Numerical methods in (non-hyperbolic) chaos

Part 3: Koopman and transfer operator discretisations I

Caroline Wormell, Sorbonne Université/CNRS

- These slideshows are made with a package called RISE: rise.readthedocs.io/en/stable/
- It is an add-on for the Jupyter notebook: jupyter.org
- I am writing code in Julia, but most programming languages also work with Jupyter.

Yesterday:

- We can estimate averages of our physical measure with *a posteriori* error bounds
- Convergence of stuff is determined by decay of correlations

How can we understand decay of correlations, or compute other quantities?

Answer: Using transfer and Koopman operators.

Koopman operator

The Koopman operator $\mathcal{K}$ of a map f is a composition operator:

$$(\mathcal{K}\phi)(x) = \phi(f(x))$$

It is commonly used in physical sciences to study complex dynamics (e.g. forecasting, system reduction...)

However, traditionally dynamicists have studied transfer operators, which are the adjoints of Koopman operators (usually with $\mu =$ Lebesgue):

$$\int_M \psi \, \mathcal{K}\phi \, \mathrm{d}\mu = \int_M \mathcal{L}\psi \, \phi \, \mathrm{d}\mu$$

So, ignoring **important** niceties of what space you are defining your transfer operator on, the transfer and Koopman operators do the same job in different ways.

Galerkin approximation

Our transfer and Koopman operators act on infinite-dimensional spaces. Numerics will require finite-dimensional appproximations.

Chaotic dynamics doesn't typically suggest a natural finite-dimensional subset of these spaces to choose, so let's imagine we choose a reasonable set of functions $\{\psi_1, \ldots, \psi_K\}$ and try and approximate the action of the relevant operator as best we can in this subspace.

Because it's computationally easy, we usually try and do our operator approximation in a Hilbert space, and this is usually $L^2(\mu)$.

(So we are just doing a method of least squares.)

Let's take

- Hilbert space H
- Vector subspace $V = \mathrm{span}\{\psi_1, \ldots, \psi_K\} \subset H$
- Element $u \in H$ that we want to approximate by an element in V as best we can.

Let's seek an element $u_V \in V$ so that $\|u - u_V\|_H$ is minimised.

Let's take

- Hilbert space H
- Vector subspace $V = \mathrm{span}\{\psi_1, \ldots, \psi_K\} \subset H$
- Element $u \in H$ that we want to approximate by an element in V as best we can.

Let's seek an element $u_V \in V$ so that $\|u - u_V\|_H$ is minimised.

This is to say that for any $w \in V$ and $\epsilon > 0$,

$$\|u - (u_V + \epsilon w)\|_H^2 - \|u - u_V\|_H^2 \geq 0$$

so

$$\Re\langle -\epsilon w, u - (u_V + \epsilon w) + u - u_V\rangle_H \geq 0$$

so

$$\Re\langle w, u - u_V\rangle_H \leq \tfrac{1}{2}\epsilon\|w\|_H^2.$$

Since ϵ can be arbitrarily small, this implies that

$$\langle w, u - u_V \rangle_H = 0$$

for all $w \in H$. It is an orthogonal projection.

It is a standard result that u_V exists and is unique. How do we compute it?

Well, we need that

$$\langle \psi_j, u - u_V \rangle_H = 0$$

for all j. So, u_V must obey

$$\langle \psi_j, u_V \rangle_H = \langle \psi_j, u \rangle_H, j = 1, \ldots, K.$$

It is a standard result that u_V exists and is unique. How do we compute it?

Well, we need that

$$\langle \psi_j, u - u_V \rangle_H = 0$$

for all j. So, u_V must obey

$$\langle \psi_j, u_V \rangle_H = \langle \psi_j, u \rangle_H, j = 1, \ldots, K.$$

Using the Hilbert space adjoint,

$$\psi_j^* u_V = \psi_j^* u, j = 1, \ldots, K$$

Let's write this in vector notation. Set the row vector

$$\Psi_0 = \begin{bmatrix} \psi_1 & \psi_2 & \cdots & \psi_K \end{bmatrix}$$

(note: if the ambient space is separable, the ψ_j could be represented as column vectors.)

Then the adjoint is

$$\Psi_0^* = \begin{bmatrix} \psi_1^* \\ \psi_2^* \\ \vdots \\ \psi_K^* \end{bmatrix}$$

and so we have

$$\Psi_0^* u_V = \Psi_0^* u$$

.

But how can we express u_V as a sum of ψ_j given that the ψ_j are not necessarily orthogonal?

If $u_V = \sum_{k=1}^{K} a_k \psi_k$, we obtain the system of equations

$$\sum_{k=1}^{K} \langle \psi_j, \psi_k \rangle_H a_k = \langle \psi_j, u \rangle_H, j = 1, \ldots, K.$$

But how can we express u_V as a sum of ψ_j given that the ψ_j are not necessarily orthogonal?

