Introduction to Applied Linear Algebra

Vectors, Matrices, and Least Squares

Julia Language Companion

Stephen Boyd and Lieven Vandenberghe

DRAFT August 26, 2018

Contents

Preface **vii**

Getting started with Julia **ix**

1 Vectors **1**
- 1.1 Vectors . . . 1
- 1.2 Vector addition . . . 9
- 1.3 Scalar-vector multiplication . . . 9
- 1.4 Inner product . . . 14
- 1.5 Complexity of vector computations . . . 15

2 Linear functions **19**
- 2.1 Linear functions . . . 19
- 2.2 Taylor approximation . . . 20
- 2.3 Regression model . . . 21

3 Norm and distance **25**
- 3.1 Norm . . . 25
- 3.2 Distance . . . 27
- 3.3 Standard deviation . . . 28
- 3.4 Angle . . . 29
- 3.5 Complexity . . . 30

4 Clustering **33**
- 4.1 Clustering . . . 33
- 4.2 A clustering objective . . . 33
- 4.3 The k-means algorithm . . . 34
- 4.4 Examples . . . 37
- 4.5 Applications . . . 37

5 Linear independence **39**
- 5.1 Linear dependence . . . 39
- 5.2 Basis . . . 39
- 5.3 Orthonormal vectors . . . 40
- 5.4 Gram–Schmidt algorithm . . . 41

6 Matrices **45**
6.1 Matrices . . . 45
6.2 Zero and identity matrices . . . 50
6.3 Transpose, addition, and norm . . . 54
6.4 Matrix-vector multiplication . . . 56
6.5 Complexity . . . 59

7 Matrix examples **61**
7.1 Geometric transformations . . . 61
7.2 Selectors . . . 61
7.3 Incidence matrix . . . 63
7.4 Convolution . . . 65

8 Linear equations **67**
8.1 Linear and affine functions . . . 67
8.2 Linear function models . . . 69
8.3 Systems of linear equations . . . 71

9 Linear dynamical systems **73**
9.1 Linear dynamical systems . . . 73
9.2 Population dynamics . . . 74
9.3 Epidemic dynamics . . . 75
9.4 Motion of a mass . . . 75
9.5 Supply chain dynamics . . . 76

10 Matrix multiplication **79**
10.1 Matrix-matrix multiplication . . . 79
10.2 Composition of linear functions . . . 80
10.3 Matrix power . . . 81
10.4 QR factorization . . . 82

11 Matrix inverses **85**
11.1 Left and right inverses . . . 85
11.2 Inverse . . . 86
11.3 Solving linear equations . . . 87
11.4 Examples . . . 90
11.5 Pseudo-inverse . . . 91

12 Least squares **93**
12.1 Least squares problem . . . 93
12.2 Solution . . . 94
12.3 Solving least squares problems . . . 95
12.4 Examples . . . 96

13 Least squares data fitting **99**
13.1 Least squares data fitting . . . 99
13.2 Validation . . . 105
13.3 Feature engineering . . . 108

14 Least squares classification **113**
14.1 Classification . . . 113
14.2 Least squares classifier . . . 114
14.3 Multi-class classifiers . . . 115

15 Multi-objective least squares **121**
15.1 Multi-objective least squares . . . 121
15.2 Control . . . 124
15.3 Estimation and inversion . . . 124
15.4 Regularized data fitting . . . 125
15.5 Complexity . . . 127

16 Constrained least squares **129**
16.1 Constrained least squares problem . . . 129
16.2 Solution . . . 131
16.3 Solving contrained least squares problems . . . 132

17 Constrained least squares applications **135**
17.1 Portfolio optimization . . . 135
17.2 Linear quadratic control . . . 137
17.3 Linear quadratic state estimation . . . 143

18 Nonlinear least squares **145**
18.1 Nonlinear equations and least squares . . . 145
18.2 Gauss–Newton algorithm . . . 145
18.3 Levenberg–Marquardt algorithm . . . 147
18.4 Nonlinear model fitting . . . 152
18.5 Nonlinear least squares classification . . . 155

19 Constrained nonlinear least squares **157**
19.1 Constrained nonlinear least squares . . . 157
19.2 Penalty algorithm . . . 157
19.3 Augmented Lagrangian algorithm . . . 159
19.4 Nonlinear control . . . 161

Appendices **163**

A The `VMLS` package **165**
A.1 Utility functions . . . 165
A.2 Algorithms . . . 166
A.3 Data sets . . . 167

Preface

This *Julia Language Companion* accompanies our book *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares* (referred to here as VMLS). It is meant to show how the ideas and methods in VMLS can be expressed and implemented in the programming language Julia.

We assume that the reader has installed Julia, or is using Juliabox online, and understands the basics of the language. We also assume that the reader has carefully read the relevant sections of VMLS. The organization of the Julia Language Companion follows VMLS, section by section.

You will see that mathematical notation and Julia syntax are pretty close, but not the same. *You must be careful to never confuse mathematical notation and Julia syntax.* In these notes we write mathematical notation (as in VMLS) in standard mathematics font, *e.g.*, $y = Ax$. Julia code, expressions, and snippets are written in a fixed-width typewriter font, *e.g.*, `y = A*x`. We encourage you to cut and paste our Julia code snippets into a Julia interactive session or notebook, to test them out, and maybe modify them and run them again. You can also use our snippets as templates for your own Julia code. At some point we will collect the Julia snippets in this companion document into Julia notebooks that you can easily run.

Julia is a very powerful language, but in this companion document we use only a small and limited set of its features. The code snippets in this companion are written so as to to be transparent and simple, and to emphasize closeness to the concepts in VMLS. Some of the code snippets could be written in a much more compact way, or in a way that would result in faster or more efficient execution.

The code snippets in this compaion document are compatible with Julia 1.0, which is not quite the same as earlier versions, like Julia 0.6. Some of the functions we use are in the standard packages `LinearAlgebra`, `SparseArrays`, and `Plots`, and a few others. We have created a simple and small Julia package `VMLS`, which includes a small number of simple Julia functions that are useful for studying the material in VMLS. The next section, *Getting started with Julia*, explains how to install these Julia packages.

We consider this companion to be a draft. We'll be updating it occasionally, adding more examples and fixing typos as we become aware of them. So you may wish to periodically check whether a newer version is available.

Stephen Boyd *Stanford, California*
Lieven Vandenberghe *Los Angeles, California*

Getting started with Julia

Installing Julia. Download Julia 1.0 from its website, and then follow the instructions to install it on your platform. You'll want to make sure it's working before proceeding to install additional packages, as described below.

Installing packages. While most of the Julia code we use in this companion is from the base or core of the Julia language, several important functions are contained in other packages that you must explicitly install. Here we explain how to do this. The installation has to be done only once.

To add (install) a Julia package you use the package manager system, which is entered by typing `]` to the Julia prompt. You'll then see the package manager prompt, `(v.10) pkg>`. If you type `?`, you'll see a list of package manager commands. You exit the package manager by typing Ctrl-C (Control-C), which returns you to the julia prompt.

To add a package called `PackageName`, type `add PackageName` to the package control prompt. It may take a few minutes to get the required packages, compile, and install them. If you type `status`, you'll see what packages are installed. If you type `up`, installed packages will be upgraded, if needed.

This companion will use functions from the following packages:

- `LinearAlgebra`
- `SparseArrays`
- `Plots`

On a few occasions we use functions from the packages `Random` and `DSP`, but we will mention this in the text when we use these functions.

Here's what it looks like when you install the `Plots` package.

```
julia> ]
(v1.0) pkg> add Plots
 Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `~/.julia/environments/v1.0/Project.toml`
 [no changes]
```

```
  Updating `~/.julia/environments/v1.0/Manifest.toml`
 [no changes]
(v1.0) pkg> status
 Status `~/.julia/environments/v1.0/Project.toml`
  [91a5bcdd] Plots v0.19.3
  [011e45e9] SparseArrays v0.4.2
  [37e2e46d] LinearAlgebra
(v1.0) pkg> ^C
julia>
```

The `VMLS` package. We have created a small package called `VMLS`. It contains a few functions that use notation closer to VMLS notation or are easier to use than the corresponding Julia functions, basic implementations of some algorithms in VMLS, and functions that generate data used in examples. The list of functions is given in Appendix A. To install `VMLS`, go to the package manager from the Julia prompt, then install it as follows.

```
julia> ]
pkg> add https://github.com/VMLS-book/VMLS.jl
pkg> ^C
julia>
```

Using packages. Once a package is installed (which needs to be done only once), you import it into Julia with the command `using`, followed by the package name, or a comma separated list of package names. This too can take some time. After executing this command you can access the functions contained in the packages.

To run any the code fragments in this companion you will need to first execute the statement

```
julia> using LinearAlgebra, SparseArrays, VMLS
```

When we use other packages (in particular, `Plots`), we include the `using` statement in the code.

Chapter 1

Vectors

1.1 Vectors

Vectors in Julia are represented by one-dimensional `Array` objects. A vector is constructed by giving the list of elements surrounded by square brackets, with the elements separated by commas or semicolons. The assignment operator `=` is used to give a name to the array. The `length` function returns the size (dimension).

```
julia> x = [ -1.1, 0.0, 3.6, -7.2 ]
4-element Array{Float64,1}:
 -1.1
  0.0
  3.6
 -7.2
julia> length(x)
4
julia> y = [ -1.1; 0.0; 3.6; -7.2 ]   # Using semicolons
4-element Array{Float64,1}:
 -1.1
  0.0
  3.6
 -7.2
julia> length(y)
4
```

The `Array{Float64,1}` displayed by Julia above each array tells us that the array is one-dimensional and its entries are floating point numbers that use 64 bits.

Some common mistakes. Don't forget the commas or semicolons between entries, and be sure to use square brackets and not parentheses. Otherwise you'll get things that makes sense in Julia, but are not vectors.

```
julia> a = [ 1 2 ]
1×2 Array{Int64,2}:
 1  2
julia> b = ( 1, 2 )
(1, 2)
```

Here `a` is a row vector, which we will encounter later; `b` is a tuple or list consisting of two scalars.

Indexing. A specific element x_i is retrieved by the expression `x[i]` where `i` is the index (which runs to 1 to n, for an n-vector). Array indexing can be also be used on the left-hand side of an assignment, to change the value of a specific element.

```
julia> x = [ -1.1, 0.0, 3.6, -7.2 ];
julia> x[3]
3.6
julia> x[3] = 4.0;
julia> x
4-element Array{Float64,1}:
 -1.1
  0.0
  4.0
 -7.2
```

The special index `end` refers to the last index of a vector. In the example above, `x[end]` and `x[length(x)]` both give the last entry, `-7.2`.

Assignment versus copying. Matlab or Octave users may be surprised by the behavior of an assignment `y = x` if `x` is an array. This expression gives a new name (or reference) `y` to the *same* array already referenced by `x`. It does not create a new copy of the array `x`.

```
julia> x = [ -1.1, 0.0, 3.6, -7.2 ];
julia> y = x;
julia> x[3] = 4.0;
julia> y
4-element Array{Float64,1}:
 -1.1
  0.0
  4.0     # The assignment to x[3] also changed y[3]
 -7.2
julia> y[1] = 2.0;
julia> x
```

```
4-element Array{Float64,1}:
  2.0     # The assignment to y[1] also changed x[1]
  0.0
  4.0
 -7.2
```

To create a new copy of an array, the function `copy` should be used.

```
julia> x = [ -1.1, 0.0, 3.6, -7.2 ];
julia> y = copy(x);
julia> x[3] = 4.0;
julia> y
4-element Array{Float64,1}:
 -1.1
  0.0
  3.6
 -7.2
julia> y[1] = 2.0;
julia> x
4-element Array{Float64,1}:
 -1.1
  0.0
  4.0
 -7.2
```

Vector equality. Equality of vectors is checked using the relational operator `==`. For two vectors (arrays) `a` and `b`, the Julia expression `a==b` evaluates to `true` if the vectors (arrays) are equal, *i.e.*, they have the same length and identical entries, and `false` otherwise.

```
julia> x = [ -1.1, 0.0, 3.6, -7.2 ];
julia> y = copy(x);
julia> y[3] = 4.0
julia> y == x
false
julia> z = x
julia> z[3] = 4.0
julia> z == x
true
```

Scalars versus 1-vectors. In the mathematical notation used in VMLS we consider a 1-vector to be the same as a number. But in Julia, 1-vectors are *not* the same as scalars (numbers). Julia distinguishes between the 1-vector (array) `[ 1.3 ]` and the number `1.3`.

```
julia> x = [ 1.3 ]
1-element Array{Float64,1}:
 1.3
julia> y = 1.3
1.3
julia> x == y
false
julia> x[1] == y
true
```

In the last line, `x[1]` is the first (and only) entry of `x`, which is indeed the number `1.3`.

Block or stacked vectors. To construct a block vector in Julia, you can use `vcat` (vertical concatenate) or the semicolon (`;`) operator. Let's construct the block vector $z = (x, y)$ with $x = (1, -2)$ and $y = (1, 1, 0)$ using the two methods.

```
julia> x = [ 1, -2 ];  y = [ 1, 1, 0 ];
julia> z = [ x; y ]      # Concatenate using semicolon
5-element Array{Int64,1}:
  1
 -2
  1
  1
  0
julia> z = vcat(x, y)    # Concatenate using vcat
5-element Array{Int64,1}:
 1
-2
 1
 1
 0
```

As in mathematical notation, you can stack vectors with scalars, *e.g.*, `[1;x;0]` creates $(1, x, 0)$.

Some common mistakes. There are a few Julia operations that look similar but do not construct a block or stacked vector. For example, `z = (x,y)` creates a list or tuple of the two vectors; `z = [x,y]` creates an array of the two vectors. Both of these are valid Julia expression, but neither of them is the stacked vector `[x;y]`.

Subvectors and slicing. As in the mathematical notation used in VMLS, the Julia expression `r:s` denotes the index range $r, r+1, \ldots, s$. (It is assumed here that `r` and `s` are positive integers with `r` the smaller of the two.) In VMLS, we use $x_{r:s}$ to denote the slice of the vector x from index r to s. In Julia you can extract a subvector or slice of a vector using an index range as the argument. You can also use index ranges to assign a slice of a vector.

```
julia> x = [ 9, 4, 3, 0, 5 ];
julia> y = x[2:4]
3-element Array{Int64,1}:
 4
 3
 0
julia> x[4:5] = [ -2, -3 ];  # Re-assign the 4 and 5 entries of x
julia> x
5-element Array{Int64,1}:
  9
  4
  3
 -2
 -3
```

Julia indexing into arrays. Julia slicing and subvectoring is much more general than the mathematical notation we use in VMLS. For example, one can use a number range with a third argument, that gives the stride, which is the increment between successive indexes. For example, the index range `1:2:5` is the list of numbers 1,3,5. The expression `x[1:2:5]` extracts the 3-vector `[9,3,5]`, *i.e.*, the first, third, and fifth entries of `x` defined above. You can also use an index range that runs backward. For any vector `z`, the Julia expression `z[end:-1:1]` is the reversed vector, *i.e.*, the vector with the same coefficients, but in opposite order.

Vector of first differences. Let's use slicing to create the $(n-1)$-vector d defined by $d_i = x_{i+1} - x_i$, for $i = 1, \ldots, n-1$, where x is an n-vector. The vector d is called the vector of (first) *differences* of x.

```
julia> x = [ 1, 0, 0, -2, 2 ];
julia> d = x[2:end] - x[1:end-1]
4-element Array{Int64,1}:
 -1
  0
 -2
  4
```

Lists of vectors. An ordered list of n-vectors might be denoted in VMLS as $a_1, \ldots, a_k$ or $a^{(1)}, \ldots, a^{(k)}$, or just as a, b, c. There are several ways to represent lists of vectors in Julia. If we give the elements of the list, separated by commas, and surrounded by square brackets, we form a one-dimensional array of vectors. If instead we use parentheses as delimiters, we obtain a *tuple* or list.

```
julia> x = [ 1.0, 0 ];   y = [ 1.0, -1.0 ];   z = [ 0, 1.0];
julia> list = [ x, y, z ]
3-element Array{Array{Float64,1},1}:
 [1.0, 0.0]
 [1.0, -1.0]
 [0.0, 1.0]
julia> list[2]  # Second element of list
2-element Array{Float64,1}:
  1.0
 -1.0
julia> list = ( x, y, z )
([1.0, 0.0], [1.0, -1.0], [0.0, 1.0])
julia> list[3]  # Third element of list
2-element Array{Float64,1}:
 0.0
 1.0
```

Note the difference between `[x, y, z]` (an array of arrays) and `[x; y; z]` (an array of numbers, obtained by concatenation). To extract the ith vector from the list of vectors, use `list[i]`. To get the jth element or coefficient of the ith vector in the list, use `list[i][j]`.

Zero vectors. In Julia a zero vector of dimension n is created using `zeros(n)`.

```
julia> zeros(3)
3-element Array{Float64,1}:
 0.0
 0.0
 0.0
```

The expression `zeros(length(a))` creates a zero vector with the same size as the vector `a`.

Unit vectors. There is no built-in Julia function for creating e_i, the ith unit vector of length n. The following code creates e_i, with $i = 2$ and $n = 4$.

```
julia> i = 2; n = 4;
julia> ei = zeros(n);   # Create a zero vector
julia> ei[i] = 1;   # Set ith entry to 1
julia> ei
```

```
4-element Array{Float64,1}:
 0.0
 1.0
 0.0
 0.0
```

Here's another way to create e_i using concatenation, using a Julia inline function.

```
julia> unit_vector(i,n) = [zeros(i-1); 1 ; zeros(n-i)]
unit_vector (generic function with 1 method)
julia> unit_vector(2,4)
4-element Array{Float64,1}:
 0.0
 1.0
 0.0
 0.0
```

Ones vector. In Julia the ones vector of dimension n, denoted $\mathbf{1}_n$ or just $\mathbf{1}$ in VMLS, is created using `ones(n)`.

```
julia> ones(2)
2-element Array{Float64,1}:
 1.0
 1.0
```

Random vectors. We do not use or refer to random vectors in VMLS, which does not assume a background in probability. However, it is sometimes useful to generate random vectors, for example to test an identity or some algorithm. In Julia, `rand(n)` generates a random vector of length n with entries that are between 0 and 1. Each time this function is called or evaluated, it gives a different vector. The variant `randn(n)` (with the extra 'n' for normal) gives an n-vector with entries that come from a normal (Gaussian) distribution. They can be positive or negative, with typical values on the order of one. Remember that every time you evaluate these functions, you get a different random vector. In particular, you will obtain different entries in the vectors below when you run the code.

```
julia> rand(2)
2-element Array{Float64,1}:
 0.831491
 0.0497708
julia> rand(2)
```

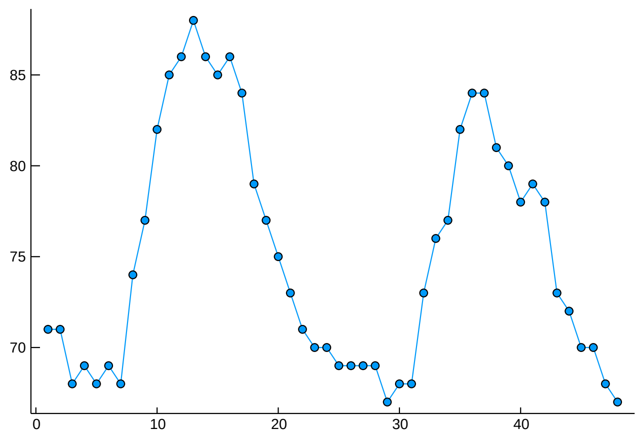


Figure 1.1 Hourly temperature in downtown Los Angeles on August 5 and 6, 2015 (starting at 12:47AM, ending at 11:47PM).

```
2-element Array{Float64,1}:
 0.189284
 0.713467
julia> randn(2)
2-element Array{Float64,1}:
  2.44544
 -0.12134
```

Plotting. There are several external packages for creating plots in Julia. One such package is `Plots.jl`, which you must add (install) via Julia's package manager control system; see page ix. Assuming the `Plots` package had been installed, you import it into Julia for use, using the command `using Plots`. (This can take some time.) After that you can access the Julia commands that create or manipulate plots.

For example, we can plot the temperature time series in Figure 1.3 of VMLS using the code below; the last line saves the plot in the file `temperature.pdf`. The result is shown in Figure 1.1.

```
julia> using Plots  # Only need to do this once per session
[ Info: Precompiling Plots [91a5bcdd-55d7-5caf-9e0b-520d859cae80]
[ Info: Precompiling GR [28b8d3ca-fb5f-59d9-8090-bfdbd6d07a71]
julia> temps = [ 71, 71, 68, 69, 68, 69, 68, 74, 77, 82, 85, 86,
 88, 86, 85, 86, 84, 79, 77, 75, 73, 71, 70, 70, 69, 69, 69,
 69, 67, 68, 68, 73, 76, 77, 82, 84, 84, 81, 80, 78, 79, 78,
 73, 72, 70, 70, 68, 67 ];
```

```
julia> plot(temps, marker = :circle, legend = false, grid = false)
julia> savefig("temperature.pdf")
```

1.2 Vector addition

Vector addition and subtraction. If `x` and `y` are vectors of the same size, `x+y` and `x-y` give their sum and difference, respectively.

```
julia> [ 0, 7, 3 ] + [ 1, 2, 0 ]  # Vector addition
3-element Array{Int64,1}:
 1
 9
 3
julia> [ 1, 9 ] - [ 1, 1]  # Vector subtraction
2-element Array{Int64,1}:
 0
 8
```

1.3 Scalar-vector multiplication

Scalar-vector multiplication and division. If `a` is a number and `x` a vector, you can express the scalar-vector product either as `a*x` or `x*a`. (Julia actually allows you to write `2.0x` for `2.0*x`. This is unambiguous because variable names cannot start with a number.) You can carry out scalar-vector division in a similar way, as `x/a`, or the less familar looking expression `a\x`.

```
julia> x = [ 0, 2, -1 ];
julia> 2.2 * x   # Scalar-vector multiplication
3-element Array{Float64,1}:
  0.0
  4.4
 -2.2
julia> x * 2.2   # Scalar-vector multiplication
3-element Array{Float64,1}:
  0.0
  4.4
 -2.2
```

```
julia> x / 3   # Scalar-vector division
3-element Array{Float64,1}:
  0.0
  0.666667
 -0.333333
julia> 3 \ x   # Scalar-vector division
  0.0
  0.666667
 -0.333333
```

Scalar-vector addition. In Julia you can add a scalar `a` and a vector `x` using `x .+ a`. The dot that precedes the plus symbol tells Julia to apply the operation to each element. (More on this below.) The meaning is that the scalar is added or subtracted to each element. (This is not standard mathematical notation; in VMLS we denote this as, *e.g.*, $x + a\mathbf{1}$, where x is an n-vector and a is a scalar.) In Julia you can also carry out scalar-vector addition with the scalar on the left.

```
julia> [ 1.1, -3.7, 0.3 ] .- 1.4  # Vector-scalar subtraction
3-element Array{Float64,1}:
 -0.3
 -5.1
 -1.1
julia> 0.7 .+ [1,-1]
2-element Array{Float64,1}:
  1.7
 -0.30000000000000004
```

Elementwise operations. Julia supports methods for carrying out an operation on every element or coefficient of a vector. To do this we add a period or dot before the operator. For example, if `x` and `y` are vectors of the same length, then `x.*y`, `x./y`, `x.\y`, `x.^y` are elementwise vector-vector operations. They result in vectors of the same length as x and y, and ith element x_iy_i, x_i/y_i, y_i/x_i, and $x_i^{y_i}$, respectively.

As an example of elementwise division, let's find the 3-vector of asset returns `r` from the (vectors of) initial and final prices of assets (see page 22 in VMLS).

```
julia> p_initial = [ 22.15, 89.32, 56.77 ];
julia> p_final = [ 23.05, 87.32, 57.13 ];
julia> r = (p_final - p_initial) ./ p_initial
3-element Array{Float64,1}:
  0.0406321
 -0.0223914
```

```
  0.00634138
```

Elementwise operations with a scalar. Elementwise operations work when one of the arguments is a scalar, in which case it is interpreted as the scalar times a ones vector of the appropriate dimension. Scalar-vector addition, described above, is a special case of this. If `a` is a scalar and `x` is a vector, then `x.^a` is a vector with ith element x_i^a, and `a.^x` is a vector with elements a^{x_i}. Like scalar-vector addition, the dot notation for elementwise operations is not standard mathematical notation and we do not use it in VMLS.

We can also use the period notation with a function that has a name, to let Julia know that the function should be applied elementwise. In this case we add the period *after* the function name to indicate that it should be applied elementwise. For example, if `x` is a vector, we can form `sin.(x)` to apply the sine function to each element of x.

The equality test operator `==` (and other relational operators like `<`, `>=`) can be made to work elementwise by preceding it with a period: `x==y` tells us whether or not `x` and `y` are the same vector; `x.==y` is a vector whose entries tell us whether the corresponding entries in `x` and `y` are the same.

```
julia> w = [1,2,2]; z = [1,2,3];
julia> w == z
false
julia> w .== z
3-element BitArray{1}:
  true
  true
  false
```

You can combine this with Julia's slicing to extract the subvector of entries that satisfy some logical condition. For example `x[abs.(x).>1]` gives the subvector of `x` consisting of the entries larger than one in magnitude.

```
julia> x = [1.1, .5, -1.5, -0.3]
4-element Array{Float64,1}:
  1.1
  0.5
 -1.5
 -0.3
julia> x[abs.(x) .> 1]
2-element Array{Float64,1}:
  1.1
 -1.5
```

Dot notation works with assignment too, allowing you to assign multiple entries of a vector to a scalar value. For example:

```
julia> x = rand(4)
4-element Array{Float64,1}:
 0.4735758513909343
 0.3554729725184458
 0.13775623085957855
 0.9227200780245117
julia> x[1:2] = [-1,1];
julia> x
4-element Array{Float64,1}:
 -1.0
  1.0
  0.13775623085957855
  0.9227200780245117
julia> x[2:3] .= 1.3;
julia> x
4-element Array{Float64,1}:
 -1.0
  1.3
  1.3
  3.0
```

Linear combination. You can form a linear combination is Julia using scalar-vector multiplication and addition.

```
julia> a = [ 1, 2 ];  b = [ 3, 4 ];
julia> alpha = -0.5; beta = 1.5;
julia> c = alpha*a + beta*b
2-element Array{Float64,1}:
 4.0
 5.0
```

To illustrate some additional Julia syntax, we create a function that takes a list of coefficients and a list of vectors as its arguments, and returns the linear combination. The lists can be represented by tuples or arrays.

```
julia> function lincomb(coeff, vectors)
       n = length(vectors[1])  # Length of vectors
       a = zeros(n);
       for i = 1:length(vectors)
           a = a + coeff[i] * vectors[i];
       end
       return a
       end
lincomb (generic function with 1 method)
julia> lincomb( ( -0.5, 1.5), ( [1, 2], [ 3, 4]) )
2-element Array{Float64,1}:
 4.0
 5.0
```

A more concise definition of the function is as follows.

```
julia> function lincomb(coeff, vectors)
       return sum( coeff[i] * vectors[i] for i = 1:length(vectors) )
       end
```

Checking properties. Let's check the distributive property

$$\beta(a+b) = \beta a + \beta b,$$

which holds for any two n-vectors a and b, and any scalar β. We'll do this for $n = 3$, and randomly generated a, b, and β. (This computation does not show that the property always holds; it only shows that it holds for the specific vectors chosen. But it's good to be skeptical and check identities with random arguments.) We use the `lincomb` function we just defined.

```
julia> a = rand(3)
3-element Array{Float64,1}:
 0.55304
 0.55801
 0.0299682
julia> b = rand(3);
3-element Array{Float64,1}:
 0.796619
 0.578865
 0.219901
julia> beta = randn()  # Generates a random scalar
beta = randn()
-0.17081925677011056
julia> lhs = beta*(a+b)
3-element Array{Float64,1}:
 -0.230548
 -0.1942
 -0.0426825
julia> rhs = beta*a + beta*b
3-element Array{Float64,1}:
 -0.230548
 -0.1942
 -0.0426825
```

Although the two vectors `lhs` and `rhs` are displayed as the same, they might not be exactly the same, due to very small round-off errors in floating point computations. When we check an identity using random numbers, we can expect that the left-hand and right-hand sides of the identity are not exactly the same, but very close to each other.

1.4 Inner product

Inner product. The inner product of n-vectors x and y is denoted as $x^T y$. In Julia, the inner product of `x` and `y` is denoted as `x'*y`.

```
julia> x = [ -1, 2, 2 ];
julia> y = [ 1, 0, -3 ];
julia> x'*y
-7
```

Net present value. As an example, the following code snippet finds the net present value (NPV) of a cash flow vector `c`, with per-period interest rate `r`.

```
julia> c = [ 0.1, 0.1, 0.1, 1.1 ];  # Cash flow vector
julia> n = length(c);
julia> r = 0.05;   # 5% per-period interest rate
julia> d = (1+r) .^ -(0:n-1)
4-element Array{Float64,1}:
 1.0
 0.952381
 0.907029
 0.863838
julia> NPV = c'*d
1.236162401468524
```

In the fourth line, to get the vector `d` we raise the scalar `1+r` elementwise to the powers given in the array `(0:n-1)`, which expands to `[0,1,...,n-1]`.

Total school-age population. Suppose that the 100-vector x gives the age distribution of some population, with x_i the number of people of age $i-1$, for $i = 1, \ldots, 100$. The total number of people with age between 5 and 18 (inclusive) is given by

$$x_6 + x_7 + \cdots + x_{18} + x_{19}.$$

We can express this as $s^T x$, where s is the vector with entries one for $i = 6, \ldots, 19$ and zero otherwise. In Julia, this is expressed as

```
julia> s = [ zeros(5); ones(14); zeros(81) ];
julia> school_age_pop = s'*x
```

Several other expressions can be used to evaluate this quantity, for example, the expression `sum(x[6:19])`, using the Julia function `sum`, which gives the sum of the entries of a vector.

1.5 Complexity of vector computations

Floating point operations. For any two numbers a and b, we have $(a+b)(a-b) = a^2 - b^2$. When a computer calculates the left-hand and right-hand side, for specific numbers a and b, they need not be exactly the same, due to very small floating point round-off errors. But they should be very nearly the same. Let's see an example of this.

```
julia> a = rand(); b = rand();
julia> lhs = (a+b) * (a-b)
-0.025420920298883976
julia> rhs = a^2 - b^2
-0.02542092029888398
julia> lhs - rhs
3.469446951953614e-18
```

Here we see that the left-hand and right-hand sides are not exactly equal, but very very close.

Complexity. You can time a Julia command by adding `@time` before the command. The timer is not very accurate for very small times, say, measured in microseconds (10^{-6} seconds). Also, you should run the command more than once; it can be a lot faster on the second or subsequent runs.

```
julia> a = randn(10^5); b = randn(10^5);
julia> @time a'*b
 0.002695 seconds (5 allocations: 176 bytes)
38.97813069037062
julia> @time a'*b
 0.000173 seconds (5 allocations: 176 bytes)
```

```
38.97813069037062
julia> c = randn(10^6); d = randn(10^6);
julia> @time c'*d
0.001559 seconds (5 allocations: 176 bytes)
1189.2960722446112
julia> @time c'*d
0.001765 seconds (5 allocations: 176 bytes)
1189.2960722446112
```

The first inner product, of vectors of length 10^5, takes around 0.00017 seconds; the second, with vectors of length 10^6 (tens times bigger), product takes around 0.0018 seconds, about 10 longer. This is predicted by the complexity of the inner product, which is $2n - 1$ flops. The computer on which the computations were done is capable of around $2 \cdot 10^6/0.001765$ flops per second, *i.e.*, around 1 Gflop/s. These timings, and the estimate of computer speed, are very approximate.

Sparse vectors. Functions for creating and manipulating sparse vectors are contained in the Julia package `SparseArrays`, so you need to install this package before you can use them; see page ix.

Sparse vectors are stored as sparse arrays, *i.e.*, arrays in which only the nonzero elements are stored. In Julia you can create a sparse vector from lists of the indices and values using the `sparsevec` function. You can also first create a sparse vector of zeros (using `spzeros(n)`) and then assign values to the nonzero entries. A sparse vector can be created from a non-sparse vector using `sparse(x)`, which returns a sparse version of `x`. `nnz(x)` gives the number of nonzero elements of a sparse vector. Sparse vectors are overloaded to work as you imagine; for example, all the usual vector operations work, and they are automatically recast as non-sparse vectors when appropriate.

```
julia> a = sparsevec( [ 123456, 123457 ], [ 1.0, -1.0 ], 10^6 )
1000000-element SparseVector{Float64,Int64} with 2 stored entries:
  [123456 ]  =  1.0
  [123457 ]  =  -1.0
julia> length(a)
1000000
julia> nnz(a)
2
julia> b = randn(10^6);  # An ordinary (non-sparse) vector
julia> @time 2*a;  # Computed efficiently!
  0.000003 seconds (7 allocations: 384 bytes)
julia> @time 2*b;
  0.003558 seconds (6 allocations: 7.630 MiB)
julia> @time a'*b;  # Computed efficiently!
  0.000003 seconds (5 allocations: 176 bytes)
```

```
julia> @time b'*b;
  0.000450 seconds (5 allocations: 176 bytes)
julia> @time c = a + b;
  0.002085 seconds (6 allocations: 7.630 MiB)
```

In the last line, the sparse vector `a` is automatically converted to an ordinary vector (array) so it can be added to the random vector; the result is a (non-sparse) vector of length 10^6.

