Last name: Nitski

First name: Osvald

Student number: 1002456987

List of collaborators (if any):

In this exercise prepared by me (Mohannad Shehadeh), I'll be demonstrating the power of projections through a variety of examples. The purpose of the exercise is for you to see how much you can do using just the basic tools that you already know as well as to motivate the upcoming material in the course. The hope is that very soon, you'll come across a problem where you can apply the techniques illustrated in this exercise to do something cool. I'll assume that you're familiar with most of the ideas already, but I'll briefly review the basic facts that you should keep in mind as you go through the exercise. If any of them seem unfamiliar or don't make sense to you, I recommend you spend some time studying your lecture notes or textbook first, or reviewing your linear algebra. If you complete and understand Parts 1 and 2, Parts 3 and 4 can be completed by pretty much copying and pasting your code, so be sure to take your time to understand fully the first parts. There's very little code to write beyond implementing the Gram-Schmidt procedure which itself should only take a few lines of code. Moreover, you should be sure to understand fully the examples in Part 0 since you can reuse the code from the examples to do almost everything you'll need to do in the exercise.

Part 0: Preliminaries

We'll start by reviewing some facts and terminology that I'll be using:

- A vector space, or a subspace of a vector space, can be represented as the span of a collection of vectors, perhaps the columns of some matrix $\mathbf{V}$
- These vectors may or may not be linearly independent
- If these vectors are linearly independent, they are called a basis for that subspace and every vector $\mathbf{x}$ in that subspace can be represented uniquely as a linear combination of these vectors, i.e., the columns of $\mathbf{V}$
- If $\mathbf{b}$ is a vector of the coefficients of this linear combination $\mathbf{x} = \mathbf{V}\mathbf{b}$
- If the columns of $\mathbf{V}$ are a basis for some subspace of higher dimensional space, in which case $\mathbf{V}$ is a tall matrix, a vector $\mathbf{x}$ in this higher dimensional space can be projected onto the span of the columns of $\mathbf{V}$
- The projection $\hat{\mathbf{x}}$ can then be represented as $\hat{\mathbf{x}} = \mathbf{V}\mathbf{b}$ for some lower dimensional vector $\mathbf{b}$
- The projection $\hat{\mathbf{x}}$ will be the closest (in Euclidean distance) vector to $\mathbf{x}$ which lives the subspace spanned by the columns of $\mathbf{V}$
- In this case $\mathbf{b}$ will be refered to either as the "projection coefficients" or the "representation of the projection in terms of $\mathbf{V}$" and will provide a lower dimensional representation of $\mathbf{x}$
- $\hat{\mathbf{x}}$ is of the same size as $\mathbf{x}$, but, given $\mathbf{V}$ we can reproduce $\hat{\mathbf{x}}$ using a lower dimensional vector $\mathbf{b}$ so we are sometimes just interested in the lower dimensional representation $\mathbf{b}$
- Suppose, for example, we wish to compare many high dimensional $\mathbf{x}$. Once projected onto an appropriate subspace, we can instead compare the corresponding lower dimensional $\mathbf{b}$ vectors.
- The projection becomes very easy to compute when we have an orthonormal basis for our subspace.
- An orthonormal basis is a basis (a linearly independent set), in which the vectors are orthogonal and normalized to have a norm of 1 each
- If a collection of vectors is orthogonal, then they are also linearly independent. Therefore, any orthogonal set of vectors is a basis for the subspace they span
- Given any collection of vectors, whether or not they are a basis, we can construct an orthonormal basis for their span using the Gram-Schmidt procedure
- Once we have an orthonormal basis $\{\mathbf{w_1}, \ldots, \mathbf{w_m}\}$ for our subspace spanned by the columns of $\mathbf{V}$, we can have the basis vectors be the columns of some matrix $\mathbf{W}$.
- If we want to project $\mathbf{x}$ onto the span of the columns of $\mathbf{W}$ which is equal to the span of the columns of $\mathbf{V}$, the projection coefficients $\mathbf{a}$ are simply

$$\mathbf{a} = \begin{bmatrix} \langle \mathbf{w_1}, \mathbf{x} \rangle \\ \langle \mathbf{w_2}, \mathbf{x} \rangle \\ \vdots \\ \langle \mathbf{w_m}, \mathbf{x} \rangle \end{bmatrix} = \mathbf{W^T}\mathbf{x}$$

- Therefore, we have $\hat{\mathbf{x}} = \mathbf{W}\mathbf{a} = \mathbf{W}\mathbf{W^T}\mathbf{x}$
- If the columns of $\mathbf{V}$ were also a basis (linearly independent), then we can also find a $\mathbf{b}$ the same size as $\mathbf{a}$, but different, such that $\hat{\mathbf{x}} = \mathbf{W}\mathbf{a} = \mathbf{W}\mathbf{W^T}\mathbf{x} = \mathbf{V}\mathbf{b}$
- In this case, $\mathbf{a}$ is the representation of the projection in our orthonormal basis, and $\mathbf{b}$ is the representation of the projection in our non-orthogonal, non-orthonormal basis
- If the the columns of $\mathbf{V}$ are not a basis (they are linearly dependent), then certainly we can represent a projection onto their span as $\hat{\mathbf{x}} = \mathbf{V}\mathbf{b}$, but the $\mathbf{b}$ will not be unique so we do not talk of it as a "representation." If we would like to talk about "representations," we'll need to find a basis, or an orthonormal basis for that span

Let's begin. Be sure to:

- Run all cells
- Remove semicolons if you'd like to see the output of a cell where I've placed a semicolon
- Use the variable names I specify in the questions

Also, I'll repeat again that the examples I provide in this part are super important and provide almost all of the code you'll need to write, so be sure to understand the preliminaries before moving on.

In [2]:
```
using Plots
using LinearAlgebra
using DelimitedFiles
```

In order to make the code you write more elegant and simple, we'll represent a collection of vectors in two ways:

- As a matrix whose columns are those vectors
- As a vector of vectors whose entires are those vectors

You can use the matrix representation when doing matrix calculations, and the vector of vectors representation when constructing a basis.

For your convenience, I've provided two functions for going back and forth between those representations and some examples

In [3]:
```
function toMatrix(V)
    return [V[j][i] for i in 1:length(V[1]), j in 1:length(V)]
end
```

Out[3]:
```
toMatrix (generic function with 1 method)
```

In [4]:
```
function toVecOfVecs(V_)
    return [V_[:,i] for i in 1:size(V_)[2]]
end
```

Out[4]:
```
toVecOfVecs (generic function with 1 method)
```

We create an orthonormal basis for some subspace of $\mathbf{R}^3$ as a collection of vectors

In [5]:
```
W = [[1/sqrt(2),0,1/sqrt(2)], [0,1,0]] # Vector whose entries are two vectors
```

Out[5]:
```
2-element Array{Array{Float64,1},1}:
 [0.7071067811865475, 0.0, 0.7071067811865475]
 [0.0, 1.0, 0.0]
```

In [6]:
```
W[1] # Gives the first vector in the basis
```

Out[6]:
```
3-element Array{Float64,1}:
 0.7071067811865475
 0.0
 0.7071067811865475
```

In [7]:
```
W[2] # Gives the second vector in the basis
```

Out[7]:
```
3-element Array{Float64,1}:
 0.0
 1.0
 0.0
```

In [8]:
```
W[1][3] # Gives the third component of the first vector in the basis
```

Out[8]:
```
0.7071067811865475
```

We use the convention that an underscore `_` denotes the matrix version of the collection of vectors:

In [9]:
```
W_ = toMatrix(W) # The vectors in W occur as the columns of W_
```

Out[9]:
```
3×2 Array{Float64,2}:
 0.707107  0.0
 0.0       1.0
 0.707107  0.0
```

In [10]:
```
W_[:,2] # Second column/vector
```

Out[10]:
```
3-element Array{Float64,1}:
 0.0
 1.0
 0.0
```

The matrix $\mathbf{W}^\mathbf{T}\mathbf{W}$ has entry i, j corresponding to the dot product of i^{th} and j^{th} column of $\mathbf{W}$ so should be diagonal if they are orthogonal and identity if they are orthonormal. You can use this as a quick sanity check:

Recall that `'` is the transpose (actually conjugate transpose, but everything is real so it's just the transpose)

