Chapter 5

Euler

5.1 Input

5.2 Arrays

An array is a sequence of numbers. The colon operator, (`:`), is one way to create an array. We have already used one form of the colon operator in a `for` loop:

```
for j = 1:9
    ...
end
```

In this case the expression `1:9` creates an array containing the integers from 1 to 9 (including both). You can create a array on the command line or assign it to a variable.

```
>> 1:9
ans =  1      2      3      4      5      6      7      8      9

>> x = 1:9
x =  1      2      3      4      5      6      7      8      9
```

There is an alternate form of the colon operator that allows you to specify the **increment** between the elements of the array. For example, the expression `1:2:9` creates the array that starts at 1, ends at 9, and takes a step of size 2 between elements.

```
>> 1:2:9
ans =  1      3      5      7      9
```

The examples we have seen so far happen to be integers, but they needn't be.

```
>> 0:0.2:1
ans =  0     0.2000     0.4000     0.6000     0.8000     1.0000
```

Also, the increment can be negative.

```
>> 1:-0.3:-1
ans = 1.0000    0.7000    0.4000    0.1000   -0.2000   -0.5000   -0.8000
```

If the increment has the wrong sign, as in `0:-1:9` and `9:1:0`, the resulting array has no elements.

```
>> 0:-1:10
ans =  Empty matrix: 1-by-0
```

The error message refers to a **matrix**. As we will see soon, an array is a kind of matrix.

5.3 Array Concatenation

Another way to create an array is to concatenate a list of numbers. The `[]` operator performs concatenation.

```
>> [1 2 3]
ans =  1     2     3
```

The vectors we used to make fractals were really arrays with two elements. The elements of the array can be arbitrary expressions.

```
>> [sin(pi/2) 10+6 exp(4)]
ans =   1.0000   16.0000   54.5982
```

It is also possible to concatenate numbers with arrays, and arrays with each other.

```
>> head = [1 2]
head =  1     2

>> head = [head 3]
head =  1     2     3

>> tail = [4 5 6]
tail =  4     5     6

>> [head tail]
ans =  1     2     3     4     5     6
```

The first example concatenates two numbers. The second example appends a number onto the array `head`. The last example concatenates two arrays.

Yet another way to create an array is with the `linspace` function. See the documentation for details.

5.4 Empty arrays and scalars

The concatenation operator makes it possible to create an empty array.

```
>> empty = []
empty =  []
```

The length of the empty array, not surprisingly, is zero.

Of course, it is also possible to have an array with a single element, like `[2]`. In linear algebra, an array with a single element is not the same thing as a number, but in MATLAB it can be used anywhere a number can be used. In contrast to an array, a single number is called a **scalar**.

5.5 Array operations

Many of the built-in operators and functions apply to arrays.

```
>> x = (0:0.2:1) * pi
x =      0    0.6283    1.2566    1.8850    2.5133    3.1416

>> sin (x)
ans =    0    0.5878    0.9511    0.9511    0.5878    0.0000
```

In the first example, each element of the array `(0:0.2:1)` is multiplied by the scalar `pi`. The parentheses are necessary because the colon operator has lower priority than multiplication.

In the second example, the `sin` function handles an array argument by computing the sine of each element.

Addition and subtraction work for both scalars and arrays. If one of the operands is a scalar, it is added to (or subtracted from) each element of the array. If both operands are arrays, they are added (or subtracted) elementwise.

```
>> x = 1:5
x =    1     2     3     4     5

>> y = x + 1
y =    2     3     4     5     6

>> y-x
ans =  1     1     1     1     1
```

As we have already seen, you can multiply the elements of an array by a scalar, but if you apply the `*` operator to two arrays, it performs **vector multiplication**, which we will get to a little later. In the meantime, we can use the `.*` operator, which performs elementwise multiplication. For example, we can square the elements of an array by multiplying it by itself.

```
>> x = 1:5
x =    1     2     3     4     5

>> x*x
??? Error using ==> *
Inner matrix dimensions must agree.

>> x.*x
ans =  1     4     9    16    25
```

Similarly, the ^ operator tries to perform vector multiplication, and fails, but the .^ operator does what we want.

```
>> x^2
??? Error using ==> ^
Matrix must be square.

>> x.^2
ans =    1     4     9    16    25
```

Division is asymmetric. We can divide an array by a scalar, but not the other way around.

```
>> x/2
ans =  0.5000    1.0000    1.5000    2.0000    2.5000

>> 2/x
??? Error using ==> /
Matrix dimensions must agree.
```

5.6 Array indexing

The () operators select elements from an array. If the index is an integer expression, it selects a single element.

```
>> x = (1:9) * 2
x =  2    4     6     8    10    12    14    16    18

>> x(1)
ans =  2

>> x(9)
ans =  18
```

The first statement creates an array and multiplies it by 2. The elements of `x` are numbered from 1 to `length(x)`. The function `length` yields the length of an array.

```
>> length(x)
ans =  9

>> x(length(x))
ans =  18
```

Alternatively, you can use the keyword `end` as an index; it refers to the last element of the array.

```
>> x(end)
ans =  18
```

It is illegal to use an index less than 1 or greater than `length(x)`.

```
>> x(0)
??? Index into matrix is negative or zero.

>> x(10)
???  Index exceeds matrix dimensions.
```

Again, the error messages refer to matrices.

5.7 Arrays as indices

Interestingly, you can use an array as an index. As long as all the elements of the index are integers, the index selects the corresponding elements of the array.

```
>> x = (1:9) * 2
x =  2     4     6     8    10    12    14    16    18

>> index = [1 3 5]
index =  1     3     5

>> x(index)
ans =  2     6    10
```

The index `[1 3 5]` selects the first, third and fifth elements of `x`.

It is common to use the colon operator to construct an array for use as an index.

```
>> x(1:2:5)
ans =  2     6    10
```

Again, this example selects the first, third and fifth elements.

When the colon operator appears as an array index, it can use the keyword `end` to refer to the last element.

```
>> x(5:end)
ans =  10    12    14    16    18
```

5.8 Plotting

One of the nicer things about MATLAB is its extensive collection of functions for plotting data. We have already seen one of those functions, `line`. In this section we will look at another: `plot`.

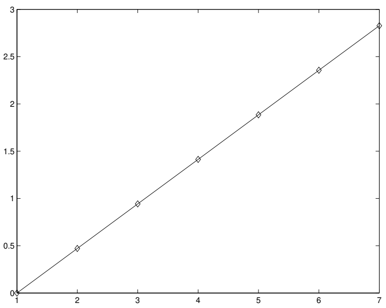
`plot` can take arguments in several forms. The simplest is a single array.

```
>> x = linspace (0, pi, 7)
x =       0    0.5236    1.0472    1.5708    2.0944    2.6180    3.1416

>> plot(x)
>> plot(x, 'bd')
>> plot(x, 'bd-')
```

The first statement demonstrates the use of `linspace` to create an array of seven elements. Invoking `plot` with the argument `x` draw a single line with the default color.

The second argument to `plot` can be a format string. The format string `'bd'` draws blue diamonds at each data point. The format `'bd-'` draws both blue diamonds and the lines connecting them. See the documentation of `plot` for details about these format strings. The result of the third `plot` looks like this.



The horizontal axis is labeled with the indices of the array, from 1 to 7. The vertical axis labels the values of the array. As we have seen before, MATLAB chooses the range of the axes so that the data fit in the window.

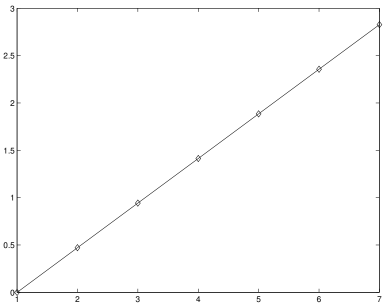
This form of the `plot` function is useful for checking the contents of an array, but for applications it is more common to plot one array as a function of another. For example, if we compute `y` as a function of `x`, then we would plot `y` versus `x`.

```
>> y = sin (x)
y =    0    0.5000    0.8660    1.0000    0.8660    0.5000    0.0000

>> plot (x, y)
>> plot (x, y, 'bd-')
```

I always have trouble remembering what order the arguments should be in. The array that should be plotted on the x axis goes first.

Again, the third argument can be a format string. The result of the second `plot` looks like this.



5.9 Euler's method

The handout from Gould and Tobochnik's book explains the general use of Euler's method to solve differential equations in the form

$$\frac{dy}{dt} = f(t, y) \tag{5.1}$$

As a physical analogy, I find it useful to think of y as representing the position of a particle (or whatever) and t as time. Then the derivative $\frac{dy}{dt}$ is the velocity of the particle.

Under this interpretation, the equation applies in any situation where we can compute the current velocity as a function of the current position and time.

There may be many solutions to equations like this, but if we know the starting point of the particle, called the **initial condition** then there is exactly one solution (as long as f is well-behaved).

For example, a falling object accelerates constantly at $g = 9.8m/s^2$, if it is near the surface of the Earth and air resistance is insignificant. If the object is released at time $t = 0$, then it's velocity at time t is

$$\frac{dy}{dt} = gt \tag{5.2}$$

If we know the initial height of the object, h, then we can compute $y(t)$, which is the height as a function of time. Another way to write the initial condition is $y(0) = h$. In this case, the velocity only depends on time and not on position.

For the coffee cup problem, the "position" is the current temperature and the "velocity" is the rate of cooling. The equation is

$$\frac{dy}{dt} = -r(y - T_s) \tag{5.3}$$

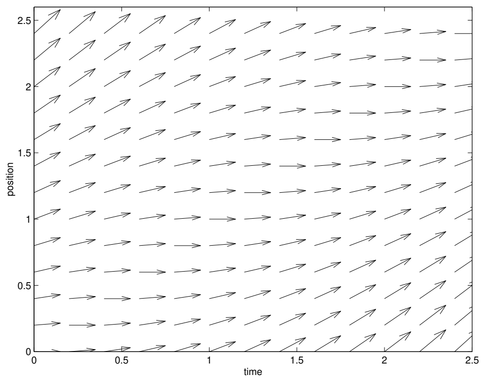
where y is the current temperature and T_s is the temperature of the surroundings. In this case the velocity depends only on position (temperature) and not on time.

Again, to compute the temperature of the coffee over time, we need to know the starting temperature, for example $y(0) = 82.3$.

To visualize the movement of the particle (or the temperature of the coffee) over time, it is useful to make an arrow plot (cleverly called a "quiver" in MATLAB). For example, in the equation

$$\frac{dy}{dt} = |y - t| \tag{5.4}$$

the velocity depends on the difference between t and y (the $|$ symbols mean "absolute value"). The arrow plot for this equation looks like this.



Along the diagonal where $x = t$, the value of the derivative is 0, so the arrow points to the right, meaning that as time moves forward, the position doesn't change. The larger the value of the derivative, the closer to vertical the arrows are.

By following the flow of the arrows from left to right, it is easy to see how the position of the particle varies with time, depending on the starting place.

5.10 MATLAB implementation

Let's get back to the first equation, for the position of an accelerating object. To apply Euler's method to this equation, it is natural to write a subfunction to compute $\frac{dy}{dt}$.

```
function res = velocity (g, t, y)
    res = g * t;
```

For the sake of generality, I have included both `t` and `y` as input variable of `velocity`, although in this case the velocity ($\frac{dy}{dt}$) is a function only of time and not position.

The code to implement Euler's method looks like this:

```
function euler (dt, te)
% euler: compute the position of a falling object as
% a function of time.
% dt is the time step in seconds.
% te is the end time in seconds.

% g is the acceleration of gravity in meters per second^2
% n is the number of time steps
% t is time in seconds
% y is the distance of the object from the origin in meters

    g  = -9.8;
    n  = te / dt;
    t = 0;
    y = 0;

    for i = 0:n-1
        y = y + velocity (g, t, y) * dt;
        t = t + dt;

        fprintf ('%f  %f\n', t, y)
    end
end
```

The comments are longer than the code! The constant g is a parameter of the problem—on Earth the acceleration of gravity is about 9.8 meters per second^2. Throughout the code, time is measured in seconds and distance in meters. Specifying the units for variables like this is very important. Since MATLAB doesn't provide support for carrying units along with values, it is important to include this information in the comments. Failure to document units carefully has been the cause of some spectacular software failures, including the complete loss of a $236 million dollar Mars probe in 1999.

The loop that performs the computation is straightforward. Each time through the loop it updates the values of y and t and prints the new values. One possible source of confusion is the order of the updates. We have to compute the new value of y using the *old* value of t. At least, that's what we have to do if we want to call the algorithm "Euler's method." In fact, is would not be unreasonable to use the new value of t instead. But that's a different algorithm.

5.11 Accumulating an array

When you compute a series of values it is often useful to collect all the results in an array so that you can, for example, plot the array at the end.

For example, the following loop computes the first n numbers in a Fibonacci series and stores them in an array:

```
function a = fibonacci3 (n)
    a = [1 1];
    for i = 2:n
        term = a(length(a)) + a(length(a)-1);
        a = [a term];
    end
end
```

We can call this function to print the first 10 numbers in the series.

```
>> fibonacci(10)

ans =   1      1      2      3      5      8     13     21     34     55     89
```

Also, since the result is an array, we can plot the result to see a representation of how the series grows. The statement `plot (fibonacci(10))` yields this plot: