The Hyperdimensional Transform: a Holographic Representation of Functions

The hyperdimensional transform allows for converting functions into high-dimensional, holographic vectors, as known in the field of *hyperdimensional computing* or *vector symbolic architectures*.

This notebook supports our paper "*The Hyperdimensional Transform: a Holgraphic Representation of Functions*". It translates the theoretical concepts into simple, interpretable code. In a few lines, we show how one can implement the transform and reproduce the results in the paper. For example, the tutorial demonstrates how the transform allows one for easily solving a differential equation.

Note that this notebook is only supporting material and is not written a stand-alone tutorial. For theory, details, explanation and more examples, we refer the reader to our paper.

First, we include some supporting code. Then we define the domain: the interval $[0, 1]$, represented by m equidistant points

In [1]:

```
using LinearAlgebra, Plots, StatsBase

# supporting code
function ϕ(x, D, l; real=true)
    ϕx = zeros(m, D)
    for i in 1:D
        signs = rand((-1,1), ceil(Int, 1/l)+1)
        shift = l * rand()
        ϕx[:,i] .= @. (1-cos(2π/l*(x+shift))) * signs[(ceil(Int, (x+shif
t)/l))]
    end
    return real ? ϕx : sign.(ϕx)
end

# compute the normalization coefficients at the discrete points via
# iteratively solving the Hammerstein equation.
function find_normalization(K; n_iter=10)
    n = sqrt.(mean(K, dims=1))[:]
    onetilde = mean(K ./ n, dims=1)[:] ./ n
    p = plot(onetilde, label="iteration 0", legend=:bottomright)
    for i in 1:n_iter
        @. n = n * sqrt(onetilde)
        onetilde .= mean(K ./ n, dims=1)[:] ./ n
        plot!(onetilde, label="iteration $i")
    end
    return p, n
end

function δ(x, D, L; real)
    ϕx = ϕ(x, D, l; real)
    p,n = find_normalization(ϕx * ϕx' / D, n_iter=10)
    return ϕx ./ n
end

function deriv_encoding(X; h=1)
    res = zero(X)
    for i in 1:size(X,1)
        l = max(i-h, 1)
        u = min(i+h, size(X,1))
        res[i,:] = (X[u,:] - X[l,:]) / (u - l) * m
    end
    return res
end

function plot_encoding(ϕₓ)
    p = plot(layout=(2,1), size=(600,600))
    plot!(p[1], x, ϕₓ[:,1:4], label=["ϕ₁(x)" "ϕ₂(x)" "ϕ₃(x)" "..."], xlabel
="x", ylabel="random function value")
    plot!(p[2], x, ϕₓ*ϕₓ[1,:] / D, label="ϕₓ*ϕ₀ / D", xlabel="x", ylabel="s
imilarity to neighbors")
    plot!(p[2], x, ϕₓ*ϕₓ[125,:] / D, label="ϕₓ*ϕ₀.₂₅ / D")
    plot!(p[2], x, ϕₓ*ϕₓ[250,:] / D, label="ϕₓ*ϕ₀.₅ / D")
    plot!(p[2], x, ϕₓ*ϕₓ[375,:] / D, label="ϕₓ*ϕ₀.₇₅ / D")
    plot!(p[2], x, ϕₓ*ϕₓ[500,:] / D, label="ϕₓ*ϕ₁ / D")
end

function plot_normalized_encoding(δₓ)
    p = plot(layout=(2,1), size=(600,600))
    plot!(p[1], x, δₓ[:,1:4], label=["δ₁(x)" "δ₂(x)" "δ₃(x)" "..."], xlabel
="x", ylabel="random function value")
    plot!(p[2], x, δₓ*δₓ[1,:] / D, label="δₓ*δ₀ / D", xlabel="x", ylabel="s
imilarity to neighbors")
    plot!(p[2], x, δₓ*δₓ[125,:] / D, label="δₓ*δ₀.₂₅ / D")
    plot!(p[2], x, δₓ*δₓ[250,:] / D, label="δₓ*δ₀.₅ / D")
    plot!(p[2], x, δₓ*δₓ[375,:] / D, label="δₓ*δ₀.₇₅ / D")
```

```
    plot!(p[2], x, δₓ*δₓ[500,:] / D, label="δₓ*δ₁ / D")
end

m = 500
x = range(0, 1, length=m);
```

Create an encoding

We create an encoding of the domain x: each point is mapped to a D-dimensoinal vector via a vector-valued function ϕ. Each component can be seen as a random function switching randomly between 1 and -1. As a result, if points x and x' are close, then $\phi(x)$ and $\phi(x')$ are similar. If they are further away, then the random switching will lead to dissimilar representations. The rate at which this similarity drops is connected to the average frequency at wich the function switches, determined by lengthscale l. As with kernel methods, the lengthscale l can be seen as a kind of bandwidth parameter.

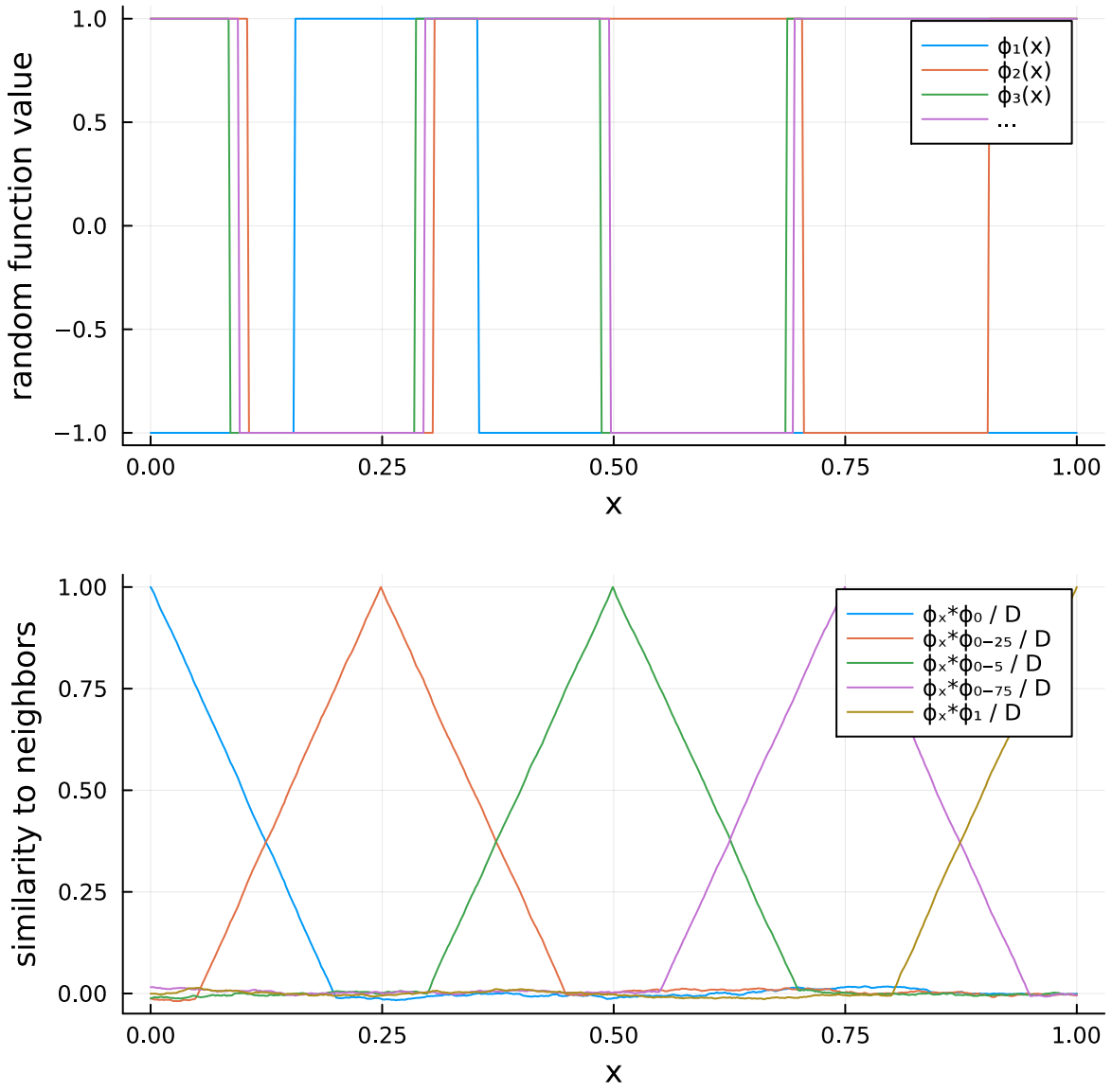
First panel: we plot a few components of the encoding ϕ for each point x in our interval.

