` package (see item 338 for how to install/load packages).
6 Probability and random values

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Generate a continuous uniform random value between 0 and 1</td>
<td><code>rand</code></td>
<td><code>runif(1)</code></td>
</tr>
<tr>
<td>201</td>
<td>Generate vector of (n) uniform random vals between 0 and 1</td>
<td><code>rand(n,1)</code> or <code>rand(1,n)</code></td>
<td><code>runif(n)</code></td>
</tr>
<tr>
<td>202</td>
<td>Generate (m \times n) matrix of uniform random values between 0 and 1</td>
<td><code>rand(m,n)</code></td>
<td><code>matrix(runif(m*n),m,n)</code> or just <code>matrix(runif(m*n),m)</code></td>
</tr>
<tr>
<td>203</td>
<td>Generate (m \times n) matrix of continuous uniform random values between (a) and (b)</td>
<td><code>a+rand(m,n)*(b-a)</code> or if you have the Statistics toolbox then <code>unifrnd(a,b,m,n)</code></td>
<td><code>matrix(runif(m*n,a,b),m)</code></td>
</tr>
<tr>
<td>204</td>
<td>Generate a random integer between 1 and (k)</td>
<td><code>floor(k*rand) + 1</code></td>
<td><code>floor(k*runif(1)) + 1</code> or <code>sample(k,1)</code></td>
</tr>
<tr>
<td>205</td>
<td>Generate (m \times n) matrix of discrete uniform random integers between 1 and (k)</td>
<td><code>floor(k*rand(m,n))+1</code> or if you have the Statistics toolbox then <code>unidrnd(k,m,n)</code></td>
<td><code>floor(k*matrix(runif(m*n),m))+1</code></td>
</tr>
<tr>
<td>206</td>
<td>Generate (m \times n) matrix where each entry is 1 with probability (p), otherwise is 0</td>
<td><code>(rand(m,n)&lt;p)*1</code> Note: multiplying by 1 turns the logical (true/false) result back into numeric values. You could also do <code>double(rand(m,n)&lt;p)</code></td>
<td><code>(matrix(runif(m,n),m)&lt;p)*1</code> (Note: multiplying by 1 turns the logical (true/false) result back into numeric values; using <code>as.numeric()</code> to do it would lose the shape of the matrix.)</td>
</tr>
<tr>
<td>207</td>
<td>Generate (m \times n) matrix where each entry is (a) with probability (p), otherwise is (b)</td>
<td><code>b + (a-b)*(rand(m,n)&lt;p)</code></td>
<td><code>b + (a-b)*(matrix(runif(m,n),m)&lt;p)</code></td>
</tr>
<tr>
<td>208</td>
<td>Generate a random integer between (a) and (b) inclusive</td>
<td><code>floor((b-a+1)*rand)+a</code> or if you have the Statistics toolbox then <code>unidrnd(b-a+1)+a-1</code></td>
<td><code>floor((b-a+1)*runif(1))+a</code></td>
</tr>
<tr>
<td>209</td>
<td>Flip a coin which comes up heads with probability (p), and perform some action if it does come up heads</td>
<td>if ((rand &lt; p)) [ \text{...some commands...} ] end</td>
<td>if ((runif(1) &lt; p)) { [ \text{...some commands...} ] }</td>
</tr>
<tr>
<td>210</td>
<td>Generate a random permutation of the integers 1, 2, \ldots, (n)</td>
<td><code>randperm(n)</code></td>
<td><code>sample(n)</code></td>
</tr>
<tr>
<td>211</td>
<td>Generate a random selection of (k) unique integers between 1 and (n) (i.e. sampling without replacement)</td>
<td><code>[s,idx]=sort(rand(n,1)); ri=idx(1:k)</code> or another way is <code>ri=randperm(n); ri=ri(1:k)</code>. Or if you have the Statistics Toolbox, then <code>randsample(n,k)</code></td>
<td><code>ri=sample(n,k)</code></td>
</tr>
<tr>
<td>212</td>
<td>Choose (k) values (with replacement) from the vector (v), storing result in (w)</td>
<td><code>L=length(v); w=v(floor(L*rand(k,1))+1)</code> Or, if you have the Statistics Toolbox, <code>w=randsample(v,k)</code></td>
<td><code>w=sample(v,k,replace=TRUE)</code></td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>---</td>
</tr>
<tr>
<td>213</td>
<td>Choose ( k ) values (without replacement) from the vector ( v ), storing result in ( w )</td>
<td>( L=\text{length}(v); \ ri=\text{randperm}(L); \ ri=ri(1:k); \ w=v(ri) ) Or, if you have the Statistics Toolbox, ( w=\text{randSample}(v,k) )</td>
<td>( \text{w=sample}(v,k,\text{replace}=\text{FALSE}) )</td>
</tr>
<tr>
<td>214</td>
<td>Generate a value from 1 to ( n ) with corresponding probabilities in vector ( pv )</td>
<td>( \text{sum} \left( \text{rand} &gt; \text{cumsum}(pv) \right) + 1 ) If entries of ( pv ) don’t sum to one, rescale them first: ( \text{sum} \left( \text{rand} &gt; \text{cumsum}(pv)/\text{sum}(pv) \right) + 1 )</td>
<td>( \text{sample}(n, 1, \text{prob}=pv) ) If the entries of ( pv ) don’t sum to one, \text{sample} automatically rescales them to do so.</td>
</tr>
<tr>
<td>215</td>
<td>Set the random-number generator back to a known state (useful to do at the beginning of a stochastic simulation when debugging, so you’ll get the same sequence of random numbers each time)</td>
<td>( \text{rand}('\text{state}', 12) ) Note: beginning in Matlab 7.7, use this instead: ( \text{RandStream}('mt19937ar', 'Seed', 12) ) though the previous method is still supported for now.</td>
<td>( \text{set.seed}(12) )</td>
</tr>
</tbody>
</table>

Note that the “*rnd,” “*pdf,” and “*cdf” functions described below are all part of the Matlab Statistics Toolbox, and not part of the core Matlab distribution.

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>216</td>
<td>Generate a random value from the binomial(( n, p )) distribution</td>
<td>( \text{binornd}(n,p) ) or ( \text{sum} \left( \text{rand}(n,1) &lt; p \right) ) will work even without the Statistics Toolbox.</td>
<td>( \text{rbinom}(1,n,p) )</td>
</tr>
<tr>
<td>217</td>
<td>Generate a random value from the Poisson distribution with parameter ( \lambda )</td>
<td>( \text{poissrnd}(\lambda) )</td>
<td>( \text{rpois}(1,\lambda) )</td>
</tr>
<tr>
<td>218</td>
<td>Generate a random value from the exponential distribution with mean ( \mu )</td>
<td>( \text{exprnd}(\mu) ) or ( -\mu \log(\text{rand}) ) will work even without the Statistics Toolbox.</td>
<td>( \text{rexp}(1, 1/\mu) )</td>
</tr>
<tr>
<td>219</td>
<td>Generate a random value from the discrete uniform distribution on integers ( 1 \ldots k )</td>
<td>( \text{unidrnd}(k) ) or ( \text{floor}(\text{rand} \times k) + 1 ) will work even without the Statistics Toolbox.</td>
<td>( \text{sample}(k,1) )</td>
</tr>
<tr>
<td>220</td>
<td>Generate ( n ) iid random values from the discrete uniform distribution on integers ( 1 \ldots k )</td>
<td>( \text{unidrnd}(k,n,1) ) or ( \text{floor}(\text{rand}(n,1) \times k) + 1 ) will work even without the Statistics Toolbox.</td>
<td>( \text{sample}(k,n,\text{replace}=\text{TRUE}) )</td>
</tr>
<tr>
<td>221</td>
<td>Generate a random value from the continuous uniform distribution on the interval ((a,b))</td>
<td>( \text{unifrnd}(a,b) ) or ( (b-a) \times \text{rand} + a ) will work even without the Statistics Toolbox.</td>
<td>( \text{runif}(1,a,b) )</td>
</tr>
<tr>
<td>222</td>
<td>Generate a random value from the normal distribution with mean ( \mu ) and standard deviation ( \sigma )</td>
<td>( \text{normrnd}(\mu,\sigma) ) or ( \mu + \sigma \times \text{randn} ) will work even without the Statistics Toolbox.</td>
<td>( \text{rnorm}(1,\mu,\sigma) )</td>
</tr>
<tr>
<td>223</td>
<td>Generate a random vector from the multinomial distribution, with ( n ) trials and probability vector ( p )</td>
<td>( \text{mnrnd}(n,p) )</td>
<td>( \text{rmultinom}(1,n,p) )</td>
</tr>
<tr>
<td>224</td>
<td>Generate ( j ) random vectors from the multinomial distribution, with ( n ) trials and probability vector ( p )</td>
<td>( \text{mnrnd}(n,p,j) ) The vectors are returned as rows of a matrix</td>
<td>( \text{rmultinom}(j,n,p) ) The vectors are returned as columns of a matrix</td>
</tr>
</tbody>
</table>

Notes:
The MATLAB “*rnd” functions above can all take additional \( r, c \) arguments to build an \( r \times c \) matrix of iid random values. E.g. \texttt{poissrnd(3.5, 4, 7)} for a \( 4 \times 7 \) matrix of iid values from the Poisson distribution with mean \( \lambda = 3.5 \). The \texttt{unidrnd(k,n,1)} command above is an example of this, to generate a \( k \times 1 \) column vector.

The first parameter of the R “*r” functions above specifies how many values are desired. E.g. to generate 28 iid random values from a Poisson distribution with mean 3.5, use \texttt{rpois(28, 3.5)}. To get a \( 4 \times 7 \) matrix of such values, use \texttt{matrix(rpois(28, 3.5),4)}.

