1 Dot products of functions

We can apply the Gram–Schmidt process to any vector space as long as we define a dot product (also called an inner product). (Technically, a continuous (“complete”) vector space equipped with an inner product is called a Hilbert space.)

For column vectors, the usual dot product is to multiply the components and add them up.

But (real-valued) functions \( f(x) \) also define a vector space (you can add, subtract, and multiply by constants). In particular, consider functions defined on the interval \( x \in [-1,1] \). The “components” of \( f \) can be viewed as its values \( f(x) \) at each point in the domain, and the obvious analogue of “summing the components” is the integral. Hence, the most obvious “dot product” of two functions in this space is:

\[
\int_{-1}^{1} f(x)g(x) \, dx
\]

Such a generalized inner product is commonly denoted \( \langle f,g \rangle \) (or \( \langle f|g \rangle \) in physics).

2 Orthogonal polynomials

In particular, let us consider a subspace of functions defined on \([-1,1]\): polynomials \( p(x) \) (of any degree).

One possible basis of polynomials is simply:

\[
1, x, x^2, x^3, \ldots
\]

(There are infinitely many polynomials in this basis because this vector space is infinite-dimensional.)

Instead, let us apply Gram–Schmidt to this basis in order to get an orthogonal basis of polynomials known as the Legendre polynomials.

2.1 Julia code

I’ll use the Polynomials package to do polynomial arithmetic for me.

However, I’ll need to define a few extra methods to perform my dot products from above, and I also want to display (“pretty print”) the polynomials a bit more nicely than the default.

```julia
In [2]: # compute the definite integral of p(x) from a to b
    function Polynomials.polyint(p::Poly, a, b)
        pi = polyint(p)
        pi(b) - pi(a)
    end
    # compute the dot product \( \langle p,q \rangle = \int p(x)q(x) \) on \([-1,1]\)
    polydot(p::Poly, q::Poly) = polyint(p*q, -1,1)
```
2.2 Gram–Schmidt on polynomials

Now, let’s apply Gram–Schmidt on the polynomials $a_i = x^i$ for $i = 0, 1, \ldots$.

Ordinarily, in Gram–Schmidt, I would normalize each result $p(x)$ by dividing by $\|p\| = \sqrt{p \cdot p}$, but that will result in a lot of annoying square roots. Instead, I will divide by $p(1)$ to result in the more conventional Legendre polynomials.

That means that to get $p_i(x)$, I will do:

$$
\hat{p}_i(x) = a_i(x) - \sum_{j=0}^{i-1} p_j(x) \frac{p_j \cdot a_i}{p_j \cdot p_j}
$$

$$
p(x) = \hat{p}(x)/\hat{p}(1)
$$

where I explicitly divide by $p_i \cdot p_i$ in the projections to compensate for the lack of normalization.

In Julia, I will use the special syntax 2 // 3 to construct the exact rational $\frac{2}{3}$, etc. This will allow me to see the exact Legendre polynomials without any roundoff errors or annoying decimals.

```julia
In [4]: p0 = a0 = Poly([1//1])
```

```
Out[4]:
1
```

```julia
In [5]: a1 = Poly([0, 1//1])
```

```
Out[5]:

x
```

```julia
In [6]: p1 = a1 - p0 * polydot(p0, a1) // polydot(p0, p0)
p1 = p1 / p1(1)
```

```
Out[6]:

x
```

Orthogonalization didn’t change $x$, because $x$ and 1 are already orthogonal under this dot product. In fact, any even power of $x$ is orthogonal to any odd power (because the dot product is the integral of an even function times an odd function).

On the other hand, $x^2$ and 1 are not orthogonal, so orthogonalizing them leads to a different polynomial of degree 2:

```julia
In [7]: a2 = Poly([0, 0, 1//1])
```

```
Out[7]:

x^2
```

```julia
In [8]: p2 = a2 - p0 * polydot(p0, a2) // polydot(p0, p0) - p1 * polydot(p1, a2) // polydot(p1, p1)
p2 = p2 / p2(1)
```

```
Out[8]:

x^2
```
It quickly gets tiresome to type in these expressions one by one, so let’s just write a function to compute the Legendre polynomials \( p_0, \ldots, p_n \):

```python
In [9]: function legendre_gramschmidt(n)
    legendre = [Poly([1//1])]
    for i = 1:n
        p = Poly([k == i ? 1//1 : 0//1 for k=0:i])
        for q in legendre
            p = p - q * (polydot(q, p) // polydot(q,q))
        end
        push!(legendre, p / p(1))
    end
    return legendre
end
```

Let’s display them more nicely with LaTeX:

```python
In [11]: foreach(p -> display("text/latex", p), L)
```

Key things to notice:

- The polynomials contain only even or only odd powers of \( x \), but not both. The reason is that the even and odd powers of \( x \) are already orthogonal under this dot product, as noted above.
A key property of Gram–Schmidt is that the first k vectors span the same space as the original first k vectors, for any k. In this case, it means that $p_0, \ldots, p_k$ span the same space as $1, x, \ldots, x^k$. That is, the $p_0, \ldots, p_k$ polynomials are an orthogonal basis for all polynomials of degree k or less.

These polynomials are very special in many ways. To get a hint of that, let’s plot them:

```
In [12]: leg = []
x = linspace(-1, 1, 300)
for i in eachindex(L)
    plot(x, L[i].(x), "-"
    push!(leg, "P_{${i-1}}(x)"")
end
plot(x, 0*x, "k--")
legend(leg)
xlabel("x")
ylabel("Legendre polynomials")
```

![Legendre polynomials graph](image)

Note that $p_n(x)$ has exactly n roots in the interval $[-1, 1]$!

### 2.2.1 Expanding a polynomial in the Legendre basis.

Now that we have an orthogonal (but not orthonormal) basis, it is easy to take an arbitrary polynomial $p(x)$ and write it in this basis:
\[ p(x) = \alpha_0 p_0(x) + \alpha_1 p_1(x) + \cdots = \sum_{i=0}^{\infty} \alpha_i p_i(x) \]

because we can get the coefficients \( \alpha_i \) merely by projecting:

\[ \alpha_i = \frac{p_i \cdot p}{p_i \cdot p_i} \]

Note, however, that this isn’t actually an infinite series: if the polynomial \( p(x) \) has degree \( d \), then \( \alpha_i = 0 \) for \( i > d \). The polynomials \( p_0, \ldots, p_d \) are a basis for the subspace of polynomials of degree \( d \) (= span of \( 1, x, \ldots, x^d \)).

Let’s see how this works for a “randomly” chosen \( p(x) \) of degree 5:

\textbf{In [13]: } p = \text{Poly}([1,3,4,7,2,5])

\textbf{Out[13]: }

\[ 1 + 3 \cdot x + 4 \cdot x^2 + 7 \cdot x^3 + 2 \cdot x^4 + 5 \cdot x^5 \]

Here are the coefficients \( \alpha_i \):

\textbf{In [14]: } \alpha = [\text{polydot}(q,p)/\text{polydot}(q,q) \text{ for } q \text{ in } L]

\textbf{Out[14]: } 6\text{-element Array\{Rational\{Int64\},1\}}:

\[ 41/15, 327/35, 80/21, 226/45, 16/35, 40/63 \]

Let’s check that the sum of \( \alpha_i p_i(x) \) gives \( p(x) \):

\textbf{In [15]: } \text{sum}(\alpha.* L) # \alpha[1]*L[1] + \alpha[2]*L[2] + \ldots + \alpha[6]*L[6]

\textbf{Out[15]: }

\[ 1 + 3 \cdot x + 4 \cdot x^2 + 7 \cdot x^3 + 2 \cdot x^4 + 5 \cdot x^5 \]

\textbf{In [16]: } \text{sum}(\alpha.* L) - p

\textbf{Out[16]: }

\[ 0/1 \]

\section*{2.3 Polynomial fits}

\subsection*{2.3.1 Review: Projections and Least-Square}

Given a matrix \( Q \) with \( n \) orthonormal columns \( q_i \), we know that the orthogonal projection

\[ p = QQ^T b = \sum_{i=1}^{n} q_i q_i^T b \]

is the closest vector in \( C(Q) \) to \( b \). That is, it minimizes the distance:

\[ \min_{p \in C(Q)} \| p - b \|. \]
### 2.3.2 Closest polynomials

Now, suppose that we have some function $f(x)$ on $x \in [-1,1]$ that is not a polynomial, and we want to find the closest polynomial of degree $n$ to $f(x)$ in the least-square sense. That is, we want to find the polynomials $p(x)$ of degree $n$ that minimizes

$$
\min_{p \in \mathcal{P}_n} \int_{-1}^{1} |f(x) - p(x)|^2 dx = \min_{p \in \mathcal{P}_n} \| f(x) - p(x) \|^2
$$

where

$$
\mathcal{P}_n = \text{span}\{1, x, x^2, \ldots, x^n\} = \text{span}\{p_0(x), p_1(x), \ldots, p_n(x)\}
$$

is the space of polynomials of degree $\leq n$, spanned by our Legendre polynomials up to degree $n$.

Presented in this context, we can see that this is the same problem as our least-square problem above, and the solution should be the same: $p(x)$ is the orthogonal projection of $f(x)$ onto $\mathcal{P}_n$, given by:

$$
p(x) = p_0(x) \frac{f(x)}{p_0} + \cdots + p_n(x) \frac{f(x)}{p_n}.
$$

Let’s try this out for $f(x) = e^x$. Because we’re lazy, we’ll have Julia compute the integrals numerically using its `quadgk` function, and fit it to polynomials of degree 5 using our Legendre polynomials from above.

```plaintext
In [17]: polydot(p::Poly, f::Function) = quadgk(x -> p(x)*f(x), -1,1, abstol=1e-13, reltol=1e-11)
Out[17]: polydot (generic function with 2 methods)
```

Now, let’s use dot products to compute the coefficients in the $p_i(x)$ expansion above for $f(x) = e^x$ (the `exp` function in Julia):

```plaintext
In [18]: coeffs = [polydot(p,exp)/polydot(p,p) for p in L]
Out[18]: 6-element Array{Float64,1}:
1.1752
1.10364
0.357814
0.0704556
0.00996513
0.00109959
```

One thing to notice is an important fact: expanding functions, especially smooth functions, in orthogonal bases like Legendre polynomials or Fourier series tends to converge very rapidly.

Let’s write out the resulting polynomial $p(x)$:

```plaintext
In [19]: p = sum(coeffs .* L)
```

Let’s plot it:

```plaintext
In [20]: plot(x, exp.(x), "r-")
plot(x, p.(x), "b--")
legend([L"e^x", L"fit $p(x)$"])
xlabel(L"x")
title(L"fitting $e^x$ to a degree-5 polynomial")
```
Out[20]: PyObject <matplotlib.text.Text object at 0x31f991610>

They are so close that you can hardly tell the difference!
Let’s plot the fits for degree 0, 1, …, 3 so that we can watch it converge:

In [21]: plot(x, exp.(x), "k--")
   for n = 1:4
       plot(x, sum([polydot(p,exp)/polydot(p,p) for p in L[1:n]] .* L[1:n]).(x), "-")
   end
legend([L"e^x", ["degree $i$ for $i=0:3$...]])
xlabel(L"x")
title(L"fitting $e^x$ to polynomials")
By degree 3, it is hard to tell the difference from $e^x$. 

**Out[21]:** PyObject <matplotlib.text.Text object at 0x322f64950>