Chapter 2

Linear functions

2.1 Linear functions

Functions in Julia. Julia provides several methods for defining functions. A simple function given by an expression such as $f(x) = x_1 + x_2 - x_4^2$ can be defined in a single line.

```
julia> f(x) = x[1] + x[2] - x[4]^2
f (generic function with 1 method)
julia> f([-1,0,1,2])
-5
```

Since the function definition refers to the first, second, and fourth elements of the argument `x`, these have to be defined when you call or evaluate `f(x)`; you'll get an error if, for example, `x` has dimension 3 or is a scalar.

Superposition. Suppose a is an n-vector. The function $f(x) = a^T x$ is linear, which means that for any n-vectors x and y, and any scalars α and β, the superposition equality

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

holds. Superposition says that evaluating f at a linear combination of two vectors is the same forming the linear combination of f evaluated at the two vectors.

Let's define the inner product function f for a specific value of a, and then verify superposition in Julia for specific values of x, y, α, and β. (This check does not show that the function is linear. It simply checks that superposition holds for these specific values.)

```
julia> a = [-2, 0, 1, -3];
julia> f(x) = a'*x  # Inner product function
f (generic function with 1 method)
julia> x = [2, 2, -1, 1];  y= [0, 1, -1, 0];
julia> alpha = 1.5;  beta = -3.7;
julia> lhs = f(alpha * x + beta * y)
-8.3
julia> rhs = alpha * f(x) + beta * f(y)
-8.3
```

For the function $f(x) = a^Tx$, we have $f(e_3) = a_3$. Let's check that this holds in our example.

```
julia> e3 = [0, 0, 1, 0];
julia> f(e3)
1.0
```

Examples. Let's define the average function in Julia, and check its value for a specific vector. (Julia's `Statistics` package contains the average function, which is called `mean`.)

```
julia> avg(x) = (ones(length(x)) / length(x))'*x;
julia> x = [1, -3, 2, -1];
julia> avg(x)
-0.25
```

The average function can be implemented more concisely as `sum(x)/length(x)`. The `avg` function is part of the `VMLS` package; once you install and then add this package, you can use the `avg` function.

2.2 Taylor approximation

Taylor approximation. The (first-order) Taylor approximation of a function $f : \mathbf{R}^n \to \mathbf{R}$, at the point z, is the affine function of x given by

$$\hat{f}(x) = f(z) + \nabla f(z)^T(x - z).$$

For x near z, $\hat{f}(x)$ is very close to $f(x)$. Let's try a numerical example (see page 36) using Julia.

```
julia> f(x) = x[1] + exp(x[2]-x[1]);   # A function
julia> # And its gradient
julia> grad_f(z) = [1-exp(z[2]-z[1]), exp(z[2]-z[1])];
julia> z = [1, 2];
julia> # Taylor approximation at z
julia> f_hat(x) = f(z) + grad_f(z)'*(x-z);
julia> # Let’s compare f and f_hat for some specific x’s
julia> f([1,2]),  f_hat([1,2])
(3.718281828459045, 3.718281828459045)
julia> f([0.96,1.98]),  f_hat([0.96,1.98])
(3.733194763964298, 3.732647465028226)
julia> f([1.10,2.11]),  f_hat([1.10,2.11])
(3.845601015016916, 3.845464646743635)
```

2.3 Regression model

Regression model. The regression model is the affine function of x given by $f(x) = x^T\beta + v$, where the n-vector β and the scalar v are the parameters in the model. The regression model is used to guess or approximate a real or observed value of the number y that is associated with x. (We'll see later how to find the parameters in a regression model, using data.)

Let's define the regression model for house sale prices described on page 39 of VMLS, and compare its prediction to the true house sale price y for a few values of x.

```
julia> # Parameters in regression model
julia> beta = [148.73, -18.85]; v = 54.40;
julia> y_hat(x) = x'*beta + v;
julia> # Evaluate regression model prediction
julia> x = [0.846, 1];  y = 115;
julia> y_hat(x), y
(161.37557999999999, 115)
julia> x = [1.324,2];  y = 234.50;
julia> y_hat(x), y
(213.61852000000002, 234.5)
```

Our first prediction is pretty bad; our second one is better.

A scatter plot of predicted and actual house prices (Figure 2.4 of VMLS) can be generated as follows. We use the `VMLS` function `house_sales_data` to obtain the vectors `price`, `area`, `beds` (see appendix A).

```
julia> D = house_sales_data();
julia> price = D["price"]
774-element Array{Float64,1}:
  94.905
  98.937
 100.309
 106.25
 107.502
 108.75
 110.7
 113.263
 116.25
 120.0
 ⋮
 229.027
 229.5
 230.0
 230.0
 232.425
 234.0
 235.0
 235.301
 235.738
julia> area = D["area"]
774-element Array{Float64,1}:
 0.941
 1.146
 0.909
 1.289
 1.02
 1.022
 1.134
 0.844
 0.795
 0.588
 ⋮
 1.358
 1.329
 1.715
 1.262
 2.28
 1.477
 1.216
 1.685
 1.362
julia> beds = D["beds"]
774-element Array{Int64,1}:
 2
 3
 3
 3
 3
 2
 2
 2
 2
 2
 ⋮
 3
 4
 4
 3
 4
 3
 3
 4
 3
julia> v = 54.4017;
julia> beta = [ 147.7251, -18.8534 ];
julia> predicted = v .+ beta[1] * area + beta[2] * beds;
julia> using Plots
julia> scatter(price, predicted, lims = (0,800));
julia> plot!([0, 800], [0, 800], linestyle = :dash);
julia> # make axes equal and add labels
julia> plot!(xlims = (0,800), ylims = (0,800), size = (500,500));
julia> plot!(xlabel = "Actual price", ylabel = "Predicted price");
```

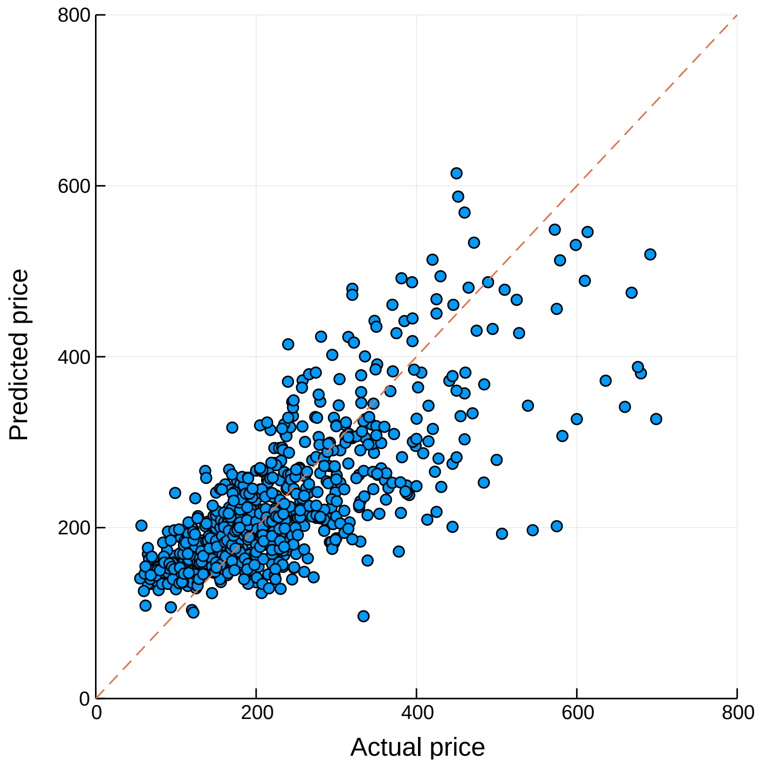


Figure 2.1 Scatter plot of actual and predicted sale prices for 774 houses sold in Sacramento during a five-day period.

Chapter 3

Norm and distance

3.1 Norm

Norm. The norm $\|x\|$ is written in Julia as `norm(x)`. (It can be evaluated several other ways too.) The `norm` function is contained in the Julia package `LinearAlgebra`, so you must install and then add this package to use it; see page ix.

```
julia> x = [ 2, -1, 2 ];
julia> norm(x)
3.0
julia> sqrt(x'*x))
3.0
julia> sqrt(sum(x.^2))
3.0
```

Triangle inequality. Let's check the triangle inequality, $\|x + y\| \leq \|x\| + \|y\|$, for some specific values of x and y.

```
julia> x = randn(10); y = randn(10);
julia> lhs = norm(x+y)
3.5830436972032644
julia> rhs = norm(x) + norm(y)
4.937368598697242
```

RMS value. The RMS value of a vector x is $\mathbf{rms}(x) = \|x\|/\sqrt{n}$. In Julia, this is expressed as `norm(x)/sqrt(length(x))`. (The `VMLS` package contains this function, so you can use it once you've installed this package.)

Let's define a vector (which represents a signal, *i.e.*, the value of some quantity at uniformly space time instances), and find its RMS value. The following code

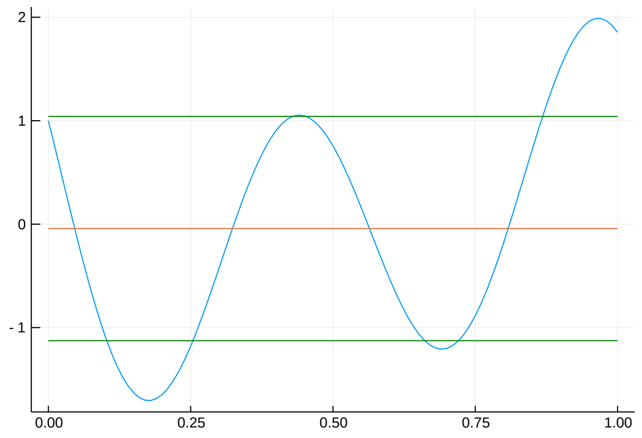


Figure 3.1 A signal x. The horizontal lines show $\mathbf{avg}(x) + \mathbf{rms}(x)$, $\mathbf{avg}(x)$, and $\mathbf{avg}(x) - \mathbf{rms}(x)$.

plots the signal, its average value, and two constant signals at $\mathbf{avg}(x) \pm \mathbf{rms}(x)$ (Figure 3.1).

```
julia> rms(x) = norm(x) / sqrt(length(x));
julia> t = 0:0.01:1;  # List of times
julia> x = cos.(8*t) - 2*sin.(11*t);
julia> avg(x)
-0.04252943783238692
julia> rms(x)
1.0837556422598
julia> using Plots
julia> plot(t, x)
julia> plot!(t, avg(x)*ones(length(x)))
julia> plot!(t, (avg(x)+rms(x))*ones(length(x)), color = :green)
julia> plot!(t, (avg(x)-rms(x))*ones(length(x)), color = :green)
julia> plot!(legend = false)
```

Chebyshev inequality. The Chebyshev inequality states that the number of entries of an n-vector x that have absolute value at least a is no more than $\|x\|^2/a^2 = n\,\mathbf{rms}(x)^2/a^2$. If this number is, say, 12.15, we can conclude that no more that 12 entries have absolute value at least a, since the number of entries is an integer. So the Chebyshev bound can be improved to be floor($\|x\|^2/a$), where floor(u) is the integer part of a positive number. Let's define a function with the Chebyshev bound, including the floor function improvement, and apply the bound to the signal found above, for a specific value of a.

```
julia> # Define Chebyshev bound function
julia> cheb_bound(x,a) = floor(norm(x)^2/a);
julia> a = 1.5;
julia> cheb_bound(x,a)
79.0
julia> # Number of entries of x with |x_i| >= a
julia> sum(abs.(x) .>= a)
20
```

In the last line, the expression `abs.(x) .>= a` creates an array with entries that are Boolean, *i.e.*, `true` or `false`, depending on whether the corresponding entry of `x` satisfies the inequality. When we sum the vector of Booleans, they are automatically converted to (re-cast as) the numbers 1 and 0, respectively.

3.2 Distance

Distance. The distance between two vectors is $\mathbf{dist}(x, y) = \|x - y\|$. This is written in Julia as `norm(x-y)`. Let's find the distance between the pairs of the three vectors u, v, and w from page 49 of VMLS.

```
julia> u = [1.8, 2.0, -3.7, 4.7];
julia> v = [0.6, 2.1, 1.9, -1.4];
julia> w = [2.0, 1.9, -4.0, 4.6];
julia> norm(u-v), norm(u-w), norm(v-w)
(8.36779540858881, 0.3872983346207417, 8.532877591996735)
```

We can see that u and w are much closer to each other than u and v, or v and w.

Nearest neighbor. We define a function that calculates the nearest neighbor of a vector in a list of vectors, and try it on the points in Figure 3.3 of VMLS.

```
julia> nearest_neighbor(x,z) = z[ argmin([norm(x-y) for y in z]) ];
julia> z = ( [2,1], [7,2], [5.5,4], [4,8], [1,5], [9,6] );
julia> nearest_neighbor([5,6], z)
2-element Array{Float64,1}:
 5.5
 4.0
julia> nearest_neighbor([3,3], z)
2-element Array{Int64,1}:
 2
 1
```

On the first line, the expression `[norm(x-y) for y in z]` uses a convenient construction in Julia. Here `z` is a list of vectors, and the expression expands to an array with elements `norm(x-z[1])`, `norm(x-z[2])`, ... The function `argmin` applied to this array returns the index of the smallest element.

De-meaning a vector. We refer to the vector $x - \mathbf{avg}(x)\mathbf{1}$ as the de-meaned version of x.

```
julia> de_mean(x) = x .- avg(x);  # Define de-mean function
julia> x = [1, -2.2, 3];
julia> avg(x)
0.6
julia> x_tilde = de_mean(x)
3-element Array{Float64,1}:
  0.4
 -2.8
  2.4
julia> avg(x_tilde)
-1.4802973661668753e-16
```

(The mean of $\tilde{x}$ is very very close to zero.)

3.3 Standard deviation

Standard deviation. We can define a function that corresponds to the VMLS definition of the standard deviation of a vector, $\mathbf{std}(x) = \|x - \mathbf{avg}(x)\mathbf{1}\|/\sqrt{n}$, where n is the length of the vector.

```
julia> x = rand(100);
julia> # VMLS definition of std
julia> stdev(x) = norm(x-avg(x))/sqrt(length(x));
julia> stdev(x)
0.292205696281305
```

This function is in the `VMLS` package, so you can use it once you've installed this package. (Julia's `Statistics` package has a similar function, `std(x)`, which computes the value $\|x - \mathbf{avg}(x)\mathbf{1}\|/\sqrt{n-1}$, where n is the length of x.)

Return and risk. We evaluate the mean return and risk (measured by standard deviation) of the four time series Figure 3.4 of VMLS.

```
julia> a = ones(10);
julia> avg(a), stdev(a)
```

```
(1.0, 0.0)
julia> b = [ 5, 1, -2, 3, 6, 3, -1, 3, 4, 1 ];
julia> avg(b), stdev(b)
(2.3, 2.41039415863879)
julia> c = [ 5, 7, -2, 2, -3, 1, -1, 2, 7, 8 ];
julia> avg(c), stdev(c)
(2.6, 3.773592452822641)
julia> d = [ -1, -3, -4, -3, 7, -1, 0, 3, 9, 5 ];
julia> avg(d), stdev(d)
(1.2, 4.308131845707603)
```

Standardizing a vector. If a vector x isn't constant (*i.e.*, at least two of its entries are different), we can standardize it, by subtracting its mean and dividing by its standard deviation. The resulting standardized vector has mean value zero and RMS value one. Its entries are called z-scores. We'll define a standardize function, and then check it with a random vector.

```
julia> function standardize(x)
       x_tilde = x .- avg(x)   # De-meaned vector
       return x_tilde/rms(x_tilde)
       end
julia> x = rand(100);
julia> avg(x), rms(x)
(0.510027255229345, 0.5883938729563185)
julia> z = standardize(x);
julia> avg(z), rms(z)
(1.965094753586527e-16, 1.0)
```

The mean or average value of the standarized vector `z` is very nearly zero.

3.4 Angle

Angle. Let's define a function that computes the angle between two vectors. We will call it `ang` because Julia already includes a function `angle` (for the phase angle of a complex number).

```
julia> # Define angle function, which returns radians
julia> ang(x,y) = acos(x'*y/(norm(x)*norm(y)));
julia> a = [1,2,-1];  b=[2,0,-3];
julia> ang(a,b)
0.9689825515916383
```

```
julia> ang(a,b)*(360/(2*pi))  # Get angle in degrees
55.51861062801842
```

Correlation coefficient. The correlation coefficient between two vectors a and b (with nonzero standard deviation) is defined as

$$\rho = \frac{\tilde{a}^T\tilde{b}}{\|\tilde{a}\|\|\tilde{b}\|},$$

where $\tilde{a}$ and $\tilde{b}$ are the de-meaned versions of a and b, respectively. There is no built-in function for correlation, so we can define one. We use function to calculate the correlation coefficients of the three pairs of vectors in Figure 3.8 in VMLS.

```
julia> function correl_coef(a,b)
       a_tilde = a .- avg(a)
       b_tilde = b .- avg(b)
       return (a_tilde'*b_tilde)/(norm(a_tilde)*norm(b_tilde))
       end
julia> a = [4.4, 9.4, 15.4, 12.4, 10.4, 1.4, -4.6, -5.6, -0.6, 7.4];
julia> b = [6.2, 11.2, 14.2, 14.2, 8.2, 2.2, -3.8, -4.8, -1.8, 4.2];
julia> correl_coef(a,b)
0.9678196342570434
julia> a = [4.1, 10.1, 15.1, 13.1, 7.1, 2.1, -2.9, -5.9, 0.1, 7.1];
julia> b = [5.5, -0.5, -4.5, -3.5, 1.5, 7.5, 13.5, 14.5, 11.5, 4.5];
julia> correl_coef(a,b)
-0.9875211120643734
julia> a = [-5.0, 0.0, 5.0, 8.0, 13.0, 11.0, 1.0, 6.0, 4.0, 7.0];
julia> b = [5.8, 0.8, 7.8, 9.8, 0.8, 11.8, 10.8, 5.8, -0.2, -3.2];
julia> correl_coef(a,b)
0.004020976661367021
```

The correlation coefficients of the three pairs of vectors are 96.8%, -98.8%, and 0.4%.

3.5 Complexity

Let's check that the time to compute the correlation coefficient of two n-vectors is approximately linear in n.

```
julia> x = randn(10^6); y = randn(10^6);
julia> @time correl_coef(x,y);
```

```
0.131375 seconds (33.01 k allocations: 16.913 MiB, 2.86% gc time)
julia> @time correl_coef(x,y);
0.023760 seconds (9 allocations: 15.259 MiB, 31.84% gc time)
julia> x = randn(10^7); y = randn(10^7);
julia> @time correl_coef(x,y)
 0.296075 seconds (9 allocations: 152.588 MiB, 30.16% gc time)
julia> @time correl_coef(x,y)
 0.118979 seconds (9 allocations: 152.588 MiB, 21.53% gc time)
```

Chapter 4

Clustering

4.1 Clustering

4.2 A clustering objective

In Julia, we can store the list of vectors in a Julia list or tuple of N vectors. If we call this list `x`, we can access the ith entry (which is a vector) using `x[i]`. To specify the clusters or group membership, we can use a list of assignments called `assignment`, where `assignment[i]` is the number of the group that vector `x[i]` is assigned to. (This is an integer between 1 and k.) (In VMLS chapter 4, we describe the assignments using a vector c or the subsets G_j.) We can store the k cluster representatives as a Julia list called `reps`, with `reps[j]` the jth cluster representative. (In VMLS we describe the representatives as the vectors $z_1, \ldots, z_k$.)

```
julia> Jclust(x,reps,assignment) =
       avg( [norm(x[i]-reps[assignment[i]])^2 for i=1:length(x)] )
Jclust (generic function with 1 method)
julia> x = [ [0,1], [1,0], [-1,1] ]
3-element Array{Array{Int64,1},1}:
 [0, 1]
 [1, 0]
 [-1, 1]
julia> reps = [ [1,1], [0,0] ]
2-element Array{Array{Int64,1},1}:
 [1, 1]
 [0, 0]
julia> assignment = [1,2,1]
3-element Array{Int64,1}:
 1
 2
 1
julia> Jclust(x,reps,assignment)
2.0
julia> assignment = [1,1,2]
1.3333333333333333
```

4.3 The k-means algorithm

We write a simple Julia implementation of the k-means algorithm and apply it to a set of points in a plane, similar to the example in Figure 4.1 of VMLS.

We first create a function `kmeans` that can be called as

```
julia> assignment, representatives = kmeans(x, k)
```

where `x` is an array of N vectors and `k` is the number of groups. The first output argument is an array of N integers, containing the computed group assignments (integers from 1 to k). The second output argument is an array of k vectors, with the k group representatives. We also include two optional keyword arguments, with a limit on the number of iterations and a tolerance used in the stopping condition.

```
function kmeans(x, k; maxiters = 100, tol = 1e-5)

N = length(x)
n = length(x[1])
distances = zeros(N)  # used to store the distance of each
                      # point to the nearest representative.
reps = [zeros(n) for j=1:k]  # used to store representatives.

# 'assignment' is an array of N integers between 1 and k.
# The initial assignment is chosen randomly.
assignment = [ rand(1:k) for i in 1:N ]

Jprevious = Inf  # used in stopping condition
for iter = 1:maxiters

    # Cluster j representative is average of points in cluster j.
    for j = 1:k
        group = [i for i=1:N if assignment[i] == j]
        reps[j] = sum(x[group]) / length(group);
    end;

    # For each x[i], find distance to the nearest representative
```

```
23     # and its group index.
24     for i = 1:N
25         (distances[i], assignment[i]) =
26             findmin([norm(x[i] - reps[j]) for j = 1:k])
27     end;
28 
29     # Compute clustering objective.
30     J = norm(distances)^2 /  N
31 
32     # Show progress and terminate if J stopped decreasing.
33     println("Iteration ", iter, ": Jclust = ", J, ".")
34     if iter > 1 && abs(J - Jprevious) < tol * J
35         return assignment, reps
36     end
37     Jprevious = J
38 end
39 
40 end
```

Initialization. As discussed in VMLS (page 76), the k-means algorithm can start from a random initial choice of representatives, or from a random assignment of the points in k groups. In this implementation, we use the second option (line 11). The Julia function `rand(1:k)` picks a random number from the set `1:k`, *i.e.*, the integers $1, \ldots, k$. On line 11 we create an array `assignment` of N elements, with each element chosen by calling `rand(1:k)`.

Updating group representatives. Lines 17–20 update the k group representatives. In line 18, we find the indexes of the points in cluster j and collect them in an array `group`. The expression `x[group]` on line 19 constructs an array from the subset of elements of `x` indexed by `group`. The function `sum` computes the sum of the elements of the array `x[group]`. Dividing by the number of elements `length(x[group])` gives the average of the vectors in the group. The result is jth the group representative. This vector is stored as the jth element in an array `reps` of length N.

Updating group assignments. On lines 24–27 we update the group assignments. The Julia function `findmin` computes both the minimum of a sequence of numbers and the position of the minimum in the sequence. The result is returned as a 2-tuple. On lines 25–26, we apply `findmin` to the array of k distances of point `x[i]` to the k representatives. We store the distance to the nearest representative in `distances[i]`, and the index of the nearest representative (*i.e.*, the new assignment of point i) in `assignment[i]`.

Clustering objective. On line 30 we compute the clustering objective J^{clust} (equation (4.1) in VMLS) as the square of the RMS value of the vector of distances.

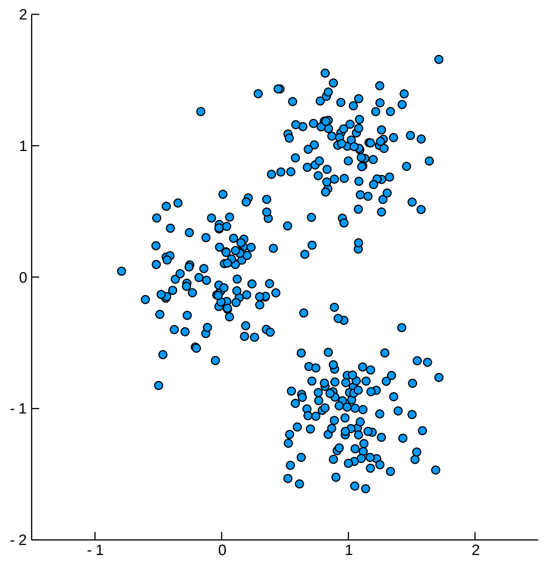


Figure 4.1 300 points in a plane.

Convergence. We terminate the algorithm when the improvement in the clustering objective becomes very small (lines 34–36).

Example. We apply the algorithm on a randomly generated set of $N = 300$ points, shown in Figure 4.1. These points were generated as follows.

```
julia> X = vcat( [ 0.3*randn(2) for i = 1:100 ],
             [ [1,1] + 0.3*randn(2) for i = 1:100 ],
             [ [1,-1] + 0.3*randn(2) for i = 1:100 ] )
julia> scatter([x[1] for x in X], [x[2] for x in X])
julia> plot!(legend = false, grid = false, size = (500,500),
       xlims = (-1.5,2.5), ylims = (-2,2))
```

On the first line we generate three arrays of vectors. Each set consists of 100 vectors chosen randomly around one of the three points $(0, 0)$, $(1, 1)$, and $(1, -1)$. The three arrays are concatenated using `vcat` to get an array of 300 points.

Next, we apply our `kmeans` function and make a figure with the three clusters (Figure 4.2).

```
julia> assignment, reps = kmeans(X, 3)
Iteration 1: Jclust = 0.8815722022603146.
Iteration 2: Jclust = 0.24189035975341422.
Iteration 3: Jclust = 0.18259342207994636.
Iteration 4: Jclust = 0.1800980527878161.
```

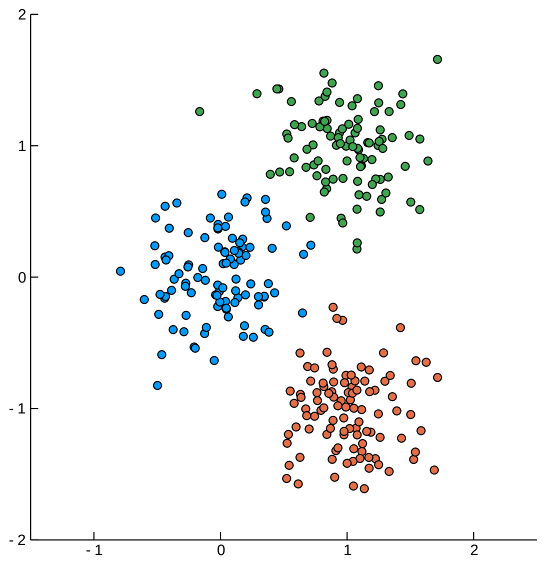


Figure 4.2 Final clustering.

```
Iteration 5: Jclust = 0.17993051934500726.
Iteration 6: Jclust = 0.17988967509836415.
Iteration 7: Jclust = 0.17988967509836415.
julia> grps  = [[X[i] for i=1:N if assignment[i] == j] for j=1:k]
julia> scatter([c[1] for c in grps[1]], [c[2] for c in grps[1]])
julia> scatter!([c[1] for c in grps[2]], [c[2] for c in grps[2]])
julia> scatter!([c[1] for c in grps[3]], [c[2] for c in grps[3]])
julia> plot!(legend = false, grid = false, size = (500,500),
       xlims = (-1.5,2.5), ylims = (-2,2))
```

4.4 Examples

4.5 Applications

Chapter 5

Linear independence

5.1 Linear dependence

5.2 Basis

Cash flow replication. Let's consider cash flows over 3 periods, given by 3-vectors. We know from VMLS page 93 that the vectors

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad l_1 = \begin{bmatrix} 1 \\ -(1+r) \\ 0 \end{bmatrix}, \quad l_2 = \begin{bmatrix} 0 \\ 1 \\ -(1+r) \end{bmatrix}$$

form a basis, wher r is the (positive) per-period interest rate. The first vector e_1 is a single payment of \$1 in period (time) $t = 1$. The second vector l_1 is loan of \$1 in period $t = 1$, paid back in period $t = 2$ with interest r. The third vector l_2 is loan of \$1 in period $t = 2$, paid back in period $t = 3$ with interest r. Let's use this basis to replicate the cash flow $c = (1, 2, -3)$ as

$$c = \alpha_1 e_1 + \alpha_2 l_1 + \alpha_3 l_2 = \alpha_1 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ -(1+r) \\ 0 \end{bmatrix} + \alpha_3 \begin{bmatrix} 0 \\ 1 \\ -(1+r) \end{bmatrix}.$$

From the third component we have $c_3 = \alpha_3(-(1+r))$, so $\alpha_3 = -c_3/(1+r)$. From the second component we have

$$c_2 = \alpha_2(-(1+r)) + \alpha_3 = \alpha_2(-(1+r)) - c_3/(1+r),$$

so $\alpha_2 = -c_2/(1+r) - c_3/(1+r)^2$. Finally from $c_1 = \alpha_1 + \alpha_2$, we have

$$\alpha_1 = c_1 + c_2/(1+r) + c_3/(1+r)^2,$$

which is the net present value (NPV) of the cash flow c.

Let's check this in Julia using an interest rate of 5% per period, and the specific cash flow $c = (1, 2, -3)$.

```
julia> r = 0.05;
julia> e1 = [1,0,0]; l1 = [1,-(1+r),0]; l2 = [0,1,-(1+r)];
julia> c = [1,2,-3];
julia> # Coefficients of expansion
julia> alpha3 = -c[3]/(1+r);
julia> alpha2 = -c[2]/(1+r) -c[3]/(1+r)^2;
julia> alpha1 = c[1] + c[2]/(1+r) + c[3]/(1+r)^2  # NPV of cash flow
0.18367346938775508
julia> alpha1*e1 + alpha2*l1 + alpha3*l2
3-element Array{Float64,1}:
  1.0
  2.0
 -3.0
```

(Later in the course we'll an automated and simple way to find the coefficients in the expansion of a vector in a basis.)

5.3 Orthonormal vectors

Expansion in an orthonormal basis. Let's check that the vectors

$$a_1 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \qquad a_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \qquad a_3 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix},$$

form an orthonormal basis, and check the expansion of $x = (1, 2, 3)$ in this basis,

$$x = (a_1^T x)a_1 + \cdots + (a_n^T x)a_n.$$

```
julia> a1 = [0,0,-1]; a2 = [1,1,0]/sqrt(2); a3 = [1,-1,0]/sqrt(2);
julia> norm(a1), norm(a2), norm(a3)
(1.0, 0.9999999999999999, 0.9999999999999999)
julia> a1'*a2, a1'*a3, a2'*a3
(0.0, 0.0, 0.0)
julia> x = [1,2,3]
3-element Array{Int64,1}:
  1
  2
  3
julia> # Get coefficients of x in orthonormal basis
julia> beta1 = a1'*x; beta2 = a2'*x; beta3 = a3'*x;
julia> # Expansion of x in basis
julia> xexp = beta1*a1 + beta2*a2 + beta3*a3
3-element Array{Float64,1}:
  1.0
  2.0
  3.0
```

5.4 Gram–Schmidt algorithm

The following is a Julia implementation of Algorithm 5.1 in VMLS (Gram–Schmidt algorithm). It takes as input an array `[ a[1], a[2], ..., a[k] ]`, containing the k vectors $a_1, \ldots, a_k$. If the vectors are linearly independent, it returns an array `[ q[1], ..., q[k] ]` with the orthonormal set of vectors computed by the Gram–Schmidt algorithm. If the vectors are linearly dependent and the Gram–Schnidt algorithm terminates early in iteration i, it returns the array `[ q[1], ..., q[i] ]` of length i.

```
 1 function gram_schmidt(a; tol = 1e-10)
 2
 3 q = []
 4 for i = 1:length(a)
 5     qtilde = a[i]
 6     for j = 1:i-1
 7         qtilde -= (q[j]'*a[i]) * q[j]
 8     end
 9     if norm(qtilde) < tol
10        println("Vectors are linearly dependent.")
11        return q
12     end
13     push!(q, qtilde/norm(qtilde))
14 end;
15 return q
16 end
```

On line 3, we initialize the output array as the empty array. In each iteration, we add the next vector to the array using the `push!` function (line 13).

Example. We apply the function to the example on page 100 of VMLS.

```
julia> a = [ [-1, 1, -1, 1], [-1, 3, -1, 3], [1, 3, 5, 7] ]
3-element Array{Array{Int64,1},1}:
 [-1, 1, -1, 1]
 [-1, 3, -1, 3]
 [1, 3, 5, 7]
julia> q = gram_schmidt(a)
3-element Array{Any,1}:
 [-0.5, 0.5, -0.5, 0.5]
 [0.5, 0.5, 0.5, 0.5]
 [-0.5, -0.5, 0.5, 0.5]
julia> # test orthnormality
julia> norm(q[1])
1.0
julia> q[1]'*q[2]
0.0
julia> q[1]'*q[3]
0.0
julia> norm(q[2])
1.0
julia> q[2]'*q[3]
0.0
julia> norm(q[3])
1.0
```

Example of early termination. If we replace a_3 with a linear combination of a_1 and a_2 the set becomes linearly dependent.

```
julia> b = [ a[1], a[2], 1.3*a[1] + 0.5*a[2] ]
3-element Array{Array{Float64,1},1}:
 [-1.0, 1.0, -1.0, 1.0]
 [-1.0, 3.0, -1.0, 3.0]
 [-1.8, 2.8, -1.8, 2.8]
julia> q = gram_schmidt(b)
Vectors are linearly dependent.
2-element Array{Any,1}:
 [-0.5, 0.5, -0.5, 0.5]
 [0.5, 0.5, 0.5, 0.5]
```

Example of independence-dimension inequality. We know that any three 2-vectors must be dependent. Let's use the Gram-Schmidt algorithm to verify this for three specific vectors.

```
julia> three_two_vectors = [ [1,1], [1,2], [-1,1] ]
julia> q = gram_schmidt(three_two_vectors)
Vectors are linearly dependent.
2-element Array{Any,1}:
 [0.707107, 0.707107]
 [-0.707107, 0.707107]
```

Chapter 6

Matrices

6.1 Matrices

Creating matrices from the entries. Matrices are represented in Julia as 2-dimensional arrays. These are constructed by giving the elements in each row, separated by space, with the rows separated by semicolons. For example, the 3×4 matrix

$$A = \begin{bmatrix} 0 & 1 & -2.3 & 0.1 \\ 1.3 & 4 & -0.1 & 0 \\ 4.1 & -1 & 0 & 1.7 \end{bmatrix}$$

is constructed in Julia as

```
julia> A = [0.0  1.0 -2.3  0.1;
            1.3  4.0 -0.1  0.0;
            4.1 -1.0  0.0 1.7]
3×4 Array{Float64,2}:
 0.0   1.0  -2.3  0.1
 1.3   4.0  -0.1  0.0
 4.1  -1.0   0.0  1.7
```

(Here, `Array{Float64,2}` above the array tells us that the array is 2-dimensional, and its entries are 64-bit floating-point numbers.) In this example, we put the different rows of the matrix on different lines, which makes the code more readable, but there is no need to do this; we get the same matrix with

```
julia> A = [0 1 -2.3 0.1; 1.3 4 -0.1 0; 4.1 -1 0 1.7]
3×4 Array{Float64,2}:
 0.0   1.0  -2.3  0.1
 1.3   4.0  -0.1  0.0
 4.1  -1.0   0.0  1.7
```

The Julia function `size(A)` gives the size, as a tuple. It can also be called as `size(A,1)` or `size(A,2)` to get only the number of rows or columns. As an example, we create a function that determines if a matrix is tall.

```
julia> m, n = size(A)
(3, 4)
julia> m
3
julia> n
4
julia> size(A,1)
3
julia> size(A,2)
4
julia> tall(X) = size(X,1)>size(X,2);
julia> tall(A)
false
```

In the function definition, the number of rows and the number of columns are combined using the relational operator `<`, which gives a Boolean.

Indexing entries. We get the i,j entry of a matrix `A` using `A[i,j]`. We can also assign a new value to an entry.

```
julia> A[2,3]  # Get 2,3 entry of A
-0.1
julia> A[1,3] = 7.5;  # Set 1,3 entry of A to 7.5
7.5
julia> A
3×4 Array{Float64,2}:
 0.0   1.0   7.5  0.1
 1.3   4.0  -0.1  0.0
 4.1  -1.0   0.0  1.7
```

Single index indexing. Julia allows you to access an entry of a matrix using only *one* index. To use this, you need to know that matrices in Julia are stored in *column-major order*. This means that a matrix can be considered as a one-dimensional array, with the first column stacked on top of the second, stacked on top of the third, and so on. For example, the elements of the matrix

$$Z = \left[\begin{array}{rrr} -1 & 0 & 2 \\ -1 & 2 & 3 \end{array}\right]$$

are stored in the order

$$-1,\ -1,\ 0,\ 2,\ 2,\ 3.$$

With single index indexing, `Z[5]` is the fifth element in this sequence.

```
julia> Z = [ -1 0 2; -1 2 -3];
julia> Z[5]
2
```

This is very much *not* standard mathematical notation, and we would never use this in VMLS. But it can be handy in some cases when you are using Julia.

Equality of matrices. `A == B` determines whether the matrices `A` and `B` are equal. The expression `A .== B` creates a matrix whose entries are Boolean, depending on whether the corresponding entries of `A` and `B` are the same. The expression `sum(A .== B)` gives the number of entries of `A` and `B` that are equal.

```
julia> B = copy(A);
julia> B[2,2] = 0;
julia> A == B
false
julia> A .== B
3×4 BitArray{2}:
 true   true  true  true
 true  false  true  true
 true   true  true  true
julia> sum(A .== B)
11
```

Row and column vectors. In Julia, as in VMLS, n-vectors are the same as $n \times 1$ matrices.

```
julia> a = [ -2.1 -3 0 ]  # A 3-row vector or 1x3 matrix
1×3 Array{Float64,2}:
 -2.1  -3.0  0.0
julia> b = [ -2.1; -3; 0 ]  # A 3-vector or 3x1 matrix
3-element Array{Float64,1}:
 -2.1
 -3.0
  0.0
```

The output reveals a small subtlety that generally won't affect you. You can see that `b` has type `Array{Float64,1}`. The final `1` means that it is a 1D array; `size(b)` gives `(3,)`, whereas you might think it would or should be `(3,1)`. So you might say that in Julia, $n \times 1$ matrices are n-vectors. This is why we say above that n-vectors and $n \times 1$ matrices are almost the same in Julia.

Slicing and submatrices. Using colon notation you can extract a submatrix.

```
julia> A = [ -1 0 1 0 ; 2 -3 0 1 ; 0 4 -2 1]
3×4 Array{Int64,2}:
 -1   0   1  0
  2  -3   0  1
  0   4  -2  1
julia> A[1:2,3:4]
2×2 Array{Float64,2}:
  1  0
  0  1
```

This is very similar to the mathematical notation in VMLS, where this submatrix would be denoted $A_{1:2,3:4}$. You can also assign a submatrix using slicing (index range) notation.

A very useful shortcut is the index range `:` which refers to the whole index range for that index. This can be used to extract the rows and columns of a matrix.

```
julia> A[:,3]  # Third column of A
3-element Array{Int64,1}:
 1
 0
-2
julia> A[2,:]  # Second row of A, returned as column vector!
4-element Array{Int64,1}:
  2
 -3
  0
  1
```

In mathematical (VMLS) notation, we say that `A[2,:]` returns the *transpose* of the second row of `A`.

As with vectors, Julia's slicing and selection is not limited to contiguous ranges of indexes. For example, we can reverse the order of the rows of a matrix `X` using

```
julia> m = size(X,1)
julia> X[m:-1:1,:]  # Matrix X with row order reversed
```

Julia's single indexing for matrices can be used with index ranges or sets. For example if `X` is an $m \times n$ matrix, `X[:]` is a vector of size mn that consists of the columns of `X` stacked on top of each other. The Julia function `reshape(X,(k,l))` gives a new $k \times l$ matrix, with the entries taken in the column-major order from `X`. (We must have $mn = kl$, *i.e.*, the original and reshaped matrix must have the same number of entries.) Neither of these is standard mathematical notation, but they can be useful in Julia.

```
julia> B = [ 1 -3 ; 2 0 ; 1 -2]
3×2 Array{Int64,2}:
 1  -3
 2   0
 1  -2
julia> B[:]
6-element Array{Int64,1}:
  1
  2
  1
 -3
  0
 -2
julia> reshape(B,(2,3))
2×3 Array{Int64,2}:
 1   1   0
 2  -3  -2
julia> reshape(B,(3,3))
ERROR: DimensionMismatch("new dimensions (3, 3) must be consistent
with array size 6")
```

Block matrices. Block matrices are constructed in Julia very much as in the standard mathematical notation in VMLS. You use ; to stack matrices, and a space to do (horizontal) concatenation. We apply this to the example on page 109 of VMLS.

```
julia> B = [ 0 2 3 ];  # 1x3 matrix
julia> C = [ -1 ];  # 1x1 matrix
julia> D = [ 2 2 1 ; 1 3 5]; # 2x3 matrix
julia> E = [4 ; 4 ];  # 2x1 matrix
julia> # construct 3x4 block matrix
julia> A = [B C ;
            D E]
3×4 Array{Int64,2}:
 0  2  3  -1
 2  2  1   4
 1  3  5   4
```

Column and row interpretation of a matrix. An $m \times n$ matrix A can be interpreted as a collection of n m-vectors (its columns) or a collection of m row vectors (its rows). Julia distinguishes between a matrix (a two-dimensional array) and an array of vectors. An array (or a tuple) of column vectors can be converted into a matrix using the horizontal concatenation function `hcat`.

```
julia> a = [ [1., 2.], [4., 5.], [7., 8.] ] # array of 2-vectors
3-element Array{Array{Float64,1},1}:
 [1.0, 2.0]
 [4.0, 5.0]
 [7.0, 8.0]
julia> A = hcat(a...)
2×3 Array{Float64,2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
```

The `...` operator in `hcat(a...)` splits the array `a` into its elements, *i.e.*, `hcat(a...)` is the same as `hcat(a[1], a[2], a[3])`, which concatenates `a[1]`, `a[2]`, `a[3]` horizontally.

Similarly, `vcat` concatenates an array of arrays vertically. This is useful when constructing a matrix from its row vectors.

```
julia> a = [ [1. 2.], [4. 5.], [7. 8.] ] # array of 1x2 matrices
3-element Array{Array{Float64,2},1}:
 [1.0 2.0]
 [4.0 5.0]
 [7.0 8.0]
julia> A = vcat(a...)
3×2 Array{Float64,2}:
 1.0  2.0
 4.0  5.0
 7.0  8.0
```

6.2 Zero and identity matrices

Zero matrices. A zero matrix of size $m \times n$ is created using `zeros(m,n)`.

```
julia> zeros(2,2)
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

Identity matrices. Identity matrices in Julia can be created many ways, for example by starting with a zero matrix and then setting the diagonal entries to one.

The `LinearAlgebra` package also contains functions for creating a special identity matrix object `I`, which has some nice features. You can use `1.0*Matrix(I,n,n)` to create an $n \times n$ identity matrix. (Multiplying by 1.0 converts the matrix into one with numerical entries; otherwise it has Boolean entries.) This expression is pretty unwieldy, so we can define a function `eye(n)` to generate an identity matrix. This function is in the `VMLS` package, so you can use it once the package is installed. (The name `eye` to denote the identity matrix I traces back to the MATLAB language.)

```
julia> eye(n) = 1.0*Matrix(I,n,n)
julia> eye(4)
4×4 Array{Float64,2}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0
```

Julia's identity matrix `I` has some useful properties. For example, when it can deduce its dimensions, you don't have to specify it. (This is the same as with common mathematical notation; see VMLS page 113.)

```
julia> A = [ 1 -1 2; 0 3 -1]
2×3 Array{Int64,2}:
 1  -1   2
 0   3  -1
julia> [A I]
2×5 Array{Int64,2}:
 1  -1   2  1  0
 0   3  -1  0  1
julia> [A ; I]
5×3 Array{Int64,2}:
 1  -1   2
 0   3  -1
 1   0   0
 0   1   0
 0   0   1
julia> B = [ 1 2 ; 3 4 ]
2×2 Array{Int64,2}:
 1  2
 3  4
julia> B + I
2×2 Array{Int64,2}:
 2  2
 3  5
```

Ones matrix. In VMLS we do not have notation for a matrix with all entries one. In Julia, such a matrix is given by `ones(m,n)`.

Diagonal matrices. In standard mathematical notation, $\mathbf{diag}(1,2,3)$ is a diagonal 3×3 matrix with diagonal entries $1, 2, 3$. In Julia such a matrix is created using the function `diagm`, provided in the `LinearAlgebra` package. To construct the diagonal matrix with diagonal entries in the vector `s`, you use `diagm(0 => s)`. This is fairly unwieldy, so the `VMLS` package defines a function `diagonal(s)`. (Note that you have to pass the diagonal entries as a vector.)

```
julia> diagonal(x) = diagm(0 => x)
diagonal (generic function with 1 method)
julia> diagonal([1,2,3])
3×3 Array{Int64,2}:
 1  0  0
 0  2  0
 0  0  3
```

A closely related Julia function `diag(X)` does the opposite: It takes the diagonal entries of the (possibly not square) matrix `X` and puts them into a vector.

```
julia> H = [0 1 -2 1; 2 -1 3 0]
2×4 Array{Int64,2}:
 0   1  -2  1
 2  -1   3  0
julia> diag(H)
2-element Array{Int64,1}:
  0
 -1
```

Random matrices. A random $m \times n$ matrix with entries between 0 and 1 is created using `rand(m,n)`. For entries that have a normal distribution, `randn(m,n)`.

```
julia> rand(2,3)
2×3 Array{Float64,2}:
 0.365832   0.381598  0.321444
 0.0317522  0.434451  0.95419
julia> randn(3,2)
3×2 Array{Float64,2}:
  0.541546    1.65458
 -0.684011   -2.12776
  0.0443909  -1.81297
```

Sparse matrices. Functions for creating and manipulating sparse matrices are contained in the `SparseArrays` package, which must be installed; see page ix. Sparse matrices are stored in a special format that exploits the property that most of the elements are zero. The `sparse` function creates a sparse matrix from three arrays that specify the row indexes, column indexes, and values of the nonzero elements. The following code creates a sparse matrix

$$A = \begin{bmatrix} -1.11 & 0 & 1.17 & 0 & 0 \\ 0.15 & -0.10 & 0 & 0 & 0 \\ 0 & 0 & -0.30 & 0 & 0 \\ 0 & 0 & 0 & 0.13 & 0 \end{bmatrix}.$$

```
julia> I = [ 1, 2, 2, 1, 3, 4 ]  # row indexes of nonzeros
julia> J = [ 1, 1, 2, 3, 3, 4 ]  # column indexes
julia> V = [ -1.11, 0.15, -0.10, 1.17, -0.30, 0.13 ]  # values
julia> A = sparse(I, J, V, 4, 5)
4×5 SparseMatrixCSC{Float64,Int64} with 6 stored entries:
  [1, 1]  =  -1.11
  [2, 1]  =  0.15
  [2, 2]  =  -0.1
  [1, 3]  =  1.17
  [3, 3]  =  -0.3
  [4, 4]  =  0.13
julia> nnz(A)
6
```

Sparse matrices can be converted to regular non-sparse matrices using the `Array` function. Applying `sparse` to a full matrix gives the equivalent sparse matrix.

```
julia> A = sparse([1, 3, 2, 1], [1, 1, 2, 3],
   [1.0, 2.0, 3.0, 4.0], 3, 3)
julia> 3×3 SparseMatrixCSC{Float64,Int64} with 4 stored entries:
  [1, 1]  =  1.0
  [3, 1]  =  2.0
  [2, 2]  =  3.0
  [1, 3]  =  4.0
julia> B = Array(A)
3×3 Array{Float64,2}:
 1.0  0.0  4.0
 0.0  3.0  0.0
 2.0  0.0  0.0
julia> B[1,3] = 0.0;
julia> sparse(B)
```

```
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [3, 1]  =  2.0
  [2, 2]  =  3.0
```

A sparse $m \times n$ zero matrix is created with `spzeros(m,n)`. To create a sparse $n \times n$ identity matrix in Julia, use `sparse(1.0I,n,n)`. This is not a particularly natural syntax, so we define a function `speye(n)` in the `VMLS` package. The `VMLS` package also includes the function `speye(n)` which creates a sparse $n \times n$ identity matrix, as well as `spdiagonal(a)`, which creates a sparse diagonal matrix with the entries of the vector `a` on its diagonal.

A useful function for creating a random sparse matrix is `sprand(m,n,d)` (with entries between 0 and 1) and `sprandn(m,n,d)` (with entries that range over all numbers). The first two arguments give the dimensions of the matrix; the last one, `d`, gives the density of nonzero entries. The nonzero entries are chosen randomly, with about mnd of them nonzero. The following code creates a random 10000×10000 sparse matrix, with a density 10^{-7}. This means that we'd expect there to be around 10 nonzero entries. (So this is a *very* sparse matrix!)

```
julia> A = sprand(10000,10000,10^-7)
10000×10000 SparseMatrixCSC{Float64,Int64} with 10 stored entries:
  [6435 ,   513]  =  0.912633
  [3274 ,  1518]  =  0.369523
  [8264 ,  2169]  =  0.875005
  [8029 ,  3513]  =  0.0670725
  [7427 ,  4122]  =  0.0376876
  [6329 ,  5078]  =  0.970446
  [3882 ,  5213]  =  0.0964994
  [102  ,  6572]  =  0.697033
  [730  ,  7936]  =  0.489414
  [7006 ,  8422]  =  0.909351
```

6.3 Transpose, addition, and norm

Transpose. In VMLS we denote the transpose of an $m \times n$ matrix A as A^T. In Julia, the transpose of `A` is given by `A'`.

```
julia>  H = [0 1 -2 1; 2 -1 3 0]
2×4 Array{Int64,2}:
 0   1  -2  1
 2  -1   3  0
```

```
julia> H'
4×2 Array{Int64,2}:
  0   2
  1  -1
 -2   3
  1   0
```

Addition, subtraction, and scalar multiplication. In Julia, addition and subtraction of matrices, and scalar-matrix multiplication, both follow standard mathematical notation.

```
julia> U = [ 0 4; 7 0; 3 1]
3×2 Array{Int64,2}:
 0  4
 7  0
 3  1
julia> V = [ 1 2; 2 3; 0 4]
3×2 Array{Int64,2}:
 1  2
 2  3
 0  4
julia> U+V
3×2 Array{Int64,2}:
 1  6
 9  3
 3  5
julia> 2.2*U
3×2 Array{Float64,2}:
  0.0  8.8
 15.4  0.0
  6.6  2.2
```

(We can also multiply a matrix on the right by a scalar.)

Julia supports some operations that are not standard mathematical ones. For example, in Julia you can add or subtract a constant from a matrix, which carries out the operation on each entry.

Elementwise operations. The syntax for elementwise vector operations described on page 10 carries over naturally to matrices. We add a period before a binary operator to change the interpretation to elementwise. For example, if A and B are matrices of the same size, then `C = A .* B` creates a matrix of the same size with elements $C_{ij} = A_{ij}B_{ij}$. We can add a period after a function name to tell Julia that the function should be applied elementwise. Thus, if `X` is a matrix, then `Y = exp.(X)` creates a matrix of the same size, with elements $Y_{ij} = \exp(X_{ij})$.

Matrix norm. In VMLS we use $\|A\|$ to denote the norm of an $m \times n$ matrix,

$$\|A\| = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2 \right)^{1/2}.$$

In standard mathematical notation, this is more often written as $\|A\|_F$, where F stands for the name Frobenius. In standard mathematical notation, $\|A\|$ usually refers to another norm of a matrix, that is beyond the scope of the topics in VMLS. In Julia, `norm(A)` gives the norm used in VMLS.

```
julia> A = [2 3 -1; 0 -1 4]
julia> norm(A)
5.5677643628300215
julia> norm(A[:])
5.5677643628300215
```

Triangle inequality. Let's check that the triangle inequality $\|A+B\| \leq \|A\|+\|B\|$ holds, for two specific matrices.

```
julia> A = [-1 0; 2 2];  B= [3 1; -3 2];
julia> norm(A + B),  norm(A) + norm(B)
(4.69041575982343, 7.795831523312719)
```

6.4 Matrix-vector multiplication

In Julia, matrix-vector multiplication has the natural syntax `y=A*x`.

```
julia> A = [0 2 -1; -2 1 1]
2×3 Array{Int64,2}:
  0  2  -1
 -2  1   1
julia> x = [2, 1, -1]
3-element Array{Int64,1}:
  2
  1
 -1
julia> A*x
2-element Array{Int64,1}:
  3
 -4
```

Difference matrix. An $(n-1) \times n$ difference matrix (equation (6.5) of VMLS) can be constructed in several ways. A simple one is the following.

```
julia> difference_matrix(n) = [-eye(n-1) zeros(n-1)] +
       [zeros(n-1), eye(n-1)];
julia> D = difference_matrix(4)
3×4 Array{Float64,2}:
 -1.0   1.0   0.0  0.0
  0.0  -1.0   1.0  0.0
  0.0   0.0  -1.0  1.0
julia> D*[-1,0,2,1]
3-element Array{Float64,1}:
  1.0
  2.0
 -1.0
```

Since a difference matrix contains many zeros, this is a good opportunity to use sparse matrices.

```
julia> difference_matrix(n) = [-speye(n-1) spzeros(n-1)] +
       [spzeros(n-1) speye(n-1)];
julia> D = difference_matrix(4)
3×4 SparseMatrixCSC{Float64,Int64} with 6 stored entries:
  [1, 1]  =  -1.0
  [1, 2]  =  1.0
  [2, 2]  =  -1.0
  [2, 3]  =  1.0
  [3, 3]  =  -1.0
  [3, 4]  =  1.0
julia> D*[-1,0,2,1]
3-element Array{Float64,1}:
  1.0
  2.0
 -1.0
```

Running sum matrix. The running sum matrix (equation (6.6) in VMLS) is a lower triangular matrix, with elements on and below the diagonal equal to one.

```
julia> function running_sum(n)  # n x n running sum matrix
       S = zeros(n,n)
       for i=1:n
           for j=1:i
               S[i,j] = 1
```

```
            end
        end
        return S
        end
running_sum (generic function with 1 method)
julia> running_sum(4)
4×4 Array{Float64,2}:
 1.0  0.0  0.0  0.0
 1.0  1.0  0.0  0.0
 1.0  1.0  1.0  0.0
 1.0  1.0  1.0  1.0
julia> running_sum(4)*[-1,1,2,0]
4-element Array{Float64,1}:
 -1.0
  0.0
  2.0
  2.0
```

An alternative construction is `tril(ones(n,n))`. This uses the function `tril`, which sets the elements of a matrix above the diagonal to zero.

Vandermonde matrix. An $m \times n$ Vandermonde matrix (equation (6.7) in VMLS) has entries t_i^{j-1} for $i = 1,\ldots,m$ and $j = 1,\ldots,n$. We define a function that takes an m-vector with elements $t_1,\ldots,t_m$ and returns the corresponding $m \times n$ Vandermonde matrix.

```
julia> function vandermonde(t,n)
        m = length(t)
        V = zeros(m,n)
        for i=1:m
            for j=1:n
                V[i,j] = t[i]^(j-1)
            end
        end
        return V
        end
vandermonde (generic function with 1 method)
julia> vandermonde([-1,0,0.5,1],5)
4×5 Array{Float64,2}:
 1.0  -1.0  1.0   -1.0    1.0
 1.0   0.0  0.0    0.0    0.0
 1.0   0.5  0.25   0.125  0.0625
 1.0   1.0  1.0    1.0    1.0
```

An alternative shorter definition uses Julia's `hcat` function.

```
julia> vandermonde(t,n) = hcat( [t.^i for i = 0:n-1]... )
vandermonde (generic function with 1 method)
julia> vandermonde([-1,0,0.5,1],5)
4×5 Array{Float64,2}:
 1.0  -1.0  1.0   -1.0    1.0
 1.0   0.0  0.0    0.0    0.0
 1.0   0.5  0.25   0.125  0.0625
 1.0   1.0  1.0    1.0    1.0
```

6.5 Complexity

Complexity of matrix-vector multiplication. The complexity of multiplying an $m \times n$ matrix by an n-vector is $2mn$ flops. This grows linearly with both m and n. Let's check this.

```
julia> A = rand(1000,10000);  x = rand(10000);
julia> @time y = A*x;
0.022960 seconds (2.01 k allocations: 127.499 KiB)
julia> @time y = A*x;
0.006321 seconds (5 allocations: 8.094 KiB)
julia> A = rand(5000,20000);  x = rand(20000);
julia> @time y = A*x;
0.084710 seconds (6 allocations: 39.297 KiB)
julia> @time y = A*x;
0.047996 seconds (6 allocations: 39.297 KiB)
```

In the second matrix-vector multiply, m increases by a factor of 5 and n increases by a factor of 2, so the complexity predicts that the computation time should be (approximately) increased by a factor of 10. As we can see, it is increased by a factor around 7.4.

The increase in efficiency obtained by sparse matrix computations is seen from matrix-vector multiplications with the difference matrix.

```
julia> n = 10^4;
julia> D = [-eye(n-1) zeros(n-1)] + [zeros(n-1) eye(n-1)];
julia> x = randn(n);
julia> @time y=D*x;
  0.051516 seconds (6 allocations: 78.359 KiB)
julia> Ds = [-speye(n-1) spzeros(n-1)] + [spzeros(n-1) speye(n-1)];
julia> @time y=Ds*x;
```

```
0.000177 seconds (6 allocations: 78.359 KiB)
```

Chapter 7

Matrix examples

7.1 Geometric transformations

Let's create a rotation matrix, and use it to rotate a set of points $\pi/3$ radians (60°). The result is in Figure 7.1.

```
julia> Rot(theta) = [cos(theta) -sin(theta); sin(theta) cos(theta)];
julia> R = Rot(pi/3)
2×2 Array{Float64,2}:
 0.5       -0.866025
 0.866025   0.5
julia> # Create a list of 2-D points
julia> points = [ [1,0], [1.5,0], [2,0], [1,0.25], [1.5, 0.25],
       [1,.5] ];
julia> # Now rotate them.
julia> rpoints = [ R*p for p in points ];
julia> # Show the two sets of points.
julia> using Plots
julia> scatter([c[1] for c in points], [c[2] for c in points])
julia> scatter!([c[1] for c in rpoints], [c[2] for c in rpoints])
julia> plot!(lims = (-0.1, 2.1), size = (500,500), legend = false)
```

7.2 Selectors

Reverser matrix. The reverser matrix can be created from an identity matrix by reversing the order of its rows. The Julia command `reverse` can be used for this purpose. (`reverse(A,dims=1)` reverses the order of the rows of a matrix;

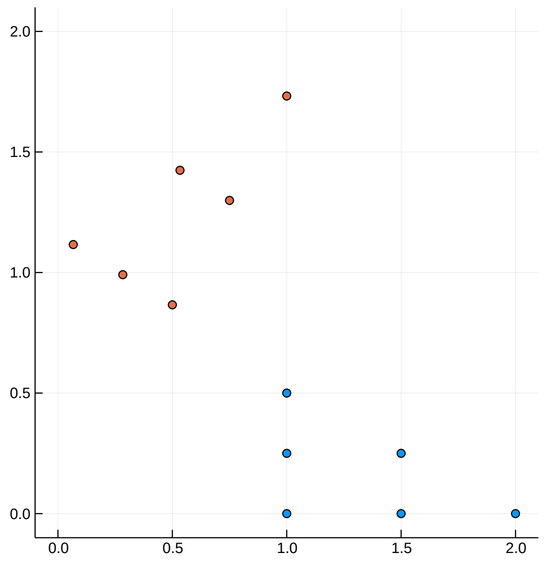


Figure 7.1 Counterclockwise rotation by 60 degrees applied to six points.

`flipdim(A,dims=2)` reverses the order of the columns.) Multiplying a vector with a reverser matrix is the same as reversing the order of its entries directly.

```
julia> reverser(n) = reverse(eye(n),dims=1)
reverser (generic function with 1 method)
julia> A = reverser(5)
5×5 Array{Float64,2}:
 0.0  0.0  0.0  0.0  1.0
 0.0  0.0  0.0  1.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0
julia> x = [1., 2., 3., 4., 5.];
julia> A*x  # Reverse x by multiplying with reverser matrix.
5-element Array{Float64,1}:
 5.0
 4.0
 3.0
 2.0
 1.0
julia> reverse(x)  # Reverse x directly.
5-element Array{Float64,1}:
 5.0
 4.0
 3.0
 2.0
 1.0
```

Permutation matrix. Let's create a permutation matrix and use it to permute the entries of a vector. In Julia, there is no reason to create a matrix to carry out the permutation, since we can do the same thing directly by passing in the permuted indexes to the vector.

```
julia> A = [0 0 1; 1 0 0; 0 1 0]
3×3 Array{Int64,2}:
 0  0  1
 1  0  0
 0  1  0
julia> x = [0.2, -1.7, 2.4]
3-element Array{Float64,1}:
  0.2
 -1.7
  2.4
julia> A*x  # Permutes entries of x to [x[3],x[1],x[2]]
3-element Array{Float64,1}:
  2.4
  0.2
 -1.7
julia> x[[3,1,2]]  # Same thing using permuted indices
3-element Array{Float64,1}:
  2.4
  0.2
 -1.7
```

7.3 Incidence matrix

Incidence matrix of a graph. We create the incidence matrix of the network shown in Figure 7.3 in VMLS.

```
julia> A = [ -1 -1 0 1 0; 1 0 -1 0 0 ; 0 0 1 -1 -1 ; 0 1 0 0 1]
4×5 Array{Int64,2}:
 -1  -1   0   1   0
```

```
  1   0  -1   0   0
  0   0   1  -1  -1
  0   1   0   0   1
julia> xcirc = [1, -1, 1, 0, 1]   # A circulation
5-element Array{Int64,1}:
  1
 -1
  1
  0
  1
julia> A*xcirc
4-element Array{Int64,1}:
 0
 0
 0
 0
julia> s = [1,0,-1,0];  # A source vector
julia> x = [0.6, 0.3, 0.6, -0.1, -0.3];  # A flow vector
julia> A*x + s   # Total incoming flow at each node
4-element Array{Float64,1}:
 1.11022e-16
 0.0
 0.0
 0.0
```

Dirichlet energy. On page 135 of VMLS we compute the Dirichlet energy of two potential vectors associated with the graph of Figure 7.2 in VMLS.

```
julia> A = [ -1 -1 0 1 0 ; 1 0 -1 0 0 ; 0 0 1 -1 -1; 0 1 0 0 1 ]
4×5 Array{Int64,2}:
 -1  -1   0   1   0
  1   0  -1   0   0
  0   0   1  -1  -1
  0   1   0   0   1
julia> vsmooth = [ 1, 2, 2, 1 ]
julia> norm(A'*vsmooth)^2  # Dirichlet energy of vsmooth
2.9999999999999996
julia> vrough = [ 1, -1, 2, -1 ]
julia> norm(A'*vrough)^2  # Dirichlet energy of vrough
27.0
```

7.4 Convolution

The Julia package `DSP` includes a convolution function `conv`. After adding this package, the command `conv(a,b)` can be used to compute the convolution of the vectors `a` and `b`. Let's use this to find the coefficients of the polynomial

$$p(x) = (1+x)(2-x+x^2)(1+x-2x^2) = 2+3x-3x^2-x^3+x^4-2x^5.$$

```
julia> Using DSP
julia> a = [1,1];  # coefficients of 1+x
julia> b = [2,-1,1];  # coefficients of 2-x+x^2
julia> c = [1,1,-2];  # coefficients of 1+x-2x^2
julia> d = conv(conv(a,b),c)  # coefficients of product
6-element Array{Int64,1}:
  2
  3
 -3
 -1
  1
 -2
```

Let's write a function that creates a Toeplitz matrix, and check it against the `conv` function. We will also check that Julia is using the very efficient method for computing the convolution.

To construct the Toeplitz matrix $T(b)$ defined in equation (7.3) of VMLS, we first create a zero matrix of the correct dimensions $((n+m-1)\times n)$ and then add the coefficients b_i one by one. Single-index indexing comes in handy for this purpose. The single-index indexes of the elements b_i in the matrix $T(b)$ are i, $i+m+n$, $i+2(m+n)$, $\ldots$, $i+(n-1)(m+n)$.

```
julia> function toeplitz(b,n)
       m = length(b)
       T = zeros(n+m-1,n)
       for i=1:m
           T[i : n+m : end] .= b[i]
       end
       return T
       end
julia> b = [-1,2,3]; a = [-2,3,-1,1];
julia> Tb = toeplitz(b, length(a))
6×4 Array{Float64,2}:
 -1.0   0.0   0.0   0.0
  2.0  -1.0   0.0   0.0
  3.0   2.0  -1.0   0.0
  0.0   3.0   2.0  -1.0
  0.0   0.0   3.0   2.0
  0.0   0.0   0.0   3.0
julia> Tb*a, conv(b,a)
([2.0, -7.0, 1.0, 6.0, -1.0, 3.0], [2, -7, 1, 6, -1, 3])
julia> m = 2000; n = 2000;
julia> b = randn(n); a=randn(m);
julia> @time ctoep = toeplitz(b,n)*a;
  0.124865 seconds (8.01 k allocations: 122.696 MiB, 5.07% gc time)
julia> @time cconv = conv(a,b);
  0.000748 seconds (164 allocations: 259.313 KiB)
julia> norm(ctoep - cconv)
2.4593600404835336e-12
```

Chapter 8

Linear equations

8.1 Linear and affine functions

Matrix-vector product function. Let's define an instance of the matrix-vector product function, and then numerically check that superpoisition holds.

```
julia> A = [-0.1 2.8 -1.6; 2.3 -0.6 -3.6] # Define 2x3 matrix A
2×3 Array{Float64,2}:
 -0.1   2.8  -1.6
  2.3  -0.6  -3.6
julia> f(x) = A*x  # Define matrix-vector product function
f (generic function with 1 method)
julia> # Let's check superposition
julia> x = [1, 2, 3];  y = [-3, -1, 2];
julia> alpha = 0.5; beta = -1.6;
julia> lhs = f(alpha*x+beta*y)
2-element Array{Float64,1}:
  9.47
 16.75
julia> rhs = alpha*f(x)+beta*f(y)
2-element Array{Float64,1}:
  9.47
 16.75
julia> norm(lhs-rhs)
1.7763568394002505e-15
julia> f([0,1,0])  # Should be second column of A
2-element Array{Float64,1}:
  2.8
 -0.6
```

De-meaning matrix. Let's create a de-meaning matrix, and check that it works on a vector.

```
julia> de_mean(n) = eye(n) .- 1/n;  # De-meaning matrix
julia> x = [0.2, 2.3, 1.0];
julia> de_mean(length(x))*x  # De-mean using matrix multiplication
3-element Array{Float64,1}:
 -0.966667
  1.13333
 -0.166667
julia> x .- avg(x)  # De-mean by subtracting mean
3-element Array{Float64,1}:
 -0.966667
  1.13333
 -0.166667
```

Examples of functions that are not linear. The componentwise absolute value and the sort function are examples of nonlinear functions. These functions are easily computed by `abs` and `sort`. By default, the `sort` function sorts in increasing order, but this can be changed by adding an optional keyword argument.

```
julia> f(x) = abs.(x)   # componentwise absolute value
f (generic function with 1 method)
julia> x = [1, 0]; y = [0, 1]; alpha = -1; beta = 2;
julia> f(alpha*x + beta*y)
2-element Array{Int64,1}:
  1
  2
julia> alpha*f(x) + beta*f(y)
2-element Array{Int64,1}:
 -1
  2
julia> f(x) = sort(x, rev = true)  # sort in decreasing order
f (generic function with 1 method)
julia> f(alpha*x + beta*y)
2-element Array{Int64,1}:
  2
 -1
julia> alpha*f(x) + beta*f(y)
2-element Array{Int64,1}:
 1
 0
```

8.2 Linear function models

Price elasticity of demand. Let's use a price elasticity of demand matrix to predict the demand for three products when the prices are changed a bit. Using this we can predict the change in total profit, given the manufacturing costs.

```
julia> p = [10, 20, 15];  # Current prices
julia> d = [5.6, 1.5, 8.6];  # Current demand (say in thousands)
julia> c = [6.5, 11.2, 9.8];  # Cost to manufacture
julia> profit = (p-c)'*d  # Current total profit
77.51999999999998
julia> # Demand elasticity matrix
julia> E = [-0.3 0.1 -0.1; 0.1 -0.5 0.05 ; -0.1 0.05 -0.4]
3×3 Array{Float64,2}:
 -0.3   0.1   -0.1
  0.1  -0.5    0.05
 -0.1   0.05  -0.4
julia> p_new = [9, 21, 14];  # Proposed new prices
julia> delta_p = (p_new-p)./p  # Fractional change in prices
3-element Array{Float64,1}:
 -0.1
  0.05
 -0.0666667
julia> delta_d = E*delta_p  # Predicted fractional change in demand
3-element Array{Float64,1}:
  0.0416667
 -0.0383333
  0.0391667
julia> d_new = d .* (1 .+ delta_d)  # Predicted new demand
3-element Array{Float64,1}:
 5.833333333333333
 1.4425
 8.936833333333333
julia> profit_new = (p_new-c)'*d_new  # Predicted new profit
66.25453333333333
```

If we trust the linear demand elasticity model, we should not make these price changes.

Taylor approximation. Consider the nonlinear function $f : \mathbf{R}^2 \to \mathbf{R}^2$ given by

$$f(x) = \left[\begin{array}{c} \|x-a\| \\ \|x-b\| \end{array} \right] = \left[\begin{array}{c} \sqrt{(x_1-a_1)^2 + (x_2-a_2)^2} \\ \sqrt{(x_1-b_1)^2 + (x_2-b_2)^2} \end{array} \right].$$

The two components of f give the distance of x to the points a and b. The function is differentiable, except when $x = a$ or $x = b$. Its derivative or Jacobian matrix is given by

$$Df(z) = \left[\begin{array}{cc} \dfrac{\partial f_1}{\partial x_1}(z) & \dfrac{\partial f_1}{\partial x_2}(z) \\[2ex] \dfrac{\partial f_2}{\partial x_1}(z) & \dfrac{\partial f_2}{\partial x_2}(z) \end{array}\right] = \left[\begin{array}{cc} \dfrac{z_1 - a_1}{\|z-a\|} & \dfrac{z_2 - a_2}{\|z-a\|} \\[2ex] \dfrac{z_1 - b_1}{\|z-b\|} & \dfrac{z_2 - b_2}{\|z-b\|} \end{array}\right].$$

Let's form the Taylor approximation of f for some specific values of a, b, and z, and then check it against the true value of f at a few points near z.

```
julia> f(x) = [ norm(x-a), norm(x-b) ];
julia> Df(z) = [ (z-a)' / norm(z-a) ;   (z-b)' / norm(z-b) ];
julia> f_hat(x) = f(z) + Df(z)*(x-z);
julia> a = [1, 0];  b = [1, 1];  z = [0, 0];
julia> f([0.1, 0.1])
2-element Array{Float64,1}:
 0.905539
 1.27279
julia> f_hat([0.1, 0.1])
2-element Array{Float64,1}:
 0.9
 1.27279
julia> f([0.5, 0.5])
2-element Array{Float64,1}:
 0.707107
 0.707107
julia> f_hat([0.5, 0.5])
2-element Array{Float64,1}:
 0.5
 0.707107
```

Regression model. We revisit the regression model for the house sales data in Section 2.3. The model is

$$\hat{y} = x^T\beta + v = \beta_1 x_1 + \beta_2 x_2 + v,$$

where $\hat{y}$ is the predicted house sale price, x_1 is the house area in 1000 square feet, and x_2 is the number of bedrooms.

In the following code we construct the 2×774 data matrix X and vector of outcomes y^{d}, for the $N = 774$ examples in the data set. We then calculate the regression model predictions $\hat{y}^{\mathrm{d}}$, the prediction errors r^{d}, and the RMS prediction error.

```
julia> # parameters in regression model
julia> beta = [148.73, -18.85];  v = 54.40;
julia> D = house_sales_data();
julia> yd = D["price"];  # vector of outcomes
julia> N = length(yd)
774
julia> X = [ D["area"]  D["beds"] ]';
julia> size(X)
(2, 774)
julia> ydhat = X'*beta .+ v;  # vector of predicted outcomes
julia> rd = yd - ydhat;  # vector of predicted errors
julia> rms(rd)  # RMS prediction error
74.84571862623022
julia> # Compare with standard deviation of prices
julia> stdev(yd)
112.7821615975651
```

8.3 Systems of linear equations

Balancing chemical reactions. We verify the linear balancing equations on page 155 of VMLS, for the simple example of electrolysis of water.

```
julia> R = [2 ; 1]
2-element Array{Int64,1}:
 2
 1
julia> P = [2 0 ; 0 2]
2×2 Array{Int64,2}:
 2  0
 0  2
julia> # Check balancing coefficients [2,2,1]
julia> coeff = [2,2,1];
julia> [R -P]*coeff
2-element Array{Int64,1}:
 0
 0
```

Chapter 9

Linear dynamical systems

9.1 Linear dynamical systems

Let's simulate a time-invariant linear dynamic system

$$x_{t+1} = Ax_t, \quad t = 1, \ldots, T,$$

with dynamics matrix

$$A = \begin{bmatrix} 0.97 & 0.10 & -0.05 \\ -0.30 & 0.99 & 0.05 \\ 0.01 & -0.04 & 0.96 \end{bmatrix}$$

and initial state $x_1 = (1, 0, -1)$. We store the state trajectory in the $n \times T$ matrix `state_traj`, with the ith column x_t. We plot the result in Figure 9.1.

```
julia> x_1 = [1,0,-1];  # initial state
julia> n = length(x_1);  T = 50;
julia> A = [ 0.97 0.10 -0.05 ; -0.3 0.99 0.05 ; 0.01 -0.04 0.96 ]
3×3 Array{Float64,2}:
  0.97   0.1   -0.05
 -0.3    0.99   0.05
  0.01  -0.04   0.96
julia> state_traj = [x_1 zeros(n,T-1) ];
julia> for t=1:T-1  # Dynamics recursion
       state_traj[:,t+1] = A*state_traj[:,t];
       end
julia> using Plots
julia> plot(1:T, state_traj', xlabel = "t",
       label = ["(x_t)_1", "(x_t)_2", "(x_t)_3"])
```

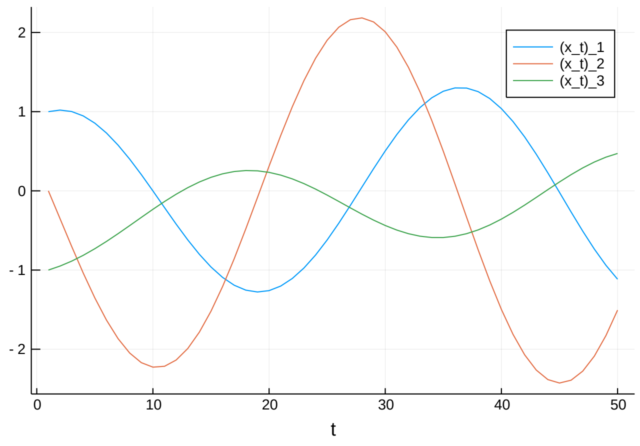


Figure 9.1 Linear dynamical system simulation.

9.2 Population dynamics

We can create a population dynamics matrix with just one simple line of Julia. The following code predicts the 2020 population distribution in the US using the data of Section 9.2 of VMLS, which are available through the `VMLS` function `population_data`. The result is shown in Figure 9.2.

```
julia> # Import 3 100-vectors: population, birth_rate, death_rate
julia> D = population_data();
julia> b = D["birth_rate"];
julia> d = D["death_rate"];
julia> A = [b'; diagonal(1 .- d[1:end-1])  zeros(length(d)-1)];
julia> x = D["population"];
julia> for k = 1:10
           global x
           x = A*x;
       end;
julia> using Plots
julia> plot(x, legend=false, xlabel = "Age",
       ylabel = "Population (millions)")
```

Note the keyword `global` in the for-loop. Without this statement, the scope of the variable `x` created by the assignment `x = A*x` would be local to the for-loop, *i.e.*, this variable does not exist outside the loop and is different from the `x` outside the loop.

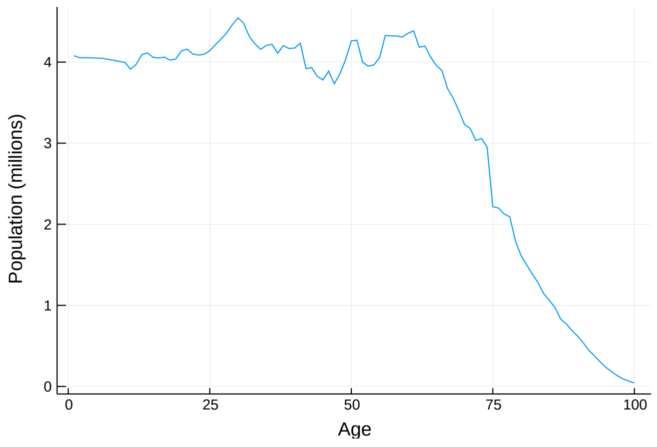


Figure 9.2 Predicted age distribution in the US in 2020.

9.3 Epidemic dynamics

Let's implement the simulation of the epidemic dynamics from VMLS §9.3. The plot is in figure 9.3.

```
julia> T = 210;
julia> A = [ 0.95 0.04 0 0 ; 0.05 0.85 0 0 ;
             0    0.10 1 0 ; 0    0.01 0 1 ];
julia> x_1 = [1,0,0,0];
julia> state_traj = [x_1 zeros(4,T-1) ];  # State trajectory
julia> for t=1:T-1  # Dynamics recursion
       state_traj[:,t+1] = A*state_traj[:,t];
       end
julia> using Plots
julia> plot(1:T, state_traj', xlabel = "Time t",
       label = ["Susceptible", "Infected", "Recovered", "Deceased"])
```

9.4 Motion of a mass

Let's simulate the discretized model of the motion of a mass in §9.4 of VMLS. See figure 9.4.

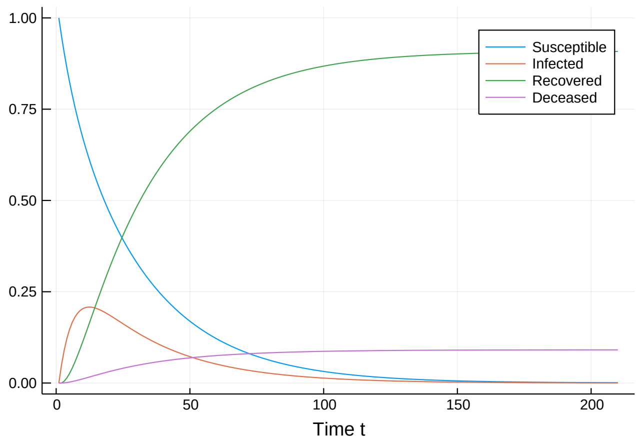


Figure 9.3 Simulation of epidemic dynamics.

```
julia> h = 0.01; m = 1; eta = 1;
julia> A = [ 1 h ; 0 1-h*eta/m ];
julia> B = [ 0 ; h/m ];
julia> x1 = [0,0];
julia> K = 600;  # simulate for K*h = 6 seconds
julia> f = zeros(K); f[50:99] .= 1.0; f[100:139] .= -1.3;
julia> X = [x1 zeros(2,K-1)];
julia> for k=1:K-1
       X[:,k+1] = A* X[:,k] + B*f[k]
       end
julia> using Plots
julia> plot(X[1,:], xlabel="k", ylabel="Position", legend=false )
julia> plot(X[2,:], xlabel="k", ylabel="Velocity", legend=false )
```

9.5 Supply chain dynamics

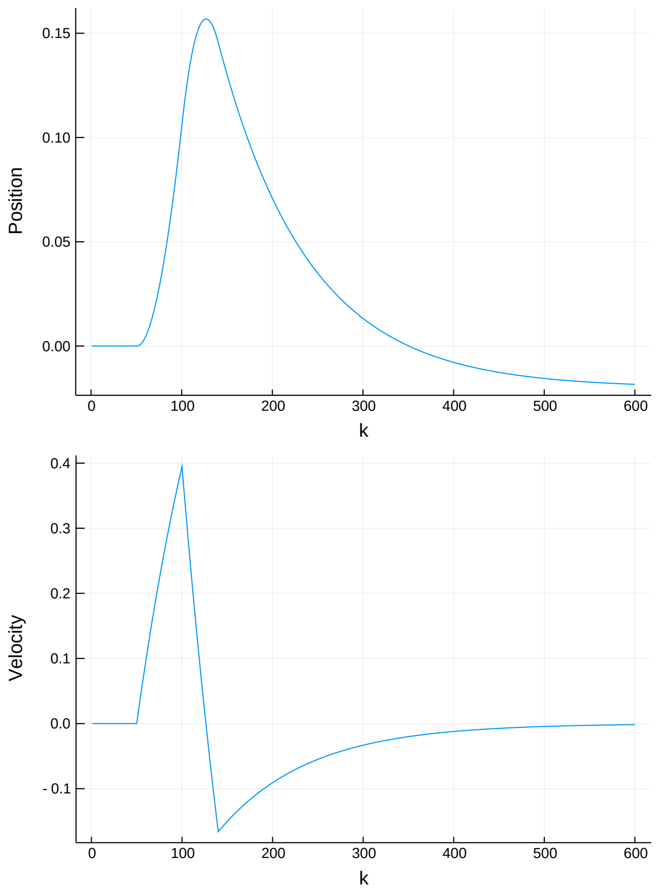


Figure 9.4 Simulation of a mass moving along a line: position (top) and velocity (bottom).

Chapter 10

Matrix multiplication

10.1 Matrix-matrix multiplication

In Julia the product of matrices `A` and `B` is obtained with `A*B`. We calculate the matrix product on page 177 of VMLS.

```
julia> A = [-1.5 3 2; 1 -1 0]
2×3 Array{Float64,2}:
 -1.5   3.0  2.0
  1.0  -1.0  0.0
julia> B = [-1 -1; 0 -2; 1 0]
3×2 Array{Int64,2}:
 -1  -1
  0  -2
  1   0
julia> C = A*B
2×2 Array{Float64,2}:
  3.5  -4.5
 -1.0   1.0
```

Gram matrix. The Gram matrix of a matrix A is the matrix $G = A^T A$. It is a symmetric matrix and the i, j element G_{ij} is the inner product of columns i and j of A.

```
julia> A = randn(10,3);
julia> G = A'*A
3×3 Array{Float64,2}:
 11.1364   -3.91865  -3.69057
 -3.91865   7.98358   2.4839
 -3.69057   2.4839   10.956
```

```
julia> # Gii is norm of column i, squared
julia> G[2,2]
7.983579175590987
julia> norm(A[:,2])^2
7.983579175590987
julia> # Gij is inner product of columns i and j
julia> G[1,3]
-3.6905664879621454
julia> A[:,1]'*A[:,3]
-3.6905664879621463
```

Complexity of matrix triple product. Let's check the associative property, which states that $(AB)C = A(BC)$ for any $m \times n$ matrix A, any $n \times p$ matrix B, and any $p \times q$ matrix B. At the same time we will see that the left-hand and right-hand sides take very different amounts of time to compute.

```
julia> m = 2000; n = 50; q = 2000; p = 2000;
julia> A = randn(m,n);  B = randn(n,p); C = randn(p,q);
julia> @time LHS = (A*B)*C;
  0.819912 seconds (245.32 k allocations: 72.557 MiB, 13.47% gc time)
julia> @time LHS = (A*B)*C;
  0.254107 seconds (8 allocations: 61.035 MiB, 20.85% gc time)
julia> @time RHS = A*(B*C);
  0.030907 seconds (9 allocations: 31.281 MiB, 13.95% gc time)
julia> @time RHS = A*(B*C);
  0.023507 seconds (8 allocations: 31.281 MiB, 3.13% gc time)
julia> norm(LHS-RHS)
5.334805188873507e-10
julia> @time D = A*B*C;   # evaluated as (A*B)*C or as A*(B*C)?
  0.220616 seconds (1.03 k allocations: 61.098 MiB, 5.09% gc time)
```

We see that evaluating `(A*B)*C` takes around 10 times as much time as evaluating `A*(B*C)`, which is predicted from the complexities. In the last line we deduce that `A*B*C` is evaluated left to right, as `(A*B)*C`. Note that for these particular matrices, this is the (much) slower order to multiply the matrices.

10.2 Composition of linear functions

Second difference matrix. We compute the second difference matrix on page 184 of VMLS.

```
julia> D(n) = [-eye(n-1) zeros(n-1)] + [zeros(n-1) eye(n-1)];
julia> D(5)
4×5 Array{Float64,2}:
 -1.0   1.0   0.0   0.0  0.0
  0.0  -1.0   1.0   0.0  0.0
  0.0   0.0  -1.0   1.0  0.0
  0.0   0.0   0.0  -1.0  1.0
julia> D(4)
3×4 Array{Float64,2}:
 -1.0   1.0   0.0  0.0
  0.0  -1.0   1.0  0.0
  0.0   0.0  -1.0  1.0
julia> Delta = D(4)*D(5)  # Second difference matrix
3×5 Array{Float64,2}:
 1.0  -2.0   1.0   0.0  0.0
 0.0   1.0  -2.0   1.0  0.0
 0.0   0.0   1.0  -2.0  1.0
```

10.3 Matrix power

The kth power of a square matrix A is denoted A^k. In Julia, this power is formed using `A^k`.

Let's form the adjacency matrix of the directed graph on VMLS page 186. Then let's find out how many cycles of length 8 there are, starting from each node. (A cycle is a path that starts and stops at the same node.)

```
julia> A = [ 0 1 0 0 1; 1 0 1 0 0; 0 0 1 1 1; 1 0 0 0 0; 0 0 0 1 0]
5×5 Array{Int64,2}:
 0  1  0  0  1
 1  0  1  0  0
 0  0  1  1  1
 1  0  0  0  0
 0  0  0  1  0
julia> A^2
5×5 Array{Int64,2}:
 1  0  1  1  0
 0  1  1  1  2
 1  0  1  2  1
 0  1  0  0  1
 1  0  0  0  0
```

```
julia> A^8
5×5 Array{Int64,2}:
 18  11  15  20  20
 25  14  21  28  26
 24  14  20  27  26
 11   6   9  12  11
  6   4   5   7   7
julia> number_of_cycles = diag(A^8)
5-element Array{Int64,1}:
 18
 14
 20
 12
  7
```

Population dynamics. Let's write the code for figure 10.2 in VMLS, which plots the contribution factor to the total US population in 2020 (ignoring immigration), for each age in 2010. The Julia plot is in figure 10.1. We can see that, not surprisingly, 20–25 year olds have the highest contributing factor, around 1.5. This means that on average, each 20-25 year old in 2010 will be responsible for around 1.5 people in 2020. This takes into account any children they may have before then, and (as a very small effect) the few of them who will no longer be with us in 2020.

```
julia> D = population_data();
julia> b = D["birth_rate"];
julia> d = D["death_rate"];
julia> # Dynamics matrix for populaion dynamics
julia> A = [b'; diagonal(1 .- d[1:end-1]) zeros(length(d)-1)];
julia> # Contribution factor to total poulation in 2020
julia> # from each age in 2010
julia> cf = ones(100)'*(A^10);  # Contribution factor
julia> using Plots
julia> plot(cf', legend = false, xlabel = "Age", ylabel = "Factor")
```

10.4 QR factorization

In Julia, the QR factorization of a matrix `A` can be found using `qr(A)`, which returns a tuple with the Q and R factors. However the matrix Q is not returned as

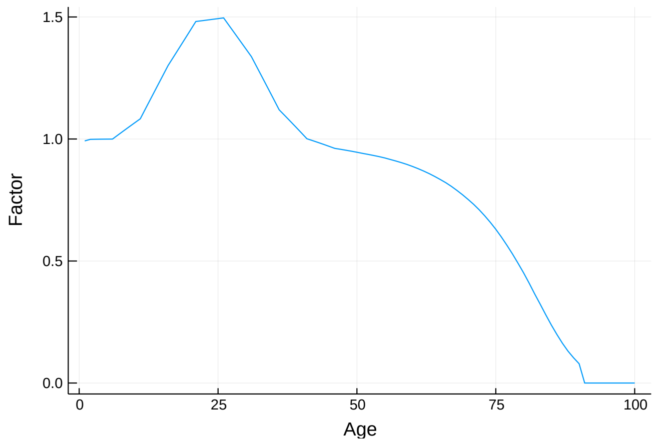


Figure 10.1 Contribution factor per age in 2010 to the total population in 2020. The value for age $i-1$ is the ith component of the row vector $\mathbf{1}^T A^{10}$.

an array, but in a special compact format. It can be converted to a regular matrix variable using the command `Matrix(Q)`. Hence, the QR factorization as defined in VMLS is computed by a sequence of two commands:

```
julia> Q, R = qr(A);
julia> Q = Matrix(Q);
```

The following example also illustates a second, but minor difference with the VMLS definition. The R factor computed by Julia may have negative elements on the diagonal, as opposed to only positive elements if we follow the definition used in VMLS. The two definitions are equivalent, because if R_{ii} is negative, one can change the sign of the ith row of R and the ith column of Q, to get an equivalent factorization with $R_{ii} > 0$. However this step is not needed in practice, since negative elements on the diagonal do not pose any problem in applications of the QR factorization.

```
julia> A = randn(6,4);
julia> Q, R = qr(A);
julia> R
4×4 Array{Float64,2}:
 1.36483  -1.21281  -0.470052  -1.40011
 0.0      -2.01191  -2.92458   -0.802368
 0.0       0.0      -1.94759    0.84228
 0.0       0.0       0.0        1.19766
julia> Q = Matrix(Q)
6×4 Array{Float64,2}:
 -0.454997    -0.442971  -0.00869016   0.121067
 -0.577229    -0.301916   0.349002    -0.370212
  0.00581297  -0.159604  -0.647797     0.28373
  0.221266     0.388812   0.223813    -0.46152
 -0.550089     0.546873  -0.507827    -0.30155
 -0.328929     0.48673    0.387945     0.681065
julia> norm(Q*R-A)
1.0046789954275695e-15
julia> Q'*Q
4×4 Array{Float64,2}:
  1.0           0.0           -2.77556e-17   2.77556e-17
  0.0           1.0           -8.32667e-17  -1.66533e-16
 -2.77556e-17  -8.32667e-17   1.0            5.55112e-17
  2.77556e-17  -1.66533e-16   5.55112e-17   1.0
```

Chapter 11

Matrix inverses

11.1 Left and right inverses

We'll see later how to find a left or right inverse, when one exists.

```
julia> A = [-3 -4; 4 6; 1 1]
3×2 Array{Int64,2}:
 -3  -4
  4   6
  1   1
julia> B = [-11 -10 16; 7 8 -11]/9  # A left inverse of A
2×3 Array{Float64,2}:
 -1.22222   -1.11111    1.77778
  0.777778   0.888889  -1.22222
julia> C = [0 -1 6; 0 1 -4]/2  # Another left inverse of A
2×3 Array{Float64,2}:
 0.0  -0.5   3.0
 0.0   0.5  -2.0
julia> # Let's check
julia> B*A
2×2 Array{Float64,2}:
  1.0          0.0
 -4.44089e-16  1.0
julia> C*A
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

11.2 Inverse

If `A` is invertible, its inverse is given by `inv(A)` (and also `A^-1`). You'll get an error if `A` is not invertible, or not square.

```
julia> A = [1 -2 3; 0 2 2; -4 -4 -4]
3×3 Array{Int64,2}:
  1  -2   3
  0   2   2
 -4  -4  -4
julia> B = inv(A)
3×3 Array{Float64,2}:
  0.0  -0.5  -0.25
 -0.2   0.2  -0.05
  0.2   0.3   0.05
julia> B*A
3×3 Array{Float64,2}:
 1.0  0.0          0.0
 0.0  1.0          2.77556e-17
 0.0  5.55112e-17  1.0
julia> A*B
3×3 Array{Float64,2}:
  1.0           1.11022e-16  0.0
  5.55112e-17   1.0          1.38778e-17
 -1.11022e-16  -2.22045e-16  1.0
```

Dual basis. The next example illustrates the dual basis provided by the rows of the inverse $B = A^{-1}$. We calculate the expansion

$$x = (b_1^T x)a_1 + \cdots + (b_n^T x)a_n$$

for a 3×3 example (see page 205 of VMLS).

```
julia> A = [ 1 0 1; 4 -3 -4; 1 -1 -2]
3×3 Array{Int64,2}:
 1   0   1
 4  -3  -4
 1  -1  -2
julia> B = inv(A)
3×3 Array{Float64,2}:
  2.0  -1.0   3.0
  4.0  -3.0   8.0
 -1.0   1.0  -3.0
```

```
julia> x = [ 0.2, -0.3, 1.2]
3-element Array{Float64,1}:
  0.2
 -0.3
  1.2
julia> rhs = (B[1,:]'*x) * A[:,1] + (B[2,:]'*x) * A[:,2]
        + (B[3,:]'*x) * A[:,3]
3-element Array{Float64,1}:
  0.2
 -0.3
  1.2
```

Inverse via QR factorization. The inverse of a matrix A can be computed from its QR factorization $A = QR$ via the formula $A^{-1} = R^{-1}Q^T$.

```
julia> A = randn(3,3);
julia> inv(A)
3×3 Array{Float64,2}:
 -0.321679   0.323945  -0.347063
 -1.0735    -5.03083   -3.24503
 -1.17582   -2.68161   -1.8496
julia> Q, R = qr(A);
julia> Q = Matrix(Q);
julia> inv(R)*Q'
3×3 Array{Float64,2}:
 -0.321679   0.323945  -0.347063
 -1.0735    -5.03083   -3.24503
 -1.17582   -2.68161   -1.8496
```

11.3 Solving linear equations

Back substitution. Let's first implement back substitution (VMLS Algorithm 11.1) in Julia, and check it. You won't need this function, since Julia has a better implementation of it built in (via the backslash operation discussed below). We give it here only to demonstrate that it works.

```
julia> function back_subst(R,b)
       n = length(b)
       x = zeros(n)
       for i=n:-1:1
           x[i] = (b[i] - R[i,i+1:n]'*x[i+1:n]) / R[i,i]
       end
       return x
       end;
julia> R = triu(randn(4,4))  # Random 4x4 upper triangular matrix
4×4 Array{Float64,2}:
 -0.498881  -0.880538  -0.745078      0.125678
  0.0       -0.922477  -0.00673699    0.30122
  0.0        0.0       -0.283035     -0.0184466
  0.0        0.0        0.0          -2.01396
julia> b = rand(4);
julia> x = back_subst(R,b);
julia> norm(R*x-b)
2.220446049250313e-16
```

The function `triu` gives the upper triangular part of a matrix, *i.e.*, it zeros out the entries below the diagonal.

Backslash notation. The Julia command for solving a set of linear equations

$$Ax = b$$

is `x=A\b`. This is faster than `x=inv(A)*b`, which first computes the inverse of A and then multiplies it with b.

```
julia> n = 5000;
julia> A = randn(n,n);  b = randn(n);  # random set of equations
julia> @time x1 = A\b;
  1.422173 seconds (14 allocations: 190.812 MiB, 0.73% gc time)
julia> norm(b-A*x1)
3.8666263141510634e-9
julia> @time x2 = inv(A)*b;
  4.550091 seconds (21 allocations: 384.026 MiB, 1.66% gc time)
julia> norm(b-A*x2)
1.682485195063787e-8
```

Julia chooses a suitable algorithm for solving the equation after checking the properties of A. For example, it will use back substitution if A is lower triangular. This explains the result in the following timing experiment.

```
julia> n = 5000;
julia> b = randn(n);
julia> A = tril(randn(n,n));  # random lower triangular matrix
julia> @time x = A\b;
  0.042580 seconds (7 allocations: 39.313 KiB)
julia> A = randn(n,n);  # random square matrix
julia> @time x = A\b;
  1.289663 seconds (14 allocations: 190.812 MiB, 0.91% gc time)
julia> n = 10000;
julia> b = randn(n);
julia> A = tril(randn(n,n));  # random lower triangular matrix
julia> @time x = A\b;
  0.157043 seconds (7 allocations: 78.375 KiB)
julia> A = randn(n,n);  # random square matrix
julia> @time x = A\b;
  9.008604 seconds (14 allocations: 763.093 MiB, 0.41% gc time)
```

When we double the size from $n = 5000$ to $n = 10000$, the solution time for the triangular equation increases from 0.04 seconds to 1.29 seconds. This is a factor of about four, consistent with the n^2 complexity of backsubstitution. For the general square system, the solution times increases from 0.16 seconds to 9.01 seconds, *i.e.*, a factor of roughly eight, as we would expect given the order n^3 complexity.

Factor-solve methods for multiple right-hand sides. A linear equation is solved by first factorizing A and then solving several simpler equations with the factors of A. This is referred to as a *factor-solve* scheme. An important application is the solution of multiple linear equations with the same coefficient matrix and different right-hand sides.

```
julia> n = 5000;
julia> A = randn(n,n); B = randn(n,2);
julia> # Solve with right-hand side B[:,1]
julia> @time x1 = A \ B[:,1];
  1.501368 seconds (17 allocations: 190.850 MiB, 0.79% gc time)
julia> # Solve with right-hand side B[:,2]
julia> @time x2 = A \ B[:,2];
  1.388827 seconds (17 allocations: 190.850 MiB, 0.81% gc time)
julia> # Naive approach for solving A*X = B
julia> @time X = [ A\B[:,1]  A\B[:,2] ];
  2.617617 seconds (35 allocations: 381.776 MiB, 3.19% gc time)
julia> # Factor-solve approach
julia> @time X = A \ B;
  1.418451 seconds (83.37 k allocations: 194.881 MiB, 0.58% gc time)
```

The factor-solve approach finds the solutions for the two right-hand sides in roughly the same time as the solution for one right-hand side. The solution time with the naïve approach is twice the time for one right-hand side.