Second panel: we plot the similarity between the components of $\phi(x)$ and $\phi(x')$ for a few fixed values of x'. The similarity is measured via the dot product, rescaled by the dimensionality D, i.e., the number of components.

In [2]:
```
D, l =  20_000, 0.2;
ϕₓ = ϕ(x, D, l, real=false);
plot_encoding(ϕₓ)
```

```
┌ Info: Precompiling GR_jll [d2c73de3-f751-5644-a686-071e5b155ba9]
└ @ Base loading.jl:1342
```

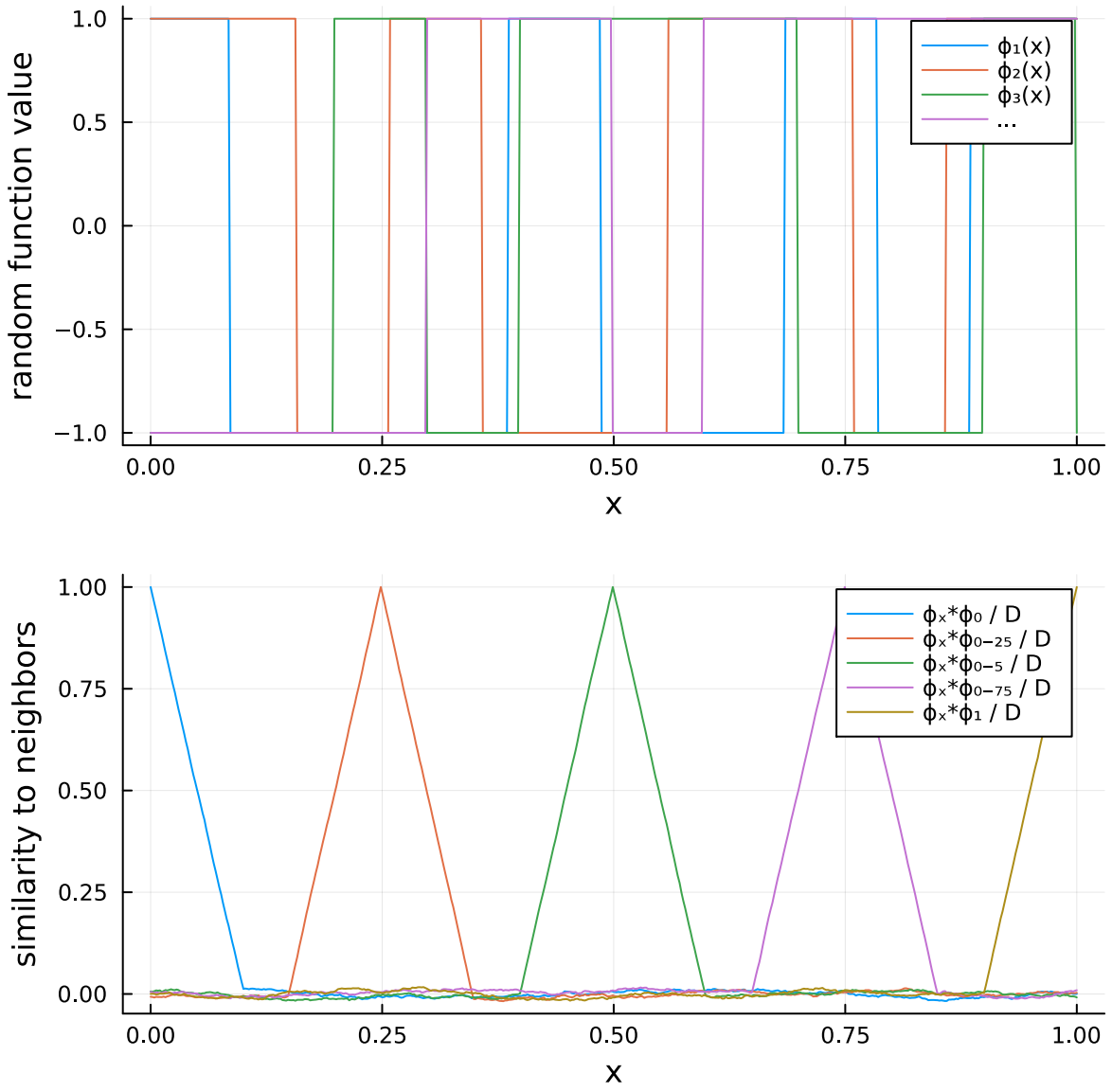
Out[2]:



If we select a shorter length scale $l = 0.1$, the components ϕ switch more frequently and the similarity drops faster. When further away than the length scale, the vectors are maximally dissimilar (similarity of random vectors).

In [3]:
```
D, l =  20_000, 0.1;
ϕₓ = ϕ(x, D, l, real=false);
plot_encoding(ϕₓ)
```

Out[3]:

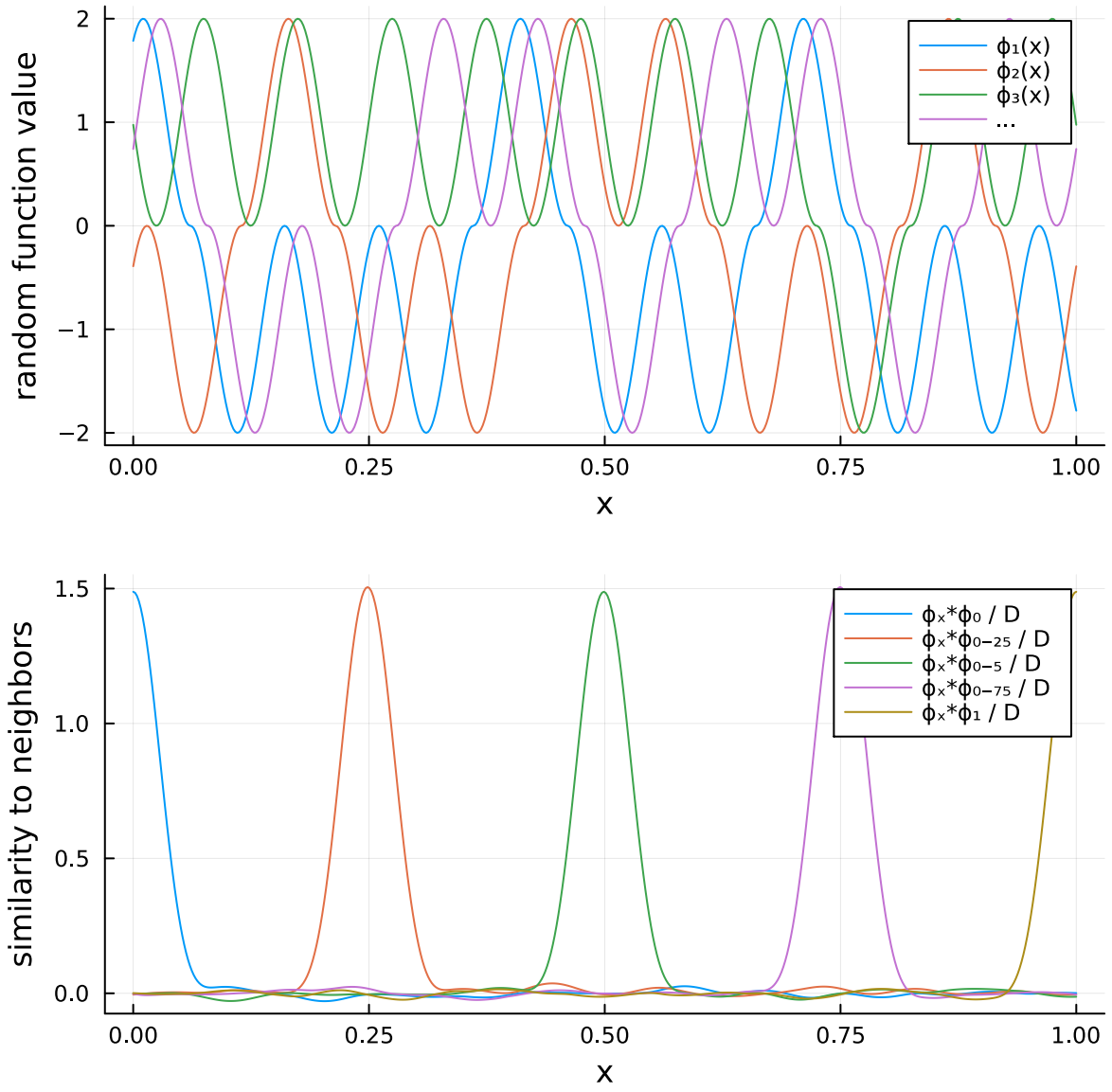


Now follows a different way of encoding that was not presented in the paper; that has a similar motivation, but has new interesting properties.

Each component is now a real-valued function that is a concatenation of pieces of cosine functions that during one period are on the postive or on the negative side, randomly. Additionally, each component also has a random phase shift to induce translation invariance. One period is exactly the length scale. This encoding is more smooth, and even differentiable. As we will see, taking the sign of this encoding leads to the very same encoding as the bipolar one.

In [4]:
```
D, l =  20_000, 0.1;
ϕₓ = ϕ(x, D, l, real=true);
plot_encoding(ϕₓ)
```

Out[4]:

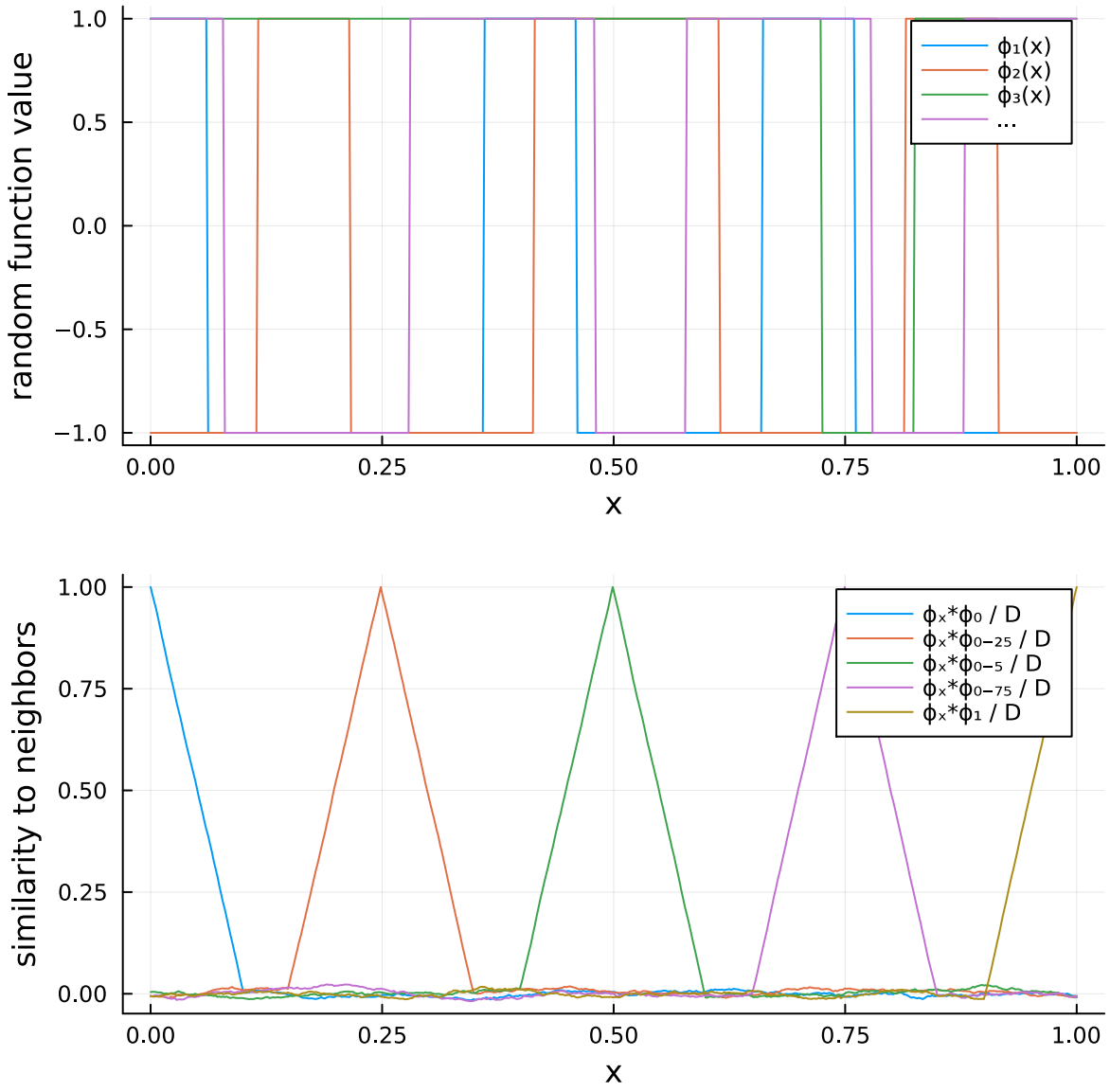


Comparing to the bipolar encoding, the similarity to the closer neighbors is a bit higher, and drops faster for further-away points.

If we take the sign of this encoding, we arrive exactly at the bipolar encoding from earlier:

In [5]:
```
ϕₓ = sign.(ϕₓ);
plot_encoding(ϕₓ)
```

Out[5]:



Create a normalized encoding

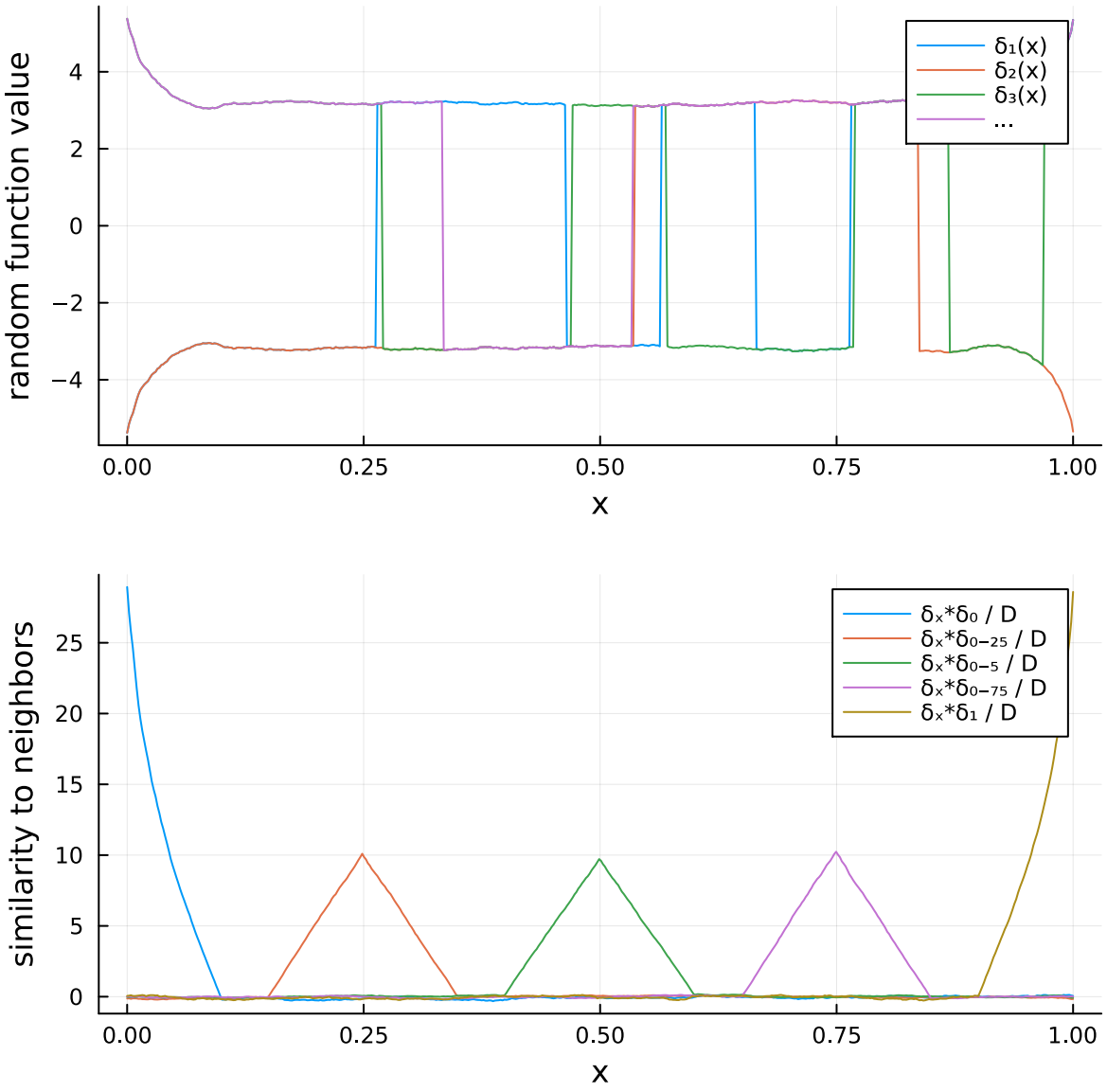
In order to have a well-defined transformation, we need some kind of normalization. The integral transform will integrate on the domain [0,1]. However, as we see in the right hand-side panels above, integration would put less weight at the boundary vectors, compared to the ones in the middle, because the boundary vectors have less similar neighbors.

Therefore, we introduced the normalized version of the encoding, denoted by δ. It may be seen as an approximation of a Dirac delta distribution with finite length scale l, represented via hyperdimensional vectors. The total mass under the curves in the right hand-side panel should be 1. The smaller the length scale l, the closer the Dirac distribution is approximated.

This normalized encoding δ is obtained by dividing the encoding ϕ by a function n that can be found by solving a Hammerstein equation. One can observe that the amplitude of ϕ at the boundaries increases to compensate for the lower weight.

In [6]:
```
D, l =  20_000, 0.1;
δₓ = δ(x, D, l, real=false);
plot_normalized_encoding(δₓ)
```

Out[6]:



The transform

With the choice of our domain, the transform is now an integral $\int_0^1 f(x)\delta(x)\mathrm{d}x$. We approximate this via our domain represented as m points. The transform is then simply a matrix multiplication of the vector holding the function evaluations in all points x, and the matrix holding the D-dimensional representations of each x; dividend by the number of points m.

We plot the D components of F, these seem indeed random, holographic.

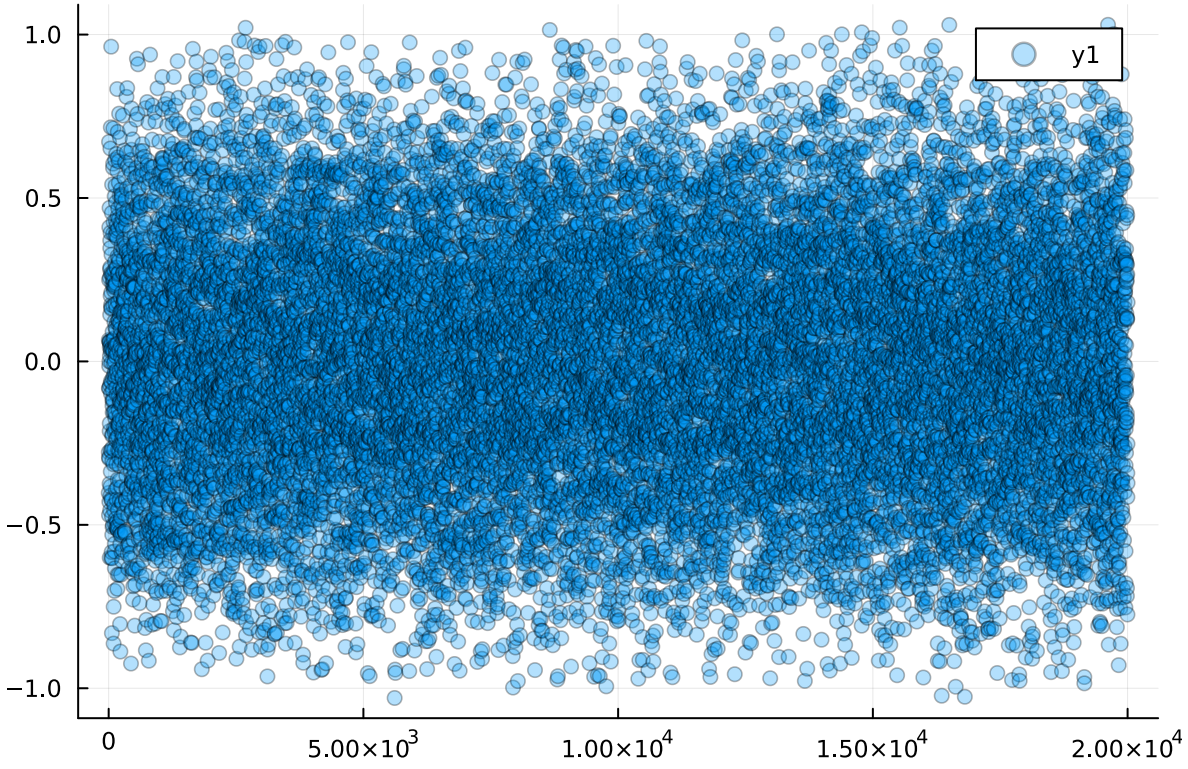
In [7]:
```
# the function
f(x) = x * sin(20x)

# the encoding
D, l =  20_000, 0.1;
δₓ = δ(x, D, l, real=true);

# the transform
F = δₓ' * f.(x)  ./ m;

scatter(F, alpha=0.3)
```

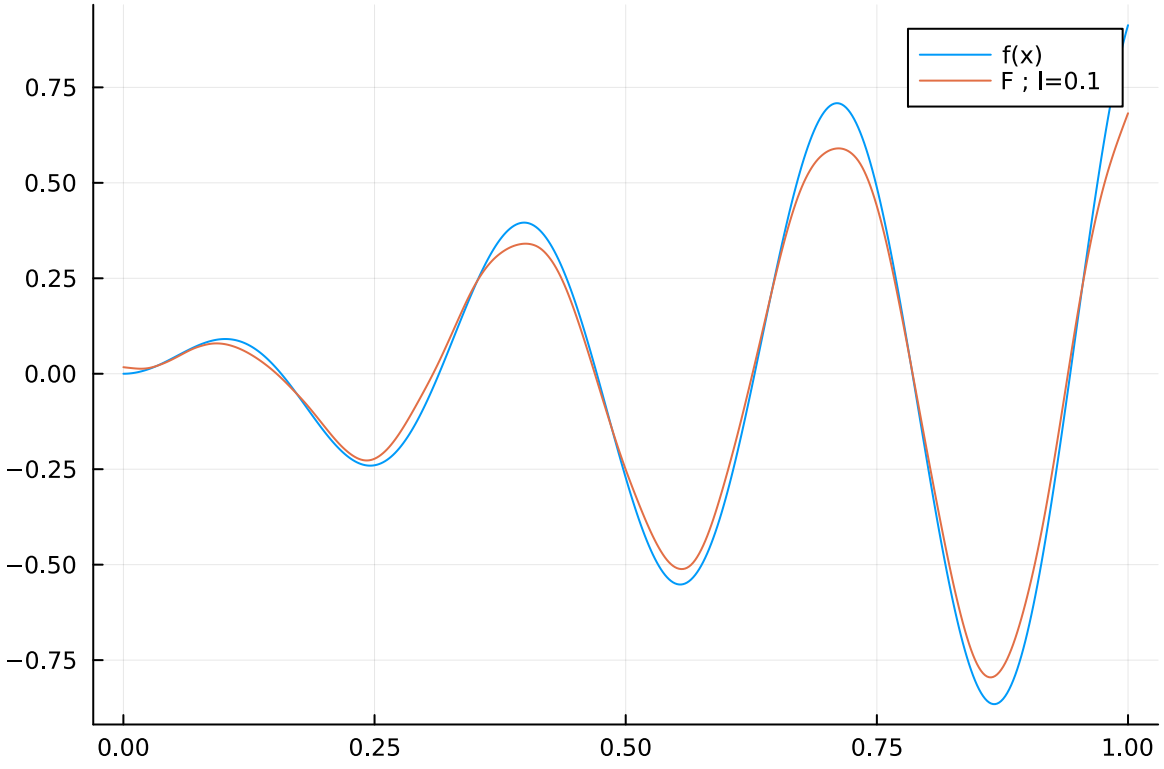
Out[7]:



Function evaluation of the inverse stransform, is a product with the delta representation. Note that the reproduced function is somewhat smoother due to the finite lengthscale. One might interpret the right hand-side panels above as the smoothing filters.

In [8]:
```
plot(x, f.(x), label="f(x)")
plot!(x, δₓ*F / D, label="F ; l=$l")
```

Out[8]:



Integrals

The Euclidean inner product $\frac{<F,F>}{D}$ approximates the function inner product $\int_0^1 f(x)f(x)\mathrm{d}x$.

Due to the finite length scale and smoothing in the representation of F (see figure above), is the reproduced norm somewhat smaller.

In [9]:
```
n1 = F' * F / D
n2 = f.(x)' * f.(x) ./ m

n1, n2
```

Out[9]:
```
(0.13903906501504792, 0.15830467214956118)
```

By setting one of the arguments of the inner product equal to the function 1, we can compute the integral of f.

In [10]:
```
o(x) = 1
O = δₓ' * o.(x) ./ m;

i1 = F' * O ./ D
i2 = f.(x)'* o.(x) ./ m

i1, i2
```

Out[10]:
```
(-0.01716952598246873, -0.017169520211703655)
```

Differentials

The derivatives of the transformed function can be computed via derivation of the the encoding, i.e.,

$$\begin{aligned}\tilde{f} &=< F, \delta >\\ \tilde{f}' &=< F, \delta' >\\ \tilde{f}'' &=< F, \delta'' >\\ &\dots\end{aligned}$$

Below, we plot the first derivatives of the transform of our function f.

the encoding

In [11]:
```
D, l =  20_000, 0.1;
δₓ = δ(x, D, l, real=true)
```

Out[11]:
```
500×20000 Matrix{Float64}:
 0.861136     -4.47225      2.06314     …   6.32735    -0.484775    -10.1313
 1.18644      -4.83819      2.46376         5.34403    -0.754091     -9.75613
 1.53452      -5.1713       2.86111         4.46632    -1.05829      -9.35907
 1.89732      -5.47049      3.24995         3.68591    -1.38818      -8.94362
 2.26833      -5.73548      3.62628         2.99581    -1.73621      -8.51346
 2.64225      -5.96653      3.98708     …   2.38984    -2.09618      -8.07206
 3.01456      -6.16364      4.32969         1.86238    -2.46278      -7.62188
 3.38131      -6.32679      4.65178         1.40848    -2.83139      -7.16496
 3.73894      -6.45587      4.95121         1.02389    -3.19788      -6.70309
 4.08397      -6.55029      5.22565         0.70493    -3.55828      -6.23759
 4.41296      -6.60938      5.47277     …   0.448513   -3.90871      -5.76984
 4.72289      -6.63297      5.69068         0.252042   -4.24573      -5.30185
 5.01086      -6.62096      5.87757         0.113207   -4.566        -4.83575
 ⋮                                      ⋱
 1.09738      -0.00194932  -0.385401       -7.14174    -1.50249       2.80965
 0.810341     -0.0459758   -0.212375       -7.27122    -1.17775       3.32279
 0.557767     -0.150432    -0.0875718  …   -7.36108    -0.880484      3.87045
 0.345115     -0.318565    -0.0158886      -7.40952    -0.616236      4.45011
 0.178102     -0.55367     -0.0023824      -7.4153     -0.390848      5.05933
 0.0627026    -0.859045    -0.0523052      -7.37664    -0.210435      5.69506
 0.00525773   -1.23793     -0.171121       -7.2917     -0.0815289     6.35361
 0.0124918    -1.69378     -0.364555   …   -7.16038    -0.0111233     7.03223
 0.0915683    -2.23049     -0.638749       -6.9834     -0.0066658     7.72881
 0.250296     -2.85279     -1.00055        -6.76233    -0.0762273     8.44232
 0.497324     -3.56564     -1.45752        -6.4974     -0.228736      9.17026
 0.842352     -4.3747      -2.01814        -6.18934    -0.474201      9.91031
```

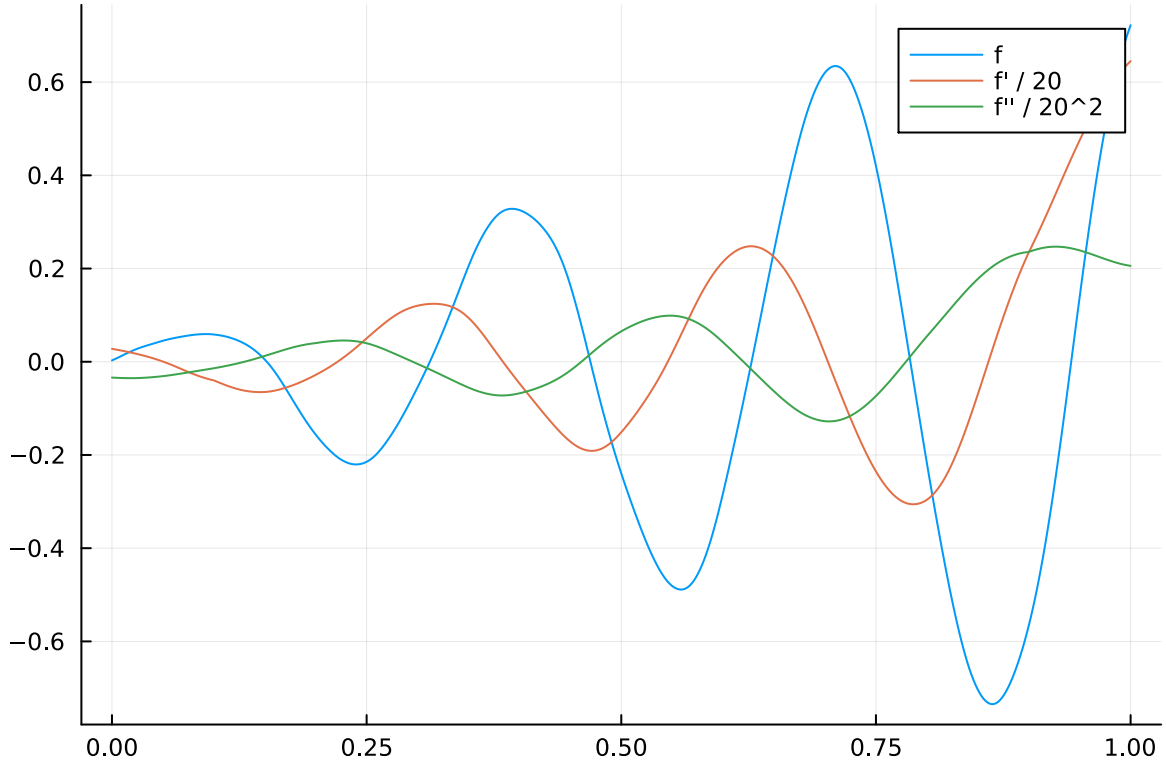
and the derivatives: a finite difference h=m*l (m:number of points in x, l: lenghtscale) is used. A finite difference allows to filter some noise, as differentiating a noisy function may increase noise.

In [12]:
```
δₓ_ = deriv_encoding(δₓ, h = round(Int, m*l));
δₓ__ = deriv_encoding(δₓ_, h = round(Int, m*l));

F = δₓ' * f.(x)  / m;

plot(x, δₓ*F / D, label="f")
plot!(x, δₓ_*F / D / 20, label="f' / 20")
plot!(x, δₓ__*F / D / 20^2, label="f'' / 20^2")
```

Out[12]:



Solve differential equations

We show how these concepts can be used to solve a differential equation. We will use a constant k, and the solution of linear ridge regression. For the encoding, we will use only 5000 dimensions allowing for fast matrix inversion, and a shorter length scale $l = 0.05$ for enough flexibility in the solution.

In [13]:
```
k = 10

# solution to ridge regression
function solve_ridge(X, B; λ=1)
    F =  (X'*X + λ*I) \ X'* B
    return F * D
end

# the encoding: fewer dimensions, shorter lengthscale
D, l =  5_000, 0.05;
δₓ = δ(x, D, l, real=true)
```

Out[13]:
```
500×5000 Matrix{Float64}:
  -9.22675       0.120946   12.2723     …  -15.412       6.87782
  -6.45296       0.627057   12.2073        -13.047       4.41687
  -4.25922       1.38927    11.8433        -10.798       2.59075
  -2.59272       2.29786    11.2661         -8.73374     1.31523
  -1.38714       3.27685    10.5301         -6.8758      0.505011
  -0.583226      4.27128     9.67026    …   -5.22833     0.0885601
  -0.133523      5.24002     8.71532        -3.79378     0.00907503
   0.000370289   6.15825     7.7001         -2.57898     0.221457
   0.154026      6.99556     6.63851        -1.58472     0.689702
   0.569353      7.70679     5.53832        -0.817647    1.37983
   1.21742       8.24181     4.42156    …   -0.293603    2.25131
   2.05984       8.55925     3.32879        -0.0302307   3.25405
   3.04958       8.63738     2.31292         0.03844     4.33237
   ⋮                                    ⋱
  -5.29222      -8.23741    -0.766701       -0.838765    6.54802
  -6.38049      -7.57498    -0.262102       -1.60698     7.48991
  -7.41217      -6.75977    -0.0193433 …    -2.58226     8.31028
  -8.33871      -5.8197      0.0582276      -3.7292      8.96715
  -9.11473      -4.79375     0.393065       -5.00602     9.4246
  -9.69661      -3.72895     1.03052        -6.36336     9.65108
 -10.0654       -2.68649     1.9714         -7.76008     9.64177
 -10.204        -1.72678     3.21247    …   -9.15401     9.39404
 -10.0796       -0.912019    4.73802       -10.4817      8.89274
  -9.68558      -0.315965    6.5278        -11.6911      8.15012
  -9.02485      -0.0183006   8.55544       -12.7304      7.18936
  -8.12083      -0.10645    10.8013        -13.5647      6.05344
```

and the derivatives.

In [14]:
```
δₓ_ = deriv_encoding(δₓ, h = round(Int, m*l/5));
δₓ__ = deriv_encoding(δₓ_, h = round(Int, m*l/5));
```

Exponential decay

The differential equation

$$\frac{1}{k}f'(x) + f(x) = 0$$

can, using the aforementioned expression of the derivative, be expressed for F

When expressed for the m points in our domain, the equation takes the form X*F = Y with X and Y computed as

In [15]:
```
X = 1 / k * δₓ_ + δₓ;
Y = zeros(m);
```

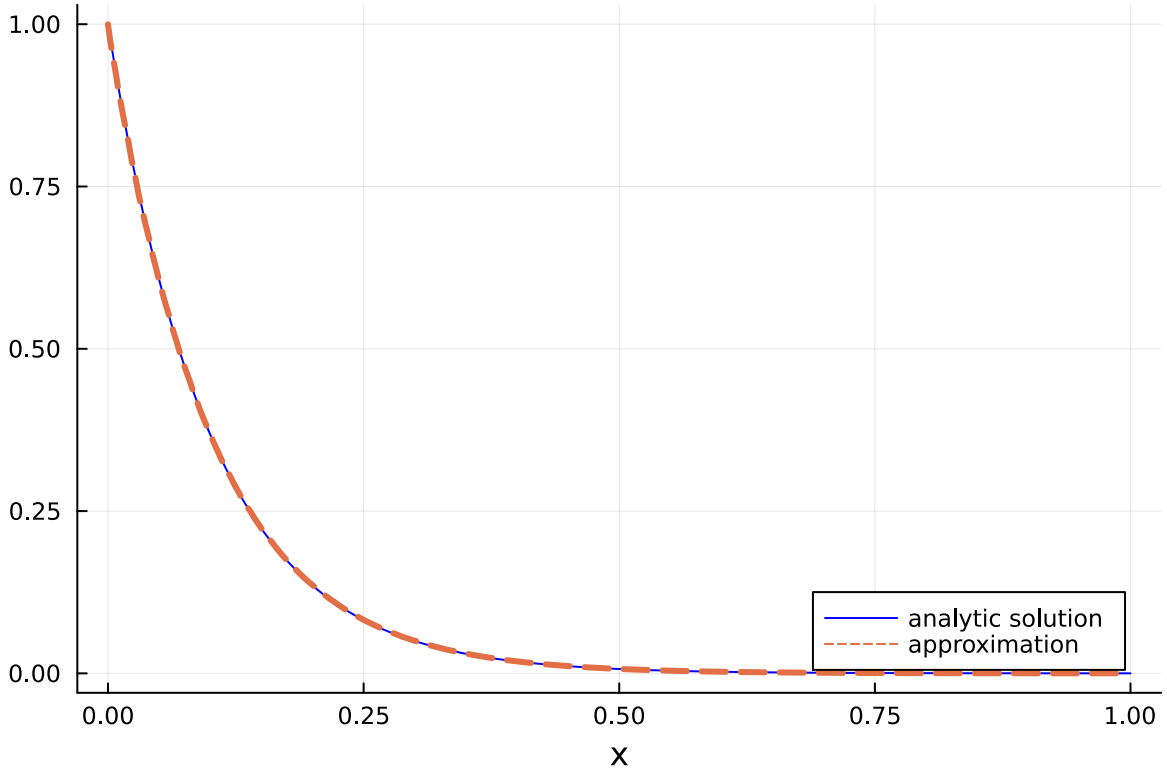
Boundary conditions can be added by concatenation. E.g. $f(0) = 1$ is translated as an additional equation in F via $< F, \delta_0 >= 1$.

In [16]:
```
X = vcat(X, δₓ[1,:]')
Y = vcat(Y, 1);
```

Now we solve as a ridge regression, trying to fit the equations as well as possible. The solution is plotted below

In [17]:
```
F = solve_ridge(X,Y, λ=1);
plot(x, c=:blue, exp.(-k*x), label="analytic solution", legend=:bottomrigh
t, xlabel="x")
plot!(x, δₓ*F / D, linestyle=:dash, linewidth=3, label="approximation")
```

Out[17]:



Harmonic oscillator

The differential equation

$$\frac{1}{k^2} f'' + f = 0$$

can, using the aforementioned expression of the derivative, be expressed for F

$$< \frac{1}{k^2} \delta_x'' + \delta_x, F >= 0\,.$$

Now, we solve this equation anolog to above.

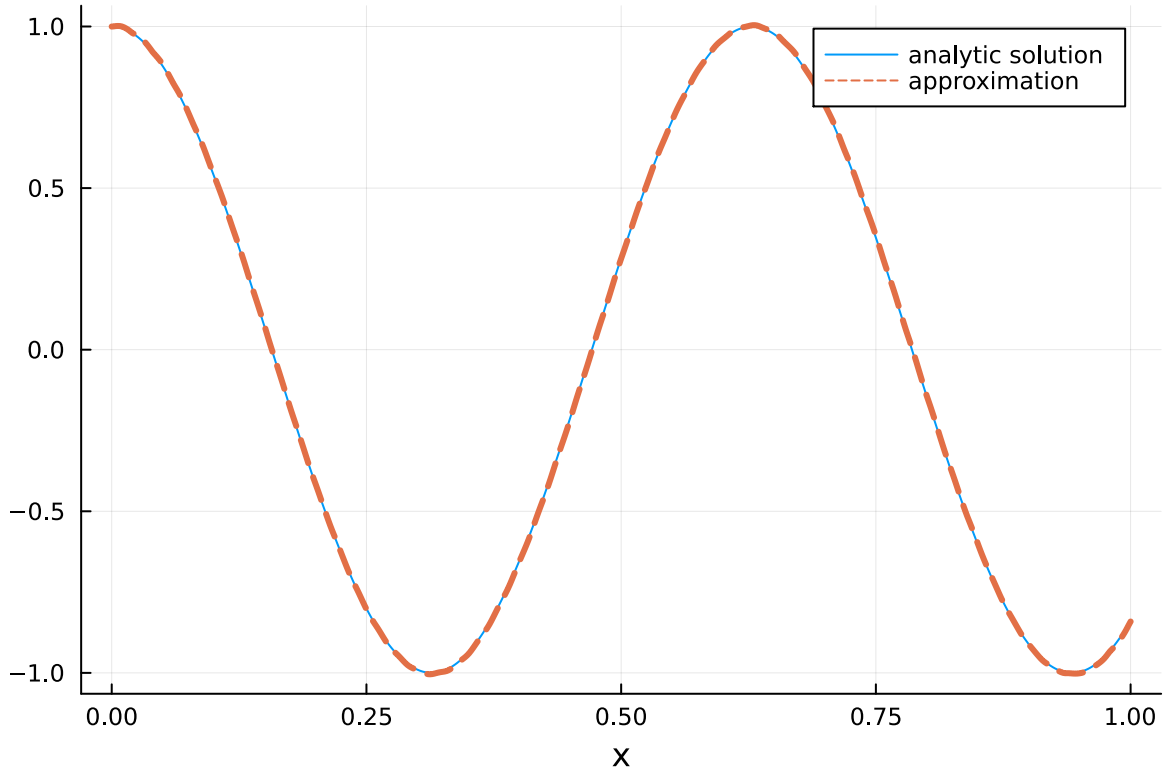
In [19]:
```
# equation
X = 1/k^2*δₓ__ + δₓ
Y = zeros(m)

# boundary condition: f(x)=1
X = vcat(X, δₓ[1,:]')
Y = vcat(Y, 1)

# solve
F = solve_ridge(X, Y, λ=1);

# plot
plot(x, cos.(k*x), label="analytic solution", xlabel="x")
plot!(x, δₓ*F / D, linestyle=:dash,linewidth=3, label="approximation")
```

Out[19]:



Damped harmonic oscillator

The differential equation

$$\frac{1}{k^2} f'' + k f' + f = 0$$

can, using the aforementioned expression of the derivative, be expressed for F

$$< \frac{1}{k^2}\delta_x'' + \frac{1}{k}\delta_x' + \delta_x, F >= 0\,.$$

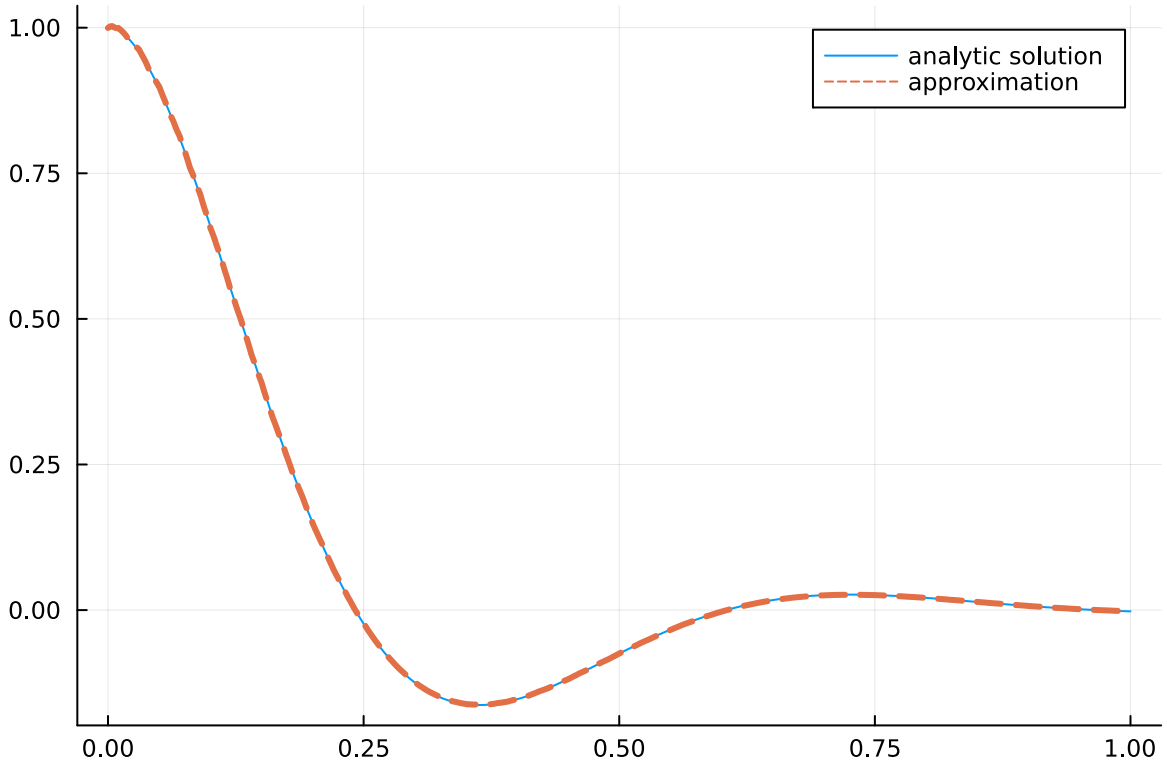
In [21]:
```
# equation
X = 1/k^2*δₓ__ + 1/k*δₓ_ + δₓ
Y = zeros(m)

# boundary conditions: f(0):1, f'(0)=0
X = vcat(X, δₓ[1,:]', δₓ_[1,:]')
Y = vcat(Y, 1, 0)

# solve
F = solve_ridge(X, Y, λ=1);

# plot
dampedosc(t) = 1/3 * exp(-5*t) *(sqrt(3)*sin(5sqrt(3)*t) + 3cos(5*sqrt(3)*
t))
plot(x, dampedosc.(x), label="analytic solution")
plot!(x, δₓ*F / D, linestyle=:dash,linewidth=3, label="approximation")
```

Out[21]:



In [ ]: