

# Chapter 2

## Graham–Schmidt Orthogonalization

In our work, we need to talk about vectors and functions using more advanced ideas. In this chapter, we begin by discussing how to integrate functions numerically with MatLab so that we can calculate the inner product of two functions in code. In the second text, we introduced the ideas of vector spaces, linear dependence and independence of vectors. Now we apply these ideas to vector spaces of functions and finish with the Graham–Schmidt Orthogonalization process.

### 2.1 Numerical Integration

We now discuss how to approximate  $\int_a^b f(x)dx$  on the interval  $[a, b]$ . These methods generate the **Newton–Cotes Formulae**. Consider the problem of integrating the function  $f$  on the finite interval  $[a, b]$ . Let's assume that the interval  $[a, b]$  has been subdivided into points  $\{x_1, \dots, x_m\}$  which are uniformly spaced; we let the function values at these points be given by  $f_i = f(x_i)$ . This gives

$$\begin{aligned}x_i &= a + \frac{i-1}{m-1}(b-a) \\x_1 &= a \\x_2 &= a + 1 \times h, \text{ where } h = \frac{m-1}{b-a} \\\dots &= \dots \\x_m &= a + (m-1) \times h\end{aligned}$$

The functions we use to build the interpolating polynomial for the  $m$  points  $x_1$  to  $x_n$  have the form

$$\begin{aligned}p_0(x) &= 1 \\p_j(x) &= \prod_{i=1}^j (x - x_i), \quad 1 \leq j \leq m.\end{aligned}$$

Thus,

$$\begin{aligned}
 p_0(x) &= 1 \\
 p_1(x) &= (x - x_1) \\
 p_2(x) &= (x - x_1)(x - x_2) \\
 p_3(x) &= (x - x_1)(x - x_2)(x - x_3) \\
 &\vdots \\
 p_{m-1}(x) &= (x - x_1)(x - x_2) \cdots (x - x_{m-1})
 \end{aligned}$$

The Newton Interpolating polynomial to  $f$  for this partition is given by the polynomial of degree  $m - 1$

$$\begin{aligned}
 P_{m-1}(x) &= c_1 p_0(x) + c_2 p_1(x) + c_3 p_2(x) + \cdots + c_m p_{m-1}(x) \\
 &= 1 + c_2 p_1(x) + c_3 p_2(x) + \cdots + c_m p_{m-1}(x)
 \end{aligned}$$

where the numbers  $c_1$  through  $c_m$  are chosen so that  $P_{m-1}(x_j) = f(x_j)$ . For convenience, let's look at a four point uniform partition of  $[a, b]$ ,  $\{x_1, x_2, x_3, x_4\}$  and figure out how to find these numbers  $c_j$ . Hence the uniform step size here is  $h = \frac{b-a}{3}$  here. The Newton interpolating polynomial in this case is given by

$$p_3(x) = c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + c_4(x - x_1)(x - x_2)(x - x_3)$$

Since we want  $p_3(x_j) = f(x_j) = f_j$ , we have

$$\begin{aligned}
 p_3(x_1) &= f_1 = c_1 \\
 p_3(x_2) &= f_2 = c_1 + c_2(x_2 - x_1) \\
 p_3(x_3) &= f_3 = c_1 + c_2(x_3 - x_1) + c_3(x_3 - x_1)(x_3 - x_2) \\
 p_3(x_4) &= f_4 = c_1 + c_2(x_4 - x_1) + c_3(x_4 - x_1)(x_4 - x_2) \\
 &\quad + c_4(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)
 \end{aligned}$$

For convenience, let  $x_{ij} = x_i - x_j$ . Then we can rewrite the above as

$$\begin{aligned}
 f_1 &= c_1 \\
 f_2 &= c_1 + c_2 x_{21} \\
 f_3 &= c_1 + c_2 x_{31} + c_3 x_{31} x_{32} \\
 f_4 &= c_1 + c_2 x_{41} + c_3 x_{41} x_{42} + c_4 x_{41} x_{42} x_{43}
 \end{aligned}$$

This is the system of equations which is in lower triangular form and so it is easy to solve.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & x_{21} & 0 & 0 \\ 1 & x_{31} & x_{31}x_{32} & 0 \\ 1 & x_{41} & x_{41}x_{42} & x_{41}x_{42}x_{43} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix}$$

We find  $c_1 = f_1$ . Then,  $c_1 + c_2x_{21} = f_2$  and letting  $f_{ij} = f_i - f_j$ , we have  $c_2 = f_{21}/x_{21}$ . Next, we have

$$c_1 + c_2x_{31} + c_3x_{31}x_{32} = f_3$$

or

$$\begin{aligned} \left(f_{21}/x_{21}\right)x_{31} + c_3x_{31}x_{32} &= f_{31} \\ c_3x_{31}x_{32} &= f_{31} - f_{21}/x_{21} \\ c_3 &= \left(f_{31}/x_{31} - f_{21}/x_{21}\right)/x_{32}. \end{aligned}$$

The final calculation is the worst. Again, for convenience, let  $g_{ij} = f_{ij}/x_{ij}$ . Then we can rewrite  $c_3$  as

$$c_3 = \left(g_{31} - g_{21}\right)/x_{32}.$$

We know

$$c_1 + c_2x_{41} + c_3x_{41}x_{42} + c_4x_{41}x_{42}x_{43} = f_4.$$

Solving for  $x_{41}x_{42}x_{43}c_4$ , we obtain

$$\begin{aligned} c_4x_{41}x_{42}x_{43} &= f_4 - c_1 - c_2x_{41} - c_3x_{41}x_{42} \\ &= f_{41} - \left(f_{21}/x_{21}\right)x_{41} - \left(\left(g_{31} - g_{21}\right)/x_{32}\right)x_{41}x_{42} \end{aligned}$$

Now divide through by  $x_{41}$  to find

$$\begin{aligned} c_4x_{42}x_{43} &= \left(f_{41}/x_{41}\right) - \left(f_{21}/x_{21}\right) - \left(\left(g_{31} - g_{21}\right)/x_{32}\right)x_{42} \\ &= g_{41} - g_{21} - \left(\left(g_{31} - g_{21}\right)/x_{32}\right)x_{42}. \end{aligned}$$

Now divide by  $x_{42}$  to solve for  $x_{43}c_4$ :

$$c_4x_{43} = \left( g_{41} - g_{21} \right) / x_{42} - \left( g_{31} - g_{21} \right) / x_{32}.$$

which tells us (finally!) that

$$c_4 = \left( \left( g_{41} - g_{21} \right) / x_{42} - \left( g_{31} - g_{21} \right) / x_{32} \right) / x_{43}.$$

Since all the differences  $x_{ij} = h$ , the uniform step size, these formulae can be then be rewritten.

$$\begin{aligned} c_1 &= f_1 \\ c_2 &= \frac{f_2 - f_1}{h} \\ c_3 &= \frac{\frac{f_3 - f_1}{2h} - \frac{f_2 - f_1}{h}}{h} \\ &= \frac{f_3 - 2f_2 + f_1}{2h^2} \\ c_4 &= \frac{f_4 - 3f_3 + 3f_2 - f_1}{6h^3} \end{aligned}$$

Although this is messy to do by hand, it is easy to do in MatLab. First, we set **f** to be the vector of function values  $f(x_1), \dots, f(x_m)$ . If **func** is the function we are using in MatLab for the function  $f$ , this is done with the lines **f = func(x)** where **x** has already been setup with a **linspace** command.

**Listing 2.1:** Implementing the differences in Matlab

```
m = length(x);
for k = 1:m-1;
    f(k+1:m) = ( f(k+1:m) - f(k) ) ./ ( x(k+1:m) - x(k) );
end
c = f;
```

For our example,  $m = 4$  and so when  $k = 1$ , we find **f(2:4) = ( f(2:4) - f(1) ) ./ ( x(2:4) - x(1) )**. This looks forbidding, but remember **f(2:4)** is the vector **f(2)**, **f(3)**, **f(4)**. So **f(2:4) - f(1)** becomes **f(2) - f(1)**, **f(3) - f(1)**, **f(4) - f(1)** or the quantities  $f_{21}, f_{31}, f_{41}$ . Then **x(2:4) - x(1)** is **x(2) - x(1)**, **x(3) - x(1)**, **x(4) - x(1)** or  $x_{21}, x_{31}, x_{41}$ . Hence, when **k = 1**, we find **f(2) = ( f(2) - f(1) ) / ( x(2) - x(1) )** or  $f_{21}/x_{21}$ , **f(3) = ( f(3) - f(1) ) / ( x(3) - x(1) )** or  $f_{31}/x_{31}$  and **f(4) = ( f(4) - f(1) ) / ( x(4) - x(1) )** or  $f_{41}/x_{41}$ . At this point the vector **f** contains the proper value of  $c_1$  in **f(1)**

and the right value of  $c_2$  in the component  $\mathbf{f}(2)$ . The values for  $c_3$  and  $c_4$  are not yet right. The next pass through the loop will set  $c_3$  correctly. When  $\mathbf{k} = 2$ , the MatLab code becomes  $\mathbf{f}(3:4) = (\mathbf{f}(3:4) - \mathbf{f}(2)) ./ (\mathbf{x}(3:4) - \mathbf{x}(2))$  which is short for  $\mathbf{f}(3) = (\mathbf{f}(3) - \mathbf{f}(2)) ./ (\mathbf{x}(3) - \mathbf{x}(2))$  and  $\mathbf{f}(4) = (\mathbf{f}(4) - \mathbf{f}(2)) ./ (\mathbf{x}(4) - \mathbf{x}(2))$ . Now remember what  $\mathbf{f}(2)$ ,  $\mathbf{f}(3)$  are at this point and plug them in. We find  $\mathbf{f}(3)$  has become  $(f_{31}/x_{31} - f_{21}/x_{21})/x_{32}$  or  $(g_{31} - g_{21})/x_{32}$ . This is the proper value  $c_3$  should have. We now have  $\mathbf{f}(4)$  is  $(f_{41}/x_{41} - f_{21}/x_{21})/x_{42}$  or  $(g_{41} - g_{21})/x_{42}$ . Finally, the last pass through the loop uses  $\mathbf{k} = 3$  and results in the line  $\mathbf{f}(4:4) = (\mathbf{f}(4:4) - \mathbf{f}(3)) ./ (\mathbf{x}(4:4) - \mathbf{x}(3))$  which is just  $\mathbf{f}(4) = (\mathbf{f}(4) - \mathbf{f}(3)) ./ (\mathbf{x}(4) - \mathbf{x}(3))$ . Now plug in what we have for  $\mathbf{f}(3)$  to obtain

$$f_4 = \left( \left( g_{41} - g_{21} \right) / x_{42} - \left( g_{31} - g_{21} \right) / x_{32} \right) / x_{43}$$

which is exactly the value that  $c_4$  should be. This careful walk through the code is what we all do when we are trying to see if our ideas actually work. We usually do it on scratch paper to make sure everything is as we expect. Typing it out is much harder!