11.4 Examples

Polynomial interpolation. We compute the interpolating polynomials in Figure 11.1 of VMLS. The following code uses the functions `vandermonde` and `linspace` from the `VMLS` package. (Th function `linspace(a,b,n)` returns a vector with n equally spaced numbers in the interval $[a, b]$.) The result is shown in Figure 11.1.

```
julia> t = [ -1.1, -0.4, 0.2, 0.8 ];
julia> A = vandermonde(t, 4)
4×4 Array{Float64,2}:
 1.0  -1.1  1.21  -1.331
 1.0  -0.4  0.16  -0.064
 1.0   0.2  0.04   0.008
 1.0   0.8  0.64   0.512
julia> b1 = [ -1.0, 1.3, 1.0, 0.5 ];
julia> c1 = A \  b1
4-element Array{Float64,1}:
  1.21096
 -0.888311
 -1.10967
  1.38648
julia> b2 = [ 1.0, 0.0, -2.0, 0 ];
julia> c2 = A \ b2
4-element Array{Float64,1}:
 -1.54129
 -3.10905
  3.33847
  3.69514
julia> using Plots
julia> ts = linspace(-1.2, 1.2, 1000);
julia> p1 = c1[1] .+ c1[2]*ts + c1[3]*ts.^2  + c1[4]*ts.^3;
julia> plot(ts, p1)
julia> scatter!(t, b1)
julia> p2 = c2[1] .+ c2[2]*ts + c2[3]*ts.^2  + c2[4]*ts.^3;
julia> plot!(ts, p2)
julia> scatter!(t, b2, maker = :square)
```

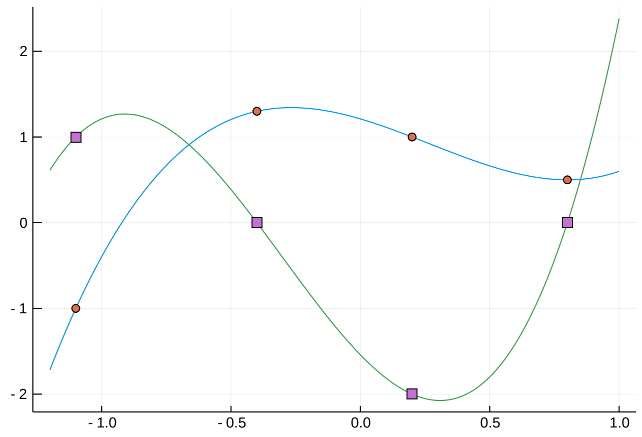


Figure 11.1 Cubic interpolants through two sets of points, shown as circles and squares.

11.5 Pseudo-inverse

In Julia, the pseudo-inverse of a matrix `A` is obtained with `pinv(A)`. We compute the pseudo-inverse for the example on page 216 of VMLS using the `pinv` function, and via the formula $A^\dagger = R^{-1}Q^T$, where $A = QR$ is the QR factorization of A.

```
julia> A = [-3 -4; 4 6; 1 1]
3×2 Array{Int64,2}:
 -3  -4
  4   6
  1   1
julia> pinv(A)
2×3 Array{Float64,2}:
 -1.22222   -1.11111    1.77778
  0.777778   0.888889  -1.22222
julia> Q, R = qr(A);
julia> Q = Matrix(Q)
3×2 Array{Float64,2}:
 -0.588348  -0.457604
  0.784465  -0.522976
  0.196116   0.719092
julia> R
2×2 Array{Float64,2}:
 5.09902   7.2563
 0.0      -0.588348
```

```
julia> R \ Q'   # pseudo-inverse from QR factors
2×3 Array{Float64,2}:
 -1.22222   -1.11111    1.77778
  0.777778   0.888889  -1.22222
```

Chapter 12

Least squares

12.1 Least squares problem

We take the small least squares problem of Figure 12.1 in VMLS and check that $\|A\hat{x} - b\|$ is less than $\|Ax - b\|$ for some other value of x.

```
julia> A = [ 2 0 ; -1 1 ; 0 2 ]
3×2 Array{Int64,2}:
  2  0
 -1  1
  0  2
julia> b = [ 1, 0, -1 ]
3-element Array{Int64,1}:
  1
  0
 -1
julia> xhat = [ 1/3, -1/3 ]
2-element Array{Float64,1}:
  0.333333
 -0.333333
julia> rhat = A*xhat -b
3-element Array{Float64,1}:
 -0.333333
 -0.666667
  0.333333
julia> norm(rhat)
0.816496580927726
julia> x = [ 1/2, -1/2 ]
2-element Array{Float64,1}:
  0.5
 -0.5
julia> r = A*x -b
3-element Array{Float64,1}:
  0.0
 -1.0
  0.0
julia> norm(r)
  1.0
```

12.2 Solution

Least squares solution formula. Let's check the solution formulas (12.5) and (12.6) in VMLS,

$$\hat{x} = (A^TA)^{-1}A^Tb = A^\dagger b$$

for the small example of Figure 12.1) (where $\hat{x} = (1/3, 1/3)$).

```
julia> inv(A'*A)*A'*b
2-element Array{Float64,1}:
  0.333333
 -0.333333
julia> pinv(A)*b
2-element Array{Float64,1}:
  0.333333
 -0.333333
julia> (A'*A)*xhat - A'*b  # Check that normal equations hold
2-element Array{Float64,1}:
  0.0
 -8.88178e-16
```

Orthogonality principle. Let's check the orthogonality principle (12.9), for the same example.

```
julia> z = [-1.1, 2.3];
julia> (A*z)'*rhat
2.220446049250313e-16
julia> z = [5.3, -1.2];
julia> (A*z)'*rhat
-6.661338147750939e-16
```

12.3 Solving least squares problems

Julia uses the backslash operator to denote the least squares approximate solution: `xhat = A\b`. (The same operator is used to solve square systems of linear equations, and we will see more uses of it in later chapters.)

```
julia> A = randn(100,20);  b = randn(100);
julia> x1 = A\b;  # Least squares using backslash operator
julia> x2 = inv(A'*A)*(A'*b);  # Using formula
julia> x3 = pinv(A)*b;  # Using pseudo-inverse
julia> Q, R = qr(A);
julia> Q = Matrix(Q);
julia> x4 = R\(Q'*b);  # Using QR factorization
julia> norm(x1-x2)
4.258136640215341e-16
julia> norm(x2-x3)
1.3328728382991758e-15
julia> norm(x3-x4)
1.3507615689695538e-15
```

Complexity. The complexity of solving the least squares problem with $m \times n$ matrix A is around $2mn^2$ flops. Let's check this in Julia by solving a few least squares problems of different dimensions.

```
julia> m = 2000; n = 500;
julia> A = randn(m,n); b = randn(m);
julia> @time x = A\b;
 0.190497 seconds (4.07 k allocations: 12.031 MiB, 28.89% gc time)
julia> @time x = A\b;
 0.120246 seconds (4.07 k allocations: 12.031 MiB)
julia> m = 4000; n = 500;
julia> A = randn(m,n); b = randn(m);
julia> @time x = A\b;
 0.248510 seconds (4.07 k allocations: 19.675 MiB)
julia> m = 2000; n = 1000;
julia> A = randn(m,n); b = randn(m);
julia> @time x = A\b;
0.418181 seconds (8.07 k allocations: 31.608 MiB, 1.66% gc time)
```

We can see that doubling m approximately doubles the computation time, and doubling n increases it by around a factor of four. The times above can be used to guess the speed of the computer on which it was carried out. For example, using the last problem solved, the number of flops is around $2mn^2 = 4 \cdot 10^9$, and it took around 0.4 seconds. This suggests a speed of around 10 Gflop/sec.

Matrix least squares. Let's solve multiple least squares problems with the same matrix A and different vectors b.

```
julia> A = randn(1000,100); B = randn(1000,10);
julia> X = A\B;
julia> # Check that third column of X is least squares solution
julia> # with third column of B
julia> x3 = A\B[:,3];
julia> norm(X[:,3]-x3)
1.1037090583270415e-16
```

12.4 Examples

Advertising purchases. We work out the solution of the optimal advertising purchase problem on page 234 of VMLS.

```
julia> R = [ 0.97  1.86  0.41;
             1.23  2.18  0.53;
             0.80  1.24  0.62;
             1.29  0.98  0.51;
             1.10  1.23  0.69;
             0.67  0.34  0.54;
             0.87  0.26  0.62;
             1.10  0.16  0.48;
             1.92  0.22  0.71;
             1.29  0.12  0.62];
julia> m, n = size(R);
julia> vdes = 1e3 * ones(m);
julia> s = R \ vdes
31 Array{Float64,2}:
   62.0766
   99.985
 1442.84
julia> rms(R*s - vdes)
132.63819026326527
```

Illumination. The following code constructs and solves the illumination problem on page 234, and plots two histograms with the pixel intensity distributions (Figure 12.1).

```
julia> n = 10;  # number of lamps
julia> lamps = [ # x, y positions of lamps and height above floor
     4.1  20.4  4;
    14.1  21.3  3.5;
    22.6  17.1  6;
     5.5  12.3  4.0;
    12.2   9.7  4.0;
    15.3  13.8  6;
    21.3  10.5  5.5;
     3.9   3.3  5.0;
    13.1   4.3  5.0;
    20.3   4.2  4.5 ];
julia> N = 25;   # grid size
julia> m = N*N;  # number of pixels
julia> # construct m x 2 matrix with coordinates of pixel centers
julia> pixels = hcat(
           reshape( collect(0.5: 1 : N) * ones(1,N), m, 1),
           reshape( ones(N,1) * collect(0.5: 1 : N)', m, 1) );
julia> # The m x n matrix A maps lamp powers to pixel intensities.
julia> # A[i,j] is inversely proportional to the squared distance of
julia> # lamp j to pixel i.
julia> A = zeros(m,n);
julia> for i=1:m
           for j=1:n
               A[i,j] = 1.0 / norm([pixels[i,:]; 0] - lamps[j,:])^2;
           end;
       end;
julia> A = (m/sum(A)) * A;   # scale elements of A
julia> # Least squares solution
julia> x = A \ ones(m,1);
julia> rms_ls = rms(A*x .- 1)
0.1403904813427606
julia> using Plots
julia> histogram(A*x, bins = (0.375:0.05:1.625),
           legend = false, ylim = (0,120))
julia> # Intensity if all lamp powers are one
julia> rms_uniform = rms(A*ones(n,1) .- 1)
0.24174131853807881
julia> histogram(A*ones(n,1), bins = (0.375:0.05:1.625),
           legend = false, ylim = (0,120))
```

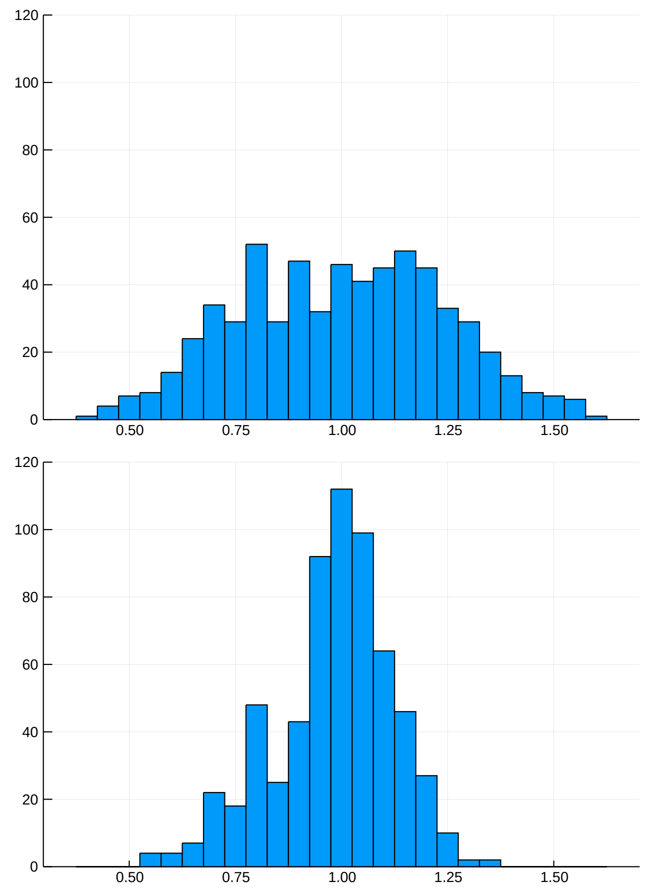


Figure 12.1 Histogram of pixel illumination values using $p = \mathbf{1}$ (top) and $\hat{p}$ (bottom). The target intensity value is one.

Chapter 13

Least squares data fitting

13.1 Least squares data fitting

Straight-line fit. A straight-line fit to time series data gives an estimate of a trend line. In Figure 13.3 of VMLS we apply this to a time series of petroleum consumption. The figure is reproduced here as Figure 13.1.

```
julia> # Petroleum consumption in thousand barrels/day
julia> consumption = petroleum_consumption_data()
julia> n = length(consumption);
julia> A = [ ones(n) 1:n ];
julia> x = A \ consumption;
julia> using Plots
julia> scatter(1980:2013, consumption, legend=false)
julia> plot!(1980:2013, A*x)
```

Estimation of trend and seasonal component. The next example is the least squares fit of a trend plus a periodic component to a time series. In VMLS this was illustrated with a time series of vehicle miles traveled in the US, per month, for 15 years (2000–2014). The following Julia code replicates Figure 13.5 in VMLS. It imports the data via the function `vehicle_miles_data`, which creates a 15×12 matrix `vmt`, with the monthly values for each of the 15 years.

```
julia> vmt = vehicle_miles_data();  # creates 15x12 matrix vmt
julia> m = 15*12;
julia> A = [ 0:(m-1)  vcat([eye(12) for i=1:15]...) ];
julia> b = reshape(vmt', m, 1);
julia> x = A \ b;
julia> using Plots
```

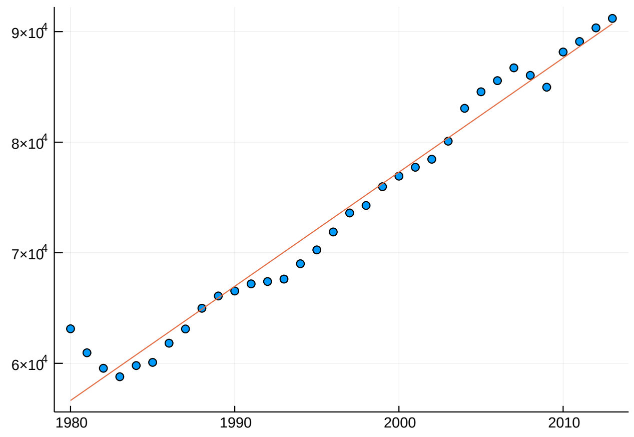


Figure 13.1 World petroleum consumption between 1980 and 2013 (dots) and least squares straight-line fit (data from www.eia.gov).

```
julia> scatter(1:m, b, markersize = 2, legend =false);
julia> plot!(1:m, A*x)
```

The matrix A in this example has size $m \times n$ where $m = 15 \cdot 12 = 180$ and $n = 13$. The first column has entries $0, 1, 2, \ldots, 179$. The remaining columns are formed by vertical stacking of 15 identity matrices of size 12×12. The Julia expression `vcat([eye(12) for i=1:15]...)` creates an array of 15 identity matrices, and then stacks them vertically. The plot produced by the code is shown in Figure 13.2.

Polynomial fit. We now discuss the polynomial fitting problem on page 255 in VMLS and the results shown in Figure 13.6. We first generate a training set of 100 points and plot them (Figure 13.3).

```
julia> # Generate training data in the interval [-1, 1].
julia> m = 100;
julia> t = -1 .+ 2*rand(m,1);
julia> y = t.^3 - t + 0.4 ./ (1 .+ 25*t.^2) + 0.10*randn(m,1);
julia> using Plots
julia> scatter(t,y,legend=false)
```

Next we define a function that fits the polynomial coefficients using least squares. We apply the function to fit polynomials of degree $2, 6, 10, 15$ to our training set.

```
julia> polyfit(t, y, p) = vandermonde(t, p) \ y
julia> theta2 = polyfit(t,y,3)
```

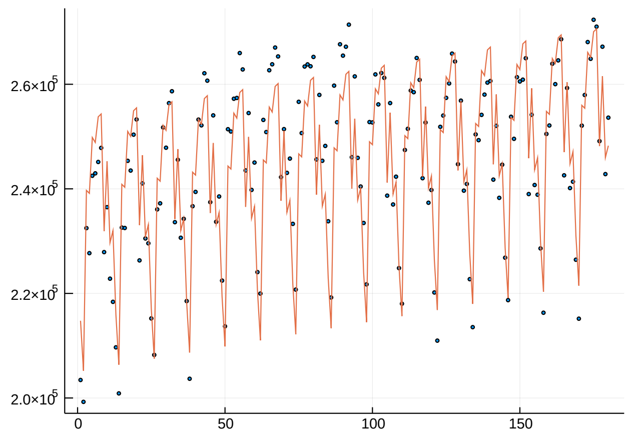


Figure 13.2 The dots show vehicle miles traveled in the US, per month, in the period January 2000–December 2014. The line shows the least squares fit of a linear trend and a seasonal component with a 12-month period.

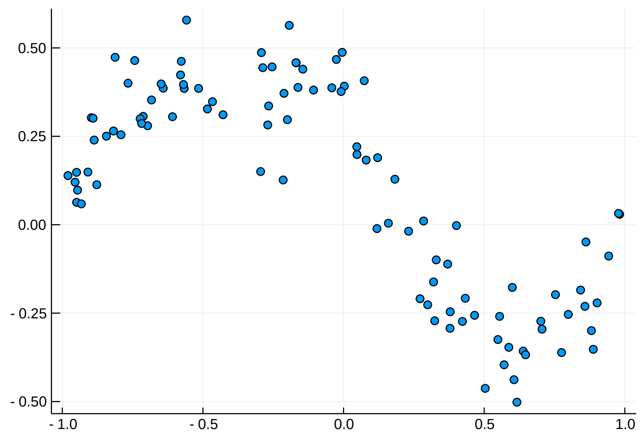


Figure 13.3 Training set used in the polynomial fitting example.

```
julia> theta6 = polyfit(t,y,7)
julia> theta10 = polyfit(t,y,11)
julia> theta15 = polyfit(t,y,16)
```

Finally, we plot the four polynomials. To simplify this, we first write a function that evaluates a polynomial at all points specified in a vector `x`. The plots are in Figure 13.4.

```
julia> polyeval(theta, x) = vandermonde(x,length(theta))*theta;
julia> t_plot = linspace(-1,1,1000);
julia> using Plots
julia> p = plot(layout=4, legend=false, ylim=(-0.7. 0.7))
julia> scatter!(t, y, subplot=1, markersize = 2)
julia> plot!(t_plot, polyeval(theta2,t_plot), subplot=1)
julia> scatter!(t, y, subplot=2, markersize = 2)
julia> plot!(t_plot, polyeval(theta6,t_plot), subplot=2)
julia> scatter!(t, y, subplot=3,markersize = 2)
julia> plot!(t_plot, polyeval(theta10,t_plot), subplot=3)
julia> scatter!(t, y, subplot=4, markersize = 2)
julia> plot!(t_plot, polyeval(theta15,t_plot), subplot=4)
```

Piecewise-linear fit. In the following code least squares is used to fit a piecewise-linear function to 100 points. It produces Figure 13.5, which is similar to Figure 13.8 in VMLS.

```
julia> # generate random data
julia> m = 100;
julia> x = -2 .+ 4*rand(m,1);
julia> y = 1 .+ 2*(x.-1) - 3*max.(x.+1,0) + 4*max.(x.-1,0)
       + 0.3*randn(m,1);
julia> # least squares fitting
julia> theta = [ ones(m)  x   max.(x.+1,0)  max.(x.-1,0) ] \ y;
julia> # plot result
julia> using Plots
julia> t = [-2.1, -1, 1, 2.1];
julia> yhat = theta[1] .+ theta[2]*t + theta[3]*max.(t.+1,0) +
       theta[4]*max.(t.-1,0);
julia> scatter(x, y, legend=false)
julia> plot!(t, yhat)
```

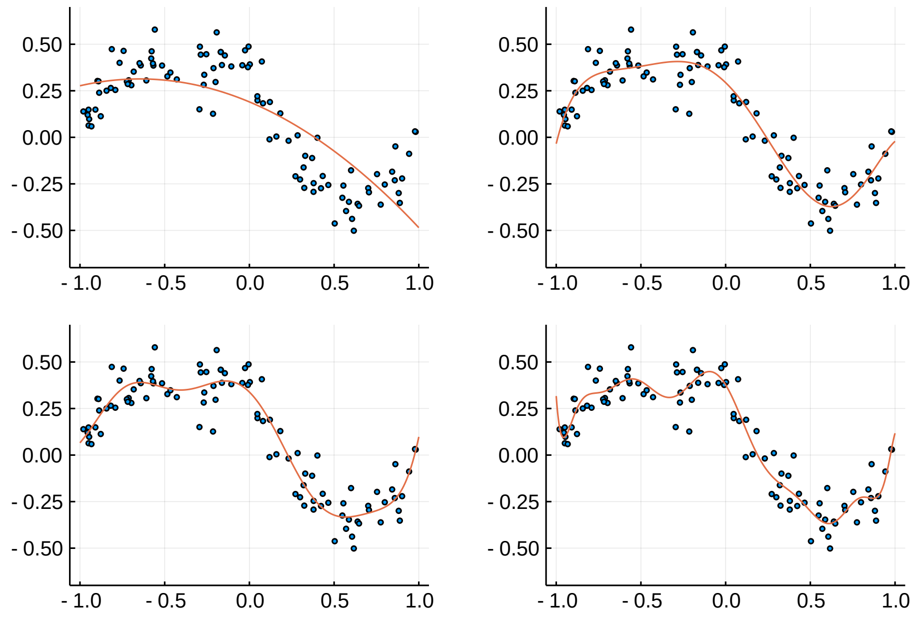
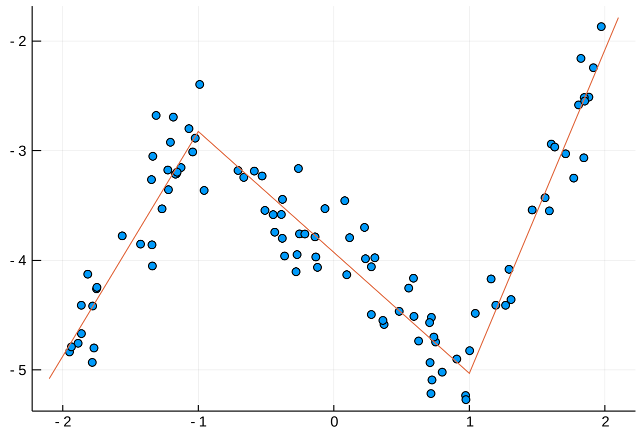


Figure 13.4 Least squares polynomial fits of degree 1, 6, 10, and 15 to 100 points



;

Figure 13.5 Piecewise-linear fit to 100 points.

House price regression. We calculate the simple regression model for predicting house sales price from area and number of bedrooms, using the data of 774 house sales in Sacramento.

```
julia> D = house_sales_data();  # creates 3 vectors: area, beds, price
julia> area = D["area"];
julia> beds = D["beds"];
julia> price = D["price"];
julia> m = length(price);
julia> A = [ ones(m) area beds ];
julia> x = A \ price
3-element Array{Float64,1}:
  54.4017
 148.725
 -18.8534
julia> rms_error = rms(price - A*x)
74.84571649590146
julia> std_prices = stdev(price)
112.7821615975651
```

Auto-regressive time series model. In the following Julia code we fit an auto-regressive model to the temperature time series discussed on page 259 of VMLS. In Figure 13.6 we compare the first five days of the model predictions with the data.

```
julia> # import time series of temperatures t
julia> t = temperature_data();
julia> N = length(t)
744
julia> stdev(t) # Standard deviation
3.05055928562933
julia> # RMS error for simple predictor zhat_{t+1} = z_t
rms(t[2:end] - t[1:end-1])
1.1602431638206119
julia> # RMS error for simple predictor zhat_{t+1} = z_{t-23}
rms(t[25:end] - t[1:end-24])
1.7338941400468744
julia> # Least squares fit of AR predictor with memory 8
julia> M = 8
julia> y = t[M+1:end];
julia> A = hcat( [ t[i:i+N-M-1] for i = M:-1:1]...);
julia> theta = A \ y;
julia> ypred = A*theta;
julia> # RMS error of LS AR fit
```

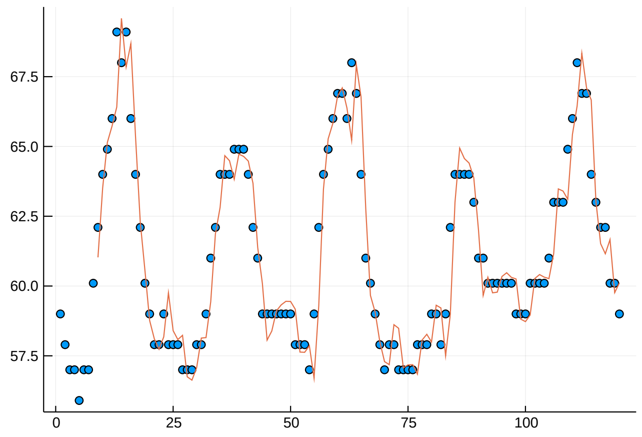


Figure 13.6 Hourly temperature at Los Angeles International Airport between 12:53AM on May 1, 2016, and 11:53PM on May 5, 2016, shown as circles. The solid line is the prediction of an auto-regressive model with eight coefficients.

```
julia> rms(ypred - y)
1.0129632612687514
julia> # Plot first five days
julia> using Plots
julia> Nplot = 24*5
julia> scatter(1:Nplot, t[1:Nplot], legend =false)
julia> plot!(M+1:Nplot, ypred[1:Nplot-M])
```

13.2 Validation

Polynomial approximation. We return to the polynomial fitting example of page 100. We continue with the data vectors `t` and `y` in the code on page 100 as the training set, and generate a test set of 100 randomly chosen points generated by the same method as used for the training set. We then fit polynomials of degree 0,. . . , 20 (*i.e.*, with $p = 1, \ldots, 21$ coefficients) and compute the RMS errors on the training set and the test set. This produces a figure similar to Figure 13.11 in VMLS, shown here as Figure 13.7.

```
julia> # Generate the test set.
julia> m = 100;
julia> t_test = -1 .+ 2*rand(m,1);
```

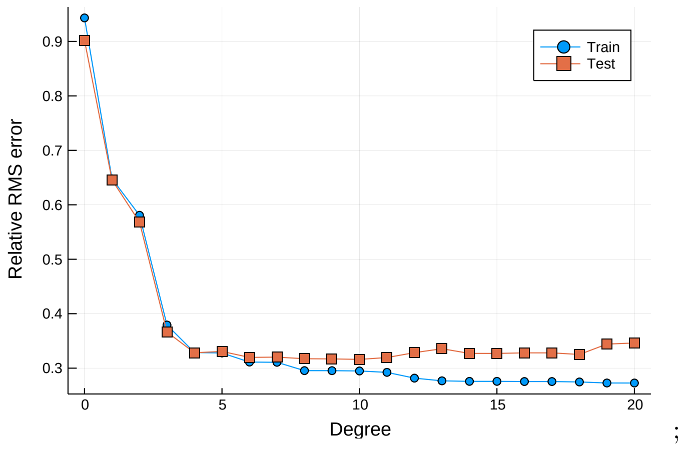


Figure 13.7 RMS error versus polynomial degree for the fitting example in figure 13.4.

```
julia> y_test = t_test.^3 - t_test + 0.4 ./ (1 .+ 25*t_test.^2)
           + 0.10*randn(m,1);
julia> error_train = zeros(21);
julia> error_test = zeros(21);
julia> for p = 1:21
           A = vandermonde(t,p)
           theta = A \ y
           error_train[p] = norm(A*theta - y) / norm(y)
           error_test[p] = norm( vandermonde(t_test, p) * theta
               - y_test) / norm(y_test);
       end
julia> using Plots
julia> plot(0:20, error_train, label = "Train", marker = :circle)
julia> plot!(0:20, error_test, label = "Test", marker = :square)
julia> plot!(xlabel="Degree", ylabel = "Relative RMS error")
```

House price regression model. On page 13.1 we used a data set of 774 house sales data to fit a simple regression model

$$\hat{y} = v + \beta_1 x_1 + \beta_2 x_2,$$

where $\hat{y}$ is the predicted sales price, x_1 is the area, and x_2 is the number of bedrooms. Here we apply cross-validation to assess the generalization ability of the simple model. We use five folds, four of size 155 (`Nfold` in the code below) and one of size 154. To choose the five folds, we create a random permutation of the indices

$1, \ldots, 774$. (We do this by calling the `randperm` function in the `Random` package.) We choose the data points indexed by the first 155 elements in the permuted list as fold 1, the next 155 as fold 2, et cetera. The output of the following code outputs is similar to Table 13.1 in VMLS (with different numbers because of the random choice of folds).

```
julia> D = house_sales_data();
julia> price = D["price"];  area = D["area"];  beds = D["beds"];
julia> N = length(price);
julia> X = [ ones(N) area beds ];
julia> nfold = div(N,5);  # size of first four folds
julia> import Random
julia> I = Random.randperm(N); # random permutation of numbers 1...N
julia> coeff = zeros(5,3);  errors = zeros(5,2);
julia> for k = 1:5
           if k == 1
               Itrain = I[nfold+1:end];
               Itest = I[1:nfold];
           elseif k == 5
               Itrain = I[1:4*nfold];
               Itest = I[4*nfold+1:end];
           else
               Itrain = I[ [1:(k-1)*nfold ; k*nfold+1 : N]]
               Itest = I[ [(k-1)*nfold+1 ; k*nfold ]];
           end;
           Ntrain = length(Itrain)
           Ntest = length(Itest)
           theta = X[Itrain,:] \ price[Itrain];
           coeff[k,:] = theta;
           rms_train = rms(X[Itrain,:] * theta - price[Itrain])
           rms_test = rms(X[Itest,:] * theta - price[Itest])
       end;
julia> coeff  # 3 coefficients for the five folds
5×3 Array{Float64,2}:
 56.4566  150.822  -20.1493
 59.4955  148.725  -20.0837
 53.7978  154.69   -21.577
 56.1279  145.629  -18.2875
 46.2149  144.207  -14.4336
julia> [rms_train rms_test]  # RMS errors for five folds
5×2 Array{Float64,2}:
 76.0359  69.9312
 75.4704  74.0014
```

```
76.194   94.3307
73.3722  27.6299
72.9218  82.1444
```

Validating time series predictions. In the next example, we return to the AR model of hourly temperatures at LAX. We divide the time series in a training set of 24 days and a test set of 7 days. We fit the AR model to the training set and calculate the RMS prediction errors on the training and test sets. Figure 13.8 shows the model predictions and the the data for the first five days of the test set.

```
julia> t = temperature_data();
julia> N = length(t);
julia> # use first 24 days as training set
julia> Ntrain = 24 * 24;  t_train = t[1:Ntrain];
julia> # use the rest as test set
julia> Ntest = N-Ntrain;  t_test = t[Ntrain+1:end];
julia> # Least squares fit of AR predictor with memory 8
julia> m = Ntrain - M;
julia> y = t_train[M+1:M+m];
julia> A = hcat( [t_train[i:i+m-1] for i=M:-1:1]...);
julia> coeff = A \ y;
julia> rms_train = rms(A*coeff-y)
1.0253577259862334
julia> ytest = t_test[M+1:end];
julia> mtest = length(ytest);
julia> ypred = hcat( [t_test[i:i+mtest-1] for i=M:-1:1]...) * coeff;
julia> rms_test = rms(ypred - ytest)
0.9755113632200967
julia> using Plots
julia> Nplot = 24*5
julia> scatter(1:Nplot, t_test[1:Nplot], legend=false)
julia> plot!(M+1:Nplot, ypred[1:Nplot-M])
```

13.3 Feature engineering

Next we compute the more complicated house price regression model of §13.3.5 of VMLS. The data are imported via the function `house_sales_data`, which returns a dictionary containing the following five vectors of length 774:

- `price`: price in thousand dollars,

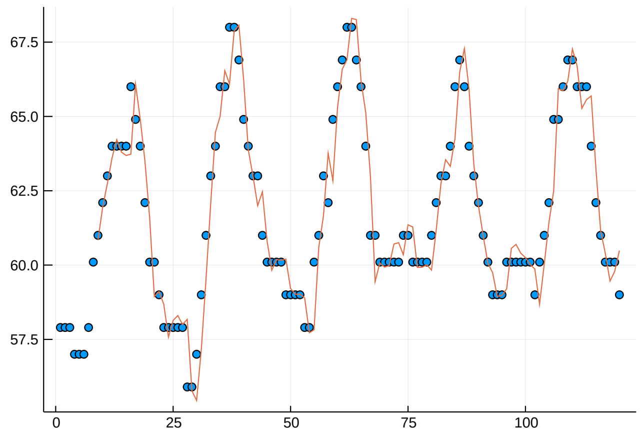


Figure 13.8 Hourly temperature at Los Angeles International Airport between 12:53AM on May 25, 2016, and 11:53PM on May 29, 2016, shown as circles. The solid line is the prediction of an auto-regressive model with eight coefficients, developed using training data from May 1 to May 24.

- `area`: house area in 1000 square feet,
- `beds`: number of bedrooms,
- `condo`: 1 if a condo, 0 otherwise,
- `location`: a number from 1–4, for the four sets of ZIP codes in Table 13.4 of VMLS.

The code computes the model and makes a scatter plot of actual and predicted prices (Figure 13.9). Note that the last three columns of the matrix `X` contain Boolean variables (`true` or `false`). We rely on the fact that Julia treats this as integers 1 and 0.

```
julia> D = house_sales_data();
julia> price = D["price"];
julia> area = D["area"];
julia> beds = D["beds"];
julia> condo = D["condo"];
julia> location = D["location"];
julia> N = length(price);
julia> X = hcat(ones(N), area, max.(area.-1.5, 0), beds, condo,
        location .== 2, location .== 3,  location .== 4 );
julia> theta = X \ price
8-element Array{Float64,1}:
  115.617
```

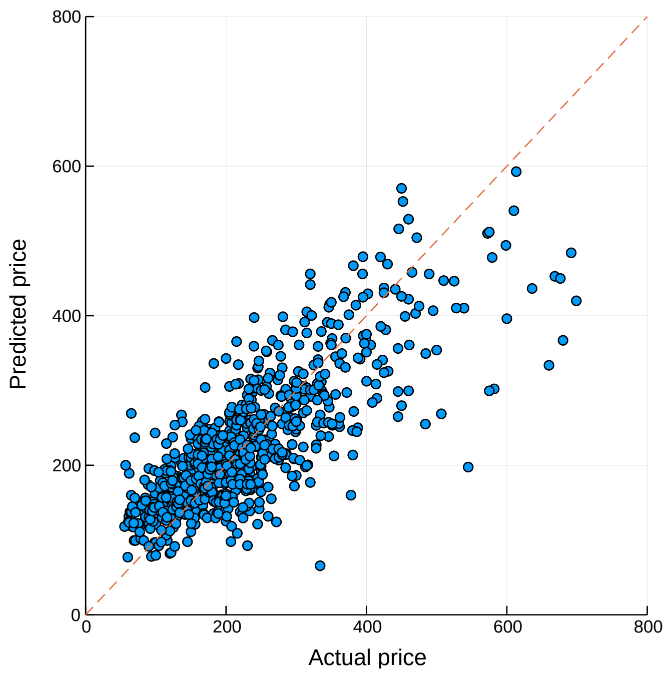


Figure 13.9 Scatter plot of actual and predicted prices for a model with eight parameters.

```
  175.413
  -42.7478
  -17.8784
  -19.0447
 -100.911
 -108.791
  -24.7652
julia> rms(X*theta - price)  # RMS prediction error
68.34428699036884
julia> using Plots
julia> scatter(price, X*theta, lims = (0,800));
julia> plot!([0, 800], [0, 800], linestyle = :dash, legend = false);
julia> plot!(xlims = (0,800), ylims = (0,800), size = (500,500));
julia> plot!(xlabel = "Actual price", ylabel = "Predicted price");
```

We finish by a cross-validation of this method. We follow the same approach as for the simple regression model on page 106, using five randomly chosen folds. The code shows the eight coefficients, and the RMS training and test errors for each fold.

```
julia> nfold = div(N,5);
julia> import Random;
julia> I = Random.randperm(N);
julia> models = zeros(8,5);  # store 8 coefficients for the 5 models
julia> errors = zeros(2,5);  # prediction errors
julia> for k = 1:5
           if k == 1
               Itrain = I[nfold+1:end];
               Itest = I[1:nfold];
           elseif k == 5
               Itrain = I[1:4*nfold];
               Itest = I[4*nfold+1:end];
           else
               Itrain = I[ [1:(k-1)*nfold ; k*nfold+1 : N]]
               Itest = I[ [(k-1)*nfold+1 ; k*nfold ]];
           end;
           Ntrain = length(Itrain)
           Ntest = length(Itest)
           theta = X[Itrain,:] \ price[Itrain];
           errors[1,k] = rms(X[Itrain,:] * theta - price[Itrain]);
           errors[2,k] = rms(X[Itest,:] * theta - price[Itest]);
           models[:,k] = theta;
    end;
julia> # display the eigth coefficients for each of the 5 folds
julia> models
8×5 Array{Float64,2}:
 121.294    110.602    142.51     103.589    101.677
 170.343    167.999    165.442    187.179    184.096
 -32.3645   -39.6754   -35.3901   -51.6821   -52.7473
 -21.0324   -16.5711   -15.0365   -17.972    -17.9788
 -17.6111   -27.6004   -19.4401   -13.7243   -16.2317
 -92.5554   -87.638   -124.744   -103.573    -97.4796
 -98.3588   -97.6417  -131.886   -111.991   -105.518
  -7.77581  -13.3224   -58.3521   -33.7546   -13.7175
julia> # display training errors (1st row) and test errors (2nd row)
julia> errors
2×5 Array{Float64,2}:
 66.1225  69.3897   66.0522  70.2625  69.16
 77.193   42.2776  115.636   73.5962  65.4291
```

Chapter 14

Least squares classification

14.1 Classification

Boolean values. Julia has the Boolean values `true` and `false`. These are automatically converted to the numbers 1 and 0 when they combined in numerical expressions. In VMLS we use the encoding (for classifiers) where True corresponds to $+1$ and False corresponds to -1. We can get our encoding from a Julia Boolean value `b` using `2*b-1`, or via the ternary conditional operation `b ? 1 : -1`.

```
julia> tf2pm1(b) = 2*b-1
julia> b = true
true
julia> tf2pm1(b)
1
julia> b = false
false
julia> tf2pm1(b)
-1
julia> b = [ true, false, true ]
3-element Array{Bool,1}:
  true
 false
  true
julia> tf2pm1.(b)
3-element Array{Int64,1}:
  1
 -1
  1
```

Confusion matrix. Let's see how we would evaluate the prediction errors and confusion matrix, given a set of data `y` and predictions `yhat`, both stored as arrays (vectors) of Boolean values, of length `N`.

```
julia> # Count errors and correct predictions
julia> Ntp(y,yhat) = sum( (y .== true) .& (yhat .== true) );
julia> Nfn(y,yhat) = sum( (y .== true) .& (yhat .== false) );
julia> Nfp(y,yhat) = sum( (y .== false) .& (yhat .== true) );
julia> Ntn(y,yhat) = sum( (y .== false) .& (yhat .== false) );
julia> error_rate(y,yhat) = (Nfn(y,yhat) + Nfp(y,yhat)) / length(y);
julia> confusion_matrix(y,yhat) = [ Ntp(y,yhat) Nfn(y,yhat);
           Nfp(y,yhat) Ntn(y,yhat) ];
julia> y = rand(Bool,100);  yhat = rand(Bool,100);
julia> confusion_matrix(y,yhat)
2×2 Array{Int64,2}:
 25  23
 29  23
julia> error_rate(y,yhat)
0.52
```

The dots that precede `==` and `&` cause them to be evaluated elementwise. When we sum the Boolean vectors, they are converted to integers. In the last section of the code we generate two random Boolean vectors, so we expect the error rate to be around 50%. In the code above, we compute the error rate from the numbers of false negatives and false positives. A more compact expression for the error rate is `avg(y .!= yhat)`. The `VMLS` package contains the function `confusion_matrix(y, yhat)`.

14.2 Least squares classifier

We can evaluate $\hat{f}(x) = \mathbf{sign}(\tilde{f}(x))$ using `ftilde(x)>0`, which returns a Boolean value.

```
julia> ftilde(x) = x'*beta .+ v  # Regression model
julia> fhat(x) = ftilde(x) > 0   # Regression classifier
```

Iris flower classification. The Iris data set contains of 150 examples of three types of iris flowers. There are 50 examples of each class. For each example, four features are provided. The following code reads in a dictionary containing three 50×4 matrices `setosa`, `versicolor`, `virginica` with the examples for each class, and then computes a Boolean classifier that distinguishes *Iris Virginica* from the the other two classes.

```
julia> D = iris_data();
julia> # Create 150x4 data matrix
julia> iris = vcat(D["setosa"], D["versicolor"], D["virginica"])
julia> # y[k] is true (1) if virginica, false (0) otherwise
julia> y = [ zeros(Bool, 50); zeros(Bool, 50); ones(Bool, 50) ];
julia> A = [ ones(150) iris ]
julia> theta = A \ (2*y .- 1)
5×1 Array{Float64,2}:
 -2.39056
 -0.0917522
  0.405537
  0.00797582
  1.10356
julia> yhat = A*theta .> 0;
julia> C = confusion_matrix(y,yhat)
2×2 Array{Int64,2}:
 46   4
  7  93
julia> err_rate = (C[1,2] + C[2,1]) / length(y)
0.07333333333333333
julia> avg(y .!= yhat)
0.07333333333333333
```

14.3 Multi-class classifiers

Multi-class error rate and confusion matrix. The overall error rate is easily evaluated as `avg(y .!= yhat)`. We can form the $K \times K$ confusion matrix from a set of N true outcomes `y` and N predictions `yhat` (each with entries among $\{1, \ldots, K\}$) by counting the number of times each pair of values occurs.

```
julia> error_rate(y, yhat) = avg(y .!= yhat);
julia> function confusion_matrix(y, yhat, K)
       C = zeros(K,K)
       for i in 1:K for j in 1:K
           C[i,j] = sum((y .== i) .& (yhat .== j))
       end end
       return C
       end;
julia> # test for K=4 on random vectors of length 100
julia> K = 4;
julia> y = rand(1:K, 100);  yhat = rand(1:K, 100);
julia> C = confusion_matrix(y, yhat, K)
4×4 Array{Float64,2}:
 4.0  9.0  8.0   5.0
 5.0  4.0  4.0  13.0
 3.0  8.0  7.0  11.0
 7.0  2.0  7.0   3.0
julia> error_rate(y, yhat),  1-sum(diag(C))/sum(C)
(0.82, 0.8200000000000001)
```

The function `confusion_matrix` is included in the `VMLS` package.

Least squares multi-class classifier. A K-class classifier (with regression model) can be expressed as

$$\hat{f}(x) = \underset{k=1,\ldots,K}{\operatorname{argmax}} \tilde{f}_k(x),$$

where $\tilde{f}_k(x) = x^T\theta_k$. The n-vectors $\theta_1, \ldots, \theta_K$ are the coefficients or parameters in the model. We can express this in matrix-vector notation as

$$\hat{f}(x) = \operatorname{argmax}(x^T\Theta),$$

where $\Theta = \begin{bmatrix} \theta_1 & \cdots & \theta_K \end{bmatrix}$ is the $n \times K$ matrix of model coefficients, and the argmax of a row vector has the obvious meaning.

Let's see how to express this in Julia. In Julia the function `argmax(u)` finds the index of the largest entry in the row or column vector `u`, *i.e.*, $\operatorname{argmax}_k u_k$. To extend this to matrices, we define a function `row_argmax` that returns a vector with, for each row, the index of the largest entry in that row.

```
julia> row_argmax(u) = [ argmax(u[i,:]) for i = 1:size(u,1) ]
julia> A = randn(4,5)
4×5 Array{Float64,2}:
  1.42552    0.766725   1.7106    -1.08668   -0.492051
 -0.507653  -0.158288  -1.37703   -0.388304   0.00290895
 -1.43499   -1.18238   -0.182795  -0.428589  -0.87592
 -2.18407    1.28363    0.749702  -0.304138   0.0165654
julia> row_argmax(A)
4-element Array{Int64,1}:
 3
 5
 3
 2
```

If a data set with N examples is stored as an $n \times N$ data matrix `X`, and `Theta` is an $n \times K$ matrix with the coefficient vectors θ_k as its columns, then we can now define a function

```
fhat(X,Theta) = row_argmax(X'*Theta)
```

to find the N-vector of predictions.

Matrix least squares. Let's use least squares to find the coefficient matrix Θ for a multi-class classifier with n features and K classes, from a data set of N examples. We will assume the data is given as an $n \times N$ matrix X and an N-vector $y^{\rm cl}$ with entries in $\{1, \ldots, K\}$ that give the classes of the examples. The least squares objective can be expressed as a matrix norm squared,

$$\|X^T\Theta - Y\|^2,$$

where Y is the $N \times K$ vector with

$$Y_{ij} = \left\{ \begin{array}{rl} 1 & y_i^{\rm cl} = j \\ -1 & y_i^{\rm cl} \neq j. \end{array} \right.$$

In other words, the rows of Y describe the classes using one-hot encoding, converted from $0/1$ to $-1/+1$ values. The least squares solution is given by $\hat{\Theta} = (X^T)^\dagger Y$.

Let's see how to express this in Julia.

```
julia> function one_hot(ycl,K)
       N = length(ycl)
       Y = zeros(N,K)
       for j in 1:K
           Y[findall(ycl .== j), j] .= 1
       end
       return Y
       end;
julia> K = 4;
julia> ycl = rand(1:K,6)
6-element Array{Int64,1}:
 4
 2
 3
 1
 2
 1
julia> Y = one_hot(ycl, K)
64 ArrayFloat64,2:
 0.0  0.0  0.0  1.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 1.0  0.0  0.0  0.0
julia> 2*Y .- 1
6×4 Array{Float64,2}:
 -1.0  -1.0  -1.0   1.0
 -1.0   1.0  -1.0  -1.0
 -1.0  -1.0   1.0  -1.0
  1.0  -1.0  -1.0  -1.0
 -1.0   1.0  -1.0  -1.0
  1.0  -1.0  -1.0  -1.0
```

Using the functions we have defined, the matrix least squares multi-class classifier can be computed in a few lines.

```
julia> function ls_multiclass(X,ycl,K)
       n, N = size(X)
       Theta = X' \ (2*one_hot(ycl,K) .- 1)
       yhat = row_argmax(X'*theta)
       return Theta, yhat
       end
```

Iris flower classification. We compute a 3-class classifier for the iris flower data set. We split the data set of 150 examples in a training set of 120 (40 per class) and a test set of 30 (10 per class). The code calls the functions we defined above.

```
julia> D = iris_data();
julia> setosa = D["setosa"];
julia> versicolor = D["versicolor"];
julia> virginica = D["virginica"];
julia> # pick three random permutations of 1,..., 50
julia> import Random
julia> I1 = Random.randperm(50);
julia> I2 = Random.randperm(50);
julia> I3 = Random.randperm(50);
julia> # training set is 40 randomly picked examples per class
julia> Xtrain = [ setosa[I1[1:40],:];
                  versicolor[I2[1:40],:];
                  virginica[I3[1:40],:] ]';  # 4x120 data matrix
julia> # add constant feature one
julia> Xtrain = [ ones(1,120); Xtrain ];  # 5x120 data matrix
julia> ytrain = [ ones(40); 2*ones(40); 3*ones(40) ];
julia> # test set is remaining 10 examples for each class
julia> Xtest = [ setosa[I1[41:end],:];
                 versicolor[I2[41:end],:]
                 virginica[I3[41:end],:] ]';  # 4x30 data matrix
julia> Xtest = [ ones(1,30); Xtest ];  # 5x30 data matrix
julia> ytest = [ones(10); 2*ones(10); 3*ones(10)];
julia> Theta, yhat = ls_multiclass(Xtrain, ytrain, 3);
julia> Ctrain = confusion_matrix(ytrain, yhat, 3)
3×3 Array{Float64,2}:
 40.0   0.0   0.0
  0.0  28.0  12.0
  0.0   6.0  34.0
julia> error_train =  error_rate(ytrain, yhat)
0.15
julia> yhat = row_argmax(Xtest'*theta)
julia> Ctest = confusion_matrix(ytest, yhat, 3)
3×3 Array{Float64,2}:
 10.0  0.0  0.0
  0.0  7.0  3.0
  0.0  2.0  8.0
julia> error_test =  error_rate(ytest, yhat)
0.16666666666666666
```

Chapter 15

Multi-objective least squares

15.1 Multi-objective least squares

Let's write a functiont that solves the multi-objective least squares problem, with given positive weights. The data are a list (or array) of coefficient matrices (of possibly different heights) `As`, a matching list of (right-hand side) vectors `bs`, and the weights, given as an array or list, `lambdas`.

```
julia> function mols_solve(As,bs,lambdas)
       k = length(lambdas);
       Atil = vcat([sqrt(lambdas[i])*As[i] for i=1:k]...)
       btil = vcat([sqrt(lambdas[i])*bs[i] for i=1:k]...)
       return Atil \ btil
       end
```

Simple example. We use the function `mols_solve` to work out a bi-criterion example similar to Figures 15.1, 15.2, and 15.3 in VMLS. We minimize the weighted sum objective

$$J_1 + \lambda J_2 = \|A_1x - b_1\|^2 + \lambda\|A_2x - b_2\|^2$$

for randomly chosen 10×5 matrices A_1, A_2 and 10-vectors b_1, b_2. The expression `lambdas = 10 .^ linspace(-4,4,200)` generates 200 values of $\lambda \in [10^{-4}, 10^4]$, equally spaced on a logarithmic scale. The code creates the three plots in Figures 15.1, 15.2, and 15.3.

```
julia> As = [randn(10,5), randn(10,5)];
julia> bs = [randn(10), randn(10)];
julia> N = 200;
julia> lambdas = 10 .^ linspace(-4,4,200);
julia> x = zeros(5,N); J1 = zeros(N); J2 = zeros(N);
julia> for k = 1:N
```

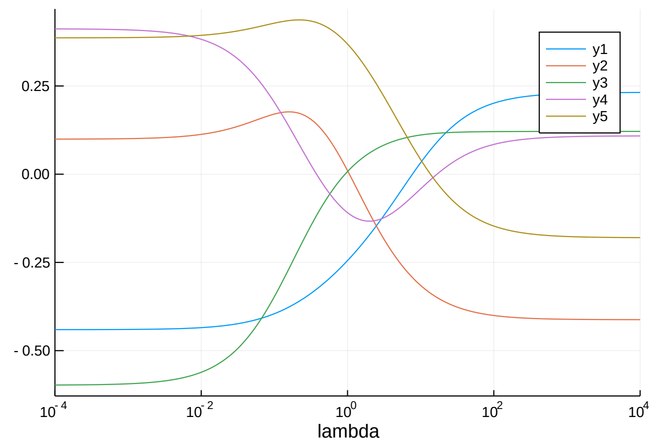


Figure 15.1 Weighted-sum least squares solution $\hat{x}(\lambda)$ as a function of λ for a bi-criterion least squares problem with five variables.

```
           x[:,k] = mols_solve(As, bs, [1, lambdas[k]])
           J1[k] = norm(As[1]*x[:,k] - bs[1])^2
           J2[k] = norm(As[2]*x[:,k] - bs[2])^2
    end;
julia> using Plots
julia> # plot solution versus lambda
julia> plot(lambdas, x', xscale = :log10, xlabel = "lambda");
julia> plot!(xlims = (1e-4,1e4));
julia> # plot two objectives versus lambda
julia> plot(lambdas, J1, xscale = :log10, label = "J_1");
julia> plot!(lambdas, J2, label = "J_2", xlabel = "lambda",
           xlims = (1e-4,1e4));
julia> # plot trade-off curve
julia> plot(J1, J2, xlabel="J1", ylabel = "J2", legend=false);
julia> # add (single-objective) end points to trade-off curve
julia> x1 = As[1] \ bs[1];
julia> x2 = As[2] \ bs[2];
julia> J1 = [norm(As[1]*x1-bs[1])^2, norm(As[1]*x2-bs[1])^2];
julia> J2 = [norm(As[2]*x1-bs[2])^2, norm(As[2]*x2-bs[2])^2];
julia> scatter!(J1,J2);
```

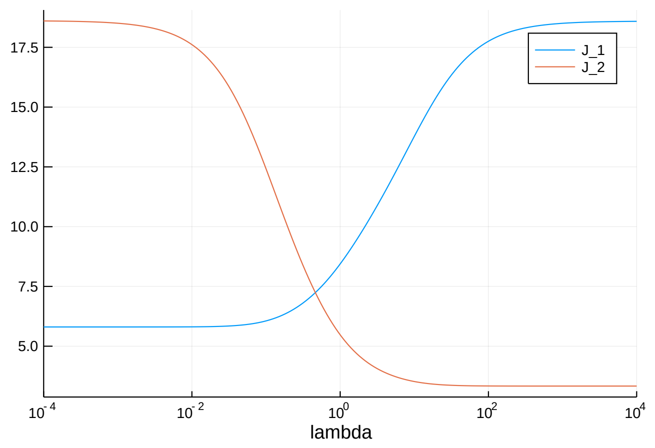


Figure 15.2 Objective functions $J_1 = \|A_1\hat{x}(\lambda)b_1\|^2$ (blue line) and $J_2 = \|A_2\hat{x}(\lambda)b_2\|^2$ (red line) as functions of λ for the bi-criterion problem in figure 15.1.

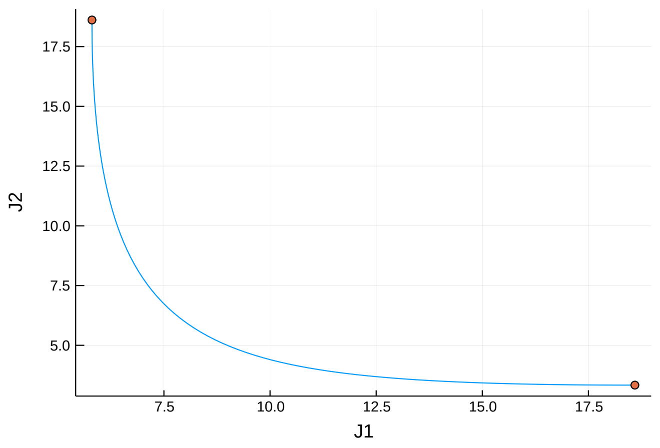


Figure 15.3 Optimal trade-off curve for the bi-criterion least squares problem of figures 15.1 and 15.2.

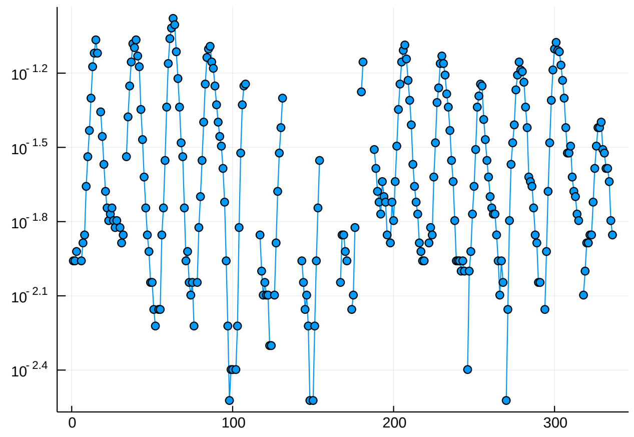


Figure 15.4 Hourly ozone level at Azusa, California, during the first 14 days of July 2014 (California Environmental Protection Agency, Air Re- sources Board, www.arb.ca.gov). Measurements start at 12AM on July 1st, and end at 11PM on July 14. Note the large number of missing measure- ments. In particular, all 4AM measurements are missing.

15.2 Control

15.3 Estimation and inversion

Estimating a periodic time series. We consider the example of Figure 15.4 in VMLS. We start by loading the data, as a vector with hourly ozone levels, for a period of 14 days. Missing measurements have a value `NaN` (for Not a Number). The `plot` command skips those values (Figure 15.4).

```
julia> ozone = ozone_data();  # a vector of length 14*24 = 336
julia> k = 14;  N = k*24;
julia> plot(1:N, ozone, yscale = :log10, marker = :circle,
          legend=false)
```

Next we use the `mols_solve` function to make a periodic fit, for the values $\lambda = 1$ and $\lambda = 10$. The Julia code `isnan` is used to find and discard the missing measurements. The results are shown in Figures 15.5 and 15.6.

```
julia> A = vcat( [eye(24) for i = 1:k]...)
julia> # periodic difference matrix
julia> D = -eye(24) + [zeros(23,1) eye(23); 1  zeros(1,23)];
julia> ind = [k for k in 1:length(ozone) if !isnan(ozone[k])];
julia> As = [A[ind,:], D]
```

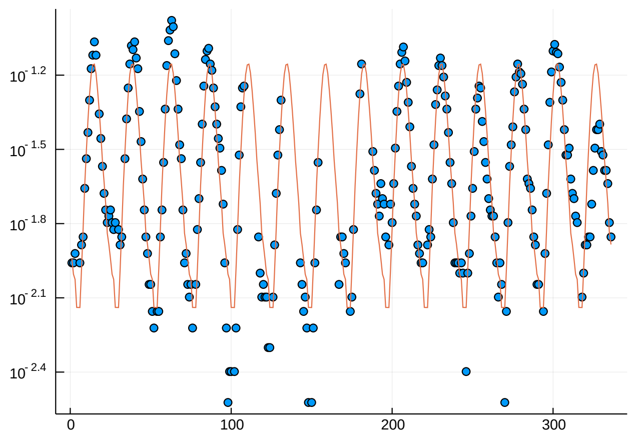


Figure 15.5 Smooth periodic least squares fit to logarithmically transformed measurements, using $\lambda = 1$.

```
julia> bs = [log.(ozone[ind]), zeros(24)]
julia> # solution for lambda = 1
julia> x = mols_solve( As, bs, [1, 1])
julia> scatter(1:N, ozone, yscale = :log10, legend=false)
julia> plot!(1:N, vcat([exp.(x) for i = 1:k]...))
julia> # solution for lambda = 100
julia> x = mols_solve( As, bs, [1, 100])
julia> scatter(1:N, ozone, yscale = :log10, legend=false)
julia> plot!(1:N, vcat([exp.(x) for i = 1:k]...))
```

15.4 Regularized data fitting

Example. Next we consider the small regularized data fitting example of page 329 of VMLS. We fit a model

$$\hat{f}(x) = \sum_{k=1}^{5} \theta_k f_k(x)$$

with basis functions $f_1(x) = 1$ and $f_{k+1}(x) = \sin(\omega_k x + \phi_k)$ for $k = 1, \ldots, 4$ to $N = 20$ data points. We use the values of ω_k, ϕ_k given in the text. We fit the model by solving a sequence of regularized least squares problems with objective

$$\sum_{i=1}^{N} (y^{(i)} - \sum_{k=1}^{5} \theta_k f_k(x^{(i)}))^2 + \lambda \sum_{k=2}^{5} \theta_k^2.$$

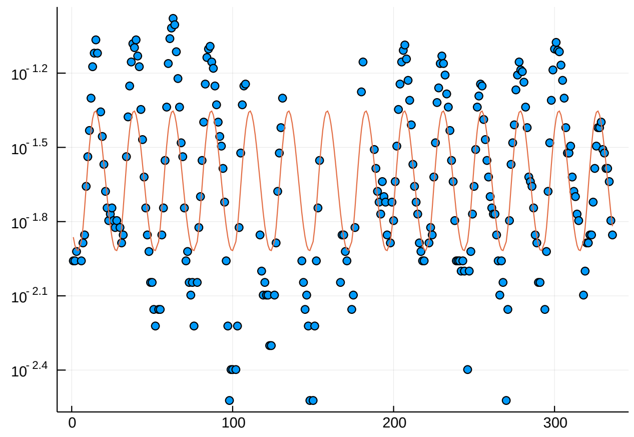


Figure 15.6 Smooth periodic least squares fit to logarithmically transformed measurements, using $\lambda = 100$.

The two plots are shown in Figures 15.7 and 15.8.

```
julia> # Import data as vectors xtrain, ytrain, xtest, ytest
julia> D = regularized_fit_data();
julia> xtrain = D["xtrain"];  ytrain = D["ytrain"];
julia> xtest = D["xtest"];  ytest = D["ytest"];
julia> N = length(ytrain);
julia> Ntest = length(ytest);
julia> p = 5;
julia> omega = [ 13.69; 3.55; 23.25; 6.03 ];
julia> phi = [ 0.21; 0.02; -1.87; 1.72 ];
julia> A = hcat(ones(N), sin.(xtrain*omega' + ones(N)*phi'));
julia> Atest = hcat(ones(Ntest),
           sin.(xtest*omega' + ones(Ntest)*phi'));
julia> npts = 100;
julia> lambdas = 10 .^ linspace(-6,6,npts);
julia> err_train = zeros(npts);
julia> err_test = zeros(npts);
julia> thetas = zeros(p,npts);
julia> for k = 1:npts
       theta = mols_solve([ A, [zeros(p-1) eye(p-1)]],
           [ ytrain, zeros(p-1) ], [1, lambdas[k]])
       err_train[k] = rms(ytrain - A*theta);
       err_test[k] = rms(ytest - Atest*theta);
       thetas[:,k] = theta;
```

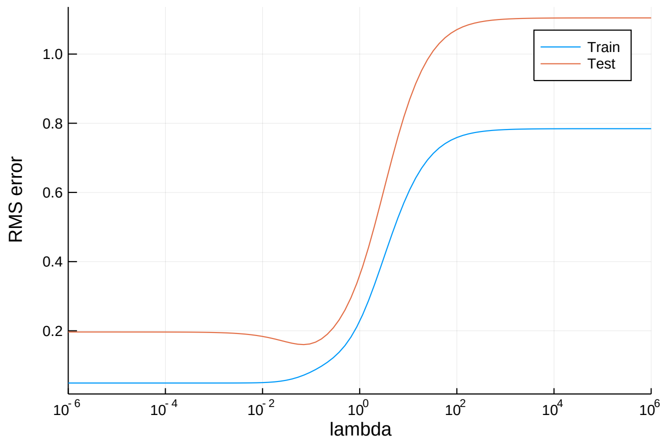


Figure 15.7 RMS training and test errors as a function of the regularization parameter λ.

```
       end;
julia> using Plots
julia> # Plot RMS errors
julia> plot(lambdas, err_train, xscale = :log10, label = "Train")
julia> plot!(lambdas, err_test, xscale = :log10, label = "Test")
julia> plot!(xlabel = "lambda", ylabel = "RMS error",
           xlim = (1e-6, 1e6));
julia> # Plot coefficients
julia> plot(lambdas, thetas', xscale = :log10)
julia> plot!(xlabel = "lambda", xlim = (1e-6, 1e6));
```

15.5 Complexity

The kernel trick. Let's check the kernel trick, described in §15.5.2, to find $\hat{x}$, the minimizer of

$$\|Ax - b\|^2 + \lambda\|x - x^{\mathrm{des}}\|^2,$$

where A is an $m \times n$ matrix and $\lambda > 0$. We'll compute $\hat{x}$ two ways. First, the naïve way, and then, using the kernel trick. We use the fact that if

$$\begin{bmatrix} A^T \\ \sqrt{\lambda} I \end{bmatrix} = QR,$$

then

$$(AA^T + \lambda I)^{-1} = (R^T Q^T Q R)^{-1} = R^{-1} R^{-T}.$$

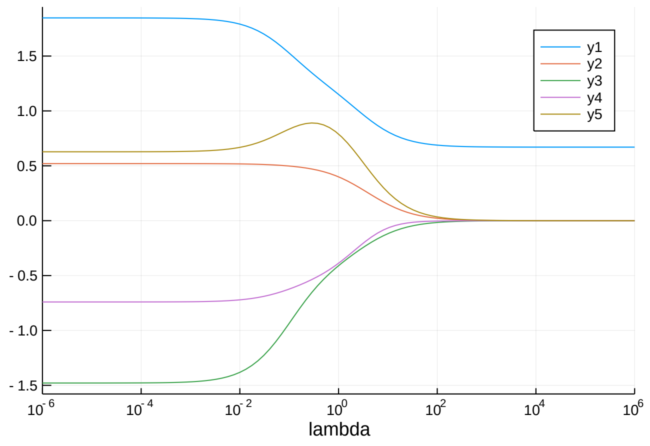


Figure 15.8 The regularization path.

```
julia> m = 100; n = 5000;
julia> A = randn(m,n); b = randn(m); xdes = randn(n);
julia> lam = 2.0;
julia> # Find x that minimizes ||Ax-b||^2 + lambda ||x||^2
julia> @time xhat1 = [A; sqrt(lam)*eye(n)] \ [b; sqrt(lam)*xdes];
 23.447045 seconds (40.08 k allocations: 1.130 GiB, 0.83% gc time)
julia> # Now use kernel trick
julia> @time begin
       Q, R = qr([A' ; sqrt(lam)*eye(m)]);
       Q = Matrix(Q);
       xhat2 = A' * (R \ (R' \ (b-A*xdes))) + xdes;
       end;
  0.025105 seconds (42 allocations: 12.114 MiB)
julia> norm(xhat1-xhat2)
1.2742623007481903e-13
```

The naïve method requires the factorization of a 5100×5100 matrix. In the second method we factor a matrix of size 5100×100.

Chapter 16

Constrained least squares

16.1 Constrained least squares problem

In the examples in this section, we use the `cls_solve` function, given later, to find the constrained least squares solution.