In [11]:
```
W_'*W_ # Should be identity if W is an orthonormal basis
```

Out[11]:
```
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

Note that due to floating point errors, you won't get exactly identity. Instead of zeros, you might just get really small numbers like 10^{-15}

In [12]:
```
toVecOfVecs(W_) # This converts to the vector of vectors representation
```

Out[12]:
```
2-element Array{Array{Float64,1},1}:
 [0.7071067811865475, 0.0, 0.7071067811865475]
 [0.0, 1.0, 0.0]
```

In [13]:
```
toVecOfVecs(W_) == W # Should be a true statement
```

Out[13]:
```
true
```

Since the columns of $\mathbf{W}$ are an orthonormal basis, the representation of the projection of some $\mathbf{x}$ onto the span of the columns of $\mathbf{W}$ can be obtained as $\mathbf{a} = \mathbf{W}^{\mathbf{T}}\mathbf{x}$ and the projection is $\hat{\mathbf{x}} = \mathbf{W}\mathbf{a}$

Let's consider projecting

$$\mathbf{x} = \begin{bmatrix} 5 \\ 3 \\ 5 \end{bmatrix}$$

which is in the span of the columns of $\mathbf{W}$ so its projection should be itself.

In [14]:
```
x = [5,3,5]
```

Out[14]:
```
3-element Array{Int64,1}:
 5
 3
 5
```

In [15]:
```
a = [dot(w,x) for w in W] # One option
```

Out[15]:
```
2-element Array{Float64,1}:
 7.071067811865475
 3.0
```

In [16]:
```
a = W_'*x # Second option, should be same output as above
```

Out[16]:
```
2-element Array{Float64,1}:
 7.071067811865475
 3.0
```

The two cells above compute the exact same thing, a matrix-vector product

In [17]:
```
x_hat = W_*a # Projection
```

Out[17]:
```
3-element Array{Float64,1}:
 4.999999999999999
 3.0
 4.999999999999999
```

This should be the same as the original $\mathbf{x}$ (with the exception of a tiny difference due to floating point error)

In [18]:
```
norm(x-x_hat) # Should be a very tiny number
```

Out[18]:
```
1.2560739669470201e-15
```

Let's now consider projecting a vector which is not in the span, so the projection should only approximate it.

Let

$$\mathbf{x} = \begin{bmatrix} 5 \\ 3 \\ 7 \end{bmatrix}$$

In [19]:
```
x = [5,3,7]
```

Out[19]:
```
3-element Array{Int64,1}:
 5
 3
 7
```

In [20]:
```
a = W_'*x
```

Out[20]:
```
2-element Array{Float64,1}:
 8.48528137423857
 3.0
```

In [21]:
```
x_hat = W_*a
```

Out[21]:
```
3-element Array{Float64,1}:
 5.999999999999999
 3.0
 5.999999999999999
```

In [22]:
```
norm(x-x_hat)
```

Out[22]:
```
1.4142135623730951
```

To check your answers, we provide a function which computes the projection of a vector `v` onto the span of any collection of vectors (vector of vectors) `V` regardless of whether or not the set is a basis, or is an orthonormal basis.

- You are not expected to understand what this function is doing for this exercise and at this stage in the course
- You can only use it to check your answers, when you compute projections, you must do it by obtaining an orthonormal basis and doing what we did above
- It doesn't provide you with the projection coefficients, so you can't use it to check all of your answers, only the projection itself

In [23]:
```
function projGeneral(v,V) # Computes projection of v onto Span(V) where V is a vector of vectors
    V = toMatrix(V)
    Q, R = qr(V)
    Q = Q[:,1:rank(V)]
    return Q*inv(Q'*Q)*Q'*v
end
```

Out[23]:
```
projGeneral (generic function with 1 method)
```

We'll demonstrate the use of this function in the following examples:

In [24]:
```
projGeneral(x, W) # This should be the same answer you got for x_hat before
```

Out[24]:
```
3-element Array{Float64,1}:
 6.0
 3.0
 6.0
```

Let's consider augmenting our basis so that our vector

$$\mathbf{x} = \begin{bmatrix} 5 \\ 3 \\ 7 \end{bmatrix}$$

is in it.

In [25]:
```
W = [[1/sqrt(2),0,1/sqrt(2)], [0,1,0]] # This is our current W
```

Out[25]:
```
2-element Array{Array{Float64,1},1}:
 [0.7071067811865475, 0.0, 0.7071067811865475]
 [0.0, 1.0, 0.0]
```

In [26]:
```
push!(W, [-3,0,1]) # Put this vector into our basis
```

Out[26]:
```
3-element Array{Array{Float64,1},1}:
 [0.7071067811865475, 0.0, 0.7071067811865475]
 [0.0, 1.0, 0.0]
 [-3.0, 0.0, 1.0]
```

In [27]:
```
W_ = toMatrix(W) # Update our matrix version
```

Out[27]:
```
3×3 Array{Float64,2}:
 0.707107  0.0  -3.0
 0.0       1.0   0.0
 0.707107  0.0   1.0
```

In [28]:
```
rank(W_)
```

Out[28]:
```
3
```

The above cell tells our columns are linearly independent and hence must be a basis for $\mathbf{R}^3$

In [29]:
```
W_'*W_
```

Out[29]:
```
3×3 Array{Float64,2}:
  1.0      0.0  -1.41421
  0.0      1.0   0.0
 -1.41421  0.0  10.0
```

The above cell tells us that our basis is not orthogonal nor orthonormal so we can't use the method of projection that we used before. Let's cheat for now and use the projection function, the answer should be $\mathbf{x}$ itself.

In [30]:
```
x
```

Out[30]:
```
3-element Array{Int64,1}:
 5
 3
 7
```

In [31]:
```
x_hat = projGeneral(x,W)
```

Out[31]:
```
3-element Array{Float64,1}:
 5.0
 3.0
 6.999999999999999
```

In [32]:
```
norm(x-x_hat) # Should be tiny
```

Out[32]:
```
8.881784197001252e-16
```

Part 1: Projecting a vector onto a plane

In this part, you'll be doing the most important part of this exercise which is writing a function to obtain orthonormal bases.

Consider the following collection of vectors:

In [33]:
```
V = Array{Float64,1}[] # Create empty vector of vectors of floats
push!(V, [1, -1, 1])
push!(V, [1, 0, 1])
push!(V, [0, 1, 0])
push!(V, [0, 0, 0])
```

Out[33]:
```
4-element Array{Array{Float64,1},1}:
 [1.0, -1.0, 1.0]
 [1.0, 0.0, 1.0]
 [0.0, 1.0, 0.0]
 [0.0, 0.0, 0.0]
```

In [34]:
```
for v in V
    println(v)
end
```

```
[1.0, -1.0, 1.0]
[1.0, 0.0, 1.0]
[0.0, 1.0, 0.0]
[0.0, 0.0, 0.0]
```

In [35]:
```
length(V) # There are four vectors in the collection
```

Out[35]:
```
4
```

In [36]:
```
V_ = toMatrix(V) # Get the matrix form
```

Out[36]:
```
3×4 Array{Float64,2}:
  1.0  1.0  0.0  0.0
 -1.0  0.0  1.0  0.0
  1.0  1.0  0.0  0.0
```

In [37]:
```
rank(V_) # Not 4 so set is linearly dependent
```

Out[37]:
```
2
```

The above cell tells us that the columns of `V` are linearly dependent and span a 2-dimensional subspace which tells us that if we compute an orthonormal basis for this subspace, the orthonormal basis should end up containing only 2 vectors

We seek to project the following vector `v` onto the span of the columns of `V`.

In [38]:
```
v = [4,7,7]
```

Out[38]:
```
3-element Array{Int64,1}:
 4
 7
 7
```

We can obtain an orthonormal basis $\mathbf{W}$ for a collection of vectors $\mathbf{V}$ using the Gram-Schmidt procedure. The following is pseudocode for the algorithm:

initialize

- Let $\mathbf{v}$ be a nonzero element of $\mathbf{V}$
- Let $\mathbf{W} = \{\mathbf{v}/||\mathbf{v}||\}$
- Remove $\mathbf{v}$ from $\mathbf{V}$

while ($\mathbf{V}$ is not empty)

- Take an element $\mathbf{v}$ out of $\mathbf{V}$ (remove it)
- Obtain the projection of $\mathbf{v}$ onto the span of the vectors in $\mathbf{W}$ (which are always orthonormal) and let it be $\hat{\mathbf{v}}$
- Let $\mathbf{e} = \mathbf{v} - \hat{\mathbf{v}}$

if ($\mathbf{e}$ is nonzero)

- Normalize $\mathbf{e}$ and add it to $\mathbf{W}$

endif

endwhile

When you implement the algorithm, instead of checking if $\mathbf{e}$ is a nonzero vector, check if its norm is greater than 10^{-7}. This is because it won't be exactly zero due to floating point error.

Exercise: Implement the algorithm described above by completing the following function. The code for the initalization has already been written for you. Once it is implemented, run the cell that follows this to obtain the orthonormal basis.

In [39]:
```
function GramSchmidt(V)

    V = copy(V) # Make a local copy so we don't modify original
    W = Array{Float64,1}[] # Create empty vector of vectors of floats

    # Get a nonzero element
    v = pop!(V) # pop! removes v from the list! We now have a copy in v
    while v == zeros(length(V[1])) # If the element is zero, keep popping unti
l we get a nonzero element
        v = pop!(V)
    end
    push!(W, v/norm(v))

    # BEGIN SOLUTION

    while length(V) > 0
        v = pop!(V)
        W_ = toMatrix(W)
        a = W_'*v
        v_hat = W_*a

        e = v - v_hat
        if norm(e) > 1e-7
            push!(W, e/norm(e))
        end
    end

    return W
end
```

Out[39]:
```
GramSchmidt (generic function with 1 method)
```

Run the following cell to compute your orthonormal basis. Feel free to create extra cells to verify correctness of your implementation.

In [40]:
```
W = GramSchmidt(V)
```

Out[40]:
```
2-element Array{Array{Float64,1},1}:
 [0.0, 1.0, 0.0]
 [0.7071067811865475, 0.0, 0.7071067811865475]
```

Exercise: Compute the projection of `v` on to the span of the columns of `V` using the orthonormal basis computed. Store the projection in a variable called `v_hat` and run the code at the end to check your answer.

In [41]:
```
# BEGIN SOLUTION
W_ = toMatrix(W)
a = W_'*v
v_hat = W_*a
```

Out[41]:
```
3-element Array{Float64,1}:
 5.499999999999999
 7.0
 5.499999999999999
```

Answer check:

In [42]:
```
v_hat_ref = projGeneral(v,V)
```

Out[42]:
```
3-element Array{Float64,1}:
 5.5
 7.0
 5.5
```

In [43]:
```
norm(v_hat_ref - v_hat)
```

Out[43]:
```
1.2560739669470201e-15
```

In [44]:
```
@assert norm(v_hat_ref - v_hat) < 10^-7
```

Part 2: Projections versus Taylor series

In this exercise, we consider the problem of approximating a sine function on the interval $(-\pi, \pi)$ using a degree 5 polynomial.

We start by creating a basis containing the functions $1, x, x^2, x^3, x^4, x^5$ evaluated at a discrete set of points to form vectors

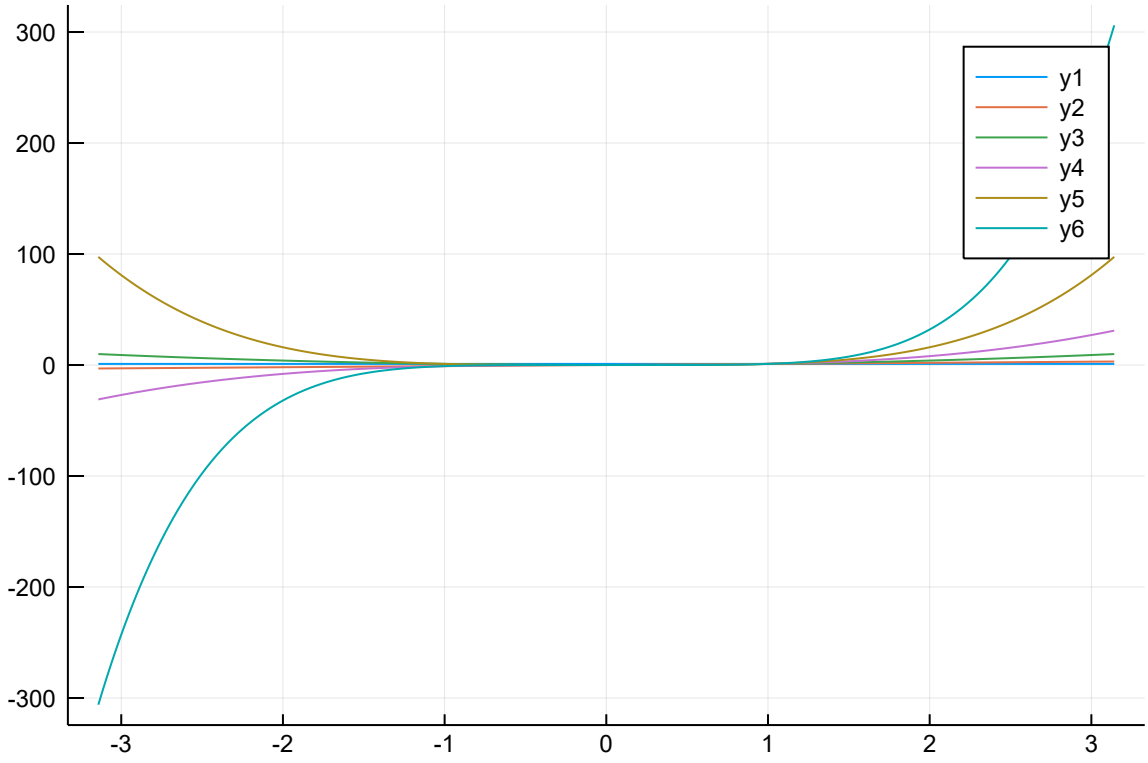
In [45]:
```
x = [i for i in LinRange(-π,π,1000)];
```

In [46]:
```
V = Array{Float64,1}[]
push!(V, x.^0)
push!(V, x.^1)
push!(V, x.^2)
push!(V, x.^3)
push!(V, x.^4)
push!(V, x.^5);
```

In [47]:
```
V_ = toMatrix(V);
```

In [48]:
```
p1 = plot()

for v in V
    plot!(p1, x, v)
end

plot(p1)
```

Out[48]:



In [49]:
```
rank(V_)
```

Out[49]:
```
6
```

The above cell tells us that these polynomials are linearly independent and hence are a basis. However, they are not orthonormal as can be seen by the following cell.

In [50]:
```
V_'*V_
```

Out[50]:
```
6×6 Array{Float64,2}:
  1000.0          -1.13687e-13  …  19559.9          1.45519e-11
    -1.13687e-13   3296.45            1.45519e-11   1.38167e5
  3296.45           -2.27374e-12      1.38167e5     1.74623e-10
    -1.81899e-12  19559.9             1.74623e-10   1.06274e6
 19559.9             1.45519e-11      1.06274e6    -1.16415e-9
     1.45519e-11     1.38167e5    …    -1.16415e-9    8.59891e6
```

A natural choice of coefficients for representing the sinusoid is the Taylor series coefficients:

In [51]:
```
b_Taylor = [0, 1, 0, -1/factorial(3), 0, 1/factorial(5)]
```

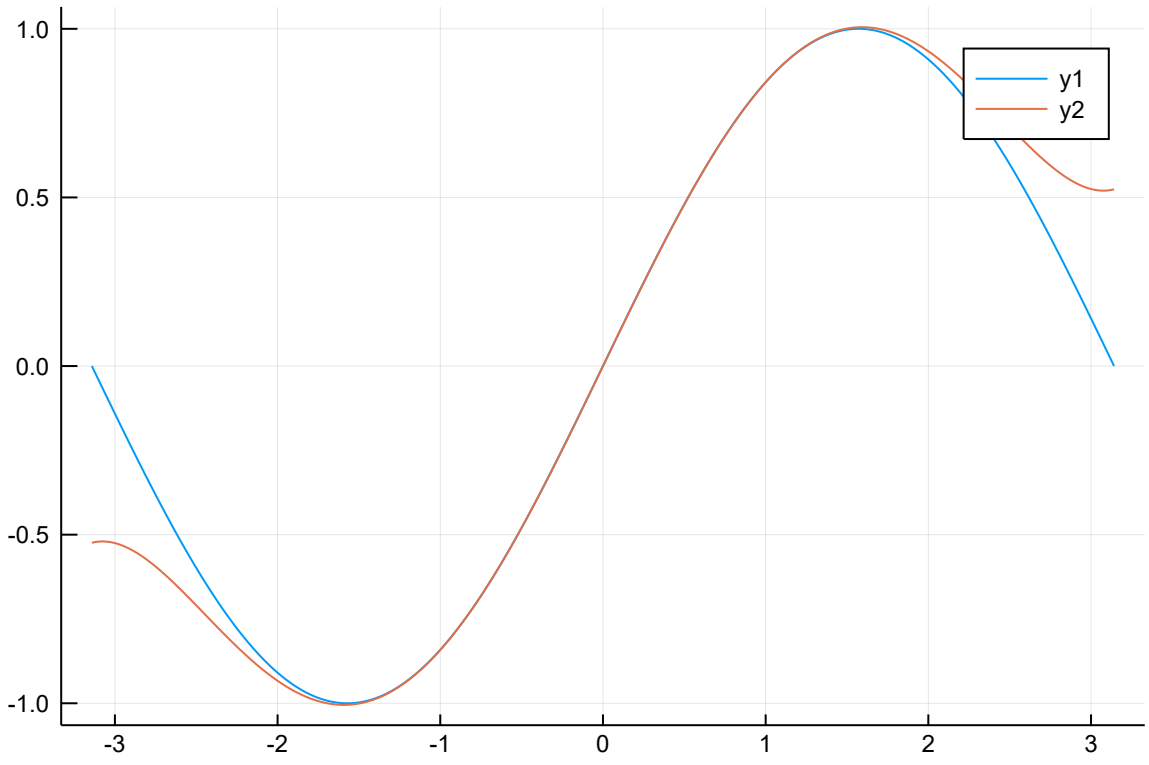
Out[51]:
```
6-element Array{Float64,1}:
  0.0
  1.0
  0.0
 -0.16666666666666666
  0.0
  0.008333333333333333
```

In [52]:
```
y_Taylor = V_*b_Taylor;
```

In [53]:
```
y = sin.(x);
```

In [54]:
```
plot(x,y)
plot!(x, y_Taylor)
```

Out[54]:



In [55]:
```
norm(y - y_Taylor)
```

Out[55]:
```
4.375951363292458
```

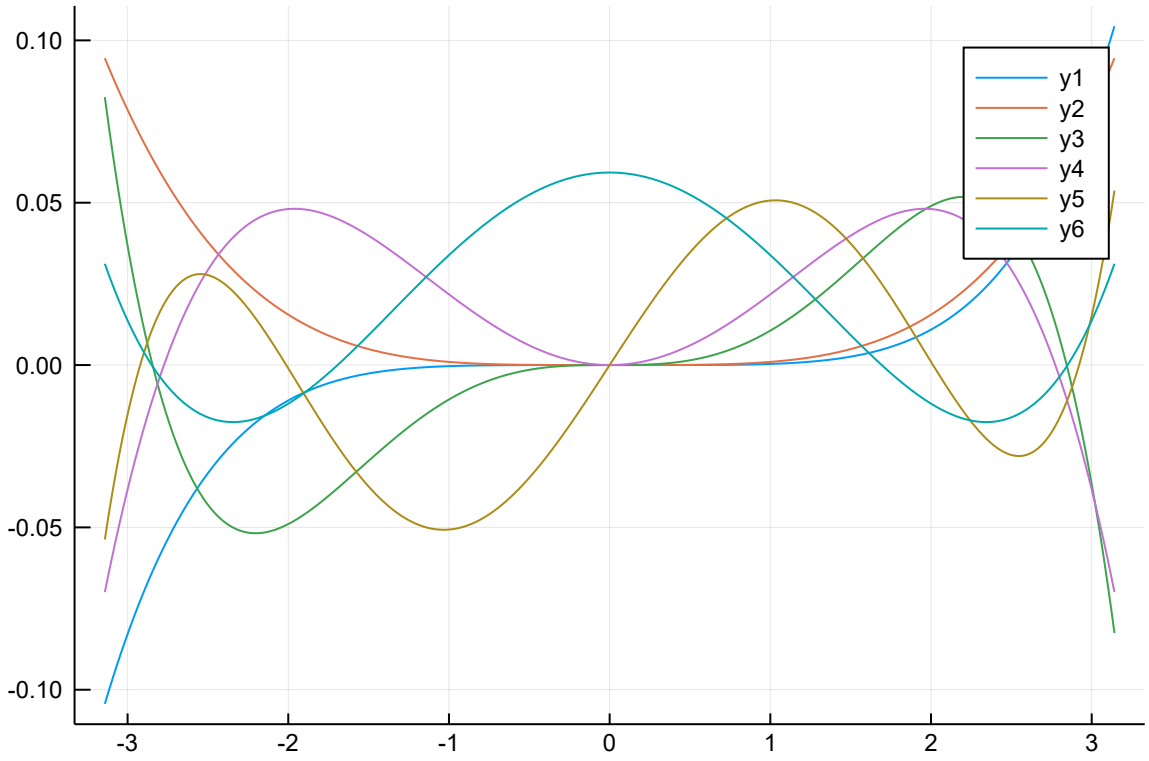
From the plot, you can see that the approximation gets considerably worse away from zero. There's no reason to believe that a truncated Taylor series is the best approximation on this interval. Instead, we will compute the closest degree 5 polynomial approximation in the sense of Euclidean distance by projecting y onto the span of V which is the 6-dimensional space of degree 5 polynomials.

Exercise: Compute an orthonormal basis for the span of the columns of `V` using your Gram-Schmidt function and store the result in a variable called `W` as in the previous exercise.

In [56]:
```
# BEGIN SOLUTION
W = GramSchmidt(V);
```

In [57]:
```
p2 = plot()

for w in W
    plot!(p2, x, w)
end

plot(p2)
```

Out[57]:

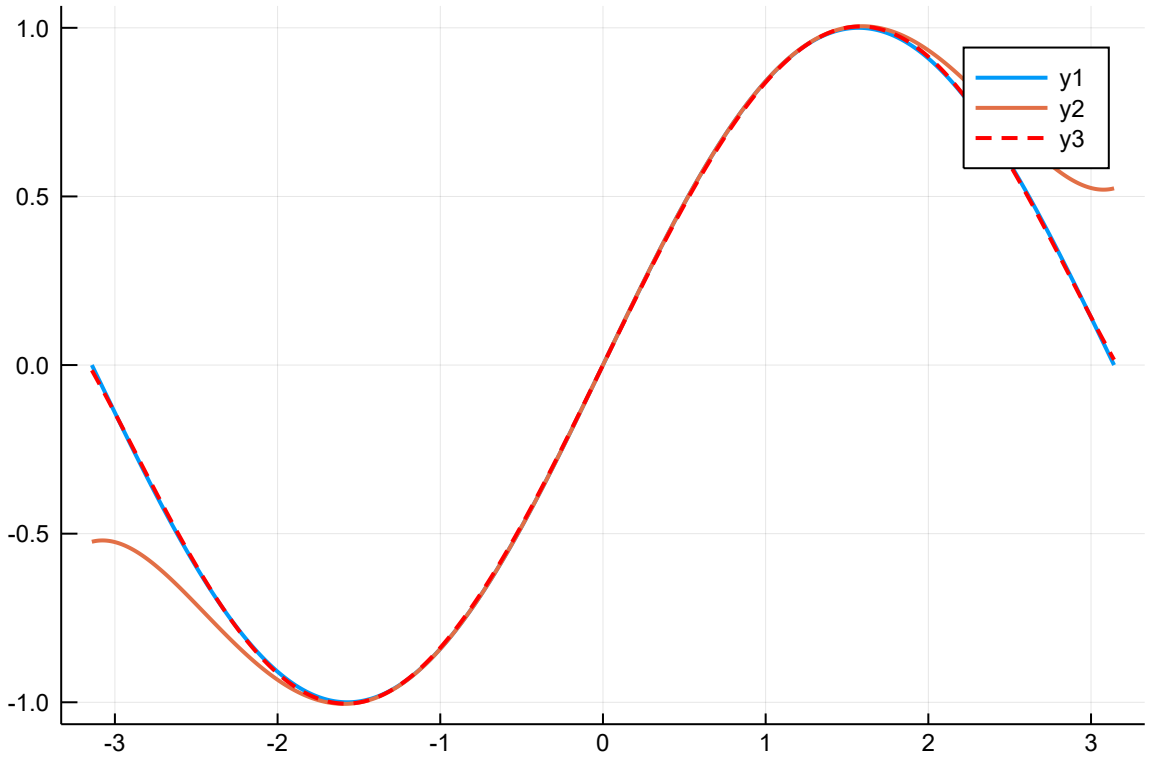


Exercise: Compute the projection of `y` onto the span of the columns of `V` and store the projection in a variable called `y_hat`. Store the representation in the orthonormal basis (projection coefficients) in a variable called `a`.

In [58]:
```
# BEGIN SOLUTION
W_ = toMatrix(W)
a = W_'*y
y_hat = W_*a;
```

In [59]:
```
plot(x,y,linewidth=2)
plot!(x, y_Taylor,linewidth=2)
plot!(x, y_hat, linestyle=:dash, linewidth=2, linecolor=:red)
```

Out[59]:



In [60]:
```
norm(y - y_hat)
```

Out[60]:
```
0.1368389425900383
```

Answer check:

In [61]:
```
y_hat_ref = projGeneral(y, V);
```

In [62]:
```
norm(y_hat-y_hat_ref)
```

Out[62]:
```
3.14660868661924e-13
```

In [63]:
```
@assert norm(y_hat_ref-y_hat) < 10^-7
```

You can see that this approximation via projection is considerably better and almost indistinguishable from sine, at least visually.

We would like to obtain the representation of the projection in terms of the original non-orthogonal basis $\mathbf{V}$. You won't have to write any code in this last bit of Part 1, but you should understand all the code I've provided because you'll need to do the same thing yourself later.

We seek a matrix $\mathbf{B}$ such that $\mathbf{VB} = \mathbf{W}$. Let $\mathbf{w_i}$ be the i^{th} column of $\mathbf{W}$ and $\mathbf{b_i}$ be the i^{th} column of $\mathbf{B}$. To obtain $\mathbf{b_i}$, we solve $\mathbf{V b_i} = \mathbf{w_i}$ for $\mathbf{b_i}$. We know that this system has a solution because both $\mathbf{V}$ and $\mathbf{W}$ have columns forming a basis for the same space.

To solve a system $\mathbf{Mu} = \mathbf{v}$ numerically for $\mathbf{u}$, we use `u = M\v`

In [64]:
```
B = [V_\w for w in W];
B_ = toMatrix(B)
```

Out[64]:
```
6×6 Array{Float64,2}:
  5.38218e-20  -3.00651e-19  -5.36005e-19  …  -7.34641e-18   0.0592928
 -2.47059e-19   3.55804e-19  -1.80009e-19      0.0762006    -1.50387e-16
 -7.31128e-20   4.50344e-19   7.03493e-19      4.3491e-17   -0.0279798
  1.53567e-19  -1.78146e-19   0.0121064       -0.0277394     4.93221e-17
  1.1431e-20    0.000970033  -3.11395e-20     -5.43746e-18   0.00254637
  0.000341019   1.53134e-19  -0.00149623   …   0.00220393   -3.29032e-18
```

In [65]:
```
sum(V_*B_ - W_) # Should be tiny
```

Out[65]:
```
7.036677698824002e-15
```

In [66]:
```
y_hat = W_*a;
```

In [67]:
```
y_hat_2 = V_*B_*a; # Should be the same as y_hat
```

In [68]:
```
norm(y_hat_2 - y_hat) # Should be tiny
```

Out[68]:
```
4.8437298154297325e-14
```

Since the projection onto our subspace can be obtained as $\mathbf{Wa} = \mathbf{VBa}$, the coefficients corresponding to the representation in terms of the non-orthogonal basis $\mathbf{V}$ are given by $\mathbf{b} = \mathbf{Ba}$

In [69]:
```
a
```

Out[69]:
```
6-element Array{Float64,1}:
  6.819240026774798
 -1.0947731436672381e-16
 16.880089163686364
 -2.3761239286920754e-15
 12.963094938594537
 -2.4210017671166817e-14
```

In [70]:
```
b_opt = B_*a
```

Out[70]:
```
6-element Array{Float64,1}:
 -1.539393758353675e-15
  0.9877956920902593
  1.1930805593746403e-15
 -0.15523104437649285
 -1.2495679206201072e-16
  0.005638621981537147
```

You can verify that the coefficients above correspond to the approximation
$\hat{y} = 0.987862x - 0.155271x^3 + 0.00564312x^5$

In [71]:
```
b_Taylor
```

Out[71]:
```
6-element Array{Float64,1}:
  0.0
  1.0
  0.0
 -0.16666666666666666
  0.0
  0.008333333333333333
```

In [72]:
```
b_opt_ref = [0, 0.987862, 0, -0.155271, 0, 0.00564312];
```

In [73]:
```
norm(b_opt-b_opt_ref) < 10^-4
```

Out[73]:
```
true
```

In [74]:
```
@assert norm(b_opt-b_opt_ref) < 10^-4
```

Let's compare the error to the original choice of Taylor series coefficients

In [75]:
```
norm(y - V_*b_Taylor)
```

Out[75]:
```
4.375951363292458
```

In [76]:
```
norm(y - V_*b_opt)
```

Out[76]:
```
0.13683894259003784
```

In [77]:
```
@assert norm(y - V_*b_opt) < 0.15
```

Exercise: Both representations `a` and `b` should have half of their entries as zero (close to zero). Why is this case? (Hint: Look at the plots of the two bases)

Answer: half of the functions are even and sine is odd, the functions are orthogonal and the projection is zero

Part 3: Projections versus Taylor Swift

In this part, we project Google Trends data on the popularity of Taylor Swift onto a basis of bump functions (https://en.wikipedia.org/wiki/Bump_function (https://en.wikipedia.org/wiki/Bump_function)).

We start by loading the data which consists of year-months and a popularity score normalized to lie between 0 and 100

In [205]:
```
TS = readdlm("TaylorSwift.csv", ',', '\n');
```

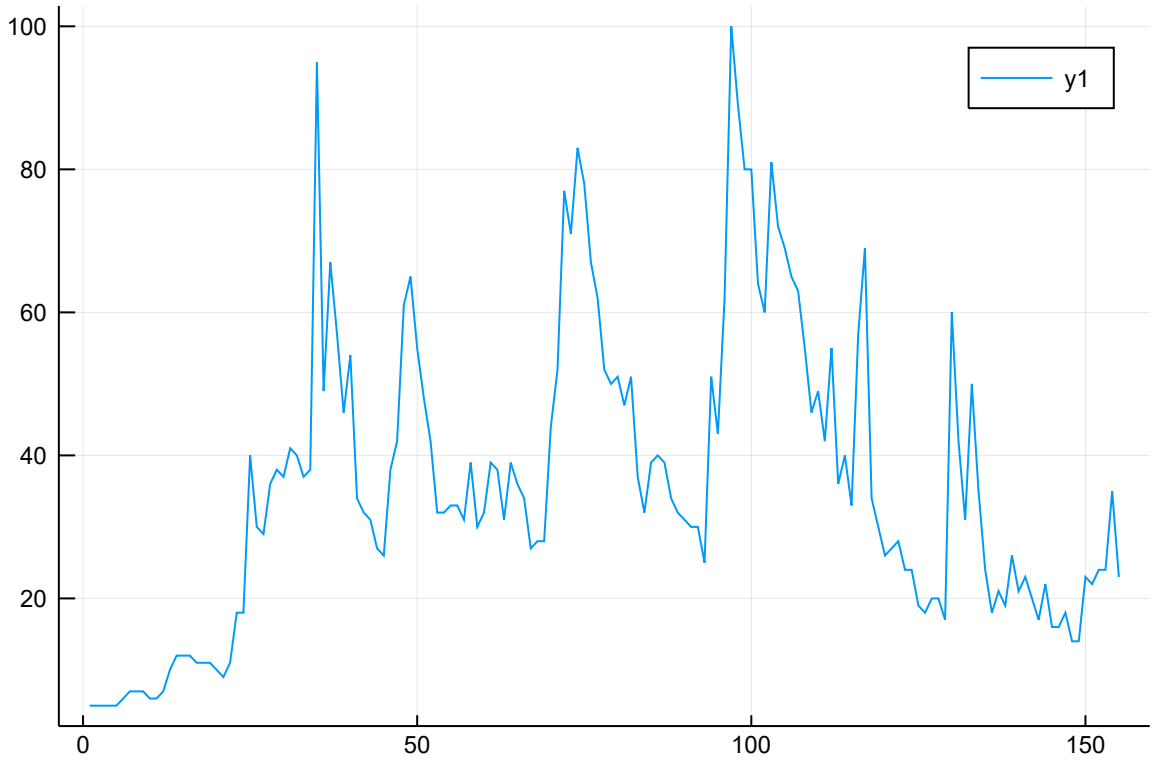
In [206]:
```
Times = TS[:,1]; # We store the dates in this variable
```

In [207]:
```
y = TS[:,2];
y = convert(Array{Float64,1}, y); # We store the popularity vector in this var
iable
```

Each point in the vector `y` contains a popularity score from 0 to 100 and the indices of the components of the vector correspond to month numbers starting from November 2006 and ending at September 2019

In [208]:
```
x = [i for i in 1:length(y)]
plot(x,y)
```

Out[208]:



We define the bump function provided here https://en.wikipedia.org/wiki/Bump_function (https://en.wikipedia.org/wiki/Bump_function) along with a scalling factor α. Larger α makes the bump narrower and smaller α makes it wider.

$$f_\alpha(x) = \begin{cases} \exp\left(-\frac{1}{1-(\alpha x)^2}\right) & \text{for} \quad |\alpha x| < 1 \\ 0 & \text{otherwise} \end{cases}$$

In [209]:
```
f(x, α) = abs(α*x) < 1 ? exp(-1.0/(1.0-(α*x)^2)) : 0 # This is fancy syntax fo
r an if-else statement
```

Out[209]:
```
f (generic function with 1 method)
```

We would like to convert this data set into just yearly data summarizing the popularity Taylor Swift from the years 2007 to 2018.

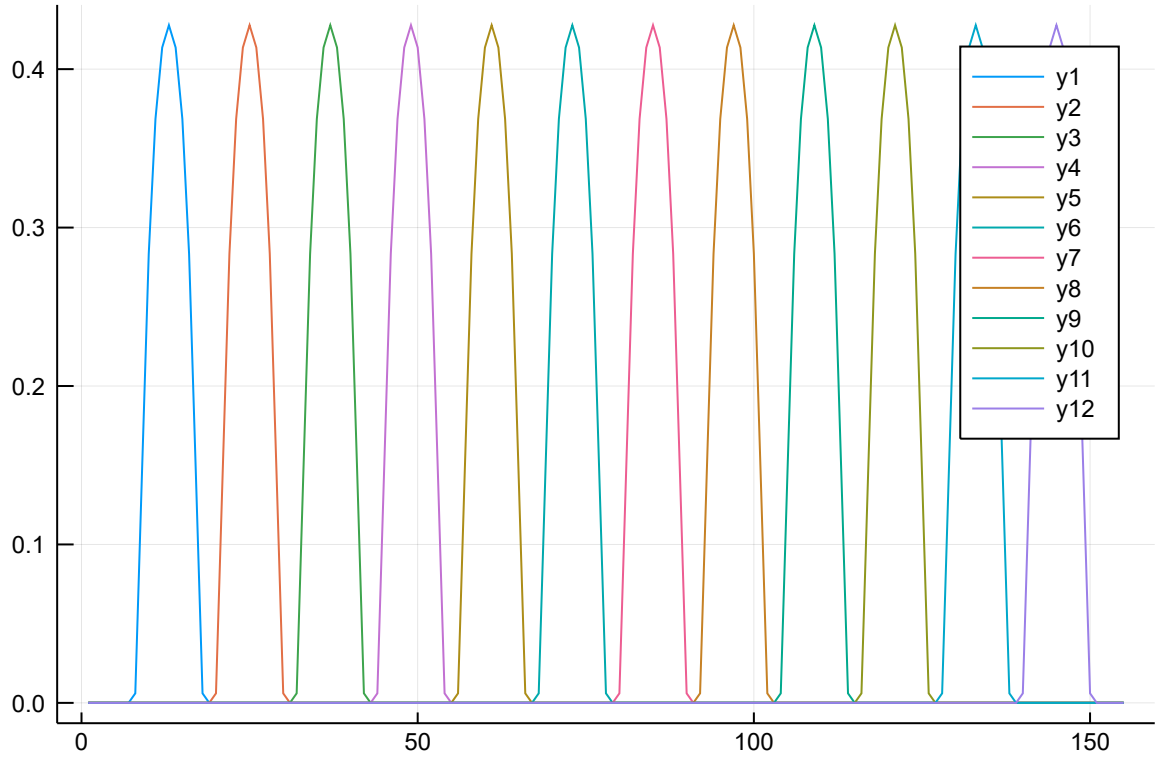
We will find a low-dimensional representation of this 155-dimensional vector of data by crudely modelling her popularity as a superposition of shifted bump functions where the bumps peak every November when she releases an album. This appears to be the case according to https://en.wikipedia.org/wiki/Taylor_Swift_discography (https://en.wikipedia.org/wiki/Taylor_Swift_discography).

We construct an orthonormal basis of 12 bump functions in the following cells and plot them. Their peaks occur every November.

In [210]:
```
V = Array{Float64,1}[]

N = 12

α = 0.18

for i = 1:N
    basis_vector = f.(x .- (1+12*i), α)
    basis_vector = basis_vector./norm(basis_vector)
    push!(V, basis_vector)
end
```

In [211]:
```
p3 = plot()

for v in V
    plot!(p3, x, v)
end

plot(p3)
```

Out[211]:



The following vector contains the dates corresponding to the indices at which the bump basis functions are centered.

In [212]:
```
Times_of_basis_vecs = [Times[1+12*i] for i = 1:N]
```

Out[212]:
```
12-element Array{SubString{String},1}:
 "2007-11"
 "2008-11"
 "2009-11"
 "2010-11"
 "2011-11"
 "2012-11"
 "2013-11"
 "2014-11"
 "2015-11"
 "2016-11"
 "2017-11"
 "2018-11"
```

Exercise: Compute the projection of `y` onto the span of the columns of `V` and store the projection in a variable called `y_hat`. Store the representation in the orthonormal basis (projection coefficients) in a variable called `a`. Note that $\mathbf{V}$ is already an orthonormal basis so you don't need to use your Gram-Schmidt. You can verify the orthonormality of our basis using checks like those we did in Part 0.

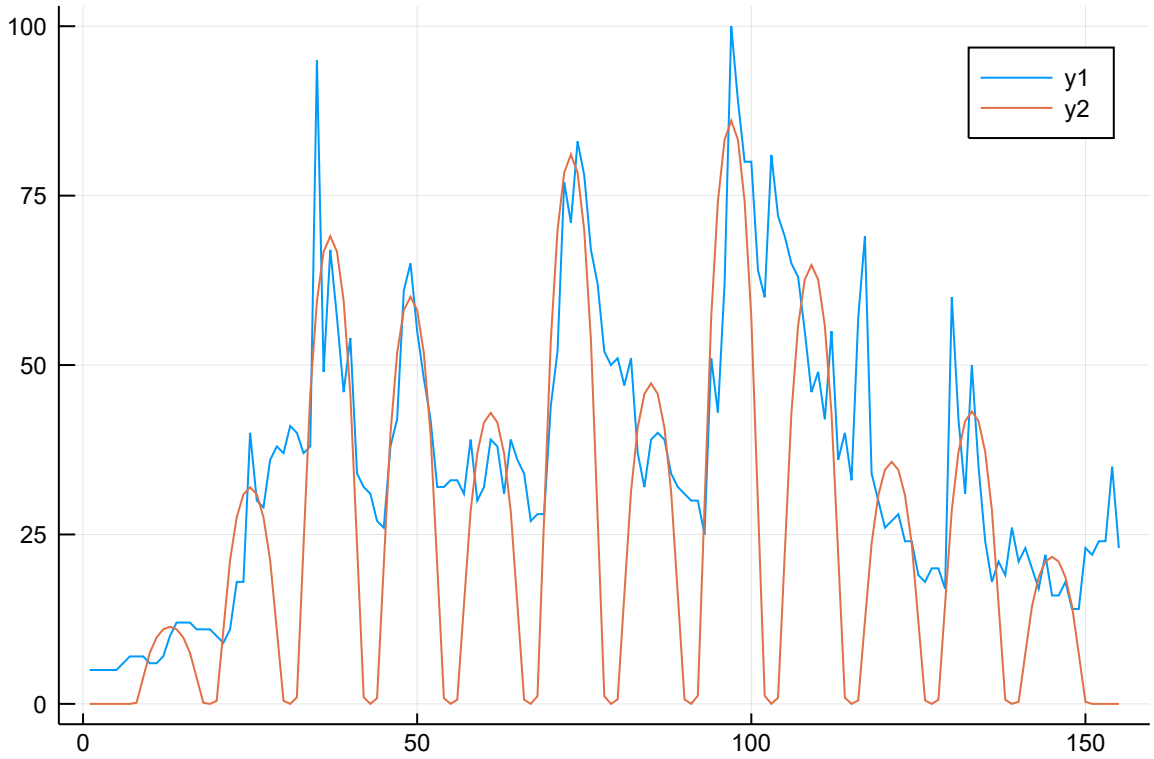
In [213]:
```
# BEGIN SOLUTION
V_ = toMatrix(V)
a = V_'*y
y_hat = V_*a
```

Out[213]:
```
155-element Array{Float64,1}:
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.16019159912892542
  3.8781616855911714
  7.539390646991026
  9.804840088105246
 11.004324126899915
 11.37904162155329
  ⋮
 20.9942873689552
 21.70918150277095
 20.9942873689552
 18.705885799305637
 14.383812399965512
  7.3988406695095925
  0.30561699450346635
  0.0
  0.0
  0.0
  0.0
  0.0
```

Plot the projection by running the following code:

In [214]:
```
plot(x,y)
plot!(x,y_hat)
```

Out[214]:



In [215]:
```
norm(y-y_hat) # This should be pretty bad
```

Out[215]:
```
270.1886245435591
```

The following code will normalize the low-dimensional representation a so that the entries lie between 0 and 100 and will then print entries and the corresponding dates sorted by the magnitudes of the entries.

In [216]:
```
a = a./maximum(a)*100;
```

In [217]:
```
for i in sortperm(a, rev=true)
    println("Date: $(Times_of_basis_vecs[i])  Popularity: $(a[i])")
end
```

```
Date: 2014-11  Popularity: 100.0
Date: 2012-11  Popularity: 94.19309940106744
Date: 2009-11  Popularity: 80.1796226588652
Date: 2015-11  Popularity: 75.23687594743443
Date: 2010-11  Popularity: 69.79749750922953
Date: 2013-11  Popularity: 54.954940507637495
Date: 2017-11  Popularity: 50.14298877962035
Date: 2011-11  Popularity: 49.895934114865454
Date: 2016-11  Popularity: 41.50186735594133
Date: 2008-11  Popularity: 37.1460876809347
Date: 2018-11  Popularity: 25.225705246053842
Date: 2007-11  Popularity: 13.222255748851868
```

From Billboard charts (https://www.billboard.com/music/taylor-swift/chart-history/hot-100 (https://www.billboard.com/music/taylor-swift/chart-history/hot-100)), we can see that Taylor Swift's number 1 and number 2 hits between 2007 and 2018 which is range covered by our bump function peaks, occured in the years 2014, 2015, 2012, 2017, 2009, 2013, 2010.

These years should coincide with the top 7 years according to our projection coefficients.

However, perhaps a naive approach might have gotten us the same results, let's check.

Here, we obtain a low-dimensional representation by simply picking out the data points for November and ignoring the remaining data.

In [218]:
```
a_naive = y[[1+12*i for i = 1:N]];
for i in sortperm(a_naive, rev=true)
    println("Date: $(Times_of_basis_vecs[i])  Popularity: $(a_naive[i])")
end
```

```
Date: 2014-11  Popularity: 100.0
Date: 2012-11  Popularity: 71.0
Date: 2009-11  Popularity: 67.0
Date: 2010-11  Popularity: 65.0
Date: 2017-11  Popularity: 50.0
Date: 2015-11  Popularity: 46.0
Date: 2008-11  Popularity: 40.0
Date: 2011-11  Popularity: 39.0
Date: 2013-11  Popularity: 39.0
Date: 2016-11  Popularity: 27.0
Date: 2018-11  Popularity: 16.0
Date: 2007-11  Popularity: 10.0
```

Exercise: This approach seems to "wrongly" squeeze 2008 into the top 7. Examining the plotted data and projection, why might this be the case?

Answer: 2013 is an anomoly in that Taylor's popularity dips in November but is high in the surrounding months, The naive approach ranks solely on the popularity during November and does not attempt to fit to the months before and after

Here, we consider taking the average of the data points in the 6 months before and after every November as the popularity for the month of November for any particular year. This way, we don't ignore the data within the years.

In [219]:
```
a_less_naive = y[1+6:144+6]
a_less_naive = reshape(a_less_naive,(12,12))
a_less_naive = sum(a_less_naive, dims = 1)/12
a_less_naive = a_less_naive[:]
a_less_naive = a_less_naive./maximum(a_less_naive)*100
for i in sortperm(a_less_naive)
    println("Date: $(Times_of_basis_vecs[i])  Popularity: $(a_less_naive[i])")
end
```

```
Date: 2007-11  Popularity: 15.126050420168067
Date: 2018-11  Popularity: 32.212885154061624
Date: 2008-11  Popularity: 40.19607843137255
Date: 2017-11  Popularity: 50.0
Date: 2016-11  Popularity: 54.48179271708683
Date: 2011-11  Popularity: 58.123249299719895
Date: 2013-11  Popularity: 67.64705882352942
Date: 2010-11  Popularity: 69.88795518207283
Date: 2009-11  Popularity: 82.6330532212885
Date: 2012-11  Popularity: 93.69747899159664
Date: 2015-11  Popularity: 94.2577030812325
Date: 2014-11  Popularity: 100.0
```

Exercise: These approach seems to overemphasize 2016 and puts it into the top 7. Examining the plotted data and projection again, why might this be the case?

Answer: During 2016 there is a downwards trend which has a high average due to the comedown from previous popularity at the start of the year; this sort of trend is not highly projected onto a bump function and the high start is ignored by the naive approach.

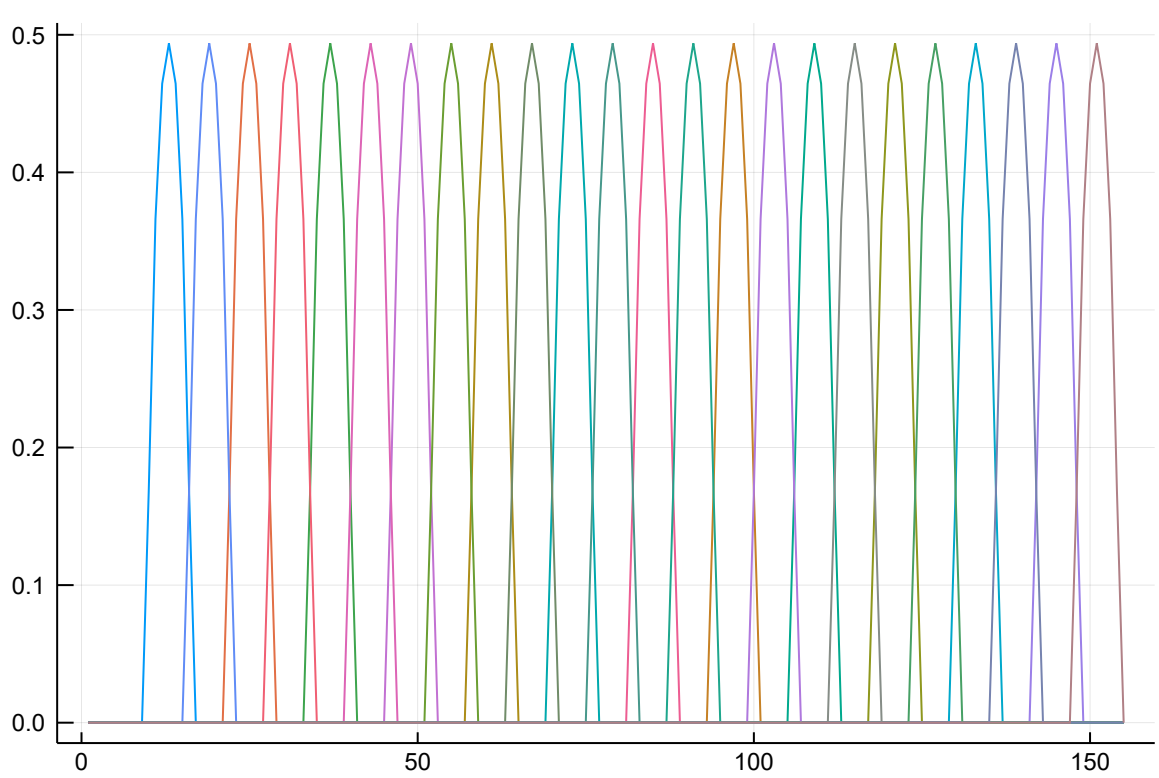
Now, we try to construct a better basis for representing the Taylor Swift data.

The following code defines a 24-dimensional non-orthonormal basis which squeezes in bump functions to lie between our previous bump functions.

In [232]:
```
V = Array{Float64,1}[]

N = 12

ϵ = 0

α = 0.24

β = 0.24 + ϵ

for i = 1:N
    basis_vector = f.(x .- (1+12*i), α)
    basis_vector = basis_vector./norm(basis_vector)
    push!(V, basis_vector)
end

for i = 1:N
    basis_vector = f.(x .- (1+6+12*i), β)
    basis_vector = basis_vector./norm(basis_vector)
    push!(V, basis_vector)
end
```

In [233]:
```
p4 = plot()

for v in V
    plot!(p4, x, v)
end

plot(p4,legend=false)
```

Out[233]:



Since the bump functions overlap, we cannot guarantee that they are orthogonal. They may or may not be orthogonal. We can check.

In [234]:
```
V_ = toMatrix(V);
V_'*V_
```

Out[234]:
```
24×24 Array{Float64,2}:
 1.0        0.0        0.0        …  0.0        0.0        0.0
 0.0        1.0        0.0           0.0        0.0        0.0
 0.0        0.0        1.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0        …  0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0283254  0.0        0.0
 0.0        0.0        0.0        …  0.0283254  0.0283254  0.0
 0.0        0.0        0.0           0.0        0.0283254  0.0283254
 0.0283254  0.0283254  0.0           0.0        0.0        0.0
 0.0        0.0283254  0.0283254     0.0        0.0        0.0
 0.0        0.0        0.0283254     0.0        0.0        0.0
 0.0        0.0        0.0        …  0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0        …  0.0        0.0        0.0
 0.0        0.0        0.0           1.0        0.0        0.0
 0.0        0.0        0.0           0.0        1.0        0.0
 0.0        0.0        0.0           0.0        0.0        1.0
```

The above matrix is not identity, so they aren't orthonormal.

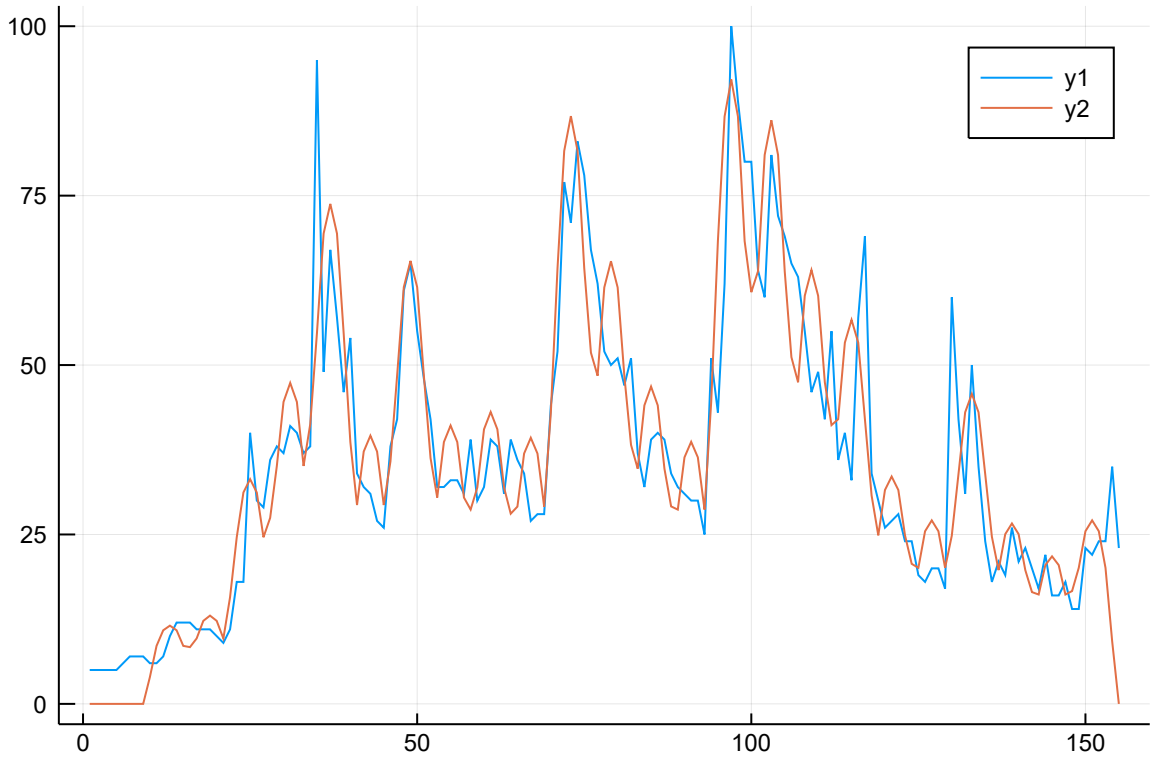
Exercise: Compute the projection of `y` onto the span of the columns of `V` and store the projection in a variable called `y_hat`. Note that you will need to obtain an orthonormal basis first this time.

In [235]:
```
# BEGIN SOLUTION
V_ortho = GramSchmidt(V)
v_ = toMatrix(V_ortho)
a = V_'*y
y_hat = V_*a
```

Out[235]:
```
155-element Array{Float64,1}:
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  9.062506003899376e-5
  3.9349977040212396
  8.558679884409399
 10.861253324204586
 11.545806050618223
  ⋮
 20.478623989394006
 21.769331191103255
 20.478623989394006
 16.13739142766102
 16.65051503333524
 20.07811560787193
 25.479588786873137
 27.08549203314787
 25.479588786873137
 20.077944736567012
  9.231174375825887
  0.00021259878530163593
```

In [236]:
```
plot(x,y)
plot!(x,y_hat)
```

Out[236]:



In [237]:
```
norm(y-y_hat)
```

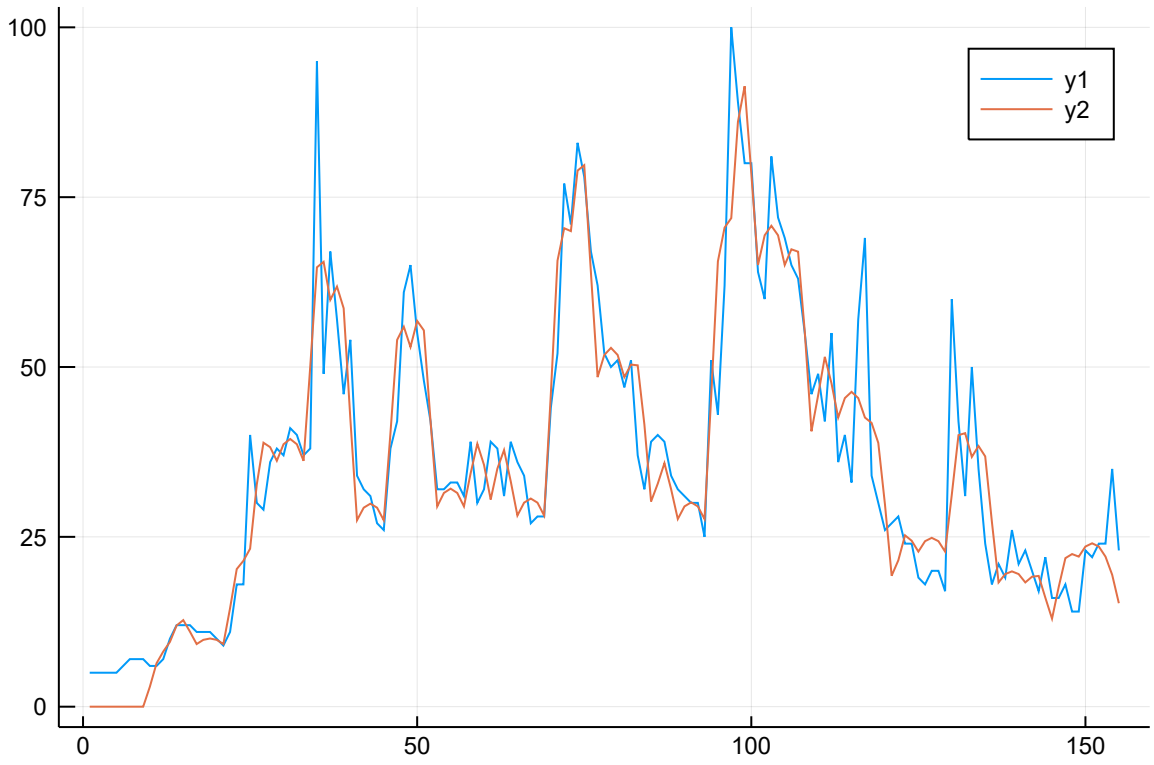
Out[237]:
```
122.8845256073332
```

In [238]:
```
V = Array{Float64,1}[]

N = 12

ϵ = -0.1

α = 0.24

β = 0.24 + ϵ

for i = 1:N
    basis_vector = f.(x .- (1+12*i), α)
    basis_vector = basis_vector./norm(basis_vector)
    push!(V, basis_vector)
end

for i = 1:N
    basis_vector = f.(x .- (1+6+12*i), β)
    basis_vector = basis_vector./norm(basis_vector)
    push!(V, basis_vector)
end
V_ = toMatrix(V);
V_'*V_
# RECOMPUTE
V_ortho = GramSchmidt(V)
V_ = toMatrix(V_ortho)
a = V_'*y
y_hat = V_*a
```

Out[238]:
```
155-element Array{Float64,1}:
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  6.765874147135439e-5
  2.9377855554764962
  6.389728286923918
  8.108780622354706
  9.533440416317578
  ⋮
 16.042246590609444
 12.954610302840607
 17.62386246698919
 21.86723054499384
 22.465488482252518
 22.089908366694036
 23.57519953719879
 24.051253901785852
 23.57519953719879
 22.08983809424877
 19.414214246659295
 15.230662651878912
```

In [230]:
```
# Recompute
V_ortho = GramSchmidt(V)
V_ = toMatrix(V_ortho)
a = V_'*y
y_hat = V_*a
plot(x,y)
plot!(x,y_hat)
```

Out[230]:



In [231]:
```
norm(y-y_hat)
```

Out[231]: 97.6896606229551

Exercise: Change the value of ϵ in the cell which constructs the basis to a nonzero (possibly negative) value and re-compute the projection. Try to find an ϵ to reduce the error to less than 110. Once you've found a choice for this value, explain why you think it's better than $\epsilon = 0$ or ϵ of a different sign.

Answer:

In [262]:
```
@assert norm(y-y_hat) < 110
```

Later in the course, you will learn how to obtain an "optimal" basis to project onto to obtain the best K-dimensional approximation of your data if that is what you're interested in rather than designing the basis based on our intuition as we did here.

However, sometimes, we'd prefer to construct our basis and subspace ourselves so that we can interpret its meaning.

Part 4: Projecting music onto a basis of notes

In this part, we consider the problem of identifying the notes in a recording of a musical chord. A chord is a bunch of notes played at once. We have recordings of three chords played on the piano each consisting of three notes played simultaneously. The `.ogg` files are audio files which you can listen to and the `.csv` files are data files containing the time-domain waveform of the sound.

We start by loading the data:

In [251]:
```
Chord = readdlm("Chord_1.csv"); # You will later have to change this to Chord_
2.csv, then Chord_3.csv
```

In [252]:
```
y = Chord[:] # This is a time-domain waveform vector
```

Out[252]:
```
7000-element Array{Float64,1}:
 -1.5297e-6
  6.2946e-6
  9.0959e-6
 -7.3607e-5
  6.8163e-5
 -1.1286e-5
 -0.0001374
  3.9927e-5
  0.00029518
  0.00014253
 -0.00021318
 -0.00067248
  0.0016772
  ⋮
 -0.003743
 -0.0019509
 -0.0011901
  0.00036732
  0.0011954
  0.0014405
  0.0010337
 -0.00041319
  0.001763
  0.0012963
 -0.00017995
  0.0013765
```

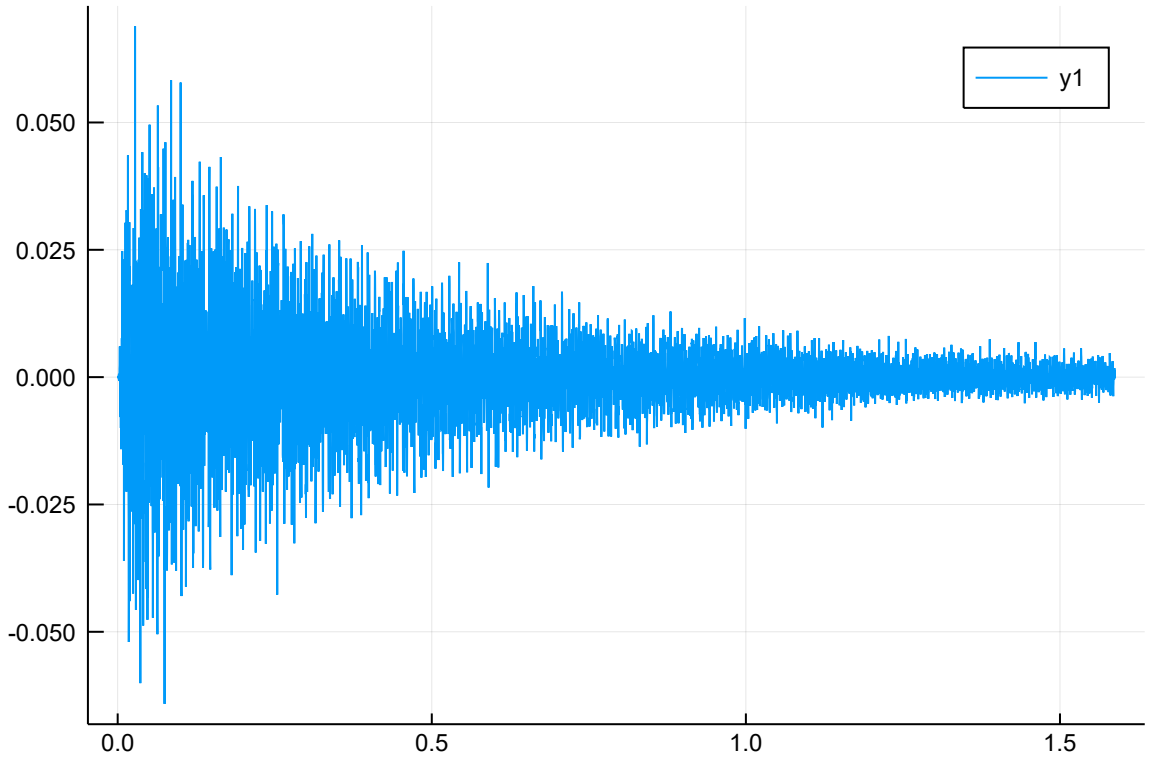
The following defines a time axis vector. The components of `y` are samples of the waveform that are 1/4410 seconds apart.

In [253]:
```
t = [i for i in 1:length(y)].*1/4410;
```

The following is a plot of the waveform:

In [254]:
```
plot(t,y)
```

Out[254]:



The chords played are restricted to a particular set of notes on the piano and our chords will consist of three notes played together. We construct a non-orthogonal basis for a 7-dimensional subspace of the 7000-dimensional space in which the recording of the chord lies. This basis is chosen to consist of cosine functions whose frequencies are those of the the notes on the piano we are restricted to.

Note that this is far from a sufficient choice of basis for accurate representation of our recording. A projection onto this basis will produce something that sounds like a synthesizer. It cannot capture variations in volume. Moreover, if it sufficed to represent the sound this way, then you wouldn't be able to tell the difference between the sound of a guitar and the sound of a piano, or the human voice or any instrument.

Nonetheless, the coefficients of the representation in this basis will be enough to tell us which 3 of the the 7 possible notes are being played.

In [255]:
```
C4 = cos.(2*π*261.626*t)
D4 = cos.(2*π*293.665*t)
E4 = cos.(2*π*329.628*t)
F4 = cos.(2*π*349.228*t)
G4 = cos.(2*π*391.995*t)
A4 = cos.(2*π*440.000*t)
B4 = cos.(2*π*493.883*t);
```