These polynomials are called **Newton interpolants** and we can use these ideas to approximate  $\int_a^b f(x)dx$  as follows. We approximate  $\int_a^b f(x)ds$  by replacing  $f$  by a Newton interpolant on the interval  $[a, b]$ . For an  $m$  point uniformly spaced partition, we will use the  $m - 1$  degree polynomial  $P_{m-1}$  whose coefficients can be computed recursively like we do above. We replace the integrand  $f$  by  $P_{m-1}$  to find

$$\begin{aligned} \int_a^b f(x)dx &\approx \int_a^b P_{m-1}(x)dx \\ &= \int_a^b \left( c_1 + \left( \sum_{k=1}^{m-1} c_{k+1} \Pi_{i=1}^k (x - x_i) \right) \right) dx \\ &= c_1(b-a) + \sum_{k=1}^{m-1} c_{k+1} \left( \int_a^b \Pi_{i=1}^k (x - x_i) \right) dx \end{aligned}$$

To figure out this approximation, we need to evaluate

$$\int_a^b \Pi_{i=1}^k (x - x_i) dx$$

Make the change of variable  $s$  defined by  $x = a + sh$ . Then at  $x = a$ ,  $s = 0$  and at  $x = b$ ,  $s = (b-a)/h = m-1$  ( $m$  points uniformly spaced means the common interval is  $(b-a)/m$ ). Further,  $x - x_i$  becomes  $(s - i + 1)h$ . Thus, we have

$$\int_a^b \Pi_{i=1}^k (x - x_i) dx = \int_0^{m-1} h^{k+1} \Pi_{i=1}^k (s - i + 1) ds$$

We will let

$$S_{m1} = m - 1$$

$$S_{m,k+1} = \int_0^{m-1} \prod_{i=1}^k (s - i + 1) ds, \quad 1 \leq k \leq m - 1$$

Next, note  $c_1(b - a) = c_1 h((b - a)/h)$  and since  $(b - a)/h = m - 1$ , we have  $c_1(b - a) = c_1 h(m - 1)$ , which gives us the final form of our approximation:

$$\begin{aligned} \int_a^b f(x) dx &\approx c_1(b - a) + \sum_{k=1}^{m-1} c_{k+1} h^{k+1} S_{m,k+1} \\ &= c_1 h(m - 1) + \sum_{k=1}^{m-1} c_{k+1} h^{k+1} S_{m,k+1} \end{aligned}$$

We can rewrite the sum above letting  $n = k + 1$  to

$$\int_a^b f(x) dx \approx c_1 h(m - 1) + \sum_{n=2}^m c_n h^n S_{mn} = \sum_{k=1}^m c_k h^k S_{mk}.$$

### 2.1.1 Evaluating $S_{mk}$

The value of  $m$  we choose to use gives rise then to what is called an  $m$  point rule and given  $m$  we can easily evaluate the needed coefficients  $S_{m1}$  through  $S_{mm}$ . Here are some calculations:

$$\begin{aligned} S_{m2} &= \int_0^{m-1} s ds \\ &= \frac{(m - 1)^2}{2} \\ S_{m3} &= \int_0^{m-1} s(s - 1) ds \\ &= \frac{(m - 1)^2}{6} (2m - 5) \\ S_{m4} &= \int_0^{m-1} s(s - 1)(s - 2) ds \\ &= \frac{(m - 1)^2 (m - 3)^2}{4} \end{aligned}$$

We will denote this Newton Polynomial approximation to this integral using  $m$  points by the symbol  $Q_{NC(m)}$ . Hence, for the case  $m = 4$ , we have

$$\begin{aligned} S_{41} &= 3 \\ S_{42} &= \frac{9}{2} \\ S_{43} &= \frac{9}{2} \\ S_{44} &= \frac{9}{4} \end{aligned}$$

This leads to the approximation

$$\begin{aligned} \int_a^b f(x)dx &\approx c_1 h S_{41} + c_2 h^2 S_{42} + c_3 h^3 S_{43} + c_4 h^4 S_{44} \\ &= 3f_1 h + \frac{9}{2} \frac{f_2 - f_1}{h} h^2 + \frac{9}{2} \frac{f_3 - 2f_2 + f_1}{h^2} h^3 + \frac{9}{4} \frac{f_4 - 3f_3 + 3f_2 - f_1}{6h^3} h^4 \\ &= 3f_1 h + \frac{9}{2} (f_2 - f_1) h + \frac{9}{4} (f_3 - 2f_2 + f_1) h + \frac{9}{24} (f_4 - 3f_3 + 3f_2 - f_1) h \\ &= 3f_1 h + \frac{9}{2} (f_2 - f_1) h + \frac{9}{4} (f_3 - 2f_2 + f_1) h + \frac{3}{8} (f_4 - 3f_3 + 3f_2 - f_1) h \\ &= \frac{3h}{8} (8f_1 + 12f_2 - 12f_1 + 6f_3 - 12f_2 + 6f_1 + f_4 - 3f_3 + 3f_2 - f_1) \\ &= \frac{3h}{8} (f_1 + 3f_2 + 3f_3 + f_4) \end{aligned}$$

But  $h$  is  $\frac{b-a}{3}$  here, so our final 4 point formula is

$$\int_a^b f(x)dx \approx \frac{b-a}{8} (f_1 + 3f_2 + 3f_3 + f_4)$$

### 2.1.2 Homework

**Exercise 2.1.1** Show that for  $m = 2$ , we get the **Trapezoidal Rule**

$$\int_a^b f(x)dx \approx (b-a) \left( \frac{1}{2}f_1 + \frac{1}{2}f_2 \right)$$

**Exercise 2.1.2** Show that for  $m = 3$ , we get the **Simpson Rule**

$$\int_a^b f(x)dx \approx \frac{b-a}{6} (f_1 + 4f_2 + f_3)$$

### 2.1.3 Matlab Implementation

We store the Newton–Cotes Weight vectors in this short piece of Matlab code

**Listing 2.2:** Storing Newton–Cotes Weight Vectors

```
function w = WNC(m)
%
% m   an integer from 2 to 5
% w   this is the Newton Cotes Weight Vector
5 %
  if m==2
    w = [1 1]'/2;
  elseif m==3
    w = [1 4 1]'/6;
10 elseif m==4
    w = [1 3 3 1]'/8;
  elseif m==5
    w = [7 32 12 32 7]'/90;
  elseif m==6
15 w = [19 75 50 50 75 19]'/288;
  elseif m==7
    w = [41 216 27 272 27 216 41]'/840;
  elseif m==8
    w = [751 3577 1323 2989 2989 1323 3577 751]'/17280;
20 elseif m==9
    w = [989 5888 -928 10496 -4540 10496 -928 5888 989]'/28350;
  else
    disp('You must use a value of m between 2 and 9');
    w = [0 0]';
25 end
```

The Newton–Cotes implementation is then given by

**Listing 2.3:** Newton–Cotes Implementation

```
function numI = QNC(fname,a,b,m)
%
% fname the name of the function which is to be integrated
% a,b   the integration interval is [a,b]
5 % m    the order of the Newton–Cotes Method to use
%       this is the same as the number of points in
%       the partition of [a,b]
%
% numI  the value of the Newton–Cotes approximation to
10 %     the integral
%
  if m >= 2 & m <= 9
    w = WNC(m);
    x = linspace(a,b,m)';
15 f = feval(fname,x);
    numI = (b-a)*(w'*f);
  else
    disp('You need to use an order m between 2 and 9');
  end
```

and finally, a script to run the numerical integration routines is as follows:



**Listing 2.4:** Numerical Integration Script

```

1 while input('Another Example? (1=yes, 0=no). ');
    fname = input('Enter function name: ');
    a = input('Enter left endpoint ');
    b = input('Enter right endpoint: ');
    s = [ 'QNC(' fname sprintf(',%6.3f,%6.3f,m )',a,b) ];
6    disp([' m      ' s])
    disp(' ')
    for m = 2:9
        numI = QNC(fname,a,b,m);
        disp(sprintf(' %2.0f      %20.16f ',m,numI))
11    end
end

```

**2.1.4 Run Time Output**

We use the functions defined in **func1.m**,  $\left(f(x) = \sin(x)\right)$ , and **func2.m**,  $\left(f(x) = e^{-x^2}\right)$ , which have matlab codes

**Listing 2.5:** Integrand Function I

```

function y = func1(x)
y = sin(x);

```

**Listing 2.6:** Integrand Function II

```

function y = func2(u)
%
3 z = -u.*u;
y = exp(z);

```

Here is our runtime. The first example is  $\int_0^\pi \sin(x)dx = 1$  and the second is  $\int_0^2 e^{-x^2}dx$  which can only be evaluated numerically and has the value approximately 0.882. You can see this by using the built in MatLab numerical integration function **quad**. The following session computes the integral using **quad**: here, we use the anonymous function **h** to pass into **quad**.

**Listing 2.7:** Using quad

```

1 h = @(x) exp(-x.^2);
c = quad(h,0,2)
c = 0.8821

```

We then compare this to the Newton–Cotes methods in the session below.

**Listing 2.8:** A Newton–Cotes session

```

ShowNC
2 Another Example? (1=yes, 0=no). 1
Enter function name: 'func1'
Enter left endpoint 0
Enter right endpoint: pi/2
m      QNC(func1, 0.000, 1.571,m )

7
2      0.7853981633974483
3      1.0022798774922104
4      1.0010049233142790
5      0.9999915654729927
12 6      0.9999952613861668
7      1.0000000258372352
8      1.0000000158229039
9      0.999999999408976
Another Example? (1=yes, 0=no). 1
17 Enter function name: 'func2'
Enter left endpoint 0
Enter right endpoint: 2.0
m      QNC(func2, 0.000, 2.000,m )

22 2      1.0183156388887342
3      0.8299444678581678
4      0.8622241875991991
5      0.8852702891231793
6      0.8838030970892903
27 7      0.8819161924221999
8      0.8819818734329694
9      0.8820864256236417
Another Example? (1=yes, 0=no). 0

```

## 2.2 Linearly Independent Functions

The ideas of linear independence and dependence are hard to grasp, so even though we went through these ideas in Peterson (2015), it is a good idea to do a repeat performance. So let's go back and think about vectors in  $\mathfrak{R}^2$ . As you know, we think of these as arrows with a tail fixed at the origin of the two dimensional coordinate system we call the  $x$ - $y$  plane. They also have a length or magnitude and this arrow makes an angle with the positive  $x$  axis. Suppose we look at two such vectors,  $\mathbf{E}$  and  $\mathbf{F}$ . Each vector has an  $x$  and a  $y$  component so that we can write

$$\mathbf{E} = \begin{bmatrix} a \\ b \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} c \\ d \end{bmatrix}$$

The cosine of the angle between them is proportional to the inner product  $\langle \mathbf{E}, \mathbf{F} \rangle = ac + bd$ . If this angle is 0 or  $\pi$ , the two vectors lie along the same line. In any case, the angle associated with  $\mathbf{E}$  is  $\tan^{-1}(\frac{b}{a})$  and for  $\mathbf{F}$ ,  $\tan^{-1}(\frac{d}{c})$ . Hence, if the two vectors lie on the same line,  $\mathbf{E}$  must be a multiple of  $\mathbf{F}$ . This means there is a number  $\beta$  so that

$$\mathbf{E} = \beta \mathbf{F}.$$

We can rewrite this as

$$\begin{bmatrix} a \\ b \end{bmatrix} = \beta \begin{bmatrix} c \\ d \end{bmatrix}$$

Now let the number 1 in front of  $\mathbf{E}$  be called  $-\alpha$ . Then the fact that  $\mathbf{E}$  and  $\mathbf{F}$  lie on the same line implies there are 2 constants  $\alpha$  and  $\beta$ , both not zero, so that

$$\alpha \mathbf{E} + \beta \mathbf{F} = 0.$$

Note we could argue this way for vectors in  $\mathbb{R}^3$  and even in  $\mathbb{R}^n$ . Of course, our ability to think of these things in terms of lying on the same line and so forth needs to be extended to situations we can no longer draw, but the idea is essentially the same. Instead of thinking of our two vectors as lying on the same line or not, we can *rethink* what is happening here and try to identify what is happening in a more abstract way. If our two vectors lie on the same line, they are not *independent* things in the sense one is a multiple of the other. As we saw above, this implies there was a linear equation connecting the two vectors which had to add up to 0. Hence, we might say the vectors were *not linearly independent* or simply, they are *linearly dependent*. Phrased this way, we are on to a way of stating this idea which can be used in many more situations. We state this as a definition.

**Definition 2.2.1** (*Two Linearly Independent Objects*)

Let  $\mathbf{E}$  and  $\mathbf{F}$  be two mathematical objects for which addition and scalar multiplication is defined. We say  $\mathbf{E}$  and  $\mathbf{F}$  are **linearly dependent** if we can find non zero constants  $\alpha$  and  $\beta$  so that

$$\alpha \mathbf{E} + \beta \mathbf{F} = 0.$$

Otherwise, we say they are **linearly independent**.

We can then easily extend this idea to any finite collection of such objects as follows.

**Definition 2.2.2** (*Finitely many Linearly Independent Objects*)

Let  $\{\mathbf{E}_i : 1 \leq i \leq N\}$  be  $N$  mathematical objects for which addition and scalar multiplication is defined. We say  $\mathbf{E}$  and  $\mathbf{F}$  are **linearly dependent** if we can find non zero constants  $\alpha_1$  to  $\alpha_N$ , not all 0, so that

$$\alpha_1 \mathbf{E}_1 + \cdots + \alpha_N \mathbf{E}_N = 0.$$

Note we have changed the way we define the constants a bit. When there are more than two objects involved, we can't say, in general, that *all* of the constants must be non zero.

### 2.2.1 Functions

We can apply these ideas to functions  $f$  and  $g$  defined on some interval  $I$ . By this we mean either

- $I$  is all of  $\mathfrak{R}$ , i.e.  $a = -\infty$  and  $b = \infty$ ,
- $I$  is half-infinite. This means  $a = -\infty$  and  $b$  is finite with  $I$  of the form  $(-\infty, b)$  or  $(-\infty, b]$ . Similarly,  $I$  could have the form  $(a, \infty)$  or  $[a, \infty)$ ,
- $I$  is an interval of the form  $(a, b)$ ,  $[a, b)$ ,  $(a, b]$  or  $[a, b]$  for finite  $a < b$ .

We would say  $f$  and  $g$  are linearly independent on the interval  $I$  if the equation

$$\alpha_1 f(t) + \alpha_2 g(t) = 0, \text{ for all } t \in I.$$

implies  $\alpha_1$  and  $\alpha_2$  must both be zero. Here is an example. The functions  $\sin(t)$  and  $\cos(t)$  are linearly independent on  $\mathfrak{R}$  because

$$\alpha_1 \cos(t) + \alpha_2 \sin(t) = 0, \text{ for all } t,$$

also implies the above equation holds for the derivative of both sides giving

$$-\alpha_1 \sin(t) + \alpha_2 \cos(t) = 0, \text{ for all } t,$$

This can be written as the system

$$\begin{bmatrix} \cos(t) & \sin(t) \\ -\sin(t) & \cos(t) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

for all  $t$ . The determinant of the matrix here is  $\cos^2(t) + \sin^2(t) = 1$  and so picking any  $t$  we like, we find the unique solution is  $\alpha_1 = \alpha_2 = 0$ . Hence, these two functions are linearly independent on  $\mathfrak{R}$ . In fact, they are linearly independent on any interval  $I$ .

This leads to another important idea. Suppose  $f$  and  $g$  are linearly independent differentiable functions on an interval  $I$ . Then, we know the system

$$\begin{bmatrix} f(t) & g(t) \\ f'(t) & g'(t) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

only has the unique solution  $\alpha_1 = \alpha_2 = 0$  for all  $t$  in  $I$ . This tells us

$$\det \left( \begin{bmatrix} f(t) & g(t) \\ f'(t) & g'(t) \end{bmatrix} \right) \neq 0$$

for all  $t$  in  $I$ . This determinant comes up a lot and it is called the **Wronskian** of the two functions  $f$  and  $g$  and it is denoted by the symbol  $W(f, g)$ . Hence, we have the implication: if  $f$  and  $g$  are linearly independent differentiable functions, then

$W(f, g) \neq 0$  for all  $t$  in  $I$ . What about the converse? If the Wronskian is never zero on  $I$ , then the system

$$\begin{bmatrix} f(t) & g(t) \\ f'(t) & g'(t) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

must have the unique solution  $\alpha_1 = \alpha_2 = 0$  at each  $t$  in  $I$  also. So the converse is true: if the Wronskian is not zero on  $I$ , then the differentiable functions  $f$  and  $g$  are linearly independent on  $I$ . We can state this formally as a theorem.

**Theorem 2.2.1** (Two Functions are Linearly Independent if and only if their Wronskian is not zero)

*If  $f$  and  $g$  are differentiable functions on  $I$ , the **Wronskian** of  $f$  and  $g$  is defined to be*

$$W(f, g) = \det \left( \begin{bmatrix} f(t) & g(t) \\ f'(t) & g'(t) \end{bmatrix} \right).$$

*where  $W(f, g)$  is the symbol for the Wronskian of  $f$  and  $g$ . Sometimes, this is just written as  $W$ , if the context is clear. Then  $f$  and  $g$  are linearly independent on  $I$  if and only if  $W(f, g)$  is non zero on  $I$ .*

*Proof* See the discussions above. ■

If  $f, g$  and  $h$  are twice differentiable on  $I$ , the Wronskian uses a third row of second derivatives and the statement that these three functions are linearly independent on  $I$  if and only if their Wronskian is non zero on  $I$  is proved essentially the same way. The appropriate theorem is

**Theorem 2.2.2** (Three Functions are Linearly Independent if and only if their Wronskian is not zero)

*If  $f, g$  and  $h$  are twice differentiable functions on  $I$ , the **Wronskian** of  $f, g$  and  $h$  is defined to be*

$$W(f, g, h) = \det \left( \begin{bmatrix} f(t) & g(t) & h(t) \\ f'(t) & g'(t) & h'(t) \\ f''(t) & g''(t) & h''(t) \end{bmatrix} \right).$$

*where  $W(f, g, h)$  is the symbol for the Wronskian of  $f$  and  $g$ . Then  $f, g$  and  $h$  are linearly independent on  $I$  if and only if  $W(f, g, h)$  is non zero on  $I$ .*

*Proof* The arguments are similar, although messier. ■

For example, to show the three functions  $f(t) = t$ ,  $g(t) = \sin(t)$  and  $h(t) = e^{2t}$  are linearly independent on  $\mathbb{R}$ , we could form their Wronskian

$$\begin{aligned}
W(f, g, h) &= \det \begin{pmatrix} t & \sin(t) & e^{2t} \\ 1 & \cos(t) & 2e^{2t} \\ 0 & -\sin(t) & 4e^{2t} \end{pmatrix} = t \begin{bmatrix} \cos(t) & 2e^{2t} \\ -\sin(t) & 4e^{2t} \end{bmatrix} - \begin{bmatrix} \sin(t) & e^{2t} \\ -\sin(t) & 4e^{2t} \end{bmatrix} \\
&= t \left( e^{2t} (4 \cos(t) + 2 \sin(t)) \right) - \left( e^{2t} (4 \sin(t) + \sin(t)) \right) \\
&= e^{2t} \left( 4t \cos(t) + 2t \sin(t) - 5 \sin(t) \right).
\end{aligned}$$

Since,  $e^{2t}$  is never zero, the question becomes is

$$4t \cos(t) + 2t \sin(t) - 5 \sin(t)$$

zero for all  $t$ ? If so, that would mean the functions  $t \sin(t)$ ,  $t \cos(t)$  and  $\sin(t)$  are linearly dependent. We could then form another Wronskian for these functions which would be rather messy. To see these three new functions are linearly independent, it is easier to just pick *three* points  $t$  from  $\Re$  and solve the resulting linearly dependence equations. Since  $t = 0$  does not give any information, let's try  $t = -\pi$ ,  $t = \frac{\pi}{4}$  and  $t = \frac{\pi}{2}$ . This gives the system

$$\begin{bmatrix} -4\pi & 0 & 0 \\ \pi & 2\frac{\pi}{4}\frac{\sqrt{2}}{2} & -5\frac{\sqrt{2}}{2} \\ 0 & 2\frac{\pi}{2} & -5 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

in the unknowns  $\alpha_1$ ,  $\alpha_2$  and  $\alpha_3$ . We see immediately  $\alpha_1 - 1 = 0$  and the remaining two by two system has determinant  $\frac{\sqrt{2}}{2}(-10\frac{\pi}{4}) + 10\frac{\pi}{2} \neq 0$ . Hence,  $\alpha_2 = \alpha_3 = 0$  too. This shows  $t \sin(t)$ ,  $t \cos(t)$  and  $\sin(t)$  are linearly independent and show the line  $4t \cos(t) + 2t \sin(t) - 5 \sin(t)$  is not zero for all  $t$ . Hence, the functions  $f(t) = t$ ,  $g(t) = \sin(t)$  and  $h(t) = e^{2t}$  are linearly independent. As you can see, these calculations become messy quickly. Usually, the Wronskian approach for more than two functions is too hard and we use the *pick three suitable points  $t_i$ , from  $I$*  approach and solve the resulting linear system. If we can show the solution is always 0, then the functions are linearly independent.

### 2.2.2 Homework

**Exercise 2.2.1** Prove  $e^t$  and  $e^{-t}$  are linearly independent on  $\Re$ .

**Exercise 2.2.2** Prove  $e^t$  and  $e^{2t}$  are linearly independent on  $\Re$ .

**Exercise 2.2.3** Prove  $f(t) = 1$  and  $g(t) = t^2$  are linearly independent on  $\Re$ .

**Exercise 2.2.4** Prove  $e^t$ ,  $e^{2t}$  and  $e^{3t}$  are linearly independent on  $\Re$ . Use the *pick three points* approach here.

**Exercise 2.2.5** Prove  $\sin(t)$ ,  $\sin(2t)$  and  $\sin(3t)$  are linearly independent on  $\mathbb{R}$ . Use the pick three points approach here.

**Exercise 2.2.6** Prove  $1$ ,  $t$  and  $t^2$  are linearly independent on  $\mathbb{R}$ . Use the pick three points approach here.

## 2.3 Vector Spaces and Basis

We can make the ideas we have been talking about more formal. If we have a set of objects  $\mathbf{u}$  with a way to add them to create new objects in the set and a way to *scale* them to make new objects, this is formally called a **Vector Space** with the set denoted by  $\mathcal{V}$ . For our purposes, we scale such objects with either real or complex numbers. If the scalars are real numbers, we say  $\mathbf{V}$  is a vector space over the reals; otherwise, it is a vector space over the complex field.

### Definition 2.3.1 (Vector Space)

Let  $\mathcal{V}$  be a set of objects  $\mathbf{u}$  with an additive operation  $\oplus$  and a scaling method  $\odot$ . Formally, this means

1. Given any  $\mathbf{u}$  and  $\mathbf{v}$ , the operation of adding them together is written  $\mathbf{u} \oplus \mathbf{v}$  and results in the creation of a new object  $\mathbf{w}$  in the vector space. This operation is *commutative* which means the order of the operation is not important. Also, this operation is associative as we can group any two objects together first, perform this addition  $\oplus$  and then do the others and the order of the grouping does not matter.
2. Given any  $\mathbf{u}$  and any number  $c$  (either real or complex, depending on the type of vector space we have), the operation  $c \odot \mathbf{u}$  creates a new object. We call such numbers *scalars*.
3. The scaling and additive operations are compatible in the sense that they satisfy the *distributive* laws for scaling and addition.
4. There is a special object called  $\mathbf{o}$  which functions as a *zero* so we always have  $\mathbf{o} \oplus \mathbf{u} = \mathbf{u} \oplus \mathbf{o} = \mathbf{u}$ .
5. There are *additive inverses* which means to each  $\mathbf{u}$  there is a unique object  $\mathbf{u}^\dagger$  so that  $\mathbf{u} \oplus \mathbf{u}^\dagger = \mathbf{o}$ .

**Comment 2.3.1** These laws imply

$$(0 + 0) \odot \mathbf{u} = (0 \odot \mathbf{u}) \oplus (0 \odot \mathbf{u})$$

which tells us  $0 \odot \mathbf{u} = 0$ . A little further thought then tells us that since

$$\begin{aligned} \mathbf{0} &= (1 - 1) \odot \mathbf{u} \\ &= (1 \odot \mathbf{u}) \oplus (-1 \odot \mathbf{u}) \end{aligned}$$

we have the additive inverse  $\mathbf{u}^\dagger = -1 \odot \mathbf{u}$ .

**Comment 2.3.2** *We usually say this much simpler. The set of objects  $\mathcal{V}$  is a vector space over its scalar field if there are two operations which we denote by  $\mathbf{u} + \mathbf{v}$  and  $c\mathbf{u}$  which generate new objects in the vector space for any  $\mathbf{u}, \mathbf{v}$  and scalar  $c$ . We then just add that these operations satisfy the usual commutative, associative and distributive laws and there are unique additive inverses.*

**Comment 2.3.3** *The objects are often called vectors and sometimes we denote them by  $\mathbf{u}$  although this notation is often too cumbersome.*

**Comment 2.3.4** *To give examples of vector spaces, it is usually enough to specify how the additive and scaling operations are done.*

- Vectors in  $\mathbb{R}^2, \mathbb{R}^3$  and so forth are added and scaled by components.
- Matrices of the same size are added and scaled by components.
- A set of functions of similar characteristics uses as its additive operator, pointwise addition. The new function  $(f \oplus g)$  is defined pointwise by  $(f \oplus g)(t) = f(t) + g(t)$ . Similarly, the new function  $c \odot f$  is defined by  $c \odot f$  is the function whose value at  $t$  is  $(cf)(t) = cf(t)$ . Classic examples are
  1.  $C[a, b]$  is the set of all functions whose domain is  $[a, b]$  that are continuous on the domain.
  2.  $C^1[a, b]$  is the set of all functions whose domain is  $[a, b]$  that are continuously differentiable on the domain.
  3.  $R[a, b]$  is the set of all functions whose domain is  $[a, b]$  that are Riemann integrable on the domain.

*There are many more, of course.*

Vector spaces have two other important ideas associated with them. We have already talked about linearly independent objects. Clearly, the kinds of objects we were focusing on were from some vector space  $\mathcal{V}$ . The first idea is that of the span of a set.

**Definition 2.3.2** *(The Span Of A Set Of Vectors)*

Given a finite set of vectors in a vector space  $\mathcal{V}$ ,  $\mathcal{W} = \{\mathbf{u}_1, \dots, \mathbf{u}_N\}$  for some positive integer  $N$ , the span of  $\mathcal{W}$  is the collection of all new vectors of the form  $\sum_{i=1}^N c_i \mathbf{u}_i$  for any choices of scalars  $c_1, \dots, c_N$ . It is easy to see  $\mathcal{W}$  is a vector space itself and since it is a subset of  $\mathcal{V}$ , we call it a *vector subspace*. The span of the set  $\mathcal{W}$  is denoted by  $Sp\mathcal{W}$ . If the set of vectors  $\mathcal{W}$  is not finite, the definition is similar but we say the span of  $\mathcal{W}$  is the set of all vectors which can be written as  $\sum_{i=1}^N c_i \mathbf{u}_i$  for some finite set of vectors  $\mathbf{u}_1, \dots, \mathbf{u}_N$  from  $\mathcal{W}$ .

Then there is the notion of a *basis* for a vector space. First, we need to extend the idea of linear independence to sets that are not necessarily finite.

**Definition 2.3.3** *(Linear Independence For Non Finite Sets)*

Given a set of vectors in a vector space  $\mathcal{V}$ ,  $\mathcal{W}$ , we say  $\mathcal{W}$  is a linearly independent subset if every finite set of vectors from  $\mathcal{W}$  is linearly independent in the usual manner.



**Definition 2.3.4** (*A Basis For A Vector Space*)

Given a set of vectors in a vector space  $\mathcal{V}$ ,  $\mathcal{W}$ , we say  $\mathcal{W}$  is a *basis* for  $\mathcal{V}$  if the span of  $\mathcal{W}$  is all of  $\mathcal{V}$  and if the vectors in  $\mathcal{W}$  are linearly independent. Hence, a basis is a linearly independent spanning set for  $\mathcal{V}$ . The number of vectors in  $\mathcal{W}$  is called the *dimension* of  $\mathcal{V}$ . If  $\mathcal{W}$  is not finite in size, then we say  $\mathcal{V}$  is an *infinite dimensional vector space*.

**Comment 2.3.5** *In a vector space like  $\mathbb{R}^n$ , the maximum size of a set of linearly independent vectors is  $n$ , the dimension of the vector space.*

**Comment 2.3.6** *Let's look at the vector space  $C[0, 1]$ , the set of all continuous functions on  $[0, 1]$ . Let  $\mathcal{W}$  be the set of all powers of  $t$ ,  $\{1, t, t^2, t^3, \dots\}$ . We can use the derivative technique to show this set is linearly independent even though it is infinite in size. Take any finite subset from  $\mathcal{W}$ . Label the resulting powers as  $\{n_1, n_2, \dots, n_p\}$ . Write down the linear dependence equation*

$$c_1 t^{n_1} + c_2 t^{n_2} + \dots + c_p t^{n_p} = 0.$$

*Take  $n_p$  derivatives to find  $c_p = 0$  and then backtrack to find the other constants are zero also. Hence  $C[0, 1]$  is an infinite dimensional vector space. It is also clear that  $\mathcal{W}$  does not span  $C[0, 1]$  as if this was true, every continuous function on  $[0, 1]$  would be a polynomial of some finite degree. This is not true as  $\sin(t)$ ,  $e^{-2t}$  and many others are not finite degree polynomials.*

## 2.4 Inner Products

Now there is an important result that we use a lot in applied work. If we have an object  $\mathbf{u}$  in a Vector Space  $\mathcal{V}$ , we often want to find to *approximate*  $\mathbf{u}$  using an element from a given subspace  $\mathcal{W}$  of the vector space. To do this, we need to add another property to the vector space. This is the notion of an *inner product*. We already know what an inner product is in a simple vector space like  $\mathbb{R}^n$ . Many vector spaces can have an inner product structure added easily. For example, in  $C[a, b]$ , since each object is continuous, each object is Riemann integrable. Hence, given two functions  $f$  and  $g$  from  $C[a, b]$ , the real number given by  $\int_a^b f(s)g(s)ds$  is well-defined. It satisfies all the usual properties that the inner product for finite dimensional vectors in  $\mathbb{R}^n$  does also. These properties are so common we will codify them into a definition for what an inner product for a vector space  $\mathcal{V}$  should behave like.

**Definition 2.4.1** (*Real Inner Product*)

Let  $\mathcal{V}$  be a vector space with the reals as the scalar field. Then a mapping  $\omega$  which assigns a pair of objects to a real number is called an inner product on  $\mathcal{V}$  if

1.  $\omega(\mathbf{u}, \mathbf{v}) = \omega(\mathbf{v}, \mathbf{u})$ ; that is, the order is not important for any two objects.
2.  $\omega(c \odot \mathbf{u}, \mathbf{v}) = c\omega(\mathbf{u}, \mathbf{v})$ ; that is, scalars in the *first slot* can be pulled out.

3.  $\omega(\mathbf{u} \oplus \mathbf{w}, \mathbf{v}) = \omega(\mathbf{u}, \mathbf{v}) + \omega(\mathbf{w}, \mathbf{v})$ , for any three objects.
4.  $\omega(\mathbf{u}, \mathbf{u}) \geq 0$  and  $\omega(\mathbf{u}, \mathbf{u}) = 0$  if and only if  $\mathbf{u} = 0$ .

These properties imply that  $\omega(\mathbf{u}, c \odot \mathbf{v}) = c\omega(\mathbf{u}, \mathbf{v})$  as well. A vector space  $\mathcal{V}$  with an inner product is called an inner product space.

**Comment 2.4.1** *The inner product is usually denoted with the symbol  $\langle, \rangle$  instead of  $\omega(\cdot, \cdot)$ . We will use this notation from now on.*

**Comment 2.4.2** *When we have an inner product, we can measure the size or magnitude of an object, as follows. We define the analogue of the euclidean norm of an object  $\mathbf{u}$  using the usual  $\|\cdot\|$  symbol as*

$$\|\mathbf{u}\| = \sqrt{\langle \mathbf{u}, \mathbf{u} \rangle}.$$

*This is called the norm induced by the inner product of the object. In  $C[a, b]$ , with the inner product  $\langle f, g \rangle = \int_a^b f(s)g(s)ds$ , the norm of a function  $f$  is thus  $\|f\| = \sqrt{\int_a^b f^2(s)ds}$ . This is called the  $L_2$  norm of  $f$ .*

It is possible to prove the Cauchy–Schwartz inequality in this more general setting also.

**Theorem 2.4.1** (General Cauchy–Schwartz Inequality)

*If  $\mathcal{V}$  is an inner product space with inner product  $\langle, \rangle$  and induced norm  $\|\cdot\|$ , then*

$$|\langle \mathbf{u}, \mathbf{v} \rangle| \leq \|\mathbf{u}\| \|\mathbf{v}\|$$

*with equality occurring if and only if  $\mathbf{u}$  and  $\mathbf{v}$  are linearly dependent.*

*Proof* The proof is different than the one you would see in a Calculus text for  $\mathbb{R}^2$ , of course, and is covered in a typical course on beginning analysis. ■

**Comment 2.4.3** *We can use the Cauchy–Schwartz inequality to define a notion of angle between objects exactly like we would do in  $\mathbb{R}^2$ . We define the angle  $\theta$  between  $\mathbf{u}$  and  $\mathbf{v}$  via its cosine as usual.*

$$\cos(\theta) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\|\mathbf{u}\| \|\mathbf{v}\|}.$$

*Hence, objects can be perpendicular or orthogonal even if we can not interpret them as vectors in  $\mathbb{R}^2$ . We see two objects are orthogonal if their inner product is 0.*

**Comment 2.4.4** *If  $\mathcal{W}$  is a finite dimensional subspace, a basis for  $\mathcal{W}$  is said to be an orthonormal basis if each object in the basis has  $L_2$  norm 1 and all of the objects are mutually orthogonal. This means  $\langle \mathbf{u}_i, \mathbf{u}_j \rangle$  is 1 if  $i = j$  and 0 otherwise. We typically let the Kronecker delta symbol  $\delta_{ij}$  be defined by  $\delta_{ij} = 1$  if  $i = j$  and 0 otherwise so that we can say this more succinctly as  $\langle \mathbf{u}_i, \mathbf{u}_j \rangle = \delta_{ij}$ .*

Now, let's return to the idea of finding the best object in a subspace  $\mathcal{W}$  to approximate a given object  $u$ . This is an easy theorem to prove.

**Theorem 2.4.2** (Best Finite Dimensional Approximation Theorem)

Let  $u$  be any object in the inner product space  $\mathcal{V}$  with inner product  $\langle, \rangle$  and induced norm  $\| \cdot \|$ . Let  $\mathcal{W}$  be a finite dimensional subspace with an orthonormal basis  $\{w_1, \dots, w_N\}$  where  $N$  is the dimension of the subspace. Then there is a unique object  $p^*$  in  $\mathcal{W}$  which satisfies

$$\|u - p^*\| = \min_{p \in \mathcal{W}} \|u - p\|$$

with

$$p^* = \sum_{i=1}^N \langle u, w_i \rangle w_i.$$

Further,  $u - p^*$  is orthogonal to the subspace  $\mathcal{W}$ .

*Proof* Any object in the subspace has the representation  $\sum_{i=1}^N a_i w_i$  for some scalars  $a_i$ . Consider the function of  $N$  variables

$$\begin{aligned} E(a_1, \dots, a_N) &= \left\langle u - \sum_{i=1}^N a_i w_i, u - \sum_{j=1}^N a_j w_j \right\rangle \\ &= \langle u, u \rangle - 2 \sum_{i=1}^N a_i \langle u, w_i \rangle + \sum_{i=1}^N \sum_{j=1}^N a_i a_j \langle w_i, w_j \rangle. \end{aligned}$$

Simplifying using the orthonormality of the basis, we find

$$E(a_1, \dots, a_N) = \langle u, u \rangle - 2 \sum_{i=1}^N a_i \langle u, w_i \rangle + \sum_{i=1}^N a_i^2.$$

This is a quadratic expression and setting the gradient of  $E$  to zero, we find the critical points

$$a_j = \langle u, w_j \rangle.$$

This is a global minimum for the function  $E$ . Hence, the optimal  $p^*$  has the form

$$p^* = \sum_{i=1}^N \langle u, w_i \rangle w_i.$$

Finally, we see

$$\begin{aligned} \langle p - p^*, w_j \rangle &= \langle p, w_j \rangle - \sum_{k=1}^N \langle p, w_k \rangle \langle w_k, w_j \rangle \\ &= \langle p, w_j \rangle - \langle p, w_j \rangle = 0, \end{aligned}$$

and hence,  $p = p^*$  is orthogonal of  $\mathcal{W}$ . ■

## 2.5 Graham–Schmidt Orthogonalization

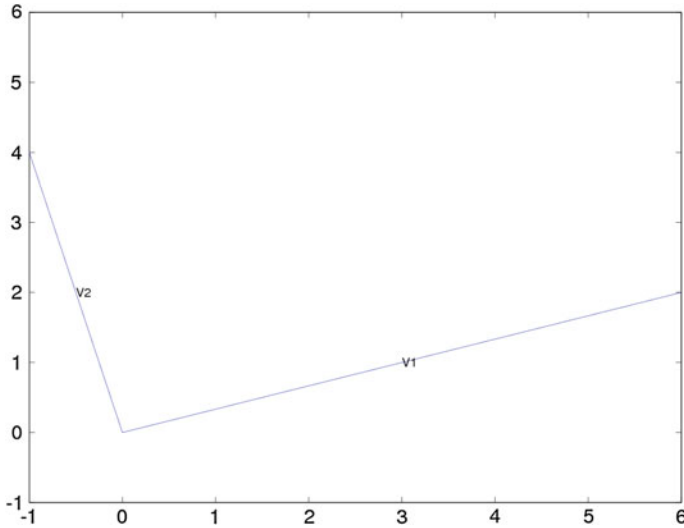
Let's assume we are given two linearly independent vectors in  $\mathfrak{N}^2$ . Fire up MatLab/Octave and enter two such vectors.

**Listing 2.9:** Setting two vectors

```
V1 = [6; 2];
V2 = [-1; 4];
```

In Fig. 2.1 we see these two vectors.

Now *project*  $V_2$  to the vector  $V_1$ . The unit vector pointing in the direction of  $V_1$  is  $E_1 = V_1/||V_1||$  and the amount of  $V_2$  that lies in the direction of  $V_1$  is given by the



**Fig. 2.1** Two linearly independent vectors in  $\mathfrak{N}^2$

inner product  $\langle V_1, E_1 \rangle$ . Hence, the vector  $\langle V_1, E_1 \rangle E_1$  in the vector portion of  $V_2$  which lies in the direction of  $V_1$ . Now subtract this from  $V_2$ . Hence, define  $W$  by

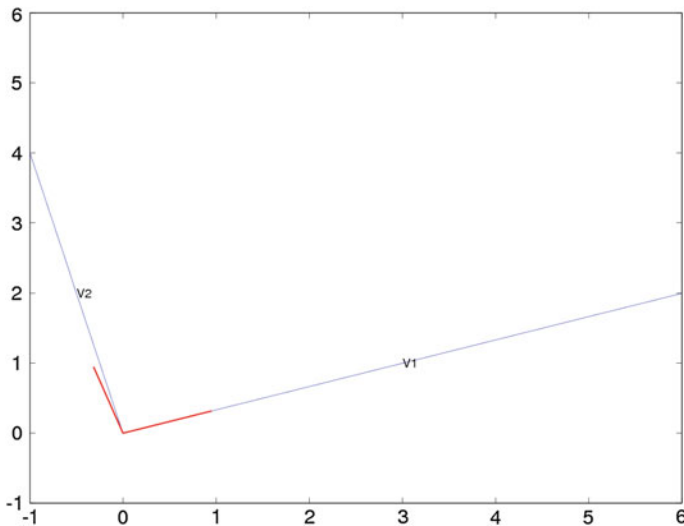
$$W = V_2 - \langle V_2, E_1 \rangle E_1.$$

Note by construction  $W$  is perpendicular to  $V_1$  as

$$\begin{aligned} \langle W, E_1 \rangle &= \langle V_2 - \langle V_2, E_1 \rangle E_1, E_1 \rangle = \langle V_2, E_1 \rangle - \langle V_2, E_1 \rangle \langle E_1, E_1 \rangle \\ &= \langle V_2, E_1 \rangle - \langle V_2, E_1 \rangle = 0, \end{aligned}$$

since  $\langle E_1, E_1 \rangle = \|E_1\|^2 = 1$ . Let  $E_2$  be the unit vector given by  $W/\|W\|$ . Then from the linearly independent vectors  $V_1$  and  $V_2$ , we have constructed two new linearly independent vectors which are mutually orthogonal,  $E_1$  and  $E_2$ . We also see both  $\{V_1, V_2\}$  and  $\{E_1, E_2\}$  are bases for  $\mathfrak{R}^2$  but the new one  $\{E_1, E_2\}$  is preferred as the basis vectors are orthogonal. We can see the new vectors in Fig. 2.2.

You should be able to see that this procedure is easy to extend to three linearly vectors in  $\mathfrak{R}^3$  and finite sets of linearly independent functions as well. This type of procedure is called **Graham–Schmidt Orthogonalization** and we can see this graphically in  $\mathfrak{R}^2$ . First we graph the vectors. This done with calls to **plot**. We start with the function **Draw2DGSO** and add the vector drawing code first. We do all of our drawing between a **hold on** and **hold off** so we keep drawing into the figure until we are done.



**Fig. 2.2** Two linearly independent vectors in  $\mathfrak{R}^2$

**Listing 2.10:** Draw2DGSO: drawing the vectors

```

function Draw2DGSO(V1,V2)
%
T = linspace(0,1,2);
hold on
5  % find equation of line for vector V1
   if V1(1) == 0
       f1 = @(t) t;
   else
       f1 = @(t) (V1(2)/V1(1))*t;
10  end
   T = [0:V1(1):V1(1)];
   plot(T,f1(T));
   text(V1(1)/2,V1(2)/2, 'v1');
   % find equation of line for vector V2
15  if V2(1) == 0
       f2 = @(t) t;
   else
       f2 = @(t) (V2(2)/V2(1))*t;
   end
20  plot(T,f2(T));
   T = [0:V2(1):V2(1)];
   text(V2(1)/2,V2(2)/2, 'v2');
hold off
end

```

Note we also use the new **text** command to add text at desired points in the graph. Next, we find the new basis vectors  $E_1$  and  $E_2$  and graph them.

**Listing 2.11:** Graph the new basis vectors

```

% Now plot e1
E1 = V1/norm(V1);
% find equation of line for vector V1
if E1(1) == 0
5  e1 = @(t) t;
else
    e1 = @(t) (E1(2)/E1(1))*t;
end
T = [0:E1(1):E1(1)];
10 plot(T,e1(T),'r','linewidth',4);
% find equation of line for vector e2
W = V2 - dot(V2,E1)*E1
E2 = W/norm(W);
% find equation of line for vector e2
15 if E2(1) == 0
    e2 = @(t) t;
else
    e2 = @(t) (E2(2)/E2(1))*t;
end
20 T = [0:E2(1):E2(1)];
plot(T,e2(T),'r','linewidth',4);

```

Now if you just did this, you will probably see the  $E_1$  and  $E_2$  don't necessarily look perpendicular in the figure. This is because by itself, MatLab/Octave does not necessarily use a square picture frame for our graph. So the  $x$  and  $y$  axes need not be

the same size. If they are not this, will make us lose the appearance of orthogonality! So to fix this, we make sure the  $x$  and  $y$  axes are the same size. This requires some messy code as follows.

**Listing 2.12:** Setting the aspect ratio

```

% set axis limits
xmin = V1(1);
if (V2(1) < xmin)
4   xmin = V2(1);
   xmax = V1(1);
else
   xmax = V2(1);
end
9   ymin = V1(2);
   if (V2(2) < ymin)
       ymin = V2(2);
       ymax = V2(2);
   else
14  ymax = V2(1);
   end
%
umin = xmin;
if (ymin < umin)
19  umin = ymin;
end
umax = xmax;
if (ymax > umax)
   umax = ymax;
24 end
axis ([umin, umax, umin, umax]);

```

This sets the axes correctly, but we still need to make sure the figure is square. We do this by setting the figure size with **figure('Position', [0,0,600,600]);** at the top of the file. The full code to see what is happening in  $\mathbb{R}^2$  is then given in the function **Draw2DGSO** given below.

**Listing 2.13:** Drawing a two dimensional Graham–Schmidt orthogonalization result: Draw2DGSO

```

function Draw2DGSO(V1,V2)
%
T = linspace(0,1,2);
figure('Position',[0,0,600,600]);
5 hold on
% find equation of line for vector V1
if V1(1) == 0
   f1 = @(t) t;
else
10  f1 = @(t) (V1(2)/V1(1))*t;
end
T = [0:V1(1):V1(1)];
plot(T,f1(T));
text(V1(1)/2,V1(2)/2, 'v1');
15 % find equation of line for vector V1
if V2(1) == 0
   f2 = @(t) t;
else
   f2 = @(t) (V2(2)/V2(1))*t;
20 end
plot(T,f2(T));

```

```

T = [0:V2(1):V2(1)];
text(V2(1)/2,V2(2)/2, 'v2');
% Now plot e1
25 E1 = V1/norm(V1);
% find equation of line for vector V1
if E1(1) == 0
    e1 = @(t) t;
else
30     e1 = @(t) (E1(2)/E1(1))*t;
end
T = [0:E1(1):E1(1)];
plot(T,e1(T),'r','linewidth',4);
% find equation of line for vector e2
35 W = V2 - dot(V2,E1)*E1;
E2 = W/norm(W);
% find equation of line for vector e2
if E2(1) == 0
    e2 = @(t) t;
40 else
    e2 = @(t) (E2(2)/E2(1))*t;
end
T = [0:E2(1):E2(1)];
plot(T,e2(T),'r','linewidth',4);
45 % set axis limits
xmin = V1(1);
if (V2(1) < xmin)
    xmin = V2(1);
    xmax = V1(1);
50 else
    xmax = V2(1);
end
ymin = V1(2);
if (V2(2) < ymin)
55     ymin = V2(2);
    ymax = V2(2);
else
    ymax = V1(2);
end
60 %
umin = xmin;
if (ymin < umin)
    umin = ymin;
end
65 umax = xmax;
if (ymax > umax)
    umax = ymax;
end
axis([umin,umax,umin,umax]);
70 hold off
end

```

It is simple to use this code. The example we just worked out would be done this way in MatLab/Octave.

#### Listing 2.14: A 2DGSO sample session

```

V1 = [6;2];
V2 = [-1;4];
Draw2DGSO(V1,V2);
4 print -dpng '2DGSO.png'

```



We can certainly do this procedure for three linearly independent vectors in  $\mathbb{R}^3$ . We graph lines using the **plot3** function. For example, to draw the line between  $V_1$  and  $V_2$ , we set up vectors  $X$ ,  $Y$  and  $Z$  as follows:

**Listing 2.15:** Setting coordinates for a line plot

```
X = [0; V1(1) ];
Y = [0; V1(2) ];
Z = [0; V1(3) ];
```

The way to look at this is that the first column of these three vectors specifies the start position  $[0, 0, 0]^T$  and the second column is the end position coordinates  $V_1^T$ . We then plot the line and add text with

**Listing 2.16:** Plotting the lines

```
plot3(X,Y,Z,'r','linewidth',2);
2 text(V1(1)/2,V1(2)/2,V1(3)/2, 'v1');
```

Note we set the width of the plotted line to be of size 2 which is thicker than size 1. Once this line is plotted, we do the others. So the code to plot the three vectors as lines starting at the origin is as follows; note, we wrap this code between **hold on** and **hold off** statements so that we draw into our figure repeatedly until we are done.

**Listing 2.17:** Plotting all the lines

```
hold on
% plot V1
X = [0; V1(1) ];
Y = [0; V1(2) ];
5 Z = [0; V1(3) ];
plot3(X,Y,Z,'r','linewidth',2);
text(V1(1)/2,V1(2)/2,V1(3)/2, 'v1');
% plot V2
X = [0; V2(1) ];
10 Y = [0; V2(2) ];
Z = [0; V2(3) ];
plot3(X,Y,Z,'r','linewidth',2);
text(V2(1)/2,V2(2)/2,V2(3)/2, 'v2');
% plot V3
15 X = [0; V3(1) ];
Y = [0; V3(2) ];
Z = [0; V3(3) ];
plot3(X,Y,Z,'r','linewidth',2);
text(V3(1)/2,V3(2)/2,V3(3)/2, 'v3');
20 hold off
```

We then do the steps of the Graham–Schmidt orthogonalization. First, we set up  $E_1 = V_1/||V_1||$  as usual. We then project  $V_2$  to  $V_1$  to obtain

$$W = V_2 - \langle V_2, E_1 \rangle E_1$$

which by construction will be perpendicular to  $E_1$ . We then set  $E_2 = W/||W||$ . Finally, we project  $V_3$  to the plane determined by  $V_1$  and  $V_2$  as follows:

$$W = V_3 - \langle V_3, E_1 \rangle E_1 - \langle V_3, E_2 \rangle E_2$$

which by construction will be perpendicular to both  $E_1$  and  $E_2$ . Finally, we let  $E_3 = W/||W||$  and we have found a new mutually orthogonal basis for  $\mathbb{R}^3 \{E_1, E_2, E_3\}$ . It is easy to put this into code. We write

**Listing 2.18:** Finding and plotting the new basis

```
% Do GSO
E1 = V1/norm(V1);
W = V2 - dot(V2,E1)*E1;
E2 = W/norm(W);
5 W = V3 - dot(V3,E1)*E1 - dot(V3,E2)*E2;
E3 = W/norm(W);
% Plot new basis
% plot E1
X = [0;E1(1)];
10 Y = [0;E1(2)];
Z = [0;E1(3)];
plot3(X,Y,Z,'b','linewidth',4);
% plot E2
X = [0;E2(1)];
15 Y = [0;E2(2)];
Z = [0;E2(3)];
plot3(X,Y,Z,'b','linewidth',4);
% plot E3
20 X = [0;E3(1)];
Y = [0;E3(2)];
Z = [0;E3(3)];
plot3(X,Y,Z,'b','linewidth',4);
```

Then to make this show up nicely, we draw lines between  $V_1$  and  $V_2$  to make the plane determined by these vectors stand out.

**Listing 2.19:** Drawing the plane between  $V_1$  and  $V_2$

```
% Draw plane determined by V1 and V2
delt = 0.05;
3 for i = 1:10
    p = i*delt;
    X = [p*V1(1);p*V2(1)];
    Y = [p*V1(2);p*V2(2)];
    Z = [p*V1(3);p*V2(3)];
8    plot3(X,Y,Z,'r','linewidth',2);
end
text((V1(1)+V2(1))/4, (V1(2)+V2(2))/4, (V1(3)+V2(3))/4, 'v1 -
    v2 Plane');
```

The full code is then given in the function **Draw3DGSO**.

**Listing 2.20:** Drawing a three dimensional Graham–Schmidt orthogonalization result: Draw3DGSO

```

function Draw3DGSO(V1,V2,V3)
%
T = linspace(0,1,2);
figure('Position',[0,0,600,600]);
5 hold on
    % plot V1
    X = [0;V1(1)];
    Y = [0;V1(2)];
    Z = [0;V1(3)];
10 plot3(X,Y,Z,'r','linewidth',2);
    text(V1(1)/2,V1(2)/2,V1(3)/2, 'v1');
    % plot V2
    X = [0;V2(1)];
    Y = [0;V2(2)];
15 Z = [0;V2(3)];
    plot3(X,Y,Z,'r','linewidth',2);
    text(V2(1)/2,V2(2)/2,V2(3)/2, 'v2');
    % plot V3
    X = [0;V3(1)];
    Y = [0;V3(2)];
20 Z = [0;V3(3)];
    plot3(X,Y,Z,'r','linewidth',2);
    text(V3(1)/2,V3(2)/2,V3(3)/2, 'v3');
    %
25 % Draw plane determined by V1 and V2
    delt = 0.05;
    for i = 1:10
        p = i*delt;
        X = [p*V1(1);p*V2(1)];
30 Y = [p*V1(2);p*V2(2)];
        Z = [p*V1(3);p*V2(3)];
        plot3(X,Y,Z,'r','linewidth',2);
    end
    text((V1(1)+V2(1))/4, (V1(2)+V2(2))/4, (V1(3)+V2(3))/4, 'v1 - v2
        Plane');
35 %
    % Do GSO
    E1 = V1/norm(V1);
    W = V2 - dot(V2,E1)*E1;
    E2 = W/norm(W);
40 W = V3 - dot(V3,E1)*E1 - dot(V3,E2)*E2;
    E3 = W/norm(W);
    % Plot new basis
    % plot E1
    X = [0;E1(1)];
    Y = [0;E1(2)];
45 Z = [0;E1(3)];
    plot3(X,Y,Z,'b','linewidth',4);
    % plot E2
    X = [0;E2(1)];
    Y = [0;E2(2)];
50 Z = [0;E2(3)];
    plot3(X,Y,Z,'b','linewidth',4);
    % plot E3
    X = [0;E3(1)];
    Y = [0;E3(2)];
55 Z = [0;E3(3)];
    plot3(X,Y,Z,'b','linewidth',4);
    hold off
end

```

It is again easy to use this code. Here is a sample session. Since we make up our vectors, we always set up a matrix  $A$  using the vectors as rows and then find the

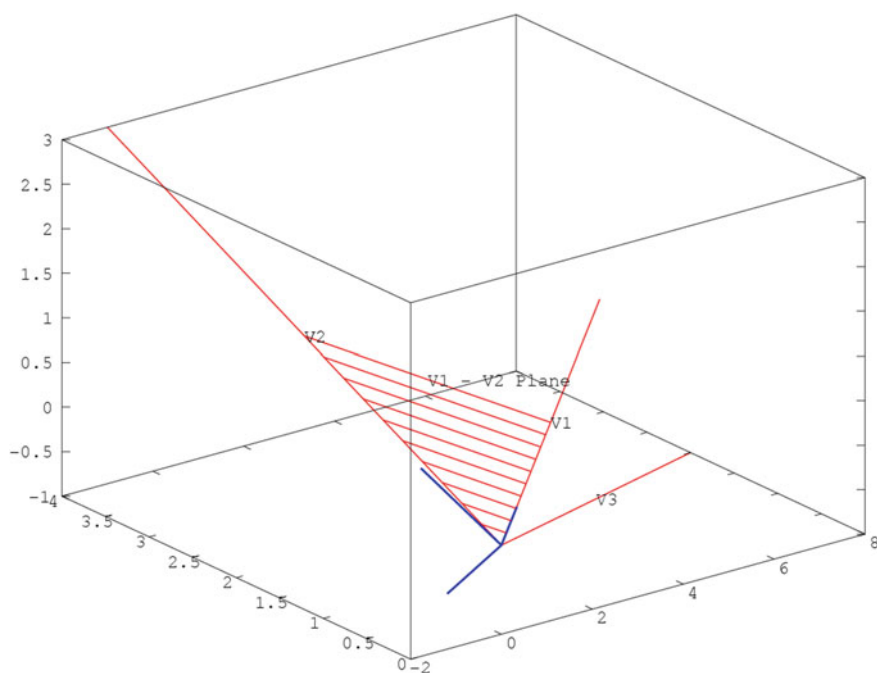
determinant of  $A$ . If that was 0, we would know our choice of vectors was not linearly independent and we would have to make up new vectors for our example.

**Listing 2.21:** A Draw3DGSO example

```

1 V1 = [6;2;1];
  V2 = [-1;4;3];
  V3 = [8;2;-1];
  A = [V1'; V2'; V3'];
  det(A)
6 ans = -48.000
  Draw3DGSO(V1,V2,V3);
  print -dpng '3DGSO.png'
```

We can use this code to generate a nice 3D plot of this procedure as shown in Fig. 2.3. If you do this in MatLab/Octave yourself, you can grab the graph and rotate it around to make sure you see it in the orientation that makes the best sense to you.



**Fig. 2.3** Three linearly independent vectors in  $\mathbb{R}^3$

### 2.5.1 Making It Formal

If we had a finite number of linearly independent objects in an inner product space  $\mathcal{V}$ , then they form a basis for their span. There is a recursive procedure to find a new orthonormal basis from these vectors for the subspace formed by their span. This is called the Graham–Schmidt Orthogonalization process. It is easiest to show it explicitly for three vectors. You can then easily generalize it to more objects. Let's assume we start with three linearly independent objects  $u$ ,  $v$  and  $w$ . We will find three orthogonal objects of length 1,  $g_1$ ,  $g_2$  and  $g_3$  to be the new basis as follows:

**First Basis Object** Set

$$g_1 = \frac{u}{||u||}.$$

**Second Basis Object**

- Subtract the part of  $v$  which lies along the object  $g_1$ .

$$h = v - \langle v, g_1 \rangle g_1.$$

- Find the length of  $h$  and set the second new basis object as follows:

$$g_2 = \frac{h}{||h||}.$$

**Third Basis Object**

- Subtract the part of  $w$  which lies along the object  $g_1$  and the object  $g_2$ .

$$h = w - \langle w, g_1 \rangle g_1 - \langle w, g_2 \rangle g_2.$$

- Find the length of  $h$  and set the third new basis object as follows:

$$g_3 = \frac{h}{||h||}.$$

It is easy to see how to generalize this to four or more objects. In fact, the procedure is the same whether we are using vectors or functions as long as the objects are linearly independent.

### 2.5.2 General MatLab GSO

To see how we could do GSO in computationally, we will now show some simple MatLab code. We will use the numerical integration code presented in this chapter

which uses Newton–Cotes formulae and also show you how to use the built-in quadrature codes in MatLab itself. In general, doing GSO on functions is *hard*, as you will see, due to numerical errors. The code we have shown you so far is hard-wired to the number of linearly independent objects. Hence, we wrote 2D and 3D GSO code separately. Now we'll try to write more general code where the argument we send into the function is a collection of linearly independent objects. The code ideas that follow are specialized to functions, but it wouldn't be hard to write similar code for sets of independent vectors in  $\mathbb{R}^n$ .

The first code is the one that uses Newton–Cotes ideas. This is written to perform GSO on powers of  $t$ , but it could be easily generalized to more general functions. Some word of explanation are in order. First, we use a generic powers of  $t$  function

**Listing 2.22:** Generic Powers of  $t$  Function

```
function y = Powers(t,n)
2 % t = independent variable
  % n = power
  y = t.^n;
end
```

We then use extensively the idea of pointers to functions which in MatLab are called *function handles*. We also use the idea of *anonymous functions* in MatLab. The basic syntax for these is something like **h = @(x)x.^2;** which sets up  $h$  as an *anonymous* function so that a session line like **h(3)** would be evaluated as  $3^2$ . The letter **h** here is the function handle which allows us to refer to this function in other code. We do the GSO using nested loops of anonymous functions using syntax like

**Listing 2.23:** Constructing an anonymous function as a loop

```
for k=M+2:N+1
  %compute next orthogonal piece
  phi = @(x) 0;
  for j = M+1:k-1
5    c = ip(f{k},g{j},a,b,r);
    phi = @(x) (phi(x)+c*g{j}(x));
  end
  psi = @(x) (f{k}(x) - phi(x));
  nf = sqrt(ip(psi,psi,a,b,r));
10 g{k} = @(x) (psi(x)/nf);
```

to calculate each of the new basis objects  $g_k$ . We start with the zero anonymous function **phi** and progressively add up the pieces we must subtract that lie along the previous  $g_j$  objects. Once this is done, we subtract this sum from the original object and then divide by the length. This then creates the new object  $g_k$ .

We also use an inner product implementation nested inside this code as follows:

**Listing 2.24:** Nested inner product code

```
function c = ip(f,g,a,b,r)
    w = WNC(r);
    s = linspace(a,b,r)';
    u = zeros(r,1);
5    u = f(s).*g(s);
    c = (b-a)*(w'*u);
end
```

which uses the function handles  $f$  and  $g$  we pass with the desired Newton–Cotes order to approximate  $\int_a^b fgds$ .

**Listing 2.25:** A Newton–Cotes GSO Code

```
function [t,z] = GrahamSchmidtNC(a,b,r,M,N,NumPoints)
%
3  % Perform Graham - Schmidt Orthogonalization
% on a set of functions 1, t, t^2, ..., t^N
%
% a = start of interval
% b = end of interval
% r = Newton-Cotes order -- can be as high as 9
% M = power of t to start at
% N = power of t to end with
% NumPoints = the number of points to use in the t
%               vector we will use for plotting the g_k's
13 %
% setup t as 1xNumPoints
t = linspace(a,b,NumPoints);
% setup x as NumPoints rows x N+1 columns
z = zeros(NumPoints,N+1);
18 %Setup function handles
f = cell(N+1,1);
g = cell(N+1,1);
for i=1:N+1
    f{i} = @(x) Powers(x,i-1);
23 end
nf = sqrt(ip(f{M+1},f{M+1},a,b,r));
g{M+1} = @(x) f{M+1}(x)/nf;
y = zeros(1,NumPoints);
d = zeros(N+1,N+1);
28 y = g{M+1}(t);
z(:,M+1) = y';
for k=M+2:N+1
    %compute next orthogonal piece
    phi = @(x) 0;
33     for j = M+1:k-1
        c = ip(f{k},g{j},a,b,r);
        phi = @(x) (phi(x)+c*g{j}(x));
    end
    psi = @(x) (f{k}(x) - phi(x));
38     nf = sqrt(ip(psi,psi,a,b,r));
    g{k} = @(x) (psi(x)/nf);
    y = g{k}(t);
    z(:,k) = y';
end
43 % find the matrix of inner products
for i=M+1:N+1
    for j=M+1:N+1
        d(i,j) = ip(g{i},g{j},a,b,r);
    end
48 end
```

```

% Print out the matrix of inner products
d

function c = ip(f,g,a,b,r)
53   w = WNC(r);
      s = linspace(a,b,r)';
      u = zeros(r,1);
      u = f(s).*g(s);
      c = (b-a)*(w'*u);
58   end

end

```

Here are some Newton–Cotes results. First, we try a low order 3 Newton–Cotes method. There is too much error here. In the snippet below,  $d$  is the matrix of  $\langle g_i, g_j \rangle$  inner products which should be an identity matrix. Note the many off diagonal terms which are close to 1 is absolute value. Much error here.

**Listing 2.26:** GSO with Newton–Cotes Order 3

```

[t,z] = GrahamSchmidtNC(0,1,3,0,5,101);

d =
5   1.0000         0         0.0000    -0.4003    -0.4003    -0.4003
      0         1.0000         0.0000    -0.1387    -0.1387    -0.1387
      0.0000         0.0000         1.0000    -0.9058    -0.9058    -0.9058
     -0.4003    -0.1387    -0.9058         1.0000         1.0000         1.0000
     -0.4003    -0.1387    -0.9058         1.0000         1.0000         1.0000
10    -0.4003    -0.1387    -0.9058         1.0000         1.0000         1.0000

```

Newton–Cotes of order 4 is somewhat better but still unacceptable.

**Listing 2.27:** GSO with Newton–Cotes Order 4

```

[t,z] = GrahamSchmidtNC(0,1,4,0,5,101);

d =
      1.0000    -0.0000         0.0000    -0.0000         0.0112    -0.0112
     -0.0000         1.0000         0.0000    -0.0000         0.0208    -0.0208
5     0.0000         0.0000         1.0000    -0.0000         0.1206    -0.1206
     -0.0000    -0.0000    -0.0000         1.0000         0.9924    -0.9924
      0.0112         0.0208         0.1206         0.9924         1.0000    -1.0000
     -0.0112    -0.0208    -0.1206    -0.9924    -1.0000         1.0000

```

We must go to Newton–Cotes of order 6 to get proper orthogonality.



**Listing 2.28:** GSO with Newton–Cotes Order 6

```
[t,z] = GrahamSchmidtNC(0,1,6,0,5,101);
2 d =
    1.0000    0.0000   -0.0000    0.0000    0.0000   -0.0000
    0.0000    1.0000   -0.0000    0.0000   -0.0000    0.0000
   -0.0000   -0.0000    1.0000    0.0000   -0.0000    0.0000
    0.0000    0.0000    0.0000    1.0000   -0.0000    0.0000
7   0.0000   -0.0000   -0.0000   -0.0000    1.0000    0.0000
   -0.0000    0.0000    0.0000    0.0000    0.0000    1.0000
```

It is much harder also, if we want to apply GSO to more powers of  $t$ , say up to  $t^{20}$ . In general, we would need a Newton–Cotes order of about 20 which would be numerically unstable itself.

Another approach is to replace the inner product calculations with the build in quadrature codes in MatLab. Here we use the function **quadl** instead of **quad**. Do a **help quad** and **help quadl** in MatLab/Octave to figure out how these two functions differ! We will find we need to fiddle with the error tolerance to get good results. This is analogous to what we had to do before by increasing the Newton–Cotes order.

**Listing 2.29:** Inner products using quad

```
function c = ip(f,g,a,b)
    v = @(x) (f(x).*g(x));
    tol = 1.0e-9;
    c = quadl(v,a,b,tol);
5 end
```

The new GSO code is as follows:

**Listing 2.30:** Graham–Schmidt Orthogonalization With quadl

```
function [t,z] = GrahamSchmidt(a,b,M,N,NumPoints)
%
% Perform Graham - Schmidt Orthogonalization
% on a set of functions 1, t, t^2, ..., t^N
5 %
% a = start of interval
% b = end of interval
% M = first power of t to use
% N = last power of t to use
10 % NumPoints = number of time points to use for time
%           for plotting of the g_k's
%
% setup t as 1xNumPoints
t = linspace(a,b,NumPoints);
15 % setup x as NumPoints rows x N+1 columns
z = zeros(NumPoints,N+1);
%Setup function handles
f = cell(N+1,1);
g = cell(N+1,1);
20 for i=1:N+1
    f{i} = @(x) Powers(x,i-1);
end
```

```

    nf = sqrt(ip(f{M+1},f{M+1},a,b));
    g{M+1} = @(x) f{M+1}(x)/nf;
25 y = zeros(1,NumPoints);
    d = zeros(N+1,N+1);
    y = g{M+1}(t);
    z(:,M+1) = y';
    for k=M+2:N+1
30 %compute next orthogonal piece
        phi = @(x) 0;
        for j = M+1:k-1
            c = ip(f{k},g{j},a,b);
            phi = @(x) (phi(x)+c*g{j}(x));
35        end
        psi = @(x) (f{k}(x) - phi(x));
        nf = sqrt(ip(psi,psi,a,b));
        g{k} = @(x) (psi(x)/nf);
        y = g{k}(t);
40        z(:,k) = y';
        end
        for i=M+1:N+1
            for j=M+1:N+1
                d(i,j) = ip(g{i},g{j},a,b);
45            end
        end
    end
    d

    function c = ip(f,g,a,b)
50        v = @(x) (f(x).*g(x));
        tol = 1.0e-9;
        c = quadl(v,a,b,tol);
    end

55 end

```

We then get similar results for the GSO.

### Listing 2.31: GSO with quadl

```

[t,z] = GrahamSchmidt(0,1,0,5,101);
d =
    1.0000    -0.0000         0     0.0000    -0.0000     0.0000
   -0.0000     1.0000     0.0000   -0.0000     0.0000   -0.0000
5         0     0.0000     1.0000   -0.0000     0.0000   -0.0000
    0.0000   -0.0000   -0.0000     1.0000     0.0000   -0.0000
   -0.0000     0.0000     0.0000     0.0000     1.0000   -0.0000
    0.0000   -0.0000   -0.0000   -0.0000   -0.0000     1.0000

```

There are other ways to do this too, but this should give the idea. The bottom line is that it is straightforward to do theoretically, but devilishly hard to do in practice!

## Reference

J. Peterson, *Calculus for Cognitive Scientists: Higher Order Models and Their Analysis*, Springer Series on Cognitive Science and Technology. (Springer Science + Business Media, Singapore 2015 in press)

Calculus for Cognitive Scientists

Partial Differential Equation Models

Peterson, J.K.

2016, XXXI, 534 p. 156 illus. in color., Hardcover

ISBN: 978-981-287-878-6