Piecewise polynomial. We fit a function $\hat{f} : \mathbf{R} \to \mathbf{R}$ to some given data, where $\hat{f}(x) = p(x)$ for $x \leq a$ and $\hat{f}(x) = q(x)$ for $x > a$, subject to $p(a) = q(a)$ and $p'(a) = q'(a)$, *i.e.*, the two polynomials have matching value and slope at the knot point a. We have data points $x_1, \ldots, x_M \leq a$ and $x_{M+1}, \ldots, x_N > a$ and corresponding values $y_1, \ldots, y_N$. In the example we take $a = 0$, polynomials p and q of degree 3, and $N = 2M = 140$. The code creates a figure similar to Figure 16.1 of VMLS (Figure 16.1). We use the `vandermonde` function from page 58.

```
julia> M = 70;  N = 2*M;
julia> xleft = rand(M) .- 1;  xright = rand(M);
julia> x = [xleft; xright]
julia> y = x.^3 - x + 0.4 ./ (1 .+ 25*x.^2) + 0.05*randn(N);
julia> n = 4;
julia> A = [ vandermonde(xleft,n)  zeros(M,n);
             zeros(M,n) vandermonde(xright,n)]
julia> b = y;
julia> C = [1  zeros(1,n-1) -1  zeros(1,n-1);
            0  1  zeros(1,n-2)  0 -1  zeros(1,n-2)];
julia> d = zeros(2);
julia> theta = cls_solve(A, b, C, d);
julia> using Plots
julia> # Evaluate and plot for 200 equidistant points on each side.
julia> Npl = 200;
julia> xpl_left = linspace(-1, 0, Npl);
```

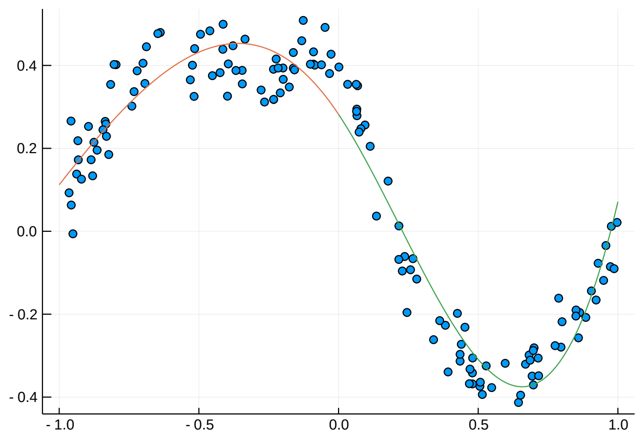


Figure 16.1 Least squares fit of two cubic polynomials to 140 points, with continuity constraints $p(0) = q(0)$ and $p'(0) = q'(0)$.

```
julia> ypl_left = vandermonde(xpl_left, 4)*theta[1:n];
julia> xpl_right = linspace(0, 1, Npl);
julia> ypl_right = vandermonde(xpl_right, 4)*theta[n+1:end];
julia> scatter(x,y, legend=false)
julia> plot!(xpl_left, ypl_left)
julia> plot!(xpl_right, ypl_right)
```

Advertising budget. We continue the advertising example of page 96 and add a total budget constraint $\mathbf{1}^T s = 1284$.

```
julia> cls_solve(R, 1e3*ones(m), ones(1,n), 1284)
3×1 Array{Float64,2}:
 315.16818459234986
 109.86643348012254
 858.9653819275276
```

Minimum norm force sequence. We compute the smallest sequence of ten forces, each applied for one second to a unit frictionless mass originally at rest, that moves the mass position one with zero velocity (VMLS page 343).

```
julia> A = eye(10);  b = zeros(10);
julia> C = [ones(1,10);  (9.5:-1:0.5)']
```

```
2×10 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 9.5  8.5  7.5  6.5  5.5  4.5  3.5  2.5  1.5  0.5
julia> d = [0,1];
julia> fln = cls_solve(A,b,C,d)
10×1 Array{Float64,2}:
  0.0545455
  0.0424242
  0.030303
  0.0181818
  0.00606061
 -0.00606061
 -0.0181818
 -0.030303
 -0.0424242
 -0.0545455
```

16.2 Solution

Let's implement the function `cls_solve_kkt`, which finds the constrained least squares solution by forming the KKT system and solving it. We allow the `b` and `d` to be matrices, so one function call can solve mutiple problems with the same A and C.

```
julia> function cls_solve_kkt(A,b,C,d)
       m, n = size(A)
       p, n = size(C)
       G = A'*A  # Gram matrix
       KKT = [2*G C'; C zeros(p,p)]  # KKT matrix
       xzhat = KKT \ [2*A'*b; d]
       return xzhat[1:n,:]
       end;
julia> A = randn(10,5); b = randn(10);
julia> C = randn(2,5); d = randn(2);
julia> x = cls_solve_kkt(A,b,C,d);
julia> C*x - d  # Check residual small
2×1 Array{Float64,2}:
 -5.551115123125783e-17
 -1.6653345369377348e-16
```

16.3 Solving contrained least squares problems

Solving constrained least squares via QR. Let's implement VMLS algorithm 16.1 and then check it against our method above, which forms and solves the KKT system.

```
julia> function cls_solve(A,b,C,d)
       m, n = size(A)
       p, n = size(C)
       Q, R = qr([A; C])
       Q = Matrix(Q)
       Q1 = Q[1:m,:]
       Q2 = Q[m+1:m+p,:]
       Qtil, Rtil = qr(Q2')
       Qtil = Matrix(Qtil)
       w = Rtil \ (2*Qtil'*Q1'*b - 2*(Rtil'\d))
       return xhat = R \ (Q1'*b - Q2'*w/2)
       end
julia> # check with KKT method
julia> m = 10; n = 5; p = 2;
julia> A = randn(m,n); b = randn(m); C = randn(p,n); d = randn(p);
julia> xKKT = cls_solve_kkt(A,b,C,d);
julia> xQR = cls_solve(A,b,C,d);
julia> norm(xKKT-xQR)
1.4931525882746458e-15
```

The function `cls_solve` is included in the `VMLS` package.

Sparse constrained least squares. Let's form and solve the system of linear equations in VMSL (16.11), and compare it to our basic method for constrained least squares. This formulation will result in a sparse set of equations to solve if A and C are sparse. (The code below just checks that the two methods agree; it does not use sparsity. Unlike the earlier `cls_solve`, it assumes `b` and `d` are vectors.)

```
julia> function cls_solve_sparse(A,b,C,d)
       m, n = size(A)
       p, n = size(C)
       bigA = [ zeros(n,n) A' C';
                A -I/2 zeros(m,p) ;
                C zeros(p,m) zeros(p,p) ]
       xyzhat = bigA \ [zeros(n) ; b ; d]
       return xhat = xyzhat[1:n]
       end
julia> m = 100; n = 50; p = 10;
julia> A = randn(m,n); b = randn(m); C = randn(p,n); d = randn(p);
julia> x1 = cls_solve(A,b,C,d);
julia> x2 = cls_solve_sparse(A,b,C,d);
julia> norm(x1-x2)
1.3344943251376455e-14
```

Solving least norm problem. In Julia, the backslash operator is used to find the least norm solution $\hat{x}$ of an under-determined set of equations $Cx = d$. Thus the backslash operator is overloaded to solve linear equations with a square coefficient matrix, find a least squares approximate solution when the coefficient matrix is tall, and find the least norm solution when the coefficient matrix is wide.

Let's solve a least norm problem using several methods, to check that they agree.

```
julia> p = 50; n = 500;
julia> C = randn(p,n);  d = randn(p);
julia> x1 = C\d;  # Solve using backslash
julia> # Solve using cls_solve, which uses KKT system
julia> x2 = cls_solve(eye(n), zeros(n), C, d);
julia> x3 = pinv(C)*d;  # Using pseudo-inverse
julia> norm(x1-x2)
5.584943800596077e-15
julia> norm(x2-x3)
5.719694159427276e-15
```

Chapter 17

Constrained least squares applications

17.1 Portfolio optimization

Compounded portfolio value. The cumulative value of a portfolio from a return time series vector r, starting from the traditional value of \$10000, is given by the value times series vector v, where

$$v_t = 10000(1 + r_1) \cdots (1 + r_{t-1}), \quad t = 1, \ldots, T.$$

In other words, we form the *cumulative product* of the vector with entries $1 + r_t$. Julia has a built-in function that does this, `cumprod`.

```
julia> # Portfolio value with re-investment, return time series r
julia> cum_value(r) = 10000 * cumprod(1 .+ r)
julia> T = 250;  # One year's worth of trading days
julia> # Generate random returns sequence with
julia> # 10% annualized return, 5% annualized risk
julia> mu = 0.10/250; sigma = 0.05/sqrt(250);
julia> r = mu .+ sigma*randn(T);
julia> v = cum_value(r);
julia> # compare final value (compounded) and average return
julia> v[T] , v[1]*(1+sum(r))
(10313.854295827463, 10348.11585318395)
julia> # plot cumulative value over the year
julia> using Plots
julia> plot(1:T, v, legend=false)
julia> plot!( xlabel = "t", ylabel = "v_t")
```

The resulting figure for a particular choice of r is shown in Figure 17.1.

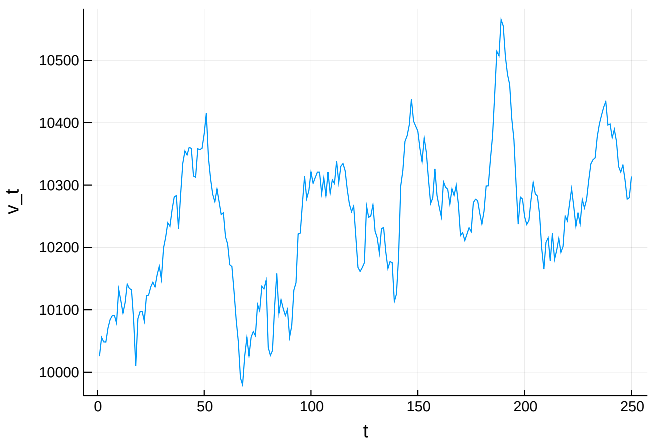


Figure 17.1 Total portfolio value over time.

Portfolio optimization. We define a function `port_opt` that evaluates the solution (17.3) of the constrained least squares problem (17.2) in VMLS, and apply to the return data in VMLS Section 17.1.3.

```
julia> function port_opt(R,rho)
       T, n = size(R)
       mu = sum(R, dims=1)'/T
       KKT = [ 2*R'*R ones(n) mu; ones(n)' 0 0; mu' 0 0]
       wz1z2 = KKT \ [2*rho*T*mu; 1; rho]
       w = wz1z2[1:n]
       return w
       end;
julia> R, Rtest = portfolio_data();
julia> T, n = size(R)
(2000, 20)
julia> rho = 0.10/250;  # Ask for 10% annual return
julia> w = port_opt(R,rho);
julia> r = R*w;  # Portfolio return time series
julia> pf_return = 250*avg(r)
0.10000000000000003
julia> pf_risk = sqrt(250)*stdev(r)
0.0865018308685463
julia> using Plots
julia> plot(1:T, cum_value(r), label= "10%")
```

This produces the curve labeled “10%” in Figure 17.2. We also included the plots for 20% and 40% annual return, and for the $1/n$ portfolio $w = (1/n)\mathbf{1}$.

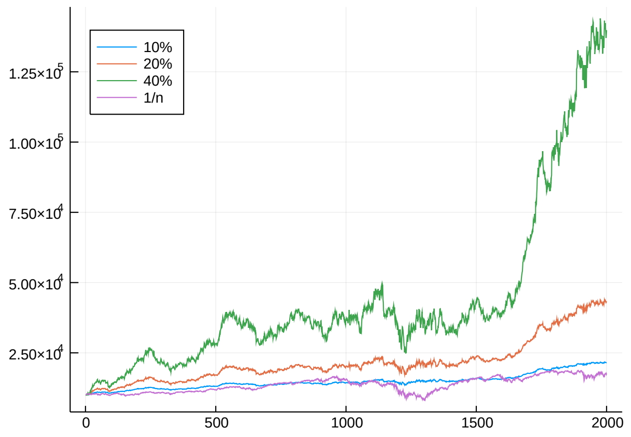


Figure 17.2 Total value over time for four portfolios: the Pareto optimal portfolios with 10%, 20%, and 40% return, and the uniform portfolio. The total value is computed using the 2000×20 daily return matrix R.

17.2 Linear quadratic control

We implement linear quadratic control, as described in VMLS §17.2, for a time-invariant system with matrices A, B, and C.

Kronecker product. To create the big matrices $\tilde{A}$ and $\tilde{C}$, we need to define block diagonal matrices with the same matrix repeated a number of times along the diagonal. There are many ways to do this in Julia. One of the simplest ways uses the `kron` function, for the Kronecker product of two matrices. The Kronecker product of an $m \times n$ matrix G and a $p \times q$ matrix H is defined as the $mp \times nq$ block matrix

$$\begin{bmatrix} G_{11}H & G_{12}H & \cdots & G_{1n}H \\ G_{21}H & G_{22}H & \cdots & G_{2n}H \\ \vdots & \vdots & & \vdots \\ G_{m1}H & G_{m2}H & \cdots & G_{mn}H \end{bmatrix}.$$

It is computed in Julia as `kron(G,H)`. If G is an $n \times n$ identity matrix, we obtain the block diagonal matrix with H repeated n times on the diagonal.

```
julia> H = randn(2,2)
2x2 Array{Float64,2}:
  1.73065  -1.33313
 -1.52245   0.0200201
julia> kron(eye(3),H)
6×6 Array{Float64,2}:
  1.73065  -1.33313     0.0      -0.0          0.0      -0.0
 -1.52245   0.0200201  -0.0       0.0        -0.0       0.0
  0.0      -0.0         1.73065  -1.33313     0.0      -0.0
 -0.0       0.0        -1.52245   0.0200201  -0.0       0.0
  0.0      -0.0         0.0      -0.0         1.73065  -1.33313
 -0.0       0.0        -0.0       0.0        -1.52245   0.0200201
```

An alternative method uses the Julia `cat` function for constructing block matrices:

```
julia> cat([H for k=1:3]..., dims=(1,2))
6×6 Array{Float64,2}:
  1.73065  -1.33313     0.0      -0.0         0.0      -0.0
 -1.52245   0.0200201  -0.0       0.0        -0.0       0.0
  0.0      -0.0         1.73065  -1.33313     0.0      -0.0
 -0.0       0.0        -1.52245   0.0200201  -0.0       0.0
  0.0      -0.0         0.0      -0.0         1.73065  -1.33313
 -0.0       0.0        -0.0       0.0        -1.52245   0.0200201
```

Linear quadratic control example. We start by writing a function `lqr` that constructs and solves the constrained least squares problem for linear quadratic control. The function returns three arrays

```
x = [ x[1], x[2], ..., x[T] ],
u = [ u[1], u[2], ..., u[T-1] ],
y = [ y[1], y[2], ..., y[T] ].
```

The first two contain the optimal solution of the problem. The third array contains $y_t = Cx_t$.

We allow the input arguments `x_init` and `x_des` to be matrices, so we can solve the same problem for different pairs of initial and end states, with one function call. If the number of columns in `x_init` and `x_des` is q, then the entries of the three output sequences `x`, `u`, `y` are matrices with q columns. The ith columns are the solution for the initial and end states specified in the ith columns of `x_init` and `x_des`.

```
function lqr(A,B,C,x_init,x_des,T,rho)
    n = size(A,1)
    m = size(B,2)
    p = size(C,1)
    q = size(x_init,2)
    Atil = [ kron(eye(T), C)  zeros(p*T,m*(T-1)) ;
        zeros(m*(T-1), n*T)  sqrt(rho)*eye(m*(T-1)) ]
    btil = zeros(p*T + m*(T-1), q)
    # We’ll construct Ctilde bit by bit
    Ctil11 = [ kron(eye(T-1), A) zeros(n*(T-1),n) ]  -
```

```
        [ zeros(n*(T-1), n) eye(n*(T-1)) ]
    Ctil12 = kron(eye(T-1), B)
    Ctil21 = [eye(n) zeros(n,n*(T-1));  zeros(n,n*(T-1)) eye(n)]
    Ctil22 = zeros(2*n,m*(T-1))
    Ctil = [Ctil11 Ctil12; Ctil21 Ctil22]
    dtil = [zeros(n*(T-1), q); x_init; x_des]
    z = cls_solve(Atil,btil,Ctil,dtil)
    x = [z[(i-1)*n+1:i*n,:] for i=1:T]
    u = [z[n*T+(i-1)*m+1 : n*T+i*m, :] for i=1:T-1]
    y = [C*xt for xt in x]
    return x, u, y
end;
```

We apply the function to the example in §17.2.1.

```
julia> A = [ 0.855  1.161  0.667;
             0.015  1.073  0.053;
            -0.084  0.059  1.022 ];
julia> B = [-0.076; -0.139; 0.342 ];
julia> C = [ 0.218  -3.597  -1.683 ];
julia> n = 3; p = 1; m = 1;
julia> x_init = [0.496; -0.745; 1.394];
julia> x_des = zeros(n,1);
```

We first plot the open-loop response of VMLS figure 17.4 in figure 17.3.

```
julia> T = 100;
julia> yol = zeros(T,1);
julia> Xol = [ x_init  zeros(n, T-1) ];
julia> for k=1:T-1
           Xol[:,k+1] = A*Xol[:,k];
       end;
julia> yol = C*Xol;
julia> using Plots
julia> plot(1:T, yol', legend = false)
```

We then solve the linear quadratic control problem with $T = 100$ and $\rho = 0.2$. The result is shown in the second row of VMLS figure 17.6 and in figure 17.4.

```
julia> rho = 0.2;
julia> T = 100;
julia> x, u, y = lqr(A,B,C,x_init,x_des,T,rho)
julia> J_input = norm(u)^2
```

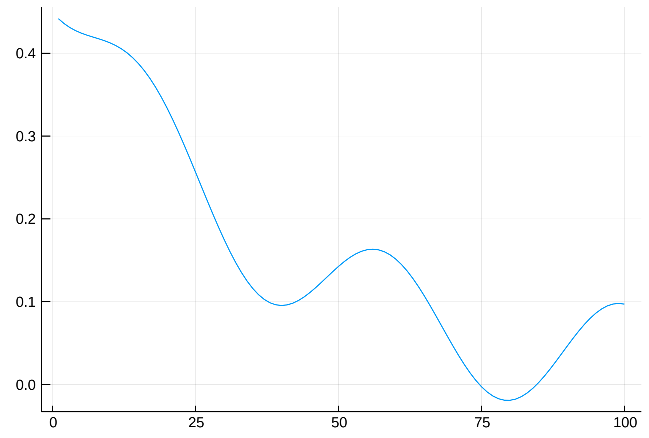


Figure 17.3 Open-loop response $CA^{t-1}x^{\text{init}}$.

```
0.7738942551160318
julia> J_output = norm(y)^2
3.7829986463323224
julia> plot(1:T-1, vcat(u...), legend = false, xlabel="t",
           ylabel= "u_t")
julia> plot(1:T, vcat(y...), legend=false, xlabel = "t",
           ylabel = "y_t")
```

Linear state feedback control. To finish the example we implement the state feedback method in VMLS section 17.2.3. The plots in figure 17.5 reproduce VMLS figure 17.7.

```
julia> # Solve LQ problem with x_init = I, x_des = 0
julia> rho = 1.0;
julia> xsf, usf, ysf = lqr(A,B,C,eye(n),zeros(n,n),T,rho);
julia> K = usf[1];
julia> # Simulate over horizon 150
julia> TT = 150;
julia> Xsf = [x_init zeros(n,TT-1)];
julia> for k=1:TT-1
           Xsf[:,k+1] = (A+B*K)*Xsf[:,k];
       end;
julia> usf = K*Xsf[:, 1:TT-1];
julia> ysf = C*Xsf;
julia> # Also compute optimal LQ solution for rho = 1.0
```

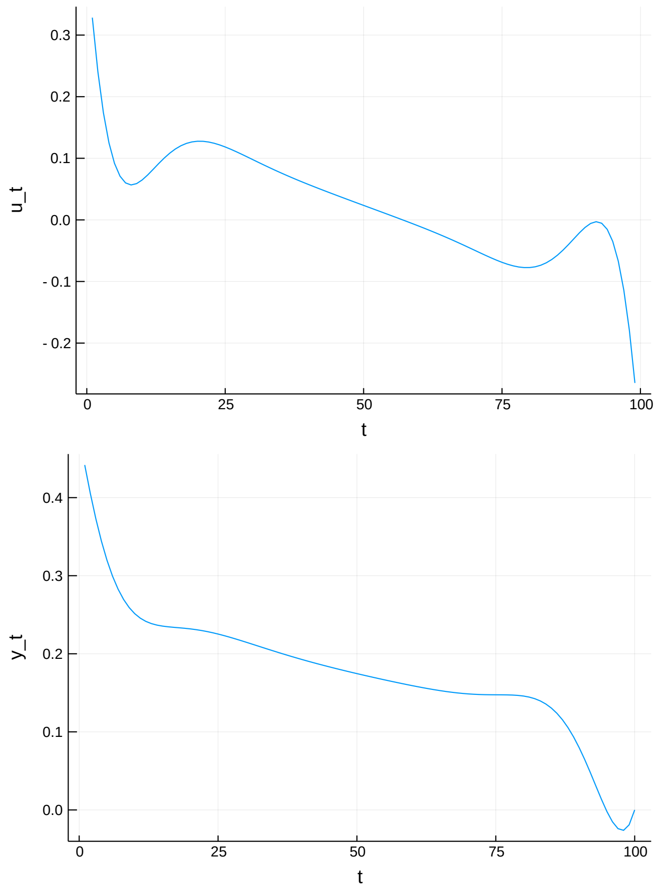
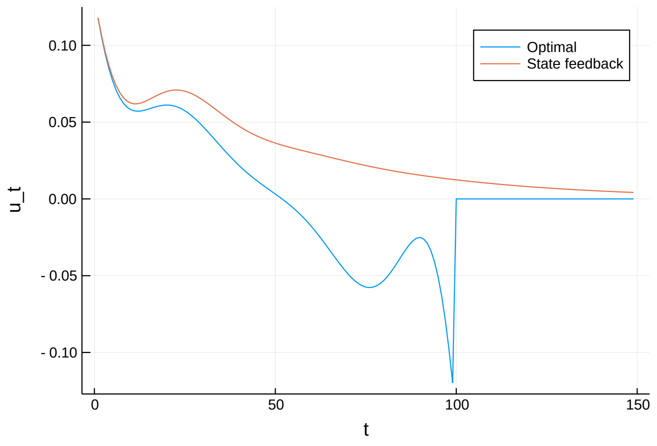


Figure 17.4 Optimal input and output for $\rho = 0.2$.



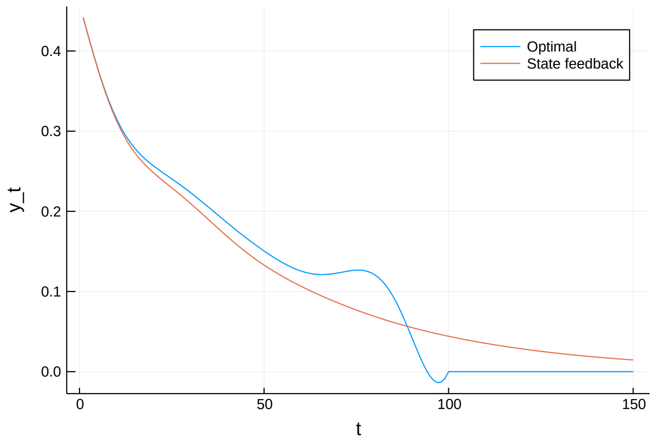


Figure 17.5 The blue curves are the solutions of the linear quadratic control problem for $\rho = 1$. The red curves are the inputs and outputs that result from the constant state feedback $u_t = Kx_t$.

.

```
julia> x, u, y = lqr(A,B,C,x_init,x_des,T,rho)
julia> # Plot the two inputs
julia> plot([vcat(u...); zeros(TT-T,1)],
           label="Optimal", xlabel = "t", ylabel = "u_t")
julia> plot!(usf', label = "State feedback")
julia> # Plot the two outputs
julia> plot_sf_y = plot([vcat(y...); zeros(TT-T,1)],
           label="Optimal", xlabel = "t", ylabel = "y_t")
julia> plot!(ysf', label = "State feedback")
```

17.3 Linear quadratic state estimation

The code for the linear quadratic estimation method is very similar to the one for linear quadratic control.

```
function lqe(A,B,C,y,T,lambda)
    n = size(A,1)
    m = size(B,2)
    p = size(C,1)
    Atil = [ kron(eye(T), C)  zeros(T*p, m*(T-1));
        zeros(m*(T-1), n*T)  sqrt(lambda)*eye(m*(T-1)) ]
    # We assume y is a p x T array, so we vectorize it
    btil = [ vcat(y...) ; zeros((T-1)*m) ]
    Ctil = [ ([ kron(eye(T-1), A) zeros(n*(T-1), n) ] +
        [ zeros(n*(T-1), n) -eye(n*(T-1)) ])  kron(eye(T-1), B) ]
    dtil = zeros(n*(T-1))
    z = cls_solve(Atil, btil, Ctil, dtil)
    x = [ z[(i-1)*n+1:i*n] for i=1:T ]
    u = [ z[n*T+(i-1)*m+1 : n*T+i*m] for i=1:T-1 ]
    y = [ C*xt for xt in x ]
    return x, u, y
end
```

We use the system matrices in §17.3.1 of VMLS. The output measurement data are read from an input file `estimation_data.jl`, which creates a 2×100 matrix `ymeas`. We compute the solution for $\lambda = 10^3$, shown in the lower-left plot of VMLS figure 17.8.

```
julia> ymeas = lq_estimation_data();
julia> A = [ eye(2)  eye(2); zeros(2,2)  eye(2) ];
julia> B = [ zeros(2,2); eye(2) ];
julia> C = [ eye(2) zeros(2,2) ];
julia> n = size(A,1);
julia> m = size(B,2);
julia> p = size(C,1);
julia> T = 100;
julia> lambda = 1e3;
julia> xest, uest, yest = lqe(A,B,C,y,T,lambda)
julia> using Plots
julia> scatter(y[1,:], y[2,:], legend = false, axis = false)
julia> plot!( [yt[1] for yt in yest], [yt[2] for yt in yest])
```

The result can be seen in figure 17.6

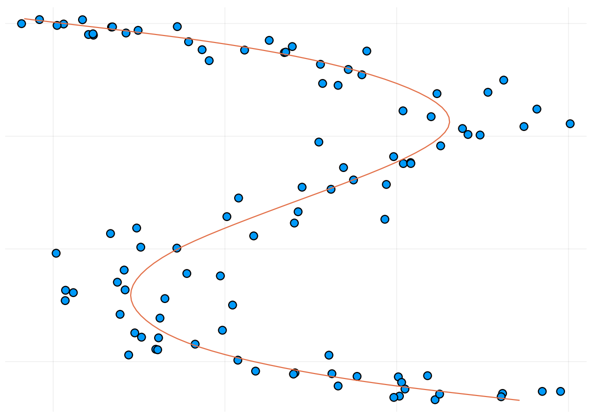


Figure 17.6 The circles show 100 noisy measurements in 2-D. The solid line is the estimated trajectory $C\hat{x}_t$ for $\lambda = 1000$.

Chapter 18

Nonlinear least squares

18.1 Nonlinear equations and least squares

18.2 Gauss–Newton algorithm

Basic Gauss–Newton algorithm. Let's first implement the basic Gauss–Newton method (algorithm 18.1 in VMLS) in Julia. In Julia, you can pass a function as an argument to another function, so we can pass `f` (the function) and also `Df` (the derivative or Jacobian matrix) to our Gauss–Newton algorithm.

```julia
function gauss_newton(f, Df, x1; kmax = 10)
    x = x1
    for k = 1:kmax
        x = x - Df(x) \ f(x)
    end
    return x
end
```

Here we simply run the algorithm for a fixed number of iterations `kmax`, specified by an optional keyword argument with default value 10. The code does not verify whether the final `x` is actually a solution, and it will break down when $Df(x^{(k)})$ has linearly dependent columns. This very simple implementation is only for illustrative purposes; the Levenberg–Marquardt algorithm described in the next section is better in every way.

Newton algorithm. The Gauss–Newton algorithm reduces to the Newton algorithm when the function maps n-vectors to n-vectors, so the function above is also an implementation of the Newton method for solving nonlinear equations. The only difference with the following function is the stopping condition. In Newton's method one terminates when $\|f(x^{(k)})\|$ is sufficiently small.

```
1  function newton(f, Df, x1; kmax = 20, tol = 1e-6)
2      x = x1
3      fnorms = zeros(1,0)
4      for k = 1:kmax
5          fk = f(x)
6          fnorms = [fnorms; norm(fk)]
7          if norm(fk) < tol
8              break
9          end;
10         x = x - Df(x) \ fk
11     end
12     return x, fnorms
13 end
```

We added a second optional argument with the tolerance in the stopping condition on line 7. The default value is 10^{-6}. We also added a second output argument `fnorms`, with the sequence $\|f(x^{(k)})\|$, so we can examine the convergence in the following examples.

Newton algorithm for $n = 1$. Our first example is a scalar nonlinear equation $f(x) = 0$ with

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{18.1}$$

(VMLS figures 18.3 and 18.4).

```
julia> f(x) = (exp(x)-exp(-x)) / (exp(x)+exp(-x));
julia> Df(x) = 4 / (exp(x) + exp(-x))^2;
```

We first try with $x^{(1)} = 0.95$.

```
julia> x, fnorms = newton(f,Df,0.95);
julia> f(x)
4.3451974324200454e-7
julia> fnorms
5-element Array{Float64,2}:
 0.7397830512740042
 0.5941663642651942
 0.23011124550034218
 0.00867002864500575
 4.3451974324200454e-7
julia> Using Plots
julia> plot(fnorms, shape=:circle, legend = false, xlabel = "k",
           ylabel = "|f|")
```

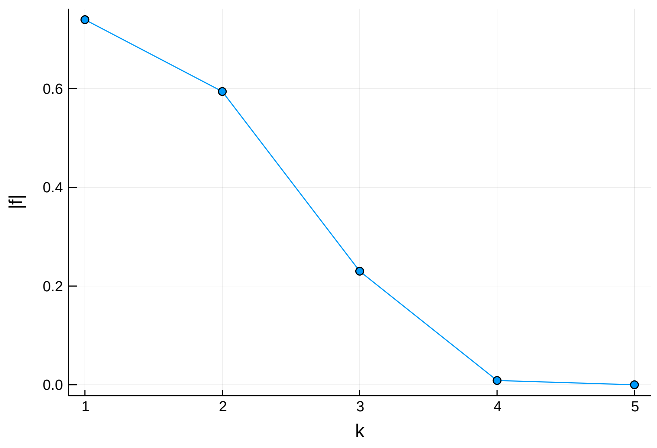


Figure 18.1 The first iterations in the Newton algorithm for solving $f(x) = 0$ for starting point $x^{(1)} = 0.95$.

The method converges very quickly, as can also be seen in figure 18.1. However it does not converge for a slightly larger starting point $x^{(1)} = 1.15$.

```
julia> x, fnorms = newton(f,Df,1.15);
julia> f(x)
NaN
julia> fnorms[1:5]
5-element Array{Float64,2}:
   0.8177540779702877
   0.8664056534177534
   0.9735568532451108
   0.9999999999999906
 NaN
```

18.3 Levenberg–Marquardt algorithm

The Gauss–Newton algorithm can fail if the derivative matrix does not have independent columns. It also does not guarantee that $\|f(x^{(k)})\|$ decreases in each iteration. Both of these shortcomings are addressed in the Levenberg–Marquardt algorithm. Below is a Julia implementation of algorithm 18.3 in VMLS. This function is included in the `Vmls` package.

```
 1 function levenberg_marquardt(f, Df, x1, lambda1; kmax=100, tol=1e-6)
 2     n = length(x1)
 3     x = x1
 4     lambda = lambda1
 5     objectives = zeros(0,1)
 6     residuals = zeros(0,1)
 7     for k = 1:kmax
 8          fk = f(x)
 9          Dfk = Df(x)
10          objectives = [objectives; norm(fk)^2]
11          residuals = [residuals; norm(2*Dfk'*fk)]
12          if norm(2*Dfk'*fk) < tol
13              break
14          end;
15          xt = x - [ Dfk; sqrt(lambda)*eye(n) ] \ [ fk; zeros(n) ]
16          if norm(f(xt)) < norm(fk)
17              lambda = 0.8*lambda
18              x = xt
19          else
20              lambda = 2.0*lambda
21          end
22     end
23     return x, Dict([ ("objectives", objectives),
24         ("residuals", residuals)])
25 end
```

Line 12 is the second stopping criterion suggested on page 393 of VMLS, and checks whether the optimality condition (18.3) is approximately satisfied. The default tolerance 10^{-6} can vary with the scale of the problem and the desired accuracy. Keep in mind that the optimality condition (18.3) is a necessary condition and does not guarantee that the solution minimizes the nonlinear least squares objective $\|f(x)\|^2$. The code limits the number of iterations to $k^{\max}$, after which it is assumed that the algorithm is failing to converge.

The function returns a dictionary with information about the sequence of iterates, including the value of $\|f(x^{(k)}\|^2$ and $\|Df(x^{(k)})^T f(x^{(k)})\|$ at each iteration.

Nonlinear equation. We apply the algorithm to the scalar function (18.1) with the starting point $x^{(1)} = 1.15$.

```
julia> f(x) = (exp.(x) - exp.(-x)) / (exp.(x) + exp.(-x));
julia> Df(x) = 4 ./ (exp.(x) + exp.(-x)).^2;
julia> x, history = levenberg_marquardt(f, Df, [1.15], 1.0);
julia> plot(sqrt.(history["objectives"][1:10]), shape = :circle,
```

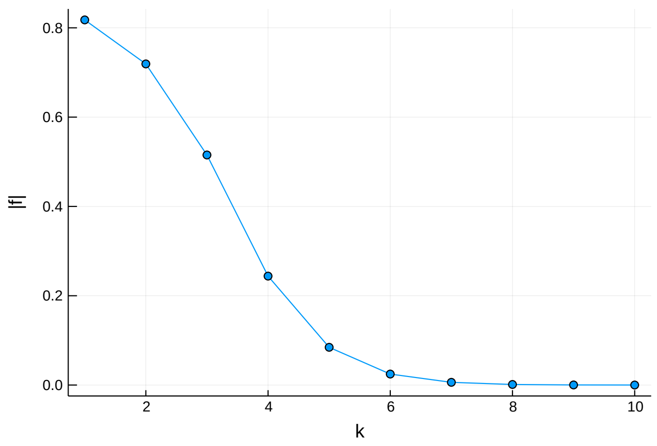


Figure 18.2 Values of $|f(x^{(k)})|$ versus the iteration number k for the Levenberg–Marquardt algorithm applied to $f(x) = (\exp(x) - \exp(-x))/(\exp(x)+\exp(-x))$. The starting point is $x^{(1)} = 1.15$ and $\lambda^{(1)} = 1$.

```
    legend = false, xlabel = "k", ylabel = "|f|");
```

The result is shown in figure 18.2.

Note that we defined $x^{(1)}$ as the array `[1.15]`, and use dot-operations in the definitions of `f` and `Df` to ensure that these functions work with vector arguments. This is important because Julia distinguishes between scalars and 1-vectors. If we call the `levenberg_marquardt` function with a scalar argument `x1`, line 15 will raise an error, because Julia does not accept subtractions of scalars and 1-vectors.

Equilibrium prices. We solve a nonlinear equation $f(p) = 0$ with two variables, where

$$f(p) = \exp(E^{\mathrm{s}} \log p + s^{\mathrm{nom}}) - \exp(E^{\mathrm{d}} \log p + d^{\mathrm{nom}}). \tag{18.2}$$

Here exp and log are interpreted as element-wise vector operations. The problem parameters are $s^{\mathrm{nom}} = (2.2, 0.3)$, $d^{\mathrm{nom}} = (3.1, 2.2)$,

$$E^{\mathrm{s}} = \begin{bmatrix} 0.5 & -0.3 \\ -0.15 & 0.8 \end{bmatrix}, \qquad E^{\mathrm{d}} = \begin{bmatrix} -0.5 & 0.2 \\ 0 & -0.5 \end{bmatrix}.$$

```
julia> snom = [2.2, 0.3];
julia> dnom = [3.1, 2.2];
julia> Es = [0.5  -.3; -0.15 0.8];
julia> Ed = [-0.5 0.2; -0.00 -0.5];
julia> f(p) = exp.(Es * log.(p) + snom) - exp.(Ed * log.(p) + dnom);
julia> function Df(p)
```

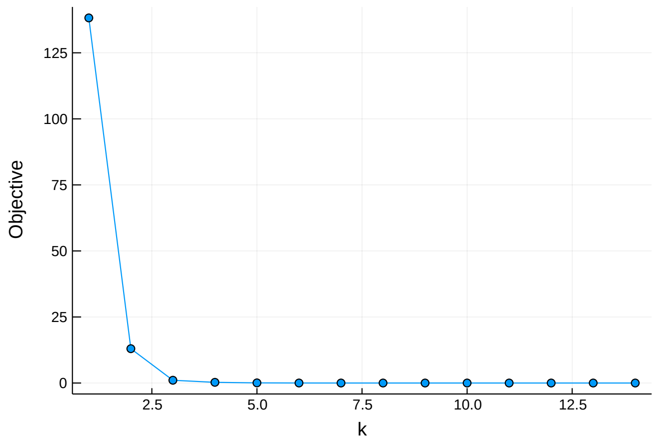


Figure 18.3 Cost function $\|f(p^{(k)}\|^2$ versus iteration number k for the example of equation (18.2).

```
       S = exp.(Es * log.(p) + snom);
       D = exp.(Ed * log.(p) + dnom);
       return [ S[1]*Es[1,1]/p[1]  S[1]*Es[1,2]/p[2];
                S[2]*Es[2,1]/p[1]  S[2]*Es[2,2]/p[2] ] -
              [ D[1]*Ed[1,1]/p[1]  D[1]*Ed[1,2]/p[2];
                D[2]*Ed[2,1]/p[1]  D[2]*Ed[2,2]/p[2] ];
       end;
julia> p, history = levenberg_marquardt(f, Df, [3, 9], 1);
julia> p
2-element Array{Float64,1}:
 5.379958476145877
 4.996349602562754
julia> using Plots
julia> plot(history["objectives"], shape = :circle, legend =false,
          xlabel = "k", ylabel = "Objective")
```

Figure 18.3 shows the plot of $\|f(p^{(k)})\|^2$ versus iteration number k.

Location from range measurements. The next example is the location from range measurements problem on page 396 in VMLS. The positions of the $m = 5$ points a_i are given as rows in a 5×2 matrix `A`. The measurements are given in a 5-vector `rhos`. To simplify the code for the functions `f(x)` and `Df(x)` we add a function `dist(x)` that computes the vector of distances $(\|x - a_1\|, \dots, \|x - a_m\|)$.

The expression for the derivative is

$$Df(x) = \begin{bmatrix} \dfrac{x_1-(a_1)_1}{\|x-a_1\|} & \dfrac{x_2-(a_1)_2}{\|x-a_1\|} \\ \vdots & \vdots \\ \dfrac{x_1-(a_m)_1}{\|x-a_m\|} & \dfrac{x_2-(a_m))_2}{\|x-a_m\|} \end{bmatrix}.$$

This can be evaluated as the product of a diagonal matrix with diagonal entries $1/\|x-a_i\|$ and the 5×2 matrix with i,j entry $(x-a_i)_j$.

We run the Levenberg–Marquardt method for three starting points and $\lambda^{(1)} = 0.1$. The plot is shown in figure 18.4.

```
julia> # Five locations ai in a 5x2 matrix.
julia> A = [ 1.8  2.5; 2.0  1.7; 1.5  1.5; 1.5  2.0; 2.5  1.5 ];
julia> # Vector of measured distances to five locations.
julia> rhos = [ 1.87288, 1.23950, 0.53672, 1.29273, 1.49353 ];
julia> # dist(x) returns a 5-vector with the distances ||x-ai||.
julia> dist(x) = sqrt.( (x[1] .- A[:,1]).^2 + (x[2] .- A[:,2]).^2 );
julia> # f(x) returns the five residuals.
julia> f(x) = dist(x) - rhos;
julia> # Df(x) is the 5x2 derivative.
julia> Df(x) = diagonal(1 ./ dist(x)) *
           [ (x[1] .- A[:,1])  (x[2] .- A[:,2]) ];
julia> # Solve with starting point (1.8,3.5) and lambda = 0.1.
julia> x1, history1 = levenberg_marquardt(f, Df, [1.8, 3.5], 0.1);
julia> x1
2-element Array{Float64,1}:
 1.1824859803827907
 0.8242289367900364
julia> # Starting point (3.0,1.5).
julia> x2, history2 = levenberg_marquardt(f, Df, [3.0, 1.5], 0.1);
julia> x2
2-element Array{Float64,1}:
 1.1824857942435818
 0.8242289466379732
julia> # Starting point (2.2,3.5).
julia> x3, history3 = levenberg_marquardt(f, Df, [2.2, 3.5], 0.1);
julia> x3
2-element Array{Float64,1}:
 2.9852664103617954
 2.1215768036188956
julia> using Plots
```

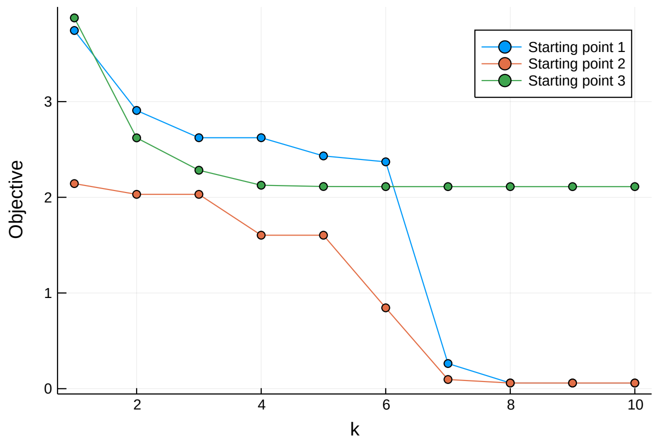


Figure 18.4 Cost function $\|f(x^{(k)})\|^2$ versus iteration number k for the three starting points in the location from range measurements example.

```
julia> plot(history1["objectives"][1:10], shape = :circle,
            label = "Starting point 1")
julia> plot!(history2["objectives"][1:10], shape = :circle,
            label = "Starting point 2")
julia> plot!(history3["objectives"][1:10], shape = :circle,
            label = "Starting point 3")
julia> plot!(xlabel = "k", ylabel = "Objective")
```

18.4 Nonlinear model fitting

Example. We fit a model

$$\hat{f}(x;\theta) = \theta_1 e^{\theta_2 x} \cos(\theta_3 x + \theta_4)$$

to $N = 60$ data points. We first generate the data.

```
julia> # Use these parameters to generate data.
julia> theta_ex = [1, -0.2, 2*pi/5, pi/3];
julia> # Choose 60 points x between 0 and 20.
julia> M = 30;
julia> xd = [5*rand(M); 5 .+ 15*rand(M)];
julia> # Evaluate function at these points.
julia> yd = theta_ex[1] * exp.(theta_ex[2]*xd) .*
           cos.(theta_ex[3] * xd .+ theta_ex[4])
julia> # Create a random perturbation of yd.
julia> N = length(xd);
julia> yd = yd .* (1 .+ 0.2*randn(N)) .+ 0.015 * randn(N);
julia> # Plot data points.
julia> using Plots
julia> scatter(xd, yd, legend=false)
```

The 60 points are shown in figure 18.5. We now run our Levenberg–Marquardt code with starting point $\theta^{(1)} = (1, 0, 1, 0)$ and $\lambda^{(1)} = 1$. The fitted model is shown in figure 18.5.

```
julia> f(theta) =  theta[1] * exp.(theta[2]*xd) .*
           cos.(theta[3] * xd .+ theta[4]) - yd;
julia> Df(theta) = hcat(
           exp.(theta[2]*xd) .* cos.(theta[3] * xd .+ theta[4]),
           theta[1] * ( xd .* exp.(theta[2]*xd) .*
               cos.(theta[3] * xd .+ theta[4])),
           -theta[1] * ( exp.(theta[2]*xd) .* xd .*
               sin.(theta[3] * xd .+ theta[4])),
           -theta[1] * ( exp.(theta[2]*xd) .*
               sin.(theta[3] * xd .+ theta[4])) );
julia> theta1 = [1, 0, 1, 0];
julia> theta, history = levenberg_marquardt(f, Df, theta1, 1.0)
julia> theta
4-element Array{Float64,1}:
  1.0065969737811806
 -0.23115179954434736
  1.2697087881931268
  1.0133243392186635
julia> # Plot the fitted model.
julia> x = linspace(0, 20, 500);
julia> y=theta[1]*exp.(theta[2]*x) .* cos.(theta[3]*x .+ theta[4]);
julia> plot!(x, y, legend = false);
```

Orthogonal distance regression. In figure 18.14 of VMLS we use orthogonal distance regression to fit a cubic polynomial

$$\hat{f}(x;\theta) = \theta_1 + \theta_2 x + \theta_3 x^2 + \theta_4 x^3$$

to $N = 25$ data points.

We first read in the data and compute the standard least squares fit.

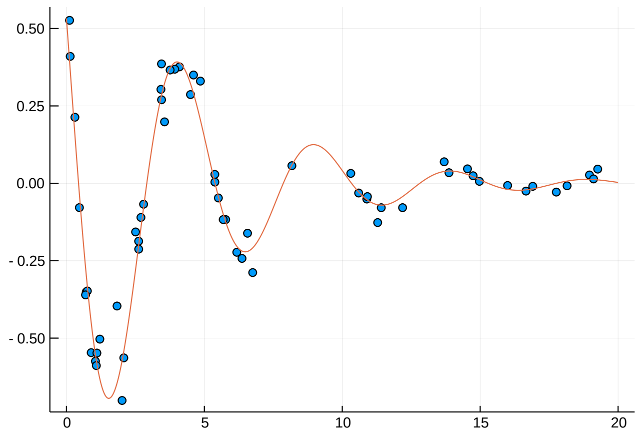


Figure 18.5 Least squares fit of a function $\hat{f}(x;\theta) = \theta_1 e^{\theta_2 x}\cos(\theta_3 x + \theta_4)$ to $N = 60$ points $(x^{(i)}, y^{(i)})$.

```
julia> xd, yd = orth_dist_reg_data();  # 2 vectors of length N = 25
julia> N = length(xd);
julia> p = 4;
julia> theta_ls = vandermonde(xd, p) \ yd;
```

The nonlinear least squares formulation on page 400 of VMLS has $p+N$ variables $\theta_1, \ldots, \theta_p, u^{(1)}, \ldots, u^{(N)}$. We will store them in that order in the nonlinear least squares vector variable. The objective is to minimize the squared norm of the $2N$-vector

$$\left[\begin{array}{c} \hat{f}(u^{(1)};\theta) - y^{(1)} \\ \vdots \\ \hat{f}(u^{(N)};\theta) - y^{(N)} \\ u^{(1)} - x^{(1)} \\ \vdots \\ u^{(N)} - x^{(N)} \end{array}\right]$$

```
function f(x)
    theta = x[1:p];
    u = x[p+1:end];
    f1 = vandermonde(u,p)*theta - yd
    f2 = u - xd
    return [f1; f2]
end;
```

```
function Df(x)
    theta = x[1:p]
    u = x[p+1:end]
    D11 = vandermonde(u,p)
    D12 = diagonal(theta[2] .+ 2*theta[3]*u .+ 3*theta[4]*u.^2)
    D21 = zeros(N,p)
    D22 = eye(N)
    return [ D11 D12; D21 D22]
end
```

We now call `levenberg_marquardt` with these two functions. A natural choice for the initial point is to use the least squares solution for the variables θ and the data points $x^{(i)}$ for the variables $u^{(i)}$. We use $\lambda^{(1)} = 0.01$.

```
julia> sol, hist = levenberg_marquardt(f, Df, [theta_ls; xd], 0.01);
julia> theta_od = sol[1:p];
```

Figure 18.6 shows the two fitted polynomials.

```
julia> using Plots
julia> scatter(xd,yd, label="", legend = :topleft);
julia> x = linspace(minimum(xd), maximum(xd), 500);
julia> y_ls = vandermonde(x, p) * theta_ls;
julia> y_od = vandermonde(x, p) * theta_od;
julia> plot!(x, y_ls, label = "LS");
julia> plot!(x, y_od, label = "Orth. dist.");
```

18.5 Nonlinear least squares classification

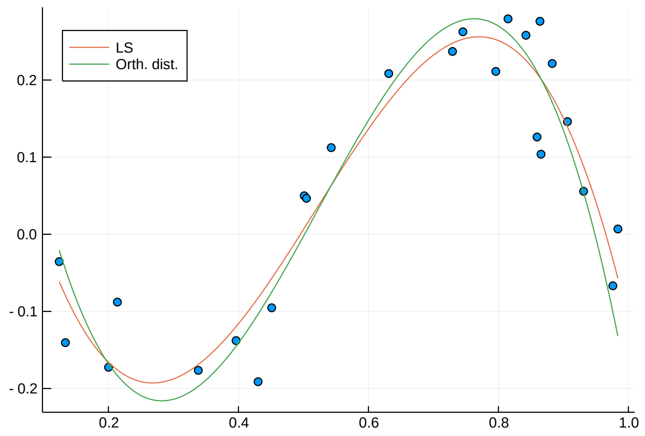


Figure 18.6 Least squares and orthogonal distance regression fit of a cubic polynomial to 25 data points.

Chapter 19

Constrained nonlinear least squares

19.1 Constrained nonlinear least squares

19.2 Penalty algorithm

Let's implement the penalty algorithm (algorithm 19.1 in VMLS).

```julia
function penalty_method(f, Df, g, Dg, x1, lambda1; kmax = 100,
        feas_tol = 1e-4, oc_tol = 1e-4)
    x = x1
    mu = 1.0
    feas_res = [norm(g(x))]
    oc_res = [norm(2*Df(x)'*f(x) + 2*mu*Dg(x)'*g(x))]
    lm_iters = zeros(Int64,0,1);
    for k=1:kmax
        F(x) = [f(x); sqrt(mu)*g(x)]
        DF(x) = [Df(x); sqrt(mu)*Dg(x)]
        x, hist = levenberg_marquardt(F,DF,x,lambda1,tol=oc_tol)
        feas_res = [feas_res; norm(g(x))]
        oc_res = [oc_res; hist["residuals"][end]]
        lm_iters = [lm_iters; length(hist["residuals"])]
        if norm(g(x)) < feas_tol
            break
        end
        mu = 2*mu
    end
    return x, Dict([ ("lm_iterations", lm_iters),
21          ("feas_res", feas_res), ("oc_res", oc_res) ])
22  end
```

On line 11 we call the function `levenberg_marquardt` of the previous chapter to minimize $\|F(x)\|^2$ where

$$F(x) = \begin{bmatrix} f(x) \\ \sqrt{\mu} g(x) \end{bmatrix}.$$

We evaluate two residuals. The "feasibility" residual $\|g(x^{(k)})\|$ is the error in the constraint $g(x) = 0$. The "optimality condition" residual is defined as

$$\|2Df(x^{(k)})^T f(x^{(k)}) + 2Dg(x^{(k)})^T z^{(k)}\|$$

where $z^{(k)} = 2\mu^{(k-1)} g(x^{(k)})$ (and we take $\mu^{(0)} = \mu^{(1)}$). On line 13, we obtain the optimality condition residual as the last residual in the Levenberg–Marquardt method. On line 20 we return the final x, and a dictionary containing the two sequences of residuals and the number of iterations used in each call to the Levenberg–Marquardt algorithm.

Example. We apply the method to a problem with two variables

$$f(x_1, x_2) = \begin{bmatrix} x_1 + \exp(-x_2) \\ x_1^2 + 2x_2 + 1 \end{bmatrix}, \qquad g(x_1, x_2) = x_1^2 + x_1^3 + x_2 + x_2^2.$$

```
julia> f(x) = [ x[1] + exp(-x[2]), x[1]^2 + 2*x[2] + 1 ];
julia> Df(x) = [ 1.0 - exp(-x[2]);  2*x[1]  2 ];
julia> g(x) = [ x[1] + x[1]^3 + x[2] + x[2]^2 ];
julia> Dg(x) = [ 1 + 3*x[1]^2  1 + 2*x[2] ];
julia> x, hist = penalty_method(f, Df, g, Dg, [0.5, -0.5], 1.0);
julia> x
2-element Array{Float64,1}:
 -3.334955140841332e-5
 -2.7682497163944097e-5
```

The following lines create a staircase plot with the residuals versus the cumulative number of Levenberg–Marquardt iterations as in VMLS figure 19.4. The result is in figure 19.1.

```
julia> using Plots
julia> cum_lm_iters = cumsum(hist["lm_iterations"], dims=1);
julia> itr = vcat([0], [[i; i] for i in cum_lm_iters]...)
julia> feas_res = vcat([
           [r;r] for r in hist["feas_res"][1:end-1]]...,
           hist["feas_res"][end]);
julia> oc_res = vcat([
```

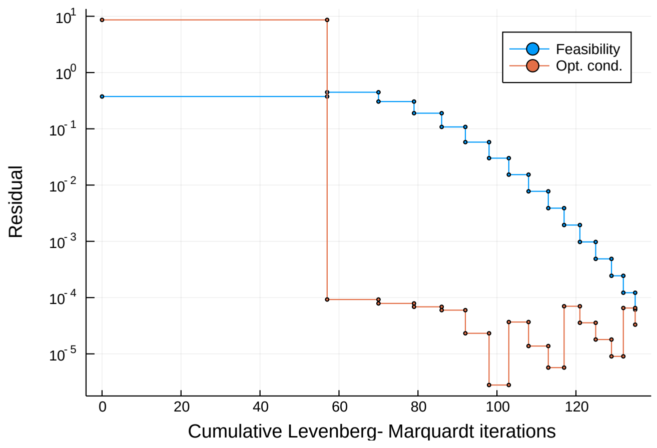


Figure 19.1 Feasibility and optimality condition errors versus the cumulative number of Levenberg–Marquardt iterations in the penalty algorithm.

```
          [r;r] for r in hist["oc_res"][1:end-1]]...,
          hist["oc_res"][end]);
julia> plot(itr, feas_res, shape=:circle, label = "Feasibility")
julia> plot!(itr, oc_res, shape=:circle, label = "Opt. cond.")
julia> plot!(yscale = :log10,
        xlabel = "Cumulative Levenberg--Marquardt iterations",
        ylabel = "Residual")
```

19.3 Augmented Lagrangian algorithm

```
function aug_lag_method(f, Df, g, Dg, x1, lambda1;  kmax = 100,
    feas_tol = 1e-4, oc_tol = 1e-4)
    x = x1
    z = zeros(length(g(x)))
    mu = 1.0
    feas_res = [norm(g(x))]
    oc_res = [norm(2*Df(x)'*f(x) + 2*mu*Dg(x)'*z)]
    lm_iters = zeros(Int64,0,1);
    for k=1:kmax
        F(x) = [f(x); sqrt(mu)*(g(x) + z/(2*mu))]
```

```
11          DF(x) = [Df(x); sqrt(mu)*Dg(x)]
12          x, hist = levenberg_marquardt(F, DF, x, lambda1, tol=oc_tol)
13          z = z + 2*mu*g(x)
14          feas_res = [feas_res; norm(g(x))]
15          oc_res = [oc_res; hist["residuals"][end]]
16          lm_iters = [lm_iters; length(hist["residuals"])]
17          if norm(g(x)) < feas_tol
18              break
19          end
20          mu = (norm(g(x)) < 0.25*feas_res[end-1]) ? mu : 2*mu
21      end
22      return x, z, Dict([ ("lm_iterations", lm_iters),
23           ("feas_res", feas_res), ("oc_res", oc_res)])
24  end
```

Here the call to the Levenberg–Marquardt algorithm on line 12 is to minimizes $\|F(x)\|^2$ where

$$F(x) = \begin{bmatrix} f(x) \\ \sqrt{\mu^{(k)}}(g(x) + z^{(k)}/(2\mu^{(k)})) \end{bmatrix}.$$

We again record the feasibility residuals $\|g(x^{(k)}\|$ and the optimality conditions residuals

$$\|2Df(x^{(k)})^T f(x^{(k)}) + 2Dg(x^{(k)})^T z^{(k)}\|,$$

and return them in a dictionary.

Example. We continue the small example.

```
julia> x, z, hist = aug_lag_method(f, Df, g, Dg, [0.5, -0.5], 1.0);
julia> x
2-element Array{Float64,1}:
 -1.8646614856169702e-5
 -1.5008567819930016e-5
julia> z
1-element Array{Float64,1}:
 -1.9999581273499105
```

The following code shows the convergence as in VMLS figure 19.4. The plot is given in figure 19.2.

```
julia> using Plots
julia> cum_lm_iters = cumsum(hist["lm_iterations"],dims=1);
julia> itr = vcat([0], [[i; i] for i in cum_lm_iters]...)
julia> feas_res = vcat([
           [r;r] for r in hist["feas_res"][1:end-1]]...,
```

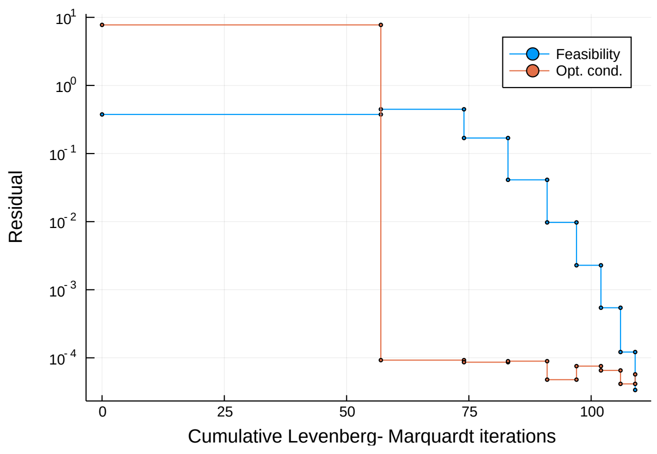


Figure 19.2 Feasibility and optimality condition errors versus the cumulative number of Levenberg–Marquardt iterations in the augmented Lagrangian algorithm.

```
           hist["feas_res"][end]);
julia> oc_res = vcat([
           [r;r] for r in hist["oc_res"][1:end-1]]...,
           hist["oc_res"][end]);
julia> plot(itr, feas_res, shape=:circle, , label = "Feasibility")
julia> plot!(itr, oc_res, shape=:circle, label = "Opt. cond.")
julia> plot!(yscale = :log10,
       xlabel = "Cumulative Levenberg-Marquardt iterations",
       ylabel = "Residual")
```

19.4 Nonlinear control

Appendices

Appendix A

The `VMLS` package

After installing the `VMLS` package as described on page ix, you can use it by typing `using VMLS` at the Julia prompt. Typing `?` followed by a function name gives a short description of the function.

The `VMLS` package includes three types of functions: simple utility functions that match the VMLS notation or are simpler to use than the corresponding Julia functions, implementations of some algorithms in VMLS, and functions that generate data for some of the examples. The algorithm implementations are meant for use in the examples of this companion and for exercises in VMLS. They are not optimized for efficiency or robustness, and do not perform any error checking of the input arguments.

A.1 Utility functions

Vector utility functions.

`avg(x)`. Returns the average of the elements of a vector or matrix (page 20).

`rms(x)`. Returns the RMS value of the elements of a vector or matrix (page 25).

`stdev(x)`. Returns the standard deviation of the elements of a vector or matrix (page 28).

`ang(x,y)`. Returns the angle in radians between non-zero vectors (page 29).

`correl_coeff(x,y)`. Returns the correlation coefficient between non-constant vectors (page 30).

Matrix utility functions.

`eye(n)`. Returns an $n \times n$ identity matrix (page 50).

`diagonal(x)`. Returns a diagonal matrix with the entries of the vector x on its diagonal (page 52).

`speye(n)`. Returns an $n \times n$ sparse identity matrix (page 54).

`spdiagonal(x)`. Returns a sparse diagonal matrix with the entries of the vector x on its diagonal (page 54).

`vandermonde(t,n)`. Returns the Vandermonde matrix with n columns and ith column t^{i-1} (page 58).

`toeplitz(a,n)`. Returns the Toeplitz matrix with n columns and the vector a in the leading positions of the first column (page 65).

Range function.

`linspace(a,b,n)`. Returns a vector with n equally spaced numbers between a and b (page 90).

Utility functions for classification.

`confusion_matrix(y,yhat,K=2)`. Returns the confusion matrix for a data vector y and the vector of predictions $\hat{y}$. If $K = 2$, the vectors y and $\hat{y}$ are Boolean. If $K > 2$, they contain integers in $\{1, \ldots, K\}$ (pages 113, 115).

`row_argmax(X)`. If X is an $m \times n$ matrix, returns an m-vector with ith element $\text{argmax}_j X_{ij}$ (page 116).

`one_hot(x,K)`. Returns the one-hot encoding of the vector x, which must have elements in $\{1, \ldots, K\}$. The one-hot encoding of an n-vector x is the $n \times K$ matrix X with $X_{ij} = 1$ if $x_i = j$ and $X_{ij} = 0$ otherwise (page 117).

A.2 Algorithms

`k_means(X,k;maxiters=100,tol=1e-5)`. Applies the k-means algorithm for k clusters to the vectors stored in `X`. The argument `X` is a one-dimensional array of N n-vectors, or an $n \times N$-matrix. The function returns a tuple with two elements. The first output argument is an array of N integers in $\{1, \ldots, k\}$ with the cluster assignments for the N data points. The second output argument is an array of k n-vectors with the k cluster representatives (page 34).

`gram_schmidt(a;tol=1e-10)`. Applies the Gram–Schmidt algorithm to the vector stored in the array `a` and returns the result as an array of vectors (page 41).

`mols_solve(As,Bs,lambdas)`. Returns the solution of the multi-objective least squares problem with coefficient matrices in the array `As`, right-hand side vectors in the array `bs`, and weights in the array `lambdas` (page 121).

`cls_solve(A,b,C,d)`. Returns the solution of the constrained least squares problem with coefficient matrices A and C, and right-hand side vectors or matrices b and d (page 132).

`levenberg_marquardt(f,Df,x1,lambda1;kmax=100,tol=1e-6)`. Applies the Levenberg–Marquardt algorithm to the function defined in `f` and `Df`, with starting point $x^{(1)}$ and initial regularization parameter $\lambda^{(1)}$. The function returns the final iterate x and a dictionary with the convergence history (page 147).

`aug_lag_method(f,Df,g,Dg,x1,lambda1;kmax=100,feas_tol=1e-4,oc_tol=1e-4)`. Applies the augmented Lagrangian method to the constrained nonlinear least squares problem defined by `f`, `Df`, `g`, `Dg`, with starting point $x^{(1)}$. The subproblems are solved using the Levenberg–Marquardt method with initial regularization parameter $\lambda^{(1)}$. Returns the final iterate x, multiplier z, and a dictionary with the convergence history (page 159).

A.3 Data sets

`house_sales_data()`. Returns a dictionary `D` with the Sacramento house sales data used in section 2.3 and chapter 13 of VMLS. The 6 items in the dictionary are vectors of length 774, with data for 774 house sales.

`D["price"]`:	selling price in 1000 dollars
`D["area"]`:	area in 1000 square feet
`D["beds"]`:	number of bedrooms
`D["baths"]`:	number of bathrooms
`D["condo"]`:	1 if a condo, 0 otherwise
`D["location"]`:	an integer between 1 and 4 indicating the location.

`population_data()`. Returns a dictionary `D` with the US population data used in section 9.2 of VMLS. The items in the dictionary are three vectors of length 100.

`D["population"]`:	2010 population in millions for ages 0,...,99
`D["birth_rate"]`:	birth rate
`D["death_rate"]`:	death rate.

`petroleum_consumption_data()`. Returns a 34-vector with the world annual petroleum consumption between 1980 and 2013, in thousand barrels/day (discussed on page 252 in VMLS).

`vehicle_miles_data()`. Returns a 15 matrix with the vehicle miles traveled in the US (in millions), per month, for the years $2000, \ldots, 2014$ (discussed on page 252 in VMLS).

`temperature_data()`. Returns a vector of length $774 = 31 \cdot 24$ with the hourly temperature at LAX in May 2016 (discussed on pages 259 and 266 in VMLS).

`iris_data()`. Returns a dictionary `D` with the Iris flower data set, discussed in sections 14.2.1 and 14.3.2 of VMLS. The items in the dictionary are:

`D["setosa"]`	a 50×4 matrix with 50 examples of Iris Setosa
`D["versicolor"]:`	a 50×4 matrix with 50 examples of Iris Versicolor
`D["virginica"]:`	a 50×4 matrix with 50 examples of Iris Virginica.

The columns give values for four features: sepal length in cm, sepal width in cm, petal length in cm, petal width in cm.

`ozone_data()`. Returns a vector of length $336 = 14 \cdot 24$ with the hourly ozone levels at Azusa, California, during the first 14 days of July 2014 (discussed on page 319 in VMLS).

`regularized_fit_data()`. Returns a dictionary `D` with data for the regularized data fitting example on page 329 of VMLS. The items in the dictionary are:

`D["xtrain"]:`	vector of length 10
`D["ytrain"]:`	vector of length 10
`D["xtest"]:`	vector of length 20
`D["ytest"]:`	vector of length 20.

`portfolio_data()`. Returns a tuple `(R, Rtest)` with data for the portfolio optimization example in section 17.1.3 of VMLS. `R` is a 2000×20 matrix with daily returns over a period of 2000 days. The first 19 columns are returns for 19 stocks; the last column is for a risk-free asset. `Rtest` is a 500×20 matrix with daily returns over a different period of 500 days.

`lq_estimation_data()`. Returns a 2×100 matrix with the measurement data for the linear quadratic state estimation example of section 17.3.1 of VMLS.

`orth_dist_reg_data()` Returns a tuple `(xd, yd)` with the data for the orthogonal distance regression example on page 400 in VMLS.