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>225</td>
<td>Probability that a random variable from the Binomial(( n, p )) distribution has value ( x ) (i.e. the density, or pdf).</td>
<td>\texttt{binopdf(x,n,p)} or \texttt{nchoosek(n,x)<em>p^x</em>(1-p)^(n-x)} will work even without the Statistics Toolbox, as long as ( n ) and ( x ) are non-negative integers and ( 0 \leq p \leq 1 ).</td>
<td>\texttt{dbinom(x,n,p)}</td>
</tr>
<tr>
<td>226</td>
<td>Probability that a random variable from the Poisson(( \lambda )) distribution has value ( x ).</td>
<td>\texttt{poisspdf(x,lambda)} or \texttt{exp(-lambda)*lambda^x / factorial(x)} will work even without the Statistics Toolbox, as long as ( x ) is a non-negative integer and ( \lambda \geq 0 ).</td>
<td>\texttt{dpois(x,lambda)}</td>
</tr>
<tr>
<td>227</td>
<td>Probability density function at ( x ) for a random variable from the exponential distribution with mean ( \mu ).</td>
<td>\texttt{exppdf(x,mu)} or \texttt{(x&gt;=0)*exp(-x/mu)/mu} will work even without the Statistics Toolbox, as long as ( \mu ) is positive.</td>
<td>\texttt{dexp(x,1/mu)}</td>
</tr>
<tr>
<td>228</td>
<td>Probability density function at ( x ) for a random variable from the Normal distribution with mean ( \mu ) and standard deviation ( \sigma ).</td>
<td>\texttt{normpdf(x,mu,sigma)} or \texttt{exp(-(x-mu)^2/(2<em>sigma^2)) / (sqrt(2</em>pi)*sigma)} will work even without the Statistics Toolbox.</td>
<td>\texttt{dnorm(x,mu,sigma)}</td>
</tr>
<tr>
<td>229</td>
<td>Probability density function at ( x ) for a random variable from the continuous uniform distribution on interval ((a, b)).</td>
<td>\texttt{unifpdf(x,a,b)} or \texttt{((x&gt;a)&amp;&amp;(x&lt;=b))/(b-a)} will work even without the Statistics Toolbox.</td>
<td>\texttt{dunif(x,a,b)}</td>
</tr>
<tr>
<td>230</td>
<td>Probability that a random variable from the discrete uniform distribution on integers 1...( n ) has value ( x ).</td>
<td>\texttt{unidpdf(x,n) or ((x==floor(x)) &amp;&amp; (x&gt;1) &amp;&amp; (x&lt;n))/n} will work even without the Statistics Toolbox, as long as ( n ) is a positive integer.</td>
<td>\texttt{dunif(x,a,b)}</td>
</tr>
<tr>
<td>231</td>
<td>Probability that a random vector from the multinomial distribution with probability vector ( \vec{p} ) has the value ( \vec{x} )</td>
<td>\texttt{mnpdf(x,p)} Note: vector ( p ) must sum to one. Also, ( x ) and ( p ) can be vectors of length ( k ), or if one or both are ( m \times k ) matrices then the computations are performed for each row.</td>
<td>\texttt{dmultinom(x,prob=p)}</td>
</tr>
</tbody>
</table>

Note: one or more of the parameters in the above “pdf” (MATLAB) or “d*” (R) functions can be vectors, but they must be the same size. Scalars are promoted to arrays of the appropriate size.
The corresponding CDF functions are below:

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>232</td>
<td>Probability that a random variable from the Binomial((n, p)) distribution is less than or equal to (x) (i.e. the cumulative distribution function, or cdf).</td>
<td><code>binocdf(x,n,p)</code></td>
<td><code>pbinom(x,n,p)</code></td>
</tr>
<tr>
<td>233</td>
<td>Probability that a random variable from the Poisson((\lambda)) distribution is less than or equal to (x).</td>
<td><code>poisscdf(x,lambda)</code></td>
<td><code>ppois(x,lambda)</code></td>
</tr>
<tr>
<td>234</td>
<td>Cumulative distribution function at (x) for a random variable from the exponential distribution with mean (\mu).</td>
<td><code>expcdf(x,mu)</code> or ((x&gt;=0)*(1-exp(-x/mu)))</td>
<td><code>pexp(x,1/mu)</code></td>
</tr>
<tr>
<td>235</td>
<td>Cumulative distribution function at (x) for a random variable from the Normal distribution with mean (\mu) and standard deviation (\sigma).</td>
<td><code>normcdf(x,mu,sigma)</code> or (1/2 - \text{erf}((-x-mu)/(sigma*sqrt(2)))/2)</td>
<td><code>pnorm(x,mu,sigma)</code></td>
</tr>
<tr>
<td>236</td>
<td>Cumulative distribution function at (x) for a random variable from the continuous uniform distribution on interval ((a, b)).</td>
<td><code>unifcdf(x,a,b)</code> or ((x&gt;a)*(min(x,b)-a)/(b-a))</td>
<td><code>punif(x,a,b)</code></td>
</tr>
<tr>
<td>237</td>
<td>Probability that a random variable from the discrete uniform distribution on integers 1…(n) is less than or equal to (x).</td>
<td><code>unidcdf(x,n)</code> or ((x&gt;=1)*min(floor(x),n)/n)</td>
<td>((x&gt;=1)*min(floor(x),n)/n)</td>
</tr>
</tbody>
</table>
7 Graphics

7.1 Various types of plotting

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>238</td>
<td>Create a new figure window</td>
<td><code>figure</code></td>
<td><code>dev.new()</code> Notes: internally, on Windows this calls <code>windows()</code>, on MacOS it calls <code>quartz()</code>, and on Linux it calls <code>X11()</code>. <code>X11()</code> is also available on MacOS: you can tell R to use it by default by doing <code>options(device='X11')</code>. In R sometime after 2.7.0, X11 graphics started doing antialiasing by default, which makes plots look smoother but takes longer to draw. If you are using X11 graphics in R and notice that figure plotting is extremely slow (especially if making many plots), do this before calling <code>dev.new()</code>: <code>X11.options(type='Xlib')</code> or <code>X11.options(antialias='none')</code>. Or just use e.g. <code>X11(type='Xlib')</code> to make new figure windows. They are uglier (lines are more jagged), but render much more quickly.</td>
</tr>
<tr>
<td>239</td>
<td>Select figure number n</td>
<td><code>figure(n)</code> (will create the figure if it doesn’t exist)</td>
<td><code>dev.set(n)</code> (returns the actual device selected; will be different from n if there is no figure device with number n)</td>
</tr>
<tr>
<td>240</td>
<td>Determine which figure window is currently active</td>
<td><code>gcf</code></td>
<td><code>dev.cur()</code></td>
</tr>
<tr>
<td>241</td>
<td>List open figure windows</td>
<td><code>get(0,'children')</code> (The 0 handle refers to the root graphics object.)</td>
<td><code>dev.list()</code></td>
</tr>
<tr>
<td>242</td>
<td>Close figure window(s)</td>
<td><code>close</code> to close the current figure window, <code>close(n)</code> to close a specified figure, and <code>close all</code> to close all figures</td>
<td><code>dev.off()</code> to close the currently active figure device, <code>dev.off(n)</code> to close a specified one, and <code>graphics.off()</code> to close all figure devices.</td>
</tr>
<tr>
<td>243</td>
<td>Plot points using open circles</td>
<td><code>plot(x,y,'o')</code></td>
<td><code>plot(x,y)</code></td>
</tr>
<tr>
<td>244</td>
<td>Plot points using solid lines</td>
<td><code>plot(x,y)</code></td>
<td><code>plot(x,y,type='l')</code> (Note: that’s a lower-case ‘L’, not the number 1)</td>
</tr>
<tr>
<td>245</td>
<td>Plotting: color, point markers, linestyle</td>
<td><code>plot(x,y,str)</code> where <code>str</code> is a string specifying color, point marker, and/or linestyle (see table below) (e.g. ‘gs--’ for green squares with dashed line)</td>
<td><code>plot(x,y,type=str1, pch=arg2,col=str3, lty=arg4)</code> See tables below for possible values of the 4 parameters</td>
</tr>
<tr>
<td>246</td>
<td>Plotting with logarithmic axes</td>
<td><code>semilogx</code>, <code>semilogy</code>, and <code>loglog</code> functions take arguments like <code>plot</code>, and plot with logarithmic scales for x, y, and both axes, respectively</td>
<td><code>plot(...)</code>, <code>log='x'</code>), <code>plot(...)</code>, <code>log='y'</code>), and <code>plot(...)</code>, <code>log='xy'</code>) plot with logarithmic scales for x, y, and both axes, respectively</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>---</td>
</tr>
<tr>
<td>247</td>
<td>Make bar graph where the (x) coordinates of the bars are in (x), and their heights are in (y)</td>
<td><code>bar(x,y)</code> or just <code>bar(y)</code> if you only want to specify heights. Note: if (A) is a matrix, <code>bar(A)</code> interprets each column as a separate set of observations, and each row as a different observation within a set. So a (20 \times 2) matrix is plotted as 2 sets of 20 observations, while a (2 \times 20) matrix is plotted as 20 sets of 2 observations.</td>
<td>Can’t do this in R; but <code>barplot(y)</code> makes a bar graph where you specify the heights, <code>barplot(y,w)</code> also specifies the widths of the bars, and <code>hist</code> can make plots like this too.</td>
</tr>
<tr>
<td>248</td>
<td>Make histogram of values in (x)</td>
<td><code>hist(x)</code></td>
<td><code>hist(x)</code></td>
</tr>
<tr>
<td>249</td>
<td>Given vector (x) containing discrete values, make a bar graph where the (x) coordinates of bars are the values, and heights are the counts of how many times the values appear in (x)</td>
<td>(v=\text{unique}(x); c=\text{hist}(x,v); ) <code>bar(v,c)</code></td>
<td><code>barplot(table(x))</code></td>
</tr>
<tr>
<td>250</td>
<td>Given vector (x) containing continuous values, lump the data into (k) bins and make a histogram / bar graph of the binned data</td>
<td><code>[c,m] = \text{hist}(x,k); \text{bar}(m,c)</code> or for slightly different plot style use <code>hist(x,k)</code></td>
<td><code>hist(x,seq(min(x), max(x), length.out=k+1))</code></td>
</tr>
<tr>
<td>251</td>
<td>Make a plot containing error-bars of height (s) above and below ((x, y)) points</td>
<td><code>errorbar(x,y,s)</code></td>
<td><code>errorbar(x,y,y+s,y-s)</code> Note: <code>errorbar</code> is part of the <code>Hmisc</code> package (see item 338 for how to install/load packages).</td>
</tr>
<tr>
<td>252</td>
<td>Make a plot containing error-bars of height (a) above and (b) below ((x, y)) points</td>
<td><code>errorbar(x,y,b,a)</code></td>
<td><code>errorbar(x,y,y+a,y-b)</code> Note: <code>errorbar</code> is part of the <code>Hmisc</code> package (see item 338 for how to install/load packages).</td>
</tr>
<tr>
<td>253</td>
<td>Other types of 2-D plots</td>
<td><code>stem(x,y)</code> and <code>stairs(x,y)</code> for other types of 2-D plots. <code>polar(theta,r)</code> to use polar coordinates for plotting.</td>
<td><code>pie(v)</code></td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>----</td>
</tr>
<tr>
<td>254</td>
<td>Make a 3-D plot of some data points with given ( x, y, z ) coordinates in the vectors ( x, y, ) and ( z ).</td>
<td>\texttt{plot3(x,y,z)} This works much like \texttt{plot}, as far as plotting symbols, line-types, and colors.</td>
<td>\texttt{cloud(z<del>x+y)} You can also use arguments \texttt{pch} and \texttt{col} as with \texttt{plot}. To make a 3-D plot with lines, do \texttt{cloud(z</del>x+y,type='l', panel.cloud=panel.3dwire)}. See the \texttt{rgl} package to interactively rotate 3-D plots (and see item 338 for how to load packages).</td>
</tr>
<tr>
<td>255</td>
<td>Surface plot of data in matrix ( A )</td>
<td>\texttt{surf(A)} You can then click on the small curved arrow in the figure window (or choose “Rotate 3D” from the “Tools” menu), and then click and drag the mouse in the figure to rotate it in three dimensions.</td>
<td>\texttt{persp(A)} You can include shading in the image via e.g. \texttt{persp(A,shade=0.5)}. There are two viewing angles you can also specify, among other parameters, e.g. \texttt{persp(A, shade=0.5, theta=50, phi=35)}.</td>
</tr>
<tr>
<td>256</td>
<td>Surface plot of ( f(x, y) = \sin(x + y)\sqrt{y} ) for 100 values of ( x ) between 0 and 10, and 90 values of ( y ) between 2 and 8</td>
<td>( x = \text{linspace}(0,10,100); ) ( y = \text{linspace}(2,8,90); ) ( [X,Y] = \text{meshgrid}(x,y); ) ( Z = \sin(X+Y).*\text{sqrt}(Y); ) \texttt{surf(X,Y,Z)} shading flat</td>
<td>( x = \text{seq}(0,10,\text{len}=100) ) ( y = \text{seq}(2,8,\text{len}=90) ) ( f = \text{function}(x,y) ) ( \text{return}(\sin(x+y)\ast\text{sqrt}(y)) ) ( z = \text{outer}(x,y,f) ) \texttt{persp(x,y,z)}</td>
</tr>
<tr>
<td>257</td>
<td>Other ways of plotting the data from the previous command</td>
<td>\texttt{mesh(X,Y,Z), surfc(X,Y,Z), surfl(X,Y,Z), contour(X,Y,Z), pcolor(X,Y,Z), waterfall(X,Y,Z). Also see the \texttt{slice} command.}</td>
<td>\texttt{contour(x,y,z)} Or do ( s=\text{expand.grid}(x=x,y=y), ) and then \texttt{wireframe(z<del>x*y,s)} or \texttt{wireframe(z</del>x*y,s,shade=TRUE)} (Note: \texttt{wireframe} is part of the \texttt{lattice} package; see item 338 for how to load packages). If you have vectors ( x, y, ) and ( z ) all the same length, you can also do \texttt{symbols(x,y,z)}.</td>
</tr>
<tr>
<td>258</td>
<td>Set axis ranges in a figure window</td>
<td>\texttt{axis([x1 x2 y1 y2])}</td>
<td>You have to do this when you make the plot, e.g. \texttt{plot(x,y,xlim=c(x1,x2), ylim=c(y1,y2))}</td>
</tr>
<tr>
<td>259</td>
<td>Add title to plot</td>
<td>\texttt{title('somestring')} \texttt{title(main='somestring')} adds a main title, \texttt{title(sub='somestring')} adds a subtitle. You can also include \texttt{main=} and \texttt{sub=} arguments in a \texttt{plot} command.</td>
<td></td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>---</td>
</tr>
<tr>
<td>261</td>
<td>Include Greek letters or symbols in plot axis labels</td>
<td>You can use basic TeX commands, e.g. <code>plot(x,y); xlabel(‘\phi^2 + \mu_{i,j}’)</code> or <code>xlabel(‘fecundity \phi’)</code> See also <code>help tex</code> and parts of <code>doc text_pprops</code> for more about building labels using general LaTeX commands</td>
<td><code>plot(x,y,xlab=expression(phi^2 + mu['i','j']))</code> or <code>plot(x,y,xlab=expression(paste('fecundity ','phi'))</code> See also <code>help(plotmath)</code> and p. 98 of the R Graphics book by Paul Murrell for more.</td>
</tr>
<tr>
<td>262</td>
<td>Change font size to 16 in plot labels</td>
<td>For the legends and numerical axis labels, use <code>set(gca, 'FontSize', 16)</code>, and for text labels on axes do e.g. <code>xlabel('my x var', 'FontSize', 16)</code></td>
<td>For on-screen graphics, do <code>par(ps=16)</code> followed by e.g. a <code>plot</code> command. For PostScript or PDF plots, add a <code>pointsize=16</code> argument, e.g. <code>pdf('myfile.pdf', width=8, height=8, pointsize=16)</code> (see items 278 and 279)</td>
</tr>
<tr>
<td>263</td>
<td>Add grid lines to plot</td>
<td><code>grid on</code> (and <code>grid off</code> to turn off)</td>
<td><code>grid()</code> Note that if you’ll be printing the plot, the default style for grid-lines is to use gray dotted lines, which are almost invisible on some printers. You may want to do e.g. <code>grid(1ty='dashed', col='black')</code> to use black dashed lines which are easier to see.</td>
</tr>
<tr>
<td>264</td>
<td>Add a text label to a plot</td>
<td><code>text(x,y,'hello')</code></td>
<td><code>text(x,y,'hello')</code></td>
</tr>
<tr>
<td>265</td>
<td>Add set of text labels to a plot. <code>xv</code> and <code>yv</code> are vectors.</td>
<td><code>s=’hi’, ’there’; text(xv,yv,s)</code></td>
<td><code>s=c(’hi’, ’there’); text(xv,yv,s)</code></td>
</tr>
<tr>
<td>266</td>
<td>Add an arrow to current plot, with tail at <code>(xt, yt)</code> and head at <code>(xh, yh)</code></td>
<td><code>annotation(’arrow’, [xt xh], [yt yh])</code> Note: coordinates should be normalized figure coordinates, not coordinates within your displayed axes. Find and download from The Mathworks the file <code>dsxy2figxy.m</code> which converts for you, then do this: <code>[fx,fy]=dsxy2figxy([xt xh], [yt yh]); annotation(’arrow’, fx, fy)</code></td>
<td><code>arrows(xt, yt, xh, yh)</code></td>
</tr>
<tr>
<td>267</td>
<td>Add a double-headed arrow to current plot, with coordinates <code>(x0, y0)</code> and <code>(x1, y1)</code></td>
<td><code>annotation(’doublearrow’, [x0 x1], [y0 y1])</code> See note in previous item about normalized figure coordinates.</td>
<td><code>arrows(x0, y0, x1, y1, code=3)</code></td>
</tr>
<tr>
<td>268</td>
<td>Add figure legend to top-left corner of plot</td>
<td><code>legend(’first’, ’second’, ’Location’, ’NorthWest’)</code></td>
<td><code>legend(’topleft’, legend=c(’first’, ’second’), col=c(’red’, ’blue’), pch=c(’*’, ’o’))</code></td>
</tr>
</tbody>
</table>

MATLAB note: sometimes you build a graph piece-by-piece, and then want to manually add a legend which doesn’t correspond with the order you put things in the plot. You can manually construct a legend by plotting “invisible” things, then building the legend using them. E.g. to make a legend with black stars and solid lines, and red circles and dashed lines: `h1=plot(0,0,’k*-’); set(h1,’Visible’, ’off’); h2=plot(0,0,’k+-’); set(h2,’Visible’, ’off’); legend([h1 h2], ’blah’, ’whoa’). Just be sure to choose coordinates for your “invisible” points within the current figure’s axis ranges.
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>269</td>
<td>Adding more things to a figure</td>
<td><code>hold on</code> means everything plotted from now on in that figure window is added to what’s already there. <code>hold off</code> turns it off. <code>clf</code> clears the figure and turns off <code>hold</code>.</td>
<td><code>points(...)</code> and <code>lines(...)</code> work like <code>plot</code>, but add to what’s already in the figure rather than clearing the figure first. <code>points</code> and <code>lines</code> are basically identical, just with different default plotting styles. Note: axes are not recalculated/redrawn when adding more things to a figure.</td>
</tr>
<tr>
<td>270</td>
<td>Plot multiple data sets at once</td>
<td><code>plot(x,y)</code> where <code>x</code> and <code>y</code> are 2-D matrices. Each column of <code>x</code> is plotted against the corresponding column of <code>y</code>. If <code>x</code> has only one column, it will be re-used.</td>
<td><code>matplot(x,y)</code> where <code>x</code> and <code>y</code> are 2-D matrices. Each column of <code>x</code> is plotted against the corresponding column of <code>y</code>. If <code>x</code> has only one column, it will be re-used.</td>
</tr>
<tr>
<td>271</td>
<td>Plot sin(2x) for x between 7 and 18</td>
<td><code>fplot('sin(2*x)', [7 18])</code></td>
<td><code>curve(sin(2*x), 7, 18, 200)</code> makes the plot, by sampling the value of the function at 200 values between 7 and 18 (if you don’t specify the number of points, 101 is the default). You could do this manually yourself via commands like <code>tmpx=seq(7,18,len=200); plot(tmpx, sin(2*tmpx))</code>.</td>
</tr>
<tr>
<td>272</td>
<td>Plot color image of integer values in matrix <code>A</code></td>
<td><code>image(A)</code> to use array values as raw indices into colormap, or <code>imagesc(A)</code> to automatically scale values first (these both draw row 1 of the matrix at the top of the image); or <code>pcolor(A)</code> (draws row 1 of the matrix at the bottom of the image). After using <code>pcolor</code>, try the commands <code>shading flat</code> or <code>shading interp</code>.</td>
<td><code>image(A)</code> (it rotates the matrix 90 degrees counterclockwise: it draws row 1 of <code>A</code> as the left column of the image, and column 1 of <code>A</code> as the bottom row of the image, so the row number is the <code>x</code> coord and column number is the <code>y</code> coord). It also rescales colors. If you are using a colormap with <code>k</code> entries, but the value <code>k</code> does not appear in <code>A</code>, use <code>image(A,zlim=c(1,k))</code> to avoid rescaling of colors. Or e.g. <code>image(A,zlim=c(0,k-1))</code> if you want values 0 through <code>k-1</code> to be plotted using the <code>k</code> colors.</td>
</tr>
<tr>
<td>273</td>
<td>Add colorbar legend to image plot</td>
<td><code>colorbar</code>, after using <code>image</code> or <code>pcolor</code>.</td>
<td>Use <code>filled.contour(A)</code> rather than <code>image(A)</code>, although it “blurs” the data via interpolation, or use <code>levelplot(A)</code> from the <code>lattice</code> package (see item 338 for how to load packages). To use a colormap with the latter, do e.g. <code>levelplot(A,col.regions=terrain.colors(100))</code>.</td>
</tr>
<tr>
<td>274</td>
<td>Set colormap in image</td>
<td><code>colormap(hot)</code>. Instead of <code>hot</code>, you can also use <code>gray</code>, <code>flag</code>, <code>jet</code> (the default), <code>cool</code>, <code>bone</code>, <code>copper</code>, <code>pink</code>, <code>hsv</code>, <code>prism</code>. By default, the length of the new colormap is the same as the currently-installed one; use e.g. <code>colormap(hot(256))</code> to specify the number of entries.</td>
<td><code>image(A,col=terrain.colors(100))</code> The parameter 100 specifies the length of the colormap. Other colormaps are <code>heat.colors()</code>, <code>topo.colors()</code>, and <code>cm.colors()</code>.</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
<td>----</td>
</tr>
<tr>
<td>275</td>
<td>Build your own colormap using Red/Green/Blue triplets</td>
<td>Use an $n \times 3$ matrix; each row gives R,G,B intensities between 0 and 1. Can use as argument with <code>colormap</code>. E.g. for 2 colors: \texttt{mycmap} = \begin{bmatrix} 0.5 &amp; 0.8 &amp; 0.2 \ 0.2 &amp; 0.2 &amp; 0.7 \end{bmatrix}</td>
<td>Use a vector of hexadecimal strings, each beginning with <code>#</code> and giving R,G,B intensities between 00 and FF. E.g. c\texttt{('#80CC33','#3333B3')}; can use as argument to \texttt{col} parameter to \texttt{image}. You can build such a vector of strings from vectors of Red, Green, and Blue intensities (each between 0 and 1) as follows (for a 2-color example): r=c\texttt{(0.5,0.2)}; g=c\texttt{(0.8,0.2)}; b=c\texttt{(0.2,0.7)}; \texttt{mycolors=rgb(r,g,b)}.</td>
</tr>
</tbody>
</table>

MATLAB plotting specifications, for use with \texttt{plot}, \texttt{fplot}, \texttt{semilogx}, \texttt{semilogy}, \texttt{loglog}, etc:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Color</th>
<th>Symbol</th>
<th>Marker</th>
<th>Symbol</th>
<th>Linestyle</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>blue</td>
<td>.</td>
<td>point (.)</td>
<td>-</td>
<td>solid line</td>
</tr>
<tr>
<td>g</td>
<td>green</td>
<td>o</td>
<td>circle (○)</td>
<td>:</td>
<td>dotted line</td>
</tr>
<tr>
<td>r</td>
<td>red</td>
<td>x</td>
<td>cross (×)</td>
<td>-.</td>
<td>dash-dot line</td>
</tr>
<tr>
<td>c</td>
<td>cyan</td>
<td>+</td>
<td>plus sign (+)</td>
<td>--</td>
<td>dashed line</td>
</tr>
<tr>
<td>m</td>
<td>magenta</td>
<td>*</td>
<td>asterisk (*)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>yellow</td>
<td>s</td>
<td>square (□)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>black</td>
<td>d</td>
<td>diamond (◇)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>white</td>
<td>v</td>
<td>triangle down (▽)</td>
<td>^</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
<td></td>
<td>triangle up (△)</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
<td>triangle right (⊳)</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td></td>
<td>pentagram star</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td>hexagram star</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\texttt{R} plotting specifications for \texttt{col} (color), \texttt{pch} (plotting character), and \texttt{type} arguments, for use with \texttt{plot}, \texttt{matplot}, \texttt{points}, and \texttt{lines}:

<table>
<thead>
<tr>
<th>col</th>
<th>Description</th>
<th>pch</th>
<th>Description</th>
<th>type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'blue'</td>
<td>Blue</td>
<td>'a'</td>
<td>a (similarly for other characters, but see &quot;.&quot; below for an exception</td>
<td>p</td>
<td>points</td>
</tr>
<tr>
<td>'green'</td>
<td>Green</td>
<td>0</td>
<td>open square</td>
<td>1</td>
<td>lines</td>
</tr>
<tr>
<td>'red'</td>
<td>Red</td>
<td>1</td>
<td>open circle</td>
<td>b</td>
<td>both</td>
</tr>
<tr>
<td>'cyan'</td>
<td>Cyan</td>
<td>2</td>
<td>triangle point-up</td>
<td>c</td>
<td>lines part only of “b”</td>
</tr>
<tr>
<td>'magenta'</td>
<td>Magenta</td>
<td>3</td>
<td>+ (plus)</td>
<td>o</td>
<td>lines, points overplotted</td>
</tr>
<tr>
<td>'yellow'</td>
<td>Yellow</td>
<td>4</td>
<td>× (cross)</td>
<td>h</td>
<td>histogram-like lines</td>
</tr>
<tr>
<td>'black'</td>
<td>Black</td>
<td>5</td>
<td>diamond</td>
<td>s</td>
<td>steps</td>
</tr>
<tr>
<td>'#RRGGBB'</td>
<td>hexadecimal specification of Red, Green, Blue</td>
<td>6</td>
<td>triangle point-down</td>
<td>S</td>
<td>another kind of steps</td>
</tr>
</tbody>
</table>

(Other names) See \texttt{colors()} for list of available color names. 

\texttt{'}\texttt{.}\texttt{'} rectangle of size 0.01 inch, 1 pixel, or 1 point (1/72 inch) depending on device; \texttt{n} no plotting (can be useful for setting up axis ranges, etc.)

(See table on next page for more)
R plotting specifications for `lty` (line-type) argument, for use with `plot`, `matplot`, `points`, and `lines`:

<table>
<thead>
<tr>
<th>lty</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>blank</td>
</tr>
<tr>
<td>1</td>
<td>solid</td>
</tr>
<tr>
<td>2</td>
<td>dashed</td>
</tr>
<tr>
<td>3</td>
<td>dotted</td>
</tr>
<tr>
<td>4</td>
<td>dotdash</td>
</tr>
<tr>
<td>5</td>
<td>longdash</td>
</tr>
<tr>
<td>6</td>
<td>twodash</td>
</tr>
</tbody>
</table>

R plotting characters, i.e. values for `pch` argument (from the book *R Graphics*, by Paul Murrell, Chapman & Hall / CRC, 2006)
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>276</td>
<td>Divide up a figure window into smaller sub-figures</td>
<td><code>subplot(m,n,k)</code> divides the current figure window into an $m \times n$ array of subplots, and draws in subplot number $k$ as numbered in “reading order,” i.e. left-to-right, top-to-bottom. E.g. <code>subplot(2,3,4)</code> selects the first sub-figure in the second row of a $2 \times 3$ array of sub-figures. You can do more complex things, e.g. <code>subplot(5,5,[1 2 6 7])</code> selects the first two subplots in the first row, and first two subplots in the second row, i.e. gives you a bigger subplot within a $5 \times 5$ array of subplots. (If you that command followed by e.g. <code>subplot(5,5,3)</code> you’ll see what’s meant by that.)</td>
<td>There are several ways to do this, e.g. using <code>layout</code> or <code>split.screen</code>, although they aren’t quite as friendly as MATLAB’s. E.g. if you let $A = \begin{bmatrix} 1 &amp; 1 &amp; 2 \ 1 &amp; 1 &amp; 3 \ 4 &amp; 5 &amp; 6 \end{bmatrix}$, then <code>layout(A)</code> will divide the figure into 6 sub-figures: you can imagine the figure divide into a $3 \times 3$ matrix of smaller blocks; sub-figure 1 will take up the upper-left $2 \times 2$ portion, and sub-figures 2–6 will take up smaller portions, according to the positions of those numbers in the matrix $A$. Consecutive plotting commands will draw into successive sub-figures; there doesn’t seem to be a way to explicitly specify which sub-figure to draw into next. To use <code>split.screen</code>, you can do e.g. <code>split.screen(c(2,1))</code> to split into a $2 \times 1$ matrix of subfigures (numbered 1 and 2). Then <code>split.screen(c(1,3),2)</code> splits sub-figure 2 into a $1 \times 3$ matrix of smaller sub-figures (numbered 3, 4, and 5). <code>screen(4)</code> will then select sub-figure number 4, and subsequent plotting commands will draw into it. A third way to accomplish this is via the commands <code>par(mfrow=)</code> or <code>par(mfcol=)</code> to split the figure window, and <code>par(mfg=)</code> to select which sub-figure to draw into. Note that the above methods are all incompatible with each other.</td>
</tr>
<tr>
<td>277</td>
<td>Force graphics windows to update</td>
<td><code>drawnow</code> (MATLAB normally only updates figure windows when a script/function finishes and returns control to the MATLAB prompt, or under a couple of other circumstances. This forces it to update figure windows to reflect any recent plotting commands.)</td>
<td>R automatically updates graphics windows even before functions/scripts finish executing, so it’s not necessary to explicitly request it. But note that some graphics functions (particularly those in the lattice package) don’t display their results when called from scripts or functions; e.g. rather than <code>levelplot(...)</code> you need to do <code>print(levelplot(...))</code>. Such functions will automatically display their plots when called interactively from the command prompt.</td>
</tr>
</tbody>
</table>
### 7.2 Printing/saving graphics

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>278</td>
<td>To print/save to a PDF file named <strong>fname.pdf</strong></td>
<td><code>print -dpdf fname</code> saves the contents of currently active figure window</td>
<td>First do <code>pdf('fname.pdf')</code>. Then, do various plotting commands to make your image, as if you were plotting in a window. Finally, do <code>dev.off()</code> to close/save the PDF file. To print the contents of the active figure window, do <code>dev.copy(device='pdf', file='fname.pdf'); dev.off()</code>. (But this will not work if you’ve turned off the display list via <code>dev.control(displaylist='inhibit')</code>). You can also simply use <code>dev.copy2pdf(file='fname.pdf')</code>.</td>
</tr>
<tr>
<td>279</td>
<td>To print/save to a PostScript file <strong>fname.ps</strong> or <strong>fname.eps</strong></td>
<td><code>print -dps fname</code> for black &amp; white PostScript; <code>print -dpsc fname</code> for color PostScript; <code>print -deps fname</code> for black &amp; white Encapsulated PostScript; <code>print -depsc fname</code> for color Encapsulated PostScript. The first two save to <strong>fname.ps</strong>, while the latter two save to <strong>fname.eps</strong>.</td>
<td><code>postscript('fname.eps'</code>, followed by your plotting commands, followed by <code>dev.off()</code> to close/save the file. Note: you may want to use <code>postscript('fname.eps', horizontal=FALSE)</code> to save your figure in portrait mode rather than the default landscape mode. To print the contents of the active figure window, do <code>dev.copy(device='postscript', file='fname.eps'); dev.off()</code>. (But this will not work if you’ve turned off the display list via <code>dev.control(displaylist='inhibit')</code>). You can also include the <code>horizontal=FALSE</code> argument with <code>dev.copy()</code>. The command <code>dev.copy2eps(file='fname.eps')</code> also saves in portrait mode.</td>
</tr>
<tr>
<td>280</td>
<td>To print/save to a JPEG file <strong>fname.jpg</strong> with jpeg quality = 90 (higher quality looks better but makes the file larger)</td>
<td><code>print -djpeg90 fname</code></td>
<td><code>jpeg('fname.jpg', quality=90)</code>, followed by your plotting commands, followed by <code>dev.off()</code> to close/save the file.</td>
</tr>
</tbody>
</table>
7.3 Animating cellular automata / lattice simulations

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>281</td>
<td>To display images of cellular automata or other lattice simulations while running in real time</td>
<td>Repeatedly use either <code>pcolor</code> or <code>image</code> to display the data. Don’t forget to call <code>drawnow</code> as well, otherwise the figure window will not be updated with each image.</td>
<td>If you simply call <code>image</code> repeatedly, there is a great deal of flickering/flashing. To avoid this, after drawing the image for the first time using e.g. <code>image(A)</code>, from then on only use <code>image(A, add=TRUE)</code>, which avoids redrawing the entire image (and the associated flicker). However, this will soon consume a great deal of memory, as all drawn images are saved in the image buffer. There are two solutions to that problem: (1) every k time steps, leave off the “add=TRUE” argument to flush the image buffer (and get occasional flickering), where you choose k to balance the flickering vs. memory-usage tradeoff; or (2) after drawing the first image, do <code>dev.control(displaylist=‘inhibit’)</code> to prohibit retaining the data. However, the latter solution means that after the simulation is done, the figure window will not be redrawn if it is resized, or temporarily obscured by another window. (A call to <code>dev.control(displaylist=‘enable’)</code> and then one final <code>image(A)</code> at the end of the simulation will re-enable re-drawing after resizing or obscuring, without consuming extra memory.)</td>
</tr>
</tbody>
</table>
## 8 Working with files

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>282</td>
<td>Create a folder (also known as a “directory”)</td>
<td>mkdir dirname</td>
<td>dir.create('dirname')</td>
</tr>
<tr>
<td>283</td>
<td>Set/change working directory</td>
<td>cd dirname</td>
<td>setwd('dirname')</td>
</tr>
<tr>
<td>284</td>
<td>Get working directory</td>
<td>pwd</td>
<td>getwd()</td>
</tr>
<tr>
<td>285</td>
<td>See list of files in current working directory</td>
<td>dir</td>
<td>dir()</td>
</tr>
<tr>
<td>286</td>
<td>Run commands in file ‘foo.m’ or ‘foo.R’ respectively</td>
<td>foo</td>
<td>source('foo.R')</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>287</td>
<td>Read data from text file “data.txt” into matrix A</td>
<td>A=load('data.txt') or A=importdata('data.txt')</td>
<td>A=as.matrix(read.table('data.txt')) This will ignore comments (anything on a line following a “#” character). To ignore comments indicated by “%”, do A=as.matrix(read.table('data.txt', comment.char='%'))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>288</td>
<td>Read data from text file “data.txt” into matrix A, skipping the first s lines of the file</td>
<td>tmp=importdata('data.txt', ' ',s); a=tmp.data</td>
<td>A=as.matrix(read.table('data.txt', skip=s))</td>
</tr>
<tr>
<td>289</td>
<td>Write data from matrix A into text file “data.txt”</td>
<td>save data.txt A -ascii</td>
<td>write(t(A), file='data.txt', ncolumn=dim(A)[2])</td>
</tr>
<tr>
<td>290</td>
<td>Save all variables/data in the workspace to a file foo (with appropriate suffix)</td>
<td>save foo.mat (MATLAB recognizes files with “.mat” suffix as binary save files). Just save with no arguments saves to matlab.mat</td>
<td>save.image(file='foo.rda') (You may use whatever filename suffix you like.) Just save.image() with no arguments saves to .RData</td>
</tr>
<tr>
<td>291</td>
<td>Reload all variables/data from a saved file foo (with appropriate suffix)</td>
<td>load foo.mat. Just load with no arguments tries to load from matlab.mat</td>
<td>load('foo.rda')</td>
</tr>
</tbody>
</table>
# 9 Miscellaneous

## 9.1 Variables

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>292</td>
<td>Assigning to variables</td>
<td><code>x = 5</code></td>
<td><code>x &lt;- 5</code> or <code>x = 5</code> Note: for compatibility with S-plus, many people prefer the first form.</td>
</tr>
<tr>
<td>293</td>
<td>From within a function, assign a value to variable <code>y</code> in the base environment (i.e. the command prompt environment)</td>
<td><code>assignin('base', 'y', 7)</code></td>
<td><code>y &lt;&lt;- 7</code></td>
</tr>
<tr>
<td>294</td>
<td>From within a function, access the value of variable <code>y</code> in the base environment (i.e. the command prompt environment)</td>
<td><code>evalin('base', 'y')</code></td>
<td><code>get('y', envir=globalenv())</code> Though note that inside a function, if there isn't a local variable <code>y</code>, then just the expression <code>y</code> will look for one in the base environment, but if there is a local <code>y</code> then that one will be used instead.</td>
</tr>
<tr>
<td>295</td>
<td>Short list of defined variables</td>
<td><code>who</code></td>
<td><code>ls()</code></td>
</tr>
<tr>
<td>296</td>
<td>Long list of defined variables</td>
<td><code>whos</code></td>
<td><code>ls.str()</code></td>
</tr>
<tr>
<td>297</td>
<td>See detailed info about the variable <code>ab</code></td>
<td><code>whos ab</code></td>
<td><code>str(ab)</code></td>
</tr>
<tr>
<td>298</td>
<td>See detailed info about all variables with “ab” in their name</td>
<td><code>whos *ab*</code></td>
<td><code>ls.str(pattern='ab')</code></td>
</tr>
<tr>
<td>299</td>
<td>Open graphical data editor, to edit the value of variable <code>A</code> (useful for editing values in a matrix, though it works for non-matrix variables as well)</td>
<td><code>openvar(A)</code>, or double-click on the variable in the Workspace pane (if it’s being displayed) of your MATLAB desktop</td>
<td><code>fix(A)</code></td>
</tr>
<tr>
<td>300</td>
<td>Clear one variable</td>
<td><code>clear x</code></td>
<td><code>rm(x)</code></td>
</tr>
<tr>
<td>301</td>
<td>Clear two variables</td>
<td><code>clear x y</code></td>
<td><code>rm(x,y)</code></td>
</tr>
<tr>
<td>302</td>
<td>Clear all variables</td>
<td><code>clear all</code></td>
<td><code>rm(list=ls())</code></td>
</tr>
<tr>
<td>303</td>
<td>See if variable <code>x</code> exists (the commands given can also take more arguments to be more specific)</td>
<td><code>exist('x')</code></td>
<td><code>exists('x')</code></td>
</tr>
<tr>
<td>304</td>
<td>See what type of object <code>x</code> is</td>
<td><code>class(x)</code></td>
<td><code>class(x)</code>, <code>typeof(x)</code>, and <code>mode(x)</code> give different aspects of the “type” of <code>x</code></td>
</tr>
<tr>
<td>305</td>
<td>(Variable names)</td>
<td>Variable names must begin with a letter, but after that they may contain any combination of letters, digits, and the underscore character. Names are case-sensitive.</td>
<td>Variable names may contain letters, digits, the period, and the underscore character. They cannot begin with a digit or underscore, or with a period followed by a digit. Names are case-sensitive.</td>
</tr>
<tr>
<td>306</td>
<td>Result of last command</td>
<td><code>ans</code> contains the result of the last command which did not assign its value to a variable. E.g. after <code>2+5; x=3</code>, then <code>ans</code> will contain 7.</td>
<td><code>.Last.value</code> contains the result of the last command, whether or not its value was assigned to a variable. E.g. after <code>2+5; x=3</code>, then <code>.Last.value</code> will contain 3.</td>
</tr>
</tbody>
</table>
### 9.2 Strings and Misc.

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
</table>
| 307 | Line continuation | If you want to break up a MATLAB command over more than one line, end all but the last line with three periods: “…” E.g.:  
\[
\begin{align*}
    x &= 3 + \ldots \\
    &\quad 4 \\
\text{or} \\
    x &= 3 \ldots \\
    &\quad + 4
\end{align*}
\] | In R, you can spread commands out over multiple lines, and nothing extra is necessary. R will continue reading input until the command is complete. However, this only works when the syntax makes it clear that the first line was not complete. E.g.:  
\[
\begin{align*}
    x &= 3 + \\
    &\quad 4 \\
\end{align*}
\] works, but  
\[
\begin{align*}
    x &= 3 \\
    &\quad + 4
\end{align*}
\] does not treat the second line as a continuation of the first. |
| 308 | Controlling formatting of output | **format short g** and **format long g** are handy; see help format | **options(digits=6)** tells R you’d like to use 6 digits of precision in values it displays (it is only a suggestion, not strictly followed) |
| 309 | Exit the program | **quit** or **exit** | **q()** or **quit()** |
| 310 | Comments | `% this is a comment` | `# this is a comment` |
| 311 | Display a string | disp('hi there') or to omit trailing newline use fprintf('hi there') | **print('hi there')** Note: to avoid having double-quotes around the displayed string, do **print('hi there', quote=FALSE)** or **print(noquote('hi there')).** Or use **cat('hi there').** But note that use of **cat** in a script won’t put newlines at the end of each string. To achieve that, either do **cat('hi there\n')** or **cat('hi there',fill=TRUE)** |
| 312 | Display a string containing single quotes | disp('It''s nice') or to omit trailing newline use fprintf('It''s nice') | **print('It\'s nice')** or **print("It's nice")** Also see cat in item above. |
| 313 | Give prompt and read numerical input from user | \[x = \text{input('Enter data:')}\] | **print('Enter data:'); x=scan()** But note: if in a script and you use the Edit → Execute menu item to run it, the selected text after the **scan** statement will be used as source for the input, rather than keyboard. Also see cat 2 items above. |
| 314 | Give prompt and read character (string) input from user | \[x = \text{input('Enter string:'},'s')\] | \[x = \text{readline('Enter string:')}\] |
| 315 | Concatenate strings | \[v = \{\text{"two hal"}, \text{"yes"}\};
\text{strcat(v{:});}\] But note that this drops trailing spaces on strings. To avoid that, instead do \[\text{strcat([v{:}]);}\] | **paste('two hal', 'yes', sep='')** |
| 316 | Concatenate strings stored in a vector | \[v = \{\text{"two "}, \text{"halves"}\};
\text{strcat(v{:});}\] But note that this drops trailing spaces on strings. To avoid that, instead do \[\text{strcat([v{:}]);}\] | \[v = c(\text{"two "}, \text{"halves"});
\text{paste(v, collapse='')}\] |
| 317 | Extract substring of a string | \[\text{text1=\"hi there\";}
\text{text2=text(2:6);}\] | **text1=\"hi there\";**
**text2=substr(text1,2,6)** |
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>Determine whether elements of a vector are in a set, and give positions of corresponding elements in the set.</td>
<td>( x = { \text{’a’}, \text{’aa’}, \text{’bc’}, \text{’c’} }; \ y = { \text{’da’}, \text{’a’}, \text{’bc’}, \text{’a’}, \text{’bc’}, \text{’aa’} }; \ [\text{tf}, \text{loc}]=\text{ismember}(x,y) ) Then <code>loc</code> contains the locations of last occurrences of elements of <code>x</code> in the set <code>y</code>, and 0 for unmatched elements.</td>
<td>( x = \text{c(’a’, ’aa’, ’bc’, ’c’)}; \ y = \text{c(’da’, ’a’, ’bc’, ’a’, ’bc’, ’aa’)}; \text{loc} = \text{match}(x,y) ) Then <code>loc</code> contains the locations of first occurrences of elements of <code>x</code> in the set <code>y</code>, and <code>NA</code> for unmatched elements.</td>
</tr>
<tr>
<td>319</td>
<td>Find indices of regular expression pattern <code>p</code> in string <code>s</code></td>
<td><code>v=regexp(s,p)</code></td>
<td><code>v=gregexpr(p,s)[[1]]</code> (The returned vector also has a “match.length” attribute giving lengths of the matches; this attribute can be removed via <code>attributes(v)=NULL</code>.)</td>
</tr>
<tr>
<td>320</td>
<td>Perform some commands only if the regular expression <code>p</code> is contained in the string <code>s</code></td>
<td><code>if (regexp(s,p)) \{ \text{...commands...} \}</code></td>
<td><code>if (grepl(p,s)) \{ \text{...commands...} \}</code></td>
</tr>
<tr>
<td>321</td>
<td>Convert number to string</td>
<td><code>num2str(x)</code></td>
<td><code>as.character(x)</code></td>
</tr>
<tr>
<td>322</td>
<td>Use <code>sprintf</code> to create a formatted string. Use <code>%d</code> for integers (“d” stands for “decimal”), i.e. base 10), <code>%f</code> for floating-point numbers, <code>%e</code> for scientific-notation floating point, <code>%g</code> to automatically choose <code>%e</code> or <code>%f</code> based on the value. You can specify field-widths/precisions, e.g. <code>%5d</code> for integers with padding to 5 spaces, or <code>%e.7f</code> for floating-point with 7 digits of precision. There are many other options too; see the docs.</td>
<td><code>x=2; y=3.5; s=sprintf(’x is %d, y is %g’, x, y)</code></td>
<td><code>x=2; y=3.5 s=sprintf(’x is %d, y is %g’, x, y)</code></td>
</tr>
<tr>
<td>323</td>
<td>Machine epsilon ( \epsilon_{\text{mach}} ), i.e. difference between 1 and the next largest double-precision floating-point number</td>
<td><code>eps</code> (See <code>help eps</code> for various other things <code>eps</code> can give.)</td>
<td><code>.Machine$double.eps</code></td>
</tr>
<tr>
<td>324</td>
<td>Pause for ( x ) seconds</td>
<td><code>pause(x)</code></td>
<td><code>Sys.sleep(x)</code></td>
</tr>
<tr>
<td>325</td>
<td>Wait for user to press any key</td>
<td><code>pause</code></td>
<td>Don’t know of a way to do this in R, but <code>scan(quiet=TRUE)</code> will wait until the user presses the Enter key</td>
</tr>
<tr>
<td>326</td>
<td>Produce a beep (or possibly a visual signal, depending on preferences set)</td>
<td><code>beep</code></td>
<td><code>alarm()</code></td>
</tr>
<tr>
<td>327</td>
<td>Measure CPU time used to do some commands</td>
<td><code>t1=cputime; \text{...commands...} ; \text{cputime}=t1</code></td>
<td><code>ti=proc.time(); \text{...commands...} ; (\text{proc.time()}-\text{ti})[1]</code></td>
</tr>
<tr>
<td>328</td>
<td>Measure elapsed (“wall-clock”) time used to do some commands</td>
<td><code>tic; \text{...commands...} ; \text{toc or t1=clock; \text{...commands...} ; etime(clock,t1)}</code></td>
<td><code>ti=proc.time(); \text{...commands...} ; (\text{proc.time()}-\text{t1})[3]</code></td>
</tr>
<tr>
<td>329</td>
<td>Print an error message and interrupt execution</td>
<td><code>error(’Problem!’)</code></td>
<td><code>stop(’Problem!’)</code></td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>------------------------------</td>
<td>--------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>330</td>
<td>Print a warning message</td>
<td><code>warning('Smaller problem!')</code></td>
<td><code>warning('Smaller problem!')</code></td>
</tr>
<tr>
<td>331</td>
<td>Putting multiple statements on one line</td>
<td>Separate statements by commas or semicolons. A semicolon at the end of a statement suppresses display of the results (also useful even with just a single statement on a line), while a comma does not.</td>
<td>Separate statements by semicolons.</td>
</tr>
<tr>
<td>332</td>
<td>Evaluate contents of a string s as command(s).</td>
<td><code>eval(s)</code></td>
<td><code>eval(parse(text=s))</code></td>
</tr>
<tr>
<td>333</td>
<td>Get a command prompt for debugging, while executing a script or function. While at that prompt, you can type expressions to see the values of variables, etc.</td>
<td>Insert the command <code>keyboard</code> in your file. Note that your prompt will change to <code>K&gt;&gt;</code>. When you are done debugging and want to continue executing the file, type <code>return</code>.</td>
<td>Insert the command <code>browser()</code> in your file. Note that your prompt will change to <code>Browse[1]&gt;</code>. When you are done debugging and want to continue executing the file, either type <code>c</code> or just press return (i.e. enter a blank line). Note, if you type <code>n</code>, you enter the step debugger.</td>
</tr>
<tr>
<td>334</td>
<td>Show where a command is which <code>sqrt</code> shows you where the file defining the <code>sqrt</code> function is (but note that many basic functions are “built in,” so the MATLAB function file is really just a stub containing documentation). This is useful if a command is doing something strange, e.g. <code>sqrt</code> isn’t working. If you’ve accidentally defined a variable called <code>sqrt</code>, then <code>which sqrt</code> will tell you, so you can <code>clear sqrt</code> to erase it so that you can go back to using the function <code>sqrt</code>.</td>
<td>R does not execute commands directly from files, so there is no equivalent command. See item 286 for reading command files in R.</td>
<td></td>
</tr>
<tr>
<td>335</td>
<td>Query/set the search path. <code>path</code> displays the current search path (the list of places MATLAB searches for commands you enter). To add a directory <code>~/foo</code> to the beginning of the search path, do <code>addpath </code><del>/foo<code> -begin</code> or to add it to the end of the path, do <code>addpath </code></del>/foo<code> -end</code> (Note: you should generally add the full path of a directory, i.e. in Linux or Mac OS-X something like <code>~/foo</code> as above or of the form <code>/usr/local/lib/foo</code>, while under Windows it would be something like <code>C:/foo</code>)</td>
<td>R does not use a search path to look for files. See item 286 for reading command files in R.</td>
<td></td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>MATLAB</td>
<td>R</td>
</tr>
<tr>
<td>-----</td>
<td>---------------------</td>
<td>----------------------------------------------------------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>336</td>
<td>Startup sequence</td>
<td>If a file <strong>startup.m</strong> exists in the startup directory for MATLAB, its contents are executed. (See the MATLAB docs for how to change the startup directory.)</td>
<td>If a file <strong>.Rprofile</strong> exists in the current directory or the user’s home directory (in that order), its contents are sourced; saved data from the file <strong>.RData</strong> (if it exists) are then loaded. If a function <strong>.First()</strong> has been defined, it is then called (so the obvious place to define this function is in your <strong>.Rprofile</strong> file).</td>
</tr>
<tr>
<td>337</td>
<td>Shutdown sequence</td>
<td>Upon typing <strong>quit</strong> or <strong>exit</strong>, MATLAB will run the script <strong>finish.m</strong> if present somewhere in the search path.</td>
<td>Upon typing <strong>q()</strong> or <strong>quit()</strong>, R will call the function <strong>.Last()</strong> if it has been defined (one obvious place to define it would be in the <strong>.Rprofile</strong> file).</td>
</tr>
<tr>
<td>338</td>
<td>Install and load a package.</td>
<td>MATLAB does not have packages. It has toolboxes, which you can purchase and install. “Contributed” code (written by end users) can simply be downloaded and put in a directory which you then add to MATLAB’s path (see item 335 for how to add things to MATLAB’s path).</td>
<td>To install e.g. the <strong>deSolve</strong> package, you can use the command <strong>install.packages(‘deSolve’)</strong>. You then need to load the package in order to use it, via the command <strong>library(‘deSolve’)</strong>. When running R again later you’ll need to load the package again to use it, but you should not need to re-install it. Note that the <strong>lattice</strong> package is typically included with binary distributions of R, so it only needs to be loaded, not installed.</td>
</tr>
</tbody>
</table>
# 10 Spatial Modeling

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>MATLAB</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>339</td>
<td>Take an $L \times L$ matrix $A$ of 0s and 1s, and “seed” fraction $p$ of the 0s (turn them into 1s), not changing entries which are already 1.</td>
<td>$A = (A</td>
<td>(\text{rand}(L) &lt; p)) \times 1;$</td>
</tr>
<tr>
<td>340</td>
<td>Take an $L \times L$ matrix $A$ of 0s and 1s, and “kill” fraction $p$ of the 1s (turn them into 0s), not changing the rest of the entries</td>
<td>$A = (A &amp; (\text{rand}(L) &lt; 1-p)) \times 1;$</td>
<td>$A = (A &amp; (\text{matrix(runif}(L^2),L) &lt; 1-p)) \times 1$</td>
</tr>
<tr>
<td>341</td>
<td>Do “wraparound” on a coordinate $\text{newx}$ that you’ve already calculated. You can replace $\text{newx}$ with $x + dx$ if you want to do wraparound on an offset $x$ coordinate.</td>
<td>$\text{mod(newx-1,L)+1}$ Note: for portability with other languages such as C which handle MOD of negative values differently, you may want to get in the habit of instead doing $\text{mod(newx-1+L,L)+1}$</td>
<td>$((\text{newx}-1) % \ L) + 1$ Note: for portability with other languages such as C which handle MOD of negative values differently, you may want to get in the habit of instead doing $((\text{newx}-1+L) % \ L) + 1$</td>
</tr>
<tr>
<td>342</td>
<td>Randomly initialize a portion of an array: set fraction $p$ of sites in rows $iy1$ through $iy2$ and columns $ix1$ through $ix2$ equal to 1 (and set the rest of the sites in that block equal to zero). Note: this assume $iy1 &lt; iy2$ and $ix1 &lt; ix2$.</td>
<td>$dx=ix2-ix1+1; dy=iy2-iy1+1;$ $A(iy1:iy2,ix1:ix2) = \ldots (\text{rand}(dy,dx) &lt; p0) \times 1;$</td>
<td>$dx=ix2-ix1+1; dy=iy2-iy1+1;$ $A[iy1:iy2,ix1:ix2] = (\text{matrix(runif(dy*dx),dy) &lt; p0}) \times 1$</td>
</tr>
</tbody>
</table>
INDEX OF MATLAB COMMANDS AND CONCEPTS

Index of MATLAB commands and concepts

', 83
', 331
.*, 82
..., 307
./, 90
^- 94
/, 89
:, 12–14
;, 331
=, 292
[], 6–8
%, 310
&, 188, 189
^, 54, 92, 93
\, 84, 91
{, 49

abs, 55, 74
acos, 60
acosh, 62
addpath, 335
all, 190
angle, 75
annotation, 266, 267
ans, 306
any, 191
arrows in plots, 266, 267
asin, 60
asinh, 62
assignin, 293
atan, 60
atanh, 62
average, see mean
axis, 258

bar, 247, 249, 250
beep, 326
binocdf, 232
binopdf, 225
binornd, 216
boolean tests
    scalar, 188
    vector, 189–191
break, 193

cd, 283
ceil, 66
ceil, 48
cell arrays, 48
    extracting elements of, 49
cellular automata animation, 281
chol, 100
circshift, 33
class, 304
clear, 300–302
clf, 269
clock, 328
close, 242
colon, see :
colorbar, 273
colormap
    building your own, 275
colormap, 274, 275
column vector, 7
comments, 310
complex numbers, 73–78
cond, 104–106
conj, 76
continue, 193
contour, 257
conv, 167
corr, 120–125
cos, 59
cosh, 61
cov, 118, 119
cputime, 327
cross, 80
csape, 180, 182, 183
cubic splines, 181, 182
    natural, 180
    not-a-knot, 184
    periodic, 183
cumprod, 137
cumsum, 133–136
cumulative distribution functions
    binomial, 232
    continuous uniform on interval (a, b), 236
    discrete uniform from 1..n, 237
    exponential, 234
    normal, 235
    Poisson, 233

debugging, 333
det, 86
diag, 22, 23
diff, 139
differential equations, see ode45
dir, 285
disp, 311, 312
doc, 4
dot, 79
INDEX OF MATLAB COMMANDS AND CONCEPTS

drawnow, 277, 281

echelon form, see matrix
eig, 96
element-by-element matrix operations, see matrix
else, 187
elseif, 187
end, 40
eps, 323
erf, 68
erfc, 69
erfcinv, 71
erfinv, 70
error, 329
errorbar, 251, 252
etime, 328
eval, 332
evalin, 294
exist, 303
exit, 309, 337
exp, 56
expcdf, 234
expm, 132
exppdf, 227
exprnd, 218
eye, 21

figure, 238, 239

file
restoring workspace from, 291
running commands in, 286
saving workspace to, 290
text
reading data from, 287, 288
saving data to, 289

find, 162–164
finish.m, 337
fliprl, 34
flipud, 35
floor, 65
fminbnd, 170, 171
fminsearch, 172, 173
font size in plots, 262
for, 185
format, 308
fplot, 271
fprintf, 311, 312
function
multi-variable
minimization, 172
minimization over first parameter only, 171
minimization over only some parameters, 173

single-variable
minimization, 170
user-written, 195
returning multiple values, 196

fzero, 169
gca, 262
gcf, 240
get, 241
Greek letters
in plot labels, 261
grid, 263
help, 1–3
helpbrowser, 4
helpdesk, 4
hilb, 46
hist, 165, 166, 248, 249
hold, 269

identity, see matrix
if, 186–188
imag, 78
image, 272, 281
imagesc, 272
importdata, 287, 288
ind2sub, 36
indexing
matrix, 10
with a single index, 11
vector, 9
input, 313, 314
inv, 87
inverse, see matrix
ismember, 318

keyboard, 333

legend, 268
length, 152, 154
linspace, 15
load, 287, 291
log, 57
log10, 58
log2, 58
loglog, 246
logspace, 16
lookfor, 5
lu, 97

matrix, 8
boolean operations on, 163, 164
changing shape of, 43
Cholesky factorization, 100
INDEX OF MATLAB COMMANDS AND CONCEPTS

- circular shift, 33
- condition number, 104–106
- containing all identical entries, 20
- containing all zeros, 19
- converting row, column to single index, 37
- converting single-index to row, column, 36
- cumulative sums of all elements of, 136
- cumulative sums of columns, 134
- cumulative sums of rows, 135
- determinant, 86
- diagonal, 22
- echelon form, 85
- eigenvalues and eigenvectors of, 96
- equation
  - solving, 84
- exponential of, 132
- extracting a column of, 28
- extracting a rectangular piece of, 31
- extracting a row of, 29
- extracting specified rows and columns of, 32
- “gluing” together, 24, 25
- identity, 21
- inverse, 87
- lower-triangular portion of, 44
- LU factorization, 97
- minimum of values of, 142
- minimum value of each column of, 143
- minimum value of each row of, 144
- modifying elements given lists of rows and columns, 38
- multiplication, 81
  - element-by-element, 82
- N-dimensional, 47
- norm, 103
- powers of, 93
- product
  - of all elements, 129
  - of columns of, 130
  - of rows of, 131
- QR factorization, 101
- rank, 95
- re-shaping its elements into a vector, 30
- reverse elements in columns, 35
- reverse elements in rows, 34
- Schur decomposition, 99
- singular value decomposition, 98
- size of, 149–151, 153, 154
- sum
  - of all elements, 126
  - of columns of, 127
  - of rows of, 128
- trace, 88
- transpose, 83
- upper-triangular portion of, 45
- max, see min
- mean, 107–109
- median, 117
- mesh, 257
- meshgrid, 26, 120, 256
- min, 141–144, 146–148
- mind, 145
- mkdir, 282
- mnpdf, 231
- mnrnd, 223, 224
- mod, 63, 341
- mode, 116
- mode of vector of values, 116
- modulo arithmetic, 63, 341
- multiple statements on one line, 331
- nchoosek, 72
- norm, 102, 103
- normcdf, 235
- normpdf, 228
- normrnd, 222
- num2str, 321
- numel, 153
- ode45, 197–199
- ones, 18, 20
- openvar, 299
- optimization, 170–173
- path, 335
- pause, 324, 325
- pcolor, 257, 272, 281
- perform some commands with probability p, 209
- permutation of integers 1..n, 210
- plot, 243–245, 270
  - Greek letters in axis labels, 261
- plot3, 254
- poisscdf, 233
- poisspdf, 226
- poissrnd, 217
- polar, 253
- polyfit, 176–178
- polynomial
  - least-squares fitted, 177–179
  - multiplication, 167
  - roots of, 168
- ppval, 180, 182, 183
- print, 278–280
- probability density functions
  - binomial, 225
  - continuous uniform on interval (a, b), 229
  - discrete uniform from 1..n, 230
  - exponential, 227
INDEX OF MATLAB COMMANDS AND CONCEPTS

- multinomial, 231
- normal, 228
- Poisson, 226
- prod, 129–131
- pwd, 284
- qr, 101
- quad, 174
- quit, 309, 337
- rand, 200–208, 214, 215
- random values
  - Bernoulli, 206
  - binomial, 216
  - continuous uniform distribution on interval (a, b), 203, 221
  - continuous uniform distribution on interval (0,1), 200–202
  - discrete uniform distribution from a..b, 208
  - discrete uniform distribution from 1..k, 205, 219, 220
  - discrete uniform distribution, 204
  - exponential, 218
  - general discrete distribution, 214
  - k unique values sampled from integers 1..n, 211
  - multinomial, 223, 224
  - normal, 222
  - Poisson, 217
  - setting the seed, 215
- randperm, 210, 211
- randsample, 211–213
- rank, 95
- rcond, 104
- real, 77
- regexp, 319, 320
- reshape, 43, 47
- roots
  - of general single-variable function, 169
  - polynomial, 168
- roots, 168
- round, 64
- row vector, 6
- rref, 85
- sampling values from a vector, 212, 213
- save, 289, 290
- schur, 99
- semilogx, 246
- semilogy, 246
- set, 262
- shading, 272
- sign, 67
- sin, 59
- sinh, 61
- size, 149–151
- slice, 257
- sort, 155, 156, 211
- sortrows, 157–160
- spline, 184
- splines, see cubic splines
- sprintf, 322
- sqrt, 53
- stairs, 253
- standard deviation, see std
- startup.m, 336
- std, 110–112
- stem, 253
- stop, 329
- strcat, 316
- string
  - concatenation, 315
  - converting number to, 321
  - pattern matching, 319, 320
  - substrings, 317
- struct, 51
- sub2ind, 37, 38
- subplot, 276
- sum, 126–128, 189
- surf, 255, 256
- surfc, 257
- surf1, 257
- svd, 98
- switch, 194
- tan, 59
- tanh, 61
- text, 264, 265
- tic, 328
- title, 259
- toc, 328
- trace, 88
- transpose, see matrix
- trapz, 175
- tril, 44
- triu, 45
- unidcdf, 237
- unidpdf, 230
- unidrnd, 219, 220
- unifcdf, 236
- unifpdf, 229
- unifrnd, 221
- unique, 165, 249
- var, 113–115
- variables
  - assigning, 292
assigning in base environment from function, 293
evaluating from base environment within function, 294
names, 305
variance, see var
vector
  boolean operations on, 161, 162
  containing all identical entries, 18
  containing all zeros, 17
  counts of binned values in, 166
  counts of discrete values in, 165
  cross product, 80
  cumulative sum of elements of, 133
  differences between consecutive elements of, 139
  dot product, 79
  minimum of values of, 141
  norm, 102
  position of first occurrence of minimum value in, 148
  product of all elements, 129
  reversing order of elements in, 27
  size of, 152
  sum of all elements, 126
  truncating, 39

warning, 330
waterfall, 257
which, 334
while, 192
who, 295
whos, 296–298
xlabel, 260–262
ylabel, 260, 261
zeros, 17, 19
Index of R commands and concepts

*, 92
/, 90
:, 12, 13
; , 331
<-, 292
<<-, 293
=, 292
?, 1, 2
[, 49
#, 310
% , 63, 341
&, 188, 189
^, 54, 94

abs, 55, 74
acos, 60
acosh, 62
alarm, 326
all, 190
any, 191
apply, 34, 35, 112, 114, 115, 130, 143, 144
Arg, 75
array, 47
arrows, 266, 267
as.character, 321
as.formula, 178
as.numeric, 165
asin, 60
asinh, 62
atan, 60
atanh, 62
average, see mean

barplot, 247, 249
boolean tests
scalar, 188
vector, 189–191
break, 193
browser, 333
c, 6, 7
cat, 311
cbind, 24, 38
celling, 66
cellular automata animation, 281
chol, 100
choose, 72
class, 304
cloud, 254
coeff, 176–179
colMeans, 108
colon, see :
colormap
building your own, 275
for image, 274
colSums, 127
column vector, 7
comments, 310
complex numbers, 73–78
Conj, 76
contour, 257
convolve, 167
cor, 121–125
cos, 59
cosh, 61
cov, 118–120
cubic splines, 181, 182, 184
natural, 180
periodic, 183
cummax, 138
cummin, 138
cumprod, 137
cumsum, 133–136
cumulative distribution functions
binomial, 232
continuous uniform on interval \((a, b)\), 236
discrete uniform from 1..\(n\), 237
exponential, 234
normal, 235
Poisson, 233
curve, 271
data.frame, 51
dbinom, 225
debugging, 333
det, 86
dev.control, 278, 279, 281
dev.copy, 278, 279
dev.copy2eps, 279
dev.copy2pdf, 278
dev.cur(), 240
dev.list, 241
dev.new, 238
dev.off, 242, 278–280
dev.set, 239
dexp, 227
diag, 21–23
diff, 139
differential equations, see lsoda
dim, 43, 151, 154
dir, 285
INDEX OF R COMMANDS AND CONCEPTS

dir.create, 282
dmultinom, 231
dnorm, 228
dpois, 226
dunif, 229
echelon form, see matrix
eig, 96
element-by-element matrix operations, see matrix
else, 187
erbar, 251, 252
eval, 332
exists, 303
exp, 56
expand, 97
expand.grid, 257
expm, 132

file
  restoring workspace from, 291
  running commands in, 286
  saving workspace to, 290
  text
    reading data from, 287, 288
    saving data to, 289
filled.contour, 273
*.First, 336
fix, 299
floor, 65
font size in plots, 262
for, 185
function
  multi-variable
    minimization, 172
    minimization over first parameter only, 171
    minimization over only some parameters, 173
  single-variable
    minimization, 170
    user-written, 195
    returning multiple values, 196
get, 294
getwd, 284
globalenv, 294
graphics
  not being displayed from scripts/functions, 277
Greek letters
  in plot labels, 261
gregexpr, 319
grepl, 320
grid, 263

help, 1, 2
help.search, 5
help.start, 4
Hilbert, 46
hist, 166, 247, 248, 250
identity, see matrix
if, 186–188
ifelse, 140
Im, 78
image, 272, 281
indexing
  matrix, 10
    with a single index, 11
  vector, 9
install.packages, 338
integrate, 174
inverse, see matrix
jpeg, 280

kappa, 280

.Last, 337
.Last.value, 306
lattice package, 257, 273, 277, 338
layout, 276
legend, 268
length, 39, 40, 152, 153
levelplot, 273, 277
library, 3, 338
lines, 269
lists, 48
  extracting elements of, 49
lm, 176–179
lm.fit, 178
load, 291
log, 57
log10, 58
log2, 58
lower.tri, 45
ls, 295
ls.str, 296, 298
lsoda, 197–199

*.Machine$double.eps, 323
match, 318
matplot, 270

matrix, 8
  boolean operations on, 163, 164
  changing shape of, 43
  Cholesky factorization, 100
  circular shift, 33
  condition number, 104–106
INDEX OF R COMMANDS AND CONCEPTS

containing all indentical entries, 20
containing all zeros, 19
converting row, column to single index, 37
converting single-index to row, column, 36
cumulative sums of all elements of, 136
cumulative sums of columns, 134
cumulative sums of rows, 135
determinant, 86
diagonal, 22
echelon form, 85
eigenvalues and eigenvectors of, 96
equation
  solving, 84
exponential of, 132
extracting a column of, 28
extracting a rectangular piece of, 31
extracting a row of, 29
extracting specified rows and columns of, 32
“gluing” together, 24, 25
identity, 21
inverse, 87
lower-triangular portion of, 44
LU factorization, 97
minimum of values of, 142
minimum value of each column of, 143
minimum value of each row of, 144
modifying elements given lists of rows and columns, 38
multiplication, 81
  element-by-element, 82
N-dimensional, 47
norm, 103
powers of, 93
product
  of all elements, 129
  of columns of, 130
  of rows of, 131
QR factorization, 101
rank, 95
re-shaping its elements into a vector, 30
reverse elements in columns, 35
reverse elements in rows, 34
Schur decomposition, 99
singular value decomposition, 98
size of, 149–151, 153, 154
sum
  of all elements, 126
  of columns of, 127
  of rows of, 128
trace, 88
transpose, 83
  upper-triangular portion of, 45
matrix, 8, 19, 20
max, see min
mean, 107
median, 117
min, 141–144, 147
Mod, 74
mode, 304
mode of vector of values, 116
modulo arithmetic, 63, 341
multiple statements on one line, 331
names, 50, 165
ncol, 150
next, 193
noquote, 311
norm, 102, 103
nrow, 149
optim, 172, 173
optimization, 170–173
optimize, 170, 171
options
  digits=, 308
order, 157–160
outer, 178, 256
packages
  installing, 338
  loading, 338
par, 262
par
  mfcol=, 276
  mfrow=, 276
parse, 332
paste, 178, 315, 316
pbinom, 232
pdf, 262, 278
perform some commands with probability p, 209
permutation of integers 1..n, 210
persp, 255, 256
pexp, 234
pie, 253
plot, 243–246
  Greek letters in axis labels, 261
    main=, 259
    sub=, 259
    xlab=, 260, 261
    xlim=, 258
    ylab=, 260, 261
    ylim=, 258
pmin, 145, 146
pnorm, 68, 69, 235
points, 269
polynomial
  least-squares fitted, 177–179
INDEX OF R COMMANDS AND CONCEPTS

multiplication, 167
roots of, 168
polyreg, 178
polyroot, 168
postscript, 279
poin, 233
print, 277, 311, 312
probability density functions
binomial, 225
continuous uniform on interval \((a, b)\), 229
discrete uniform from 1..n, 230
exponential, 227
multinomial, 231
normal, 228
Poisson, 226
proc.time, 327, 328
prod, 129–131
punif, 236
q, 309, 337
qnorm, 70, 71
qr, 95, 101
quartz, 238
quit, 309, 337
rand, 207
random values
Bernoulli, 206
binomial, 216
continuous uniform distribution on interval \((a, b)\), 203, 221
continuous uniform distribution on interval \((0.1)\), 200, 202
continuous uniform distribution on interval \((0.1, 201
1)
discrete uniform distribution from \(a..b\), 208
discrete uniform distribution from 1..\(k\), 205, 219, 220
discrete uniform distribution, 204
exponential, 218
general discrete distribution, 214
\(k\) unique values sampled from integers 1..n, 211
multinomial, 223, 224
normal, 222
Poisson, 217
setting the seed, 215
rbind, 25
rbinom, 216
rcond, 104, 106
.RData, 336
Re, 77
read.table, 287, 288
readline, 314
rep, 17, 18
rev, 27
rexp, 218
rgb, 275
rm, 300–302
rmultinom, 223, 224
rnorm, 222
roots
of general single-variable function, 169
polynomial, 168
round, 64
row vector, 6
rowMeans, 109
rpois, 217
,.Rprofile, 336
runif, 200–206, 208, 221
sample, 204, 210–214, 219, 220
sampling values from a vector, 212, 213
save.image, 290
scan, 313, 325
Schur, 99
sd, 110–112
seq, 14–16
set.seed, 215
setwd, 283
sign, 67
sin, 59
sinh, 61
solve, 84, 87, 89, 91
sort, 155, 156
source, 286
spline, 180, 181, 183
splines, see cubic splines
split.screen, 276
sprintf, 322
sqrt, 53
standard deviation, see sd
str, 297
string
concatenation, 315
converting number to, 321
pattern matching, 319, 320
substrings, 317
substr, 317
sum, 126, 128, 189
svd, 98
switch, 194
symbols, 257
Sys.sleep, 324
t, 83
INDEX OF R COMMANDS AND CONCEPTS

- **table**, 116, 165, 249
- **tan**, 59
- **tanh**, 61
- **text**, 264, 265
- **title**, 259, 260
- **transpose**, see **matrix**
- **typeof**, 304

- **uniroot**, 169
- **upper.tri**, 44

- **var**, 113–115, 119

- **variables**
  - assigning, 292
  - assigning in base environment from function, 293
  - evaluating from base environment within function, 294
  - names, 305

- **variance**, see **var**

- **vector**
  - boolean operations on, 161, 162
  - containing all identical entries, 18
  - containing all zeros, 17
  - counts of binned values in, 166
  - counts of discrete values in, 165
  - cross product, 80
  - cumulative sum of elements of, 133
  - differences between consecutive elements of, 139
  - dot product, 79
  - minimum of values of, 141
  - norm, 102
  - position of first occurance of minimum value in, 148
  - product of all elements, 129
  - reversing order of elements in, 27
  - size of, 152
  - sum of all elements, 126
  - truncating, 39

- **vector**, 48

- **warning**, 330
- **which**, 162–164
- **which.max**, see **which.min**
- **which.min**, 148
- **while**, 192
- **windows**, 238
- **wireframe**, 257
- **write**, 289

- **x11**, 238