A note on the frequencies: Two consecutive notes on the piano differ by factor of $2^{1/12}$. Here, we have the frequencies of what's called a C major scale where some notes are skipped so that the difference between two consecutive notes is a factor of factor of $2^{1/12}$ or $2^{2/12}$. For the C major scale, the notes that are skipped coincide with the 5 black keys, so these notes correspond to the 7 unique white keys on the piano which are repeated. The reference frequency is that of A4 from which all others are computed. 440 Hz is called the standard concert pitch.

Moreover, the next note in this sequence is C5 which will have a frequency which is double that of C4 making it orthogonal to it since sinusoids of frequencies that are integer multiples of the same frequency are orthogonal. However, we stop short of C5, so all of the sinusoids here are not orthogonal. They are, however, linearly independent.

In [256]:
```
Basis_vec_names = ["C4", "D4", "E4", "F4", "G4", "A4", "B4"]
```

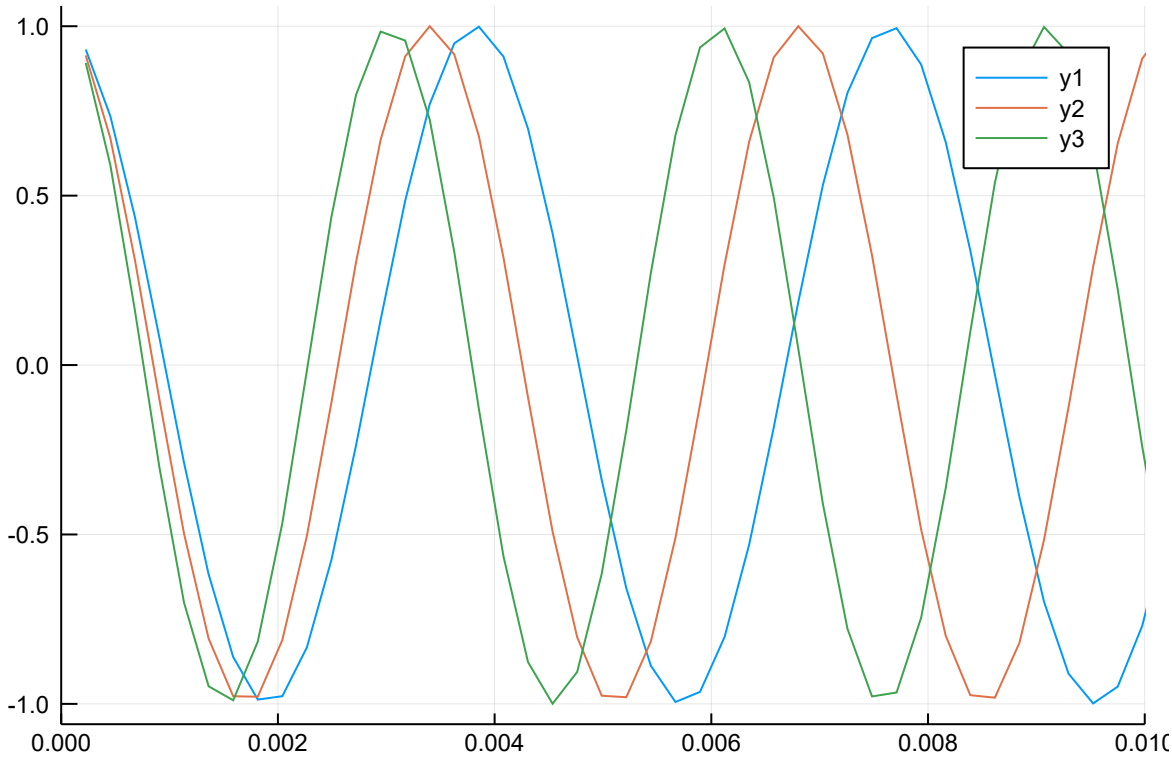
Out[256]:
```
7-element Array{String,1}:
 "C4"
 "D4"
 "E4"
 "F4"
 "G4"
 "A4"
 "B4"
```

In [257]:
```
V = [C4, D4, E4, F4, G4, A4, B4];
```

Note that the following plot is "zoomed in", i.e., the x axis range is limited, else it would just look like a blob because the period of the sinusoids is small compared to their duration

In [258]:
```
plot(t,V[1],xlim=(0,0.01))
plot!(t,V[2])
plot!(t,V[3])
```

Out[258]:



Exercise: Compute the representation of the projection of `y` onto the span of `V` and store the representation of the projection in terms of `V` in a vector called `b`. Note that you will need to do the same thing we did to obtain the approximation of the sine function in Part 2 where we obtained a change of basis matrix to convert the representation in terms of the orthonormal basis to one in terms of the non-orthogonal basis `V`. You can copy and paste code from there. Also, store your projection in a variable called `y_hat`.

In [259]:
```
# BEGIN SOLUTION
V_ = toMatrix(V)
W = GramSchmidt(V)
W_ = toMatrix(W)
B = [V_\w for w in W];
B_ = toMatrix(B)
a = W_'*y
y_hat = W_*a
b = B_*a
```

Out[259]:
```
7-element Array{Float64,1}:
  0.004159384779100875
 -1.8961780387686693e-5
  0.0020832495149037565
  0.0003596203922286806
  8.788766871434734e-8
  0.0006071806932206849
  3.04167206922945e-5
```

Exercise: By running the two code cells below which print the components of `b` sorted by magnitude, identify the three notes of each chord and enter your answers below. You will have to change the file name in the cell which loads the data and re-run your code for each file. I.e., you should identify the 3 notes of the three chords and list 9 notes in total below.

Answer

Three notes of Chord_1: C4, E4, A4

Three notes of Chord_2: F4, D4, A4

Three notes of Chord_3: C4, E4, G4

In [260]:
```
b = abs.(b)
```

Out[260]:
```
7-element Array{Float64,1}:
 0.004159384779100875
 1.8961780387686693e-5
 0.0020832495149037565
 0.0003596203922286806
 8.788766871434734e-8
 0.0006071806932206849
 3.04167206922945e-5
```

In [261]:
```
for i in sortperm(b,rev=true)
    println("Note: $(Basis_vec_names[i]) Coeff.: $(b[i])")
end
```

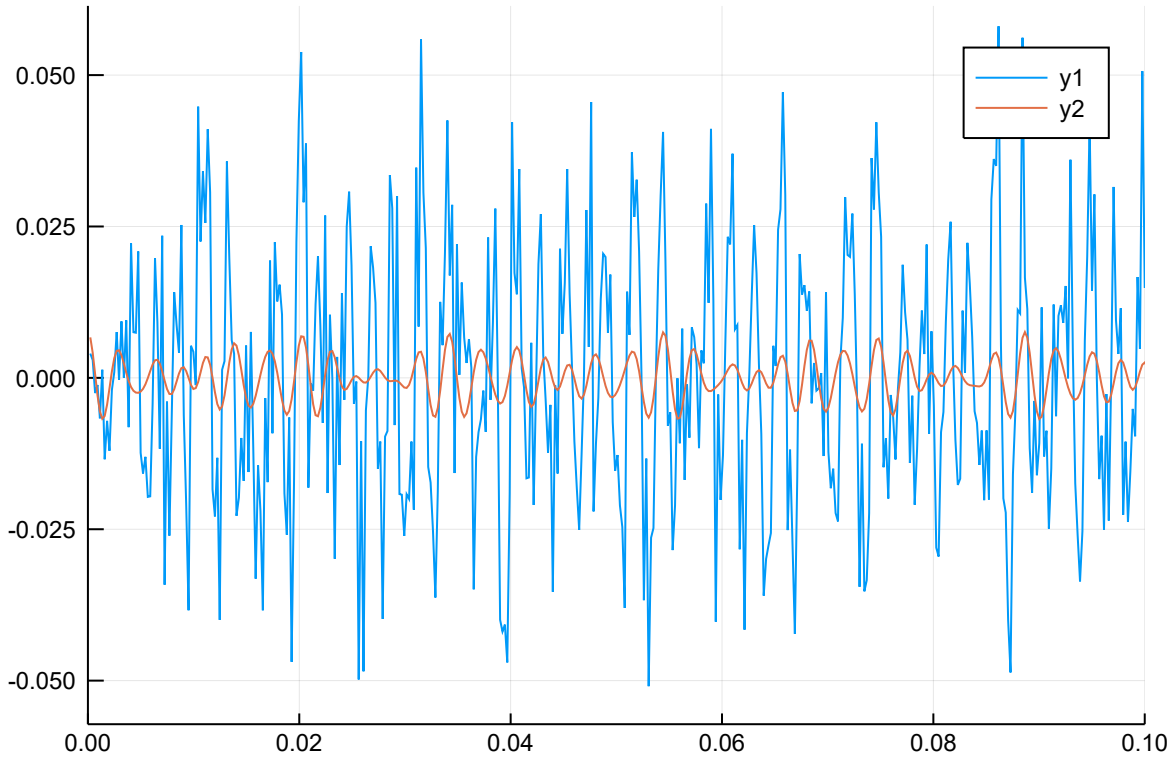
```
Note: C4 Coeff.: 0.004159384779100875
Note: E4 Coeff.: 0.0020832495149037565
Note: A4 Coeff.: 0.0006071806932206849
Note: F4 Coeff.: 0.0003596203922286806
Note: B4 Coeff.: 3.04167206922945e-5
Note: D4 Coeff.: 1.8961780387686693e-5
Note: G4 Coeff.: 8.788766871434734e-8
```

If you would like to verify your answers and you trust your music skills more than you trust your code, you can try to play the chord here https://www.apronus.com/music/flashpiano.htm (https://www.apronus.com/music/flashpiano.htm). Choose the setting "computer keyboard plays from C4 to C6" and play the chord with your computer keyboard and then listen to the corresponding audio file and compare.

Exercise: You can run the code cell below to plot the `y` and the projection `y_hat` over an interval of 0.1 seconds. From this plot, you can see that the projection is a bad approximation of the original signal. Why could we expect this method of identifying the notes to still work? Write your answer in the Markdown cell after the plot.

In [250]:
```
plot(t,y, xlim=(0,0.1))
plot!(t,y_hat)
```

Out[250]:



Answer: Though the is extra information in the higher dimensional raw audio that captures attenuation, instrument features, recording artifacts etc. this information should not align well with pure tones and should be discarded by the projection. Thus is has little effect on the projection as well.