If $u_V = \sum_{k=1}^{K} a_k \psi_k$, we obtain the system of equations

$$\sum_{k=1}^{K} \langle \psi_j, \psi_k \rangle_H a_k = \langle \psi_j, u \rangle_H, j = 1, \ldots, K.$$

Or in vector notation,

$$u_V = \mathbf{\Psi}_0 \mathbf{a}$$

for a column vector $\mathbf{a} = \{a_k\}_{k \in \mathrm{N}}$, so

$$\mathbf{\Psi}_0^* \mathbf{\Psi}_0 \mathbf{a} = \mathbf{\Psi}_0^* u.$$

Supposing the ψ_j are all linearly independent, we can then find the coefficients of u_v:

$$\mathbf{a} = (\Psi_0^* \Psi_0)^{-1} \Psi_0^* u.$$

Furthermore,

$$u_V = \Psi_0 (\Psi_0^* \Psi_0)^{-1} \Psi_0^* u.$$

Supposing the ψ_j are all linearly independent, we can then find the coefficients of u_v:

$$\mathbf{a} = (\Psi_0^* \Psi_0)^{-1} \Psi_0^* u.$$

Furthermore,

$$u_V = \Psi_0 (\Psi_0^* \Psi_0)^{-1} \Psi_0^* u.$$

We can define the projection operator

$$\mathcal{P}_{\Psi_0} u := \Psi_0 (\Psi_0^* \Psi_0)^{-1} \Psi_0^* u$$

so $\mathcal{P}_{\Psi_0} u = u_V$. Note this is linear!

Computational example of least squares

Let's suppose our Hilbert space is $\mathbb{R}^{10}$ with the usual Euclidean inner product

$$\langle v, w \rangle_H = \sum_{i=1}^{10} \bar{v}_i w_i.$$

So a vector v's adjoint v^* is just the conjugate transpose!

Computational example of least squares

Let's suppose our Hilbert space is $\mathbb{R}^{10}$ with the usual Euclidean inner product

$$\langle v, w \rangle_H = \sum_{i=1}^{10} \bar{v}_i w_i.$$

So a vector v's adjoint v^* is just the conjugate transpose!

In [2]:
```
d = 10
u = randn(d)
```

Out[2]:
```
10-element Vector{Float64}:
 -0.049334746093896875
 -1.092632645771358
 -0.49812436214735445
 -1.9875527999103249
  0.5559897989515565
 -0.6515643404193722
  0.7508811289246993
 -1.1205913933387566
  1.7096459377960074
 -0.47427065340092267
```

Computational example of least squares

Let's suppose our Hilbert space is $\mathbb{R}^{10}$ with the usual Euclidean inner product

$$\langle v, w \rangle_H = \sum_{i=1}^{10} \bar{v}_i w_i.$$

So a vector v's adjoint v^* is just the conjugate transpose!

In [2]:
```
d = 10
u = randn(d)
```

Out[2]:
```
10-element Vector{Float64}:
 -0.049334746093896875
 -1.092632645771358
 -0.49812436214735445
 -1.9875527999103249
  0.5559897989515565
 -0.6515643404193722
  0.7508811289246993
 -1.1205913933387566
  1.7096459377960074
 -0.47427065340092267
```

In [3]:
```
w = rand(d)
u' * w, sum(u[i] * w[i] for i = 1:d)
```

Out[3]: (-1.6336526756092042, -1.6336526756092042)

Let's say the basis elements are

$$\psi_j = \sum_{i=1}^{j} e_i, j = 1, \ldots, K$$

Let's say the basis elements are

$$\psi_j = \sum_{i=1}^{j} e_i, j = 1, \ldots, K$$

In [4]:

```
K = 5
Psi_0 = Array{Float64}(undef,d,K)
for j = 1:K
    psi_j = [(i<=j) for i = 1:d]
    Psi_0[:,j] = psi_j
end
Psi_0
```

Out[4]:

```
10×5 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0
 0.0  1.0  1.0  1.0  1.0
 0.0  0.0  1.0  1.0  1.0
 0.0  0.0  0.0  1.0  1.0
 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
```

Let's try to approximate a vector in this basis:

Let's try to approximate a vector in this basis:

In [5]:
```
u = collect(1.0:d)
println(u)
```

```
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

In [6]:

```
a = inv(Psi_0' * Psi_0) * (Psi_0' * u)
println(a)
```

```
[-1.0, -1.0, -1.0, -1.0, 5.0]
```

In [6]:
```
a = inv(Psi_0' * Psi_0) * (Psi_0' * u)
println(a)
```

```
[-1.0, -1.0, -1.0, -1.0, 5.0]
```

In [7]:
```
u_V = Psi_0 * a
println(u_V)
```

```
[1.0, 2.0, 3.0, 4.0, 5.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

In fact, $\Psi_0^+ = (\Psi_0^* \Psi_0)^{-1} \Psi_0^*$ is what is known as the *pseudo-inverse* of Ψ_0 (and extends naturally to situations where $\Psi_0^* \Psi_0$ is not invertible).

In fact, $\Psi_0^+ = (\Psi_0^* \Psi_0)^{-1} \Psi_0^*$ is what is known as the *pseudo-inverse* of Ψ_0 (and extends naturally to situations where $\Psi_0^* \Psi_0$ is not invertible).

Suppose a matrix W has singular value decomposition

$$W = U \begin{pmatrix} \Sigma & \\ & 0 \end{pmatrix} V^T,$$

where U, V are unitary matrices and Σ is a diagonal matrix with positive entries, then

Then

$$W^+ = V \begin{pmatrix} \Sigma^{-1} & \\ & 0 \end{pmatrix} U^T.$$

If you have the Euclidean norm, there are specific routines for pseudo-inverse.

In Julia at least, you can use the left division operator `\`. Satisfying but probably worse...

(This also does regular inverses such as $A^{-1}b =$ `A\b`.)

In [8]:
```
Psi_0 = randn(10000,500) # random basis vectors...
u = rand(10000)
@time inv(Psi_0'*Psi_0)*(Psi_0'*u)
@time Psi_0 \ u;
```

```
  0.049745 seconds (11 allocations: 4.071 MiB)
  1.572294 seconds (2.00 M allocations: 153.236 MiB, 8.11% gc t
ime, 47.27% compilation time)
```

How to discretise an operator?

How to discretise an operator?

Well, everything's linear, but let's explain:

Let's apply a bounded linear operator $\mathcal{A} : H \to H$ to our $u \in H$, then approximate it in V.

We have $(\mathcal{A}u)_V = \mathcal{P}_{\Psi_0}\mathcal{A}u$, and its basis coefficients are $\Psi_0^+\mathcal{A}u$.

So our projection of $\mathcal{A}$ is $\mathcal{P}_{\Psi_0}\mathcal{A} : H \to V$.

How to discretise an operator?

Well, everything's linear, but let's explain:

Let's apply a bounded linear operator $\mathcal{A} : H \to H$ to our $u \in H$, then approximate it in V.

We have $(\mathcal{A}u)_V = \mathcal{P}_{\Psi_0}\mathcal{A}u$, and its basis coefficients are $\Psi_0^+\mathcal{A}u$.

So our projection of $\mathcal{A}$ is $\mathcal{P}_{\Psi_0}\mathcal{A} : H \to V$.

But, on some level our operator domain is still H, which could be infinite dimensional.

We can just restrict $\mathcal{P}_{\Psi_0}\mathcal{A} : V \to V$.

So if we start with a function $\sum_{j=1}^{K} b_j \psi_j = \Psi_0 \mathbf{b} \in V$, the output coefficients are

$$\Psi_0^+ \mathcal{A} \Psi_0 \mathbf{b}.$$

If we define

$$\Psi_1 := \mathcal{A}\Psi_0 = \begin{bmatrix} \mathcal{A}\psi_1 & \mathcal{A}\psi_2 & \cdots & \mathcal{A}\psi_K \end{bmatrix}$$

Then our matrix representation of $\mathcal{P}_{\Psi_0}\mathcal{A}$ is

$$A := \Psi_0^+ \Psi_1 = (\Psi_0^* \Psi_0)^{-1} (\Psi_0^* \Psi_1).$$

So if we start with a function $\sum_{j=1}^{K} b_j \psi_j = \Psi_0 \mathbf{b} \in V$, the output coefficients are

$$\Psi_0^+ \mathcal{A} \Psi_0 \mathbf{b}.$$

If we define

$$\Psi_1 := \mathcal{A}\Psi_0 = \begin{bmatrix} \mathcal{A}\psi_1 & \mathcal{A}\psi_2 & \cdots & \mathcal{A}\psi_K \end{bmatrix}$$

Then our matrix representation of $\mathcal{P}_{\Psi_0}\mathcal{A}$ is

$$A := \Psi_0^+ \Psi_1 = (\Psi_0^* \Psi_0)^{-1} (\Psi_0^* \Psi_1).$$

Example: if $\mathcal{A} = \mathcal{K} =$ Koopman operator

$$\Psi_1 := \mathcal{K}\Psi_0 = \begin{bmatrix} \psi_1 \circ f & \psi_2 \circ f & \cdots & \psi_K \circ f \end{bmatrix}$$

Example

Example

For an example where everything is super nice, let's choose a perturbation of the doubling map on $[-1, 1]$

$$f(x) = \mathfrak{d}(x) * (1 + 0.2(1 - \mathfrak{d}(x)^2))$$

where $\mathfrak{d}(x) = 2(x \mod 1) - 1$.

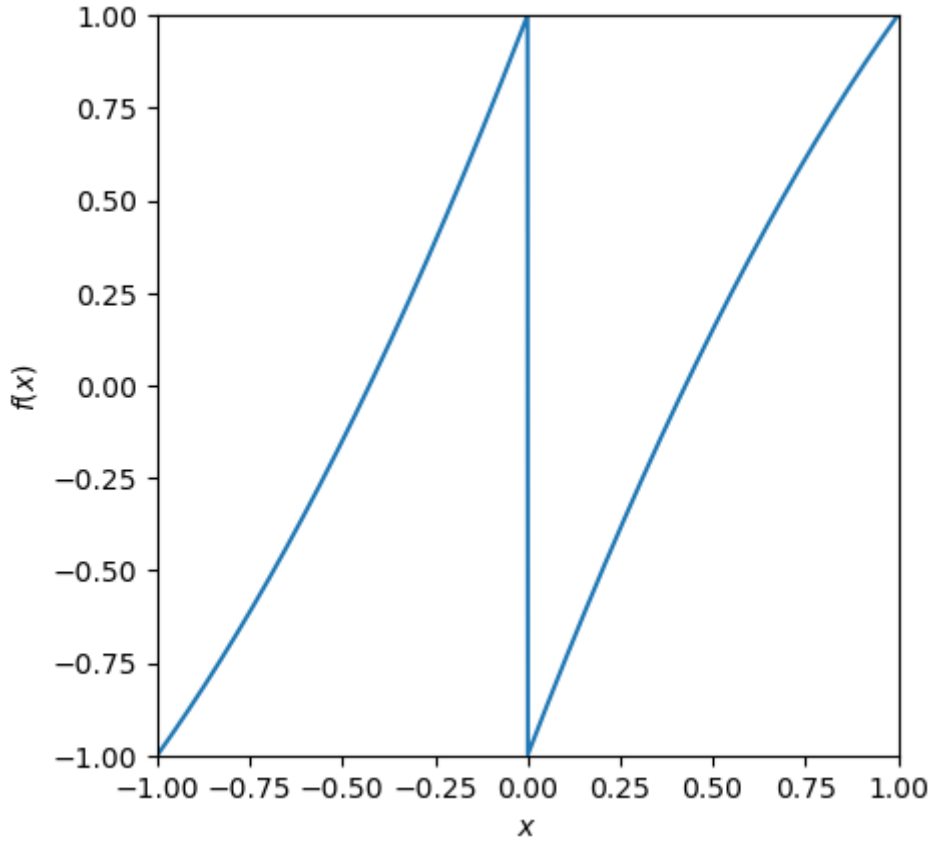
Example

For an example where everything is super nice, let's choose a perturbation of the doubling map on $[-1,1]$

$$f(x) = \mathfrak{d}(x) * (1 + 0.2(1 - \mathfrak{d}(x)^2))$$

where $\mathfrak{d}(x) = 2(x \mod 1) - 1$.

In [9]:
```
f(x) = (x>0 ? 2x-1 : 2x+1)+0.6(x*(1-abs(x)))
plot(-1:0.001:1,f.(-1:0.001:1));
xlim(-1,1);ylim(-1,1);xlabel("\$x\$");ylabel("\$f(x)\$")
gca().set_aspect("equal")
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
f(x)
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x

We will have Hilbert space $H = L^2(\mathrm{d}x)$ and monomial basis functions

$$\psi_j(x) = x^{j-1}, j = 1, \ldots, K$$

In [16]:

```
K = 16
Psi0 = [x->x^(j-1) for j = 1:K]
Psi1 = [x->f(x)^(j-1) for j = 1:K];
```

In [17]:
```
using QuadGK
integrate(psij,psik) = quadgk(x->psij(x)*psik(x),-1,0,1)[1]  # integrat
```

Out[17]:
```
integrate (generic function with 1 method)
```

In [17]:
```
using QuadGK
integrate(psij,psik) = quadgk(x->psij(x)*psik(x),-1,0,1)[1]  # integrat
```

Out[17]:
```
integrate (generic function with 1 method)
```

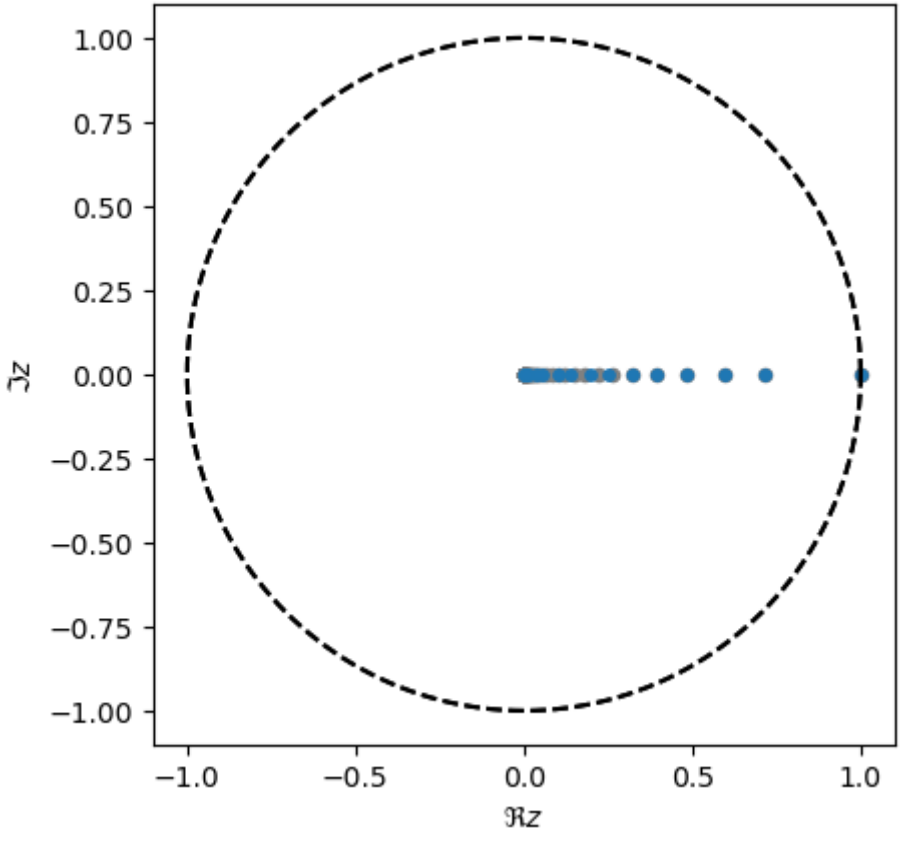
In [18]:
```
Psi00prod =  [integrate(Psi0[j],Psi0[k]) for j = 1:K, k =1:K]
Psi01prod =  [integrate(Psi0[j],Psi1[k]) for j = 1:K, k =1:K]
Koop = Psi00prod \ Psi01prod
```

Out[18]:
```
16×16 Matrix{Float64}:
  1.0     -2.77581e-8       0.787188     …      0.245195
-3.04092e-9
  0.0     -7.93639          1.56591e-7          1.39511e-7
-0.495753
  0.0      2.78653e-6     -17.8868             -9.39313
3.05252e-7
  0.0    152.949           -5.753e-6           -5.12525e-6
12.9935
  0.0     -4.57731e-5     154.135             105.504
-5.01399e-6
  0.0  -1163.24             6.18572e-5   …      5.51064e-5    -
115.966
 -0.0      0.000282188   -659.746            -523.035
3.09098e-5
  0.0   4555.51            -0.000295327        -0.000263094
491.527
  0.0     -0.000828388   1554.9               1328.14
-9.07357e-5
```

-0.0 -9847.18 0.000727001 0.000647651 -1
115.02
0.0 0.00124772 -2019.24 … -1799.31
0.000136663
-0.0 11868.8 -0.000961135 -0.000856231 1
392.35
0.0 -0.000931223 1353.07 1234.36
-0.000101996
-0.0 -7469.1 0.000648282 0.000577527 -
905.04
-0.0 0.000272813 -365.091 -335.585
2.98805e-5
0.0 1911.38 -0.00017511 … -0.000155999
240.678

In [19]:

```
using LinearAlgebra
spectrumplot(true_eigs; c="grey",s=20) # true eigenvalues: here's one 1
spectrumplot(eigvals(Koop); s= 15);
```



The function space projection $\mathcal{P}_V = \Psi_0 \Psi_0^+$ is self-adjoint in H, so in theory very nice and well-conditioned, but it may not be so in function space.

A good thing to look out for is the conditioning of your basis functions:

The function space projection $\mathcal{P}_V = \Psi_0 \Psi_0^+$ is self-adjoint in H, so in theory very nice and well-conditioned, but it may not be so in function space.

A good thing to look out for is the conditioning of your basis functions:

In [20]:
```
cond(Psi00prod)
```

Out[20]:
```
5.285116058923926e10
```

$$\mathrm{cond}(K) = \|K\| \|K^{-1}\|$$

is a good estimate of sensitive your matrix K is to numerical error when you invert it:

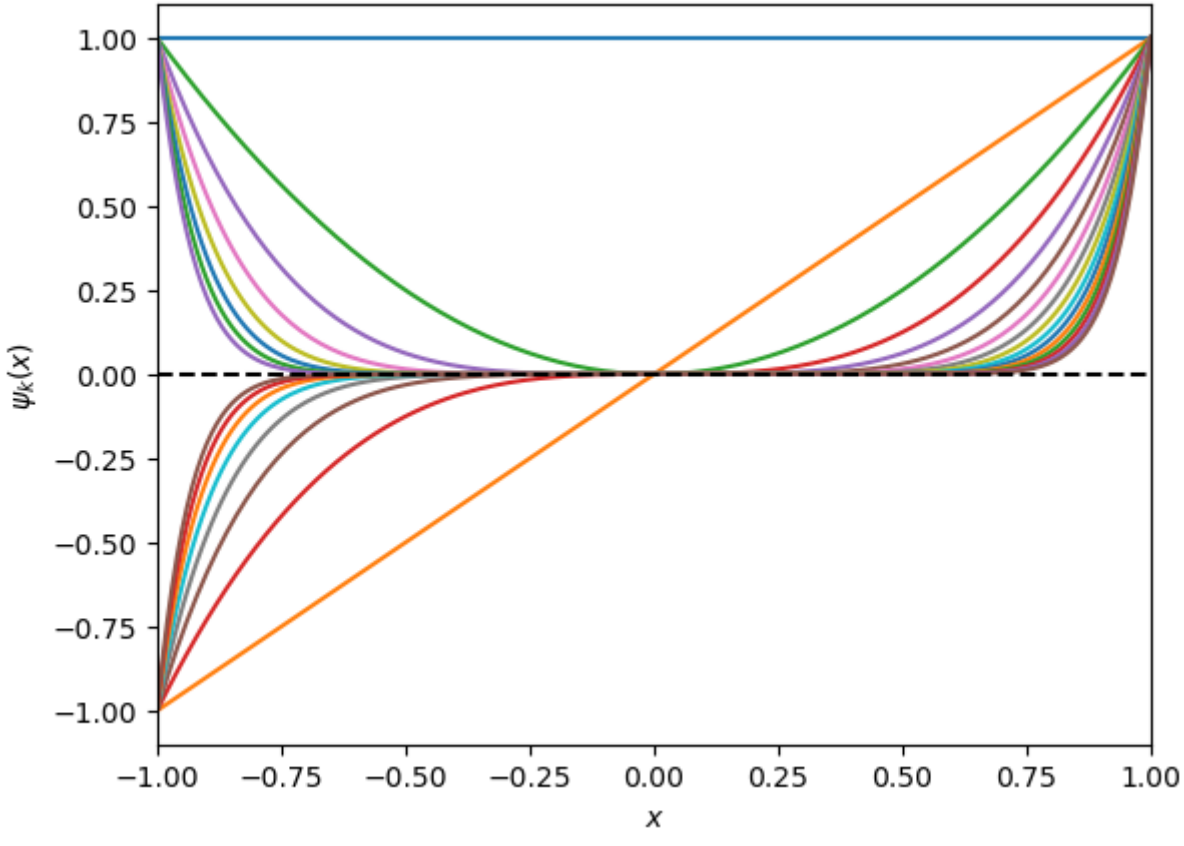
In [21]:
```
Psi00prod2 = Psi00prod + 1e-8*randn(K,K)/K # some random numerical erro
Koop2 = Psi00prod2  \ Psi01prod
Koop2 - Koop
```

Out[21]:
```
16×16 Matrix{Float64}:
 -1.34966e-5      -0.0297775    -0.0163526  …    -0.0160845
-0.00299556
  4.42936e-5       1.8303       -0.0123312       -0.0111
0.21235
  0.00129361       2.69665       1.78962          1.73679
0.276592
 -0.00110133     -68.6786        0.761349         0.650773
-7.95305
 -0.0207106      -40.8474      -31.0649         -29.9072
-4.26466
  0.0066529      748.942       -10.725      …    -8.94397
86.6191
  0.125697       235.731       199.082          190.636
25.0068
 -0.00993071   -3611.58         60.8513          49.9538      -
417.324
 -0.36528       -654.908      -601.565         -573.819
-70.4739
 -0.0225328     8956.29       -169.174         -137.321       1
034.22
  0.546445       941.293       926.678      …   881.355
102.6
  0.0841445   -11907.4         244.977          197.144      -1
374.29
 -0.405902      -674.721      -704.198         -668.221
```

-74.3997
-0.0889824 8066.85 -177.396 -141.783
930.651
0.118512 190.824 209.38 198.316
21.2632
0.0317294 -2186.64 50.7319 … 40.3203 -
252.183

The problem is our monomial basis functions all look very similar!

In [26]:
```
for (j,psi) in enumerate(Psi0)
plot(-1:0.01:1,psi.(-1:0.01:1),label="\$\\psi_{$j}\$")
end
# legend()
plot([-1,1],[0,0],"k--")
xlim(-1,1);xlabel("\$x\$"); ylabel("\$\\psi_k(x)\$");
```



Let's start with a more varied-looking basis, the Chebyshev polynomials:

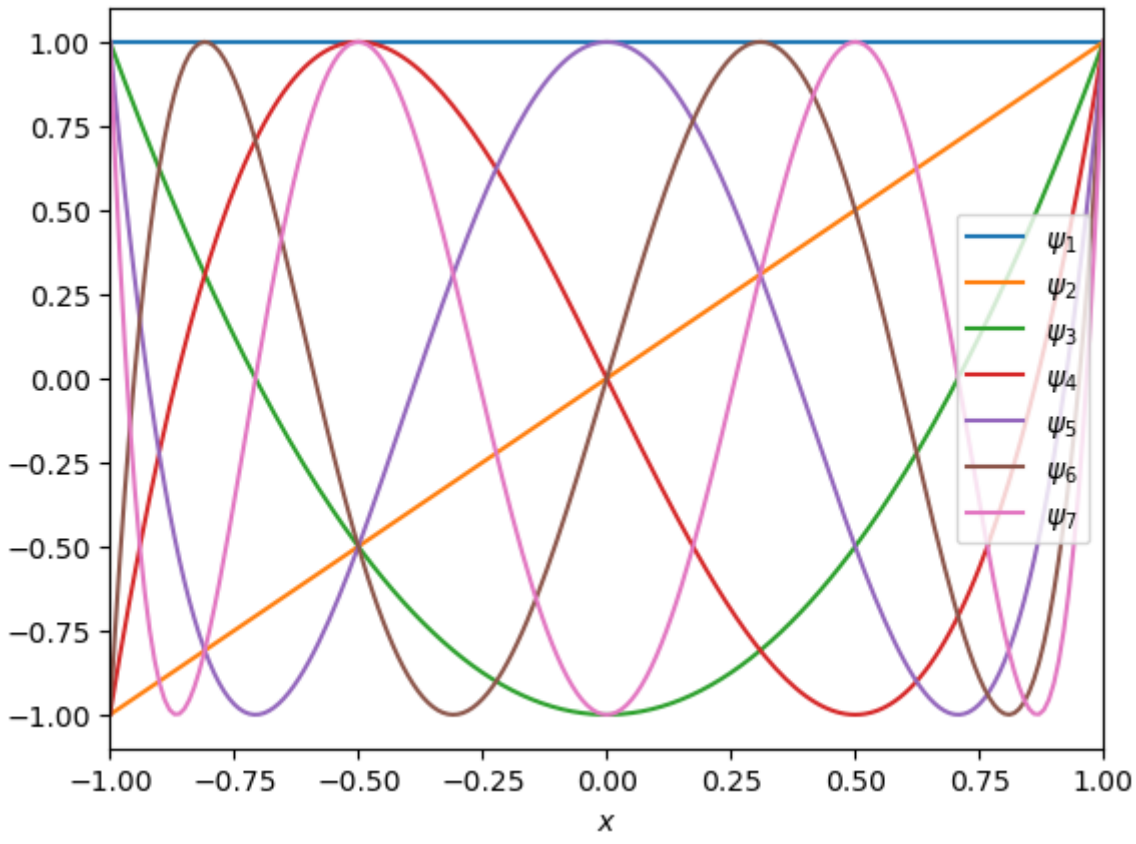
Let's start with a more varied-looking basis, the Chebyshev polynomials:

In [293]:
```
cheby(x,k) = cos(k*acos(x))# Chebyshev polynomials
K = 10
Psi0 = [x->cheby(x,j-1) for j = 1:K]
Psi1 = [x->cheby(f(x),j-1) for j = 1:K];
```

Let's start with a more varied-looking basis, the Chebyshev polynomials:

In [293]:
```
cheby(x,k) = cos(k*acos(x))# Chebyshev polynomials
K = 10
Psi0 = [x->cheby(x,j-1) for j = 1:K]
Psi1 = [x->cheby(f(x),j-1) for j = 1:K];
```

In [19]:
```
for j = 1:7
    plot(-1:0.01:1,Psi0[j].(-1:0.01:1),label="\$\\psi_{$j}\$")
end
legend()
xlim(-1,1);xlabel("\$x\$");
```



Out[19]: PyObject Text(0.5, 24.0, 'x')

In [20]:
```
Psi00prod =  [integrate(Psi0[j],Psi0[k]) for j = 1:K, k =1:K]
Psi01prod =  [integrate(Psi0[j],Psi1[k]) for j = 1:K, k =1:K]
Koop = Psi00prod \ Psi01prod
```

Out[20]:
```
10×10 Matrix{Float64}:
 1.0   4.52971e-15  -0.0214127     5.0921e-15   …  -0.125391
1.22947e-15
 0.0   0.788235      8.50137e-15   0.0402168       -3.70934e-15
-0.224303
 0.0   6.54614e-15   0.764387      1.25415e-14     -0.239389
1.81346e-15
 0.0   0.292259      9.70532e-15   0.494942        -5.01597e-15
-0.230122
 0.0   3.21595e-15   0.557615      9.376e-15       -0.266114
1.79302e-15
 0.0  -0.272908      5.0691e-15    0.771012     …  -2.66837e-15
-0.242802
 0.0   4.76528e-15  -0.283377      8.35098e-15     -0.254304
1.41133e-15
 0.0   0.139016      2.35355e-15  -0.350378        -7.94364e-16
-0.245897
 0.0   2.08377e-15   0.254976      7.53789e-15     -0.174268
-1.44164e-15
 0.0  -0.189962      1.88236e-15   0.16302         -1.33698e-15
-0.167224
```

In [20]:
```
Psi00prod =  [integrate(Psi0[j],Psi0[k]) for j = 1:K, k =1:K]
Psi01prod =  [integrate(Psi0[j],Psi1[k]) for j = 1:K, k =1:K]
Koop = Psi00prod \ Psi01prod
```

Out[20]:
```
10×10 Matrix{Float64}:
 1.0   4.52971e-15  -0.0214127     5.0921e-15   …  -0.125391
1.22947e-15
 0.0   0.788235      8.50137e-15   0.0402168       -3.70934e-15
-0.224303
 0.0   6.54614e-15   0.764387      1.25415e-14     -0.239389
1.81346e-15
 0.0   0.292259      9.70532e-15   0.494942        -5.01597e-15
-0.230122
 0.0   3.21595e-15   0.557615      9.376e-15       -0.266114
1.79302e-15
 0.0  -0.272908      5.0691e-15    0.771012     …  -2.66837e-15
-0.242802
 0.0   4.76528e-15  -0.283377      8.35098e-15     -0.254304
1.41133e-15
 0.0   0.139016      2.35355e-15  -0.350378        -7.94364e-16
-0.245897
 0.0   2.08377e-15   0.254976      7.53789e-15     -0.174268
-1.44164e-15
 0.0  -0.189962      1.88236e-15   0.16302         -1.33698e-15
-0.167224
```

In [21]:
```
cond(Psi00prod)
```

Out[21]:
```
13.508581342539028
```

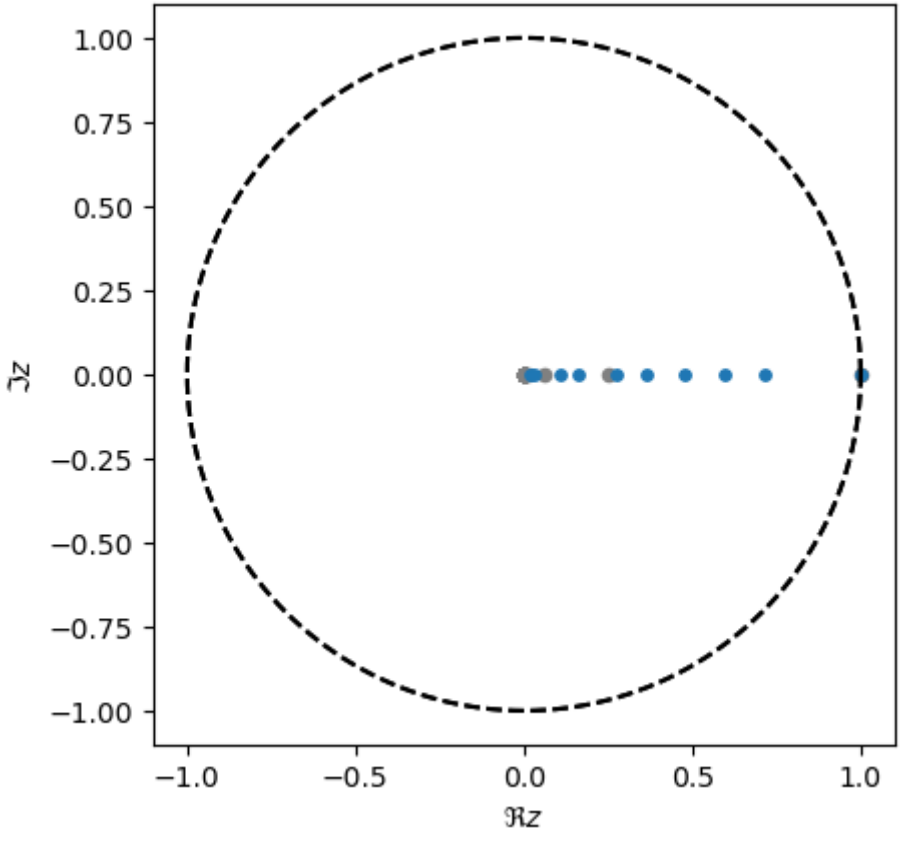
In [20]:
```
Psi00prod =  [integrate(Psi0[j],Psi0[k]) for j = 1:K, k =1:K]
Psi01prod =  [integrate(Psi0[j],Psi1[k]) for j = 1:K, k =1:K]
Koop = Psi00prod \ Psi01prod
```

Out[20]:
```
10×10 Matrix{Float64}:
 1.0   4.52971e-15  -0.0214127     5.0921e-15   …  -0.125391
1.22947e-15
 0.0   0.788235      8.50137e-15   0.0402168       -3.70934e-15
-0.224303
 0.0   6.54614e-15   0.764387      1.25415e-14     -0.239389
1.81346e-15
 0.0   0.292259      9.70532e-15   0.494942        -5.01597e-15
-0.230122
 0.0   3.21595e-15   0.557615      9.376e-15       -0.266114
1.79302e-15
 0.0  -0.272908      5.0691e-15    0.771012     …  -2.66837e-15
-0.242802
 0.0   4.76528e-15  -0.283377      8.35098e-15     -0.254304
1.41133e-15
 0.0   0.139016      2.35355e-15  -0.350378        -7.94364e-16
-0.245897
 0.0   2.08377e-15   0.254976      7.53789e-15     -0.174268
-1.44164e-15
 0.0  -0.189962      1.88236e-15   0.16302         -1.33698e-15
-0.167224
```

In [21]:
```
cond(Psi00prod)
```

Out[21]:
```
13.508581342539028
```

In [22]:
```
spectrumplot(0.25.^(0:100),c="grey",s=20) # true eigenvalues
spectrumplot(eigvals(Koop),s=15)
```



Out[22]:
```
PyObject Text(24.156249999999986, 0.5, '$\\Im z$')
```

How to do it without doing integrals?

In these examples, we have computed the integrals between the ψ_j's and $\psi_k \circ f$'s using a numerical integrator. In higher dimensions this becomes slow to impossible.

Or, we might not know our measure μ or f explicitly!

We can approximate the continuous measure μ by a discrete measure

$$\mu_N := \frac{1}{N} \sum_{n=1}^{N} \delta_{x_n}$$

for some appropriate points $\{x_n\}$. Then, we are interested in approximating our operator $\mathcal{A}$ by least-squares with respect to the inner product

$$\langle \phi, \chi \rangle_{L^2(\mu_N)} = \frac{1}{N} \sum_{n=1}^{N} \bar{\phi}(x_n)\chi(x_n).$$

Thus, it makes sense to represent a function ϕ as a column vector $\{\phi(x_n)\}_{n=1,\ldots,N}$ with the Euclidean inner product. So we get:

$$\Psi_0 = \begin{bmatrix} \psi_1(x_1) & \psi_2(x_1) & \cdots & \psi_K(x_1) \\ \vdots & \vdots & & \vdots \\ \psi_1(x_N) & \psi_2(x_N) & \cdots & \psi_K(x_N) \end{bmatrix}$$

and similarly for Ψ_1.

In [338]:
```
N = 200000
x = range(-1,1,length=N)

K = 40
Psi0 = [cheby(x[n],j-1) for n = 1:N, j = 1:K]
Psi1 = [cheby(f(x[n]),j-1) for n = 1:N, j=1:K]
# for a map with a hole: Psi1 = [(f(x[n])<0.7)*cheby(f(x[n]),j-1) for n
```

Out[338]:
```
200000×40 Matrix{Float64}:
 1.0  -1.0       1.0       -1.0       …  -1.0       1.0       -
1.0
 1.0  -0.999986  0.999944  -0.999874     -0.980895  0.979852  -
0.978781
 1.0  -0.999972  0.999888  -0.999748     -0.961912  0.959839  -
0.957713
 1.0  -0.999958  0.999832  -0.999622     -0.943049  0.939961  -
0.936794
 1.0  -0.999944  0.999776  -0.999496     -0.924308  0.920217  -
0.916024
 1.0  -0.99993   0.99972   -0.99937   …  -0.905687  0.900608  -
0.895402
 1.0  -0.999916  0.999664  -0.999244     -0.887186  0.881131  -
0.874928
 1.0  -0.999902  0.999608  -0.999118     -0.868805  0.861787  -
0.854601
 1.0  -0.999888  0.999552  -0.998992     -0.850542  0.842576  -
0.834421
 1.0  -0.999874  0.999496  -0.998866     -0.832398  0.823496  -
0.814386
 1.0  -0.99986   0.99944   -0.99874   …  -0.814372  0.804547  -
```

```
0.794497
 1.0  -0.999846  0.999384  -0.998614     -0.796463  0.785729  -
0.774752
 1.0  -0.999832  0.999328  -0.998488     -0.778672  0.76704   -
0.755151
 ⋮                                    ⋱
 1.0   0.999846  0.999384   0.998614      0.796463  0.785729
0.774752
 1.0   0.99986   0.99944    0.99874       0.814372  0.804547
0.794497
 1.0   0.999874  0.999496   0.998866  …   0.832398  0.823496
0.814386
 1.0   0.999888  0.999552   0.998992      0.850542  0.842576
0.834421
 1.0   0.999902  0.999608   0.999118      0.868805  0.861787
0.854601
 1.0   0.999916  0.999664   0.999244      0.887186  0.881131
0.874928
 1.0   0.99993   0.99972    0.99937       0.905687  0.900608
0.895402
 1.0   0.999944  0.999776   0.999496  …   0.924308  0.920217
0.916024
 1.0   0.999958  0.999832   0.999622      0.943049  0.939961
0.936794
 1.0   0.999972  0.999888   0.999748      0.961912  0.959839
0.957713
 1.0   0.999986  0.999944   0.999874      0.980895  0.979852
0.978781
 1.0   1.0       1.0        1.0           1.0       1.0
1.0
```

Now Ψ_0, Ψ_1 are matrices and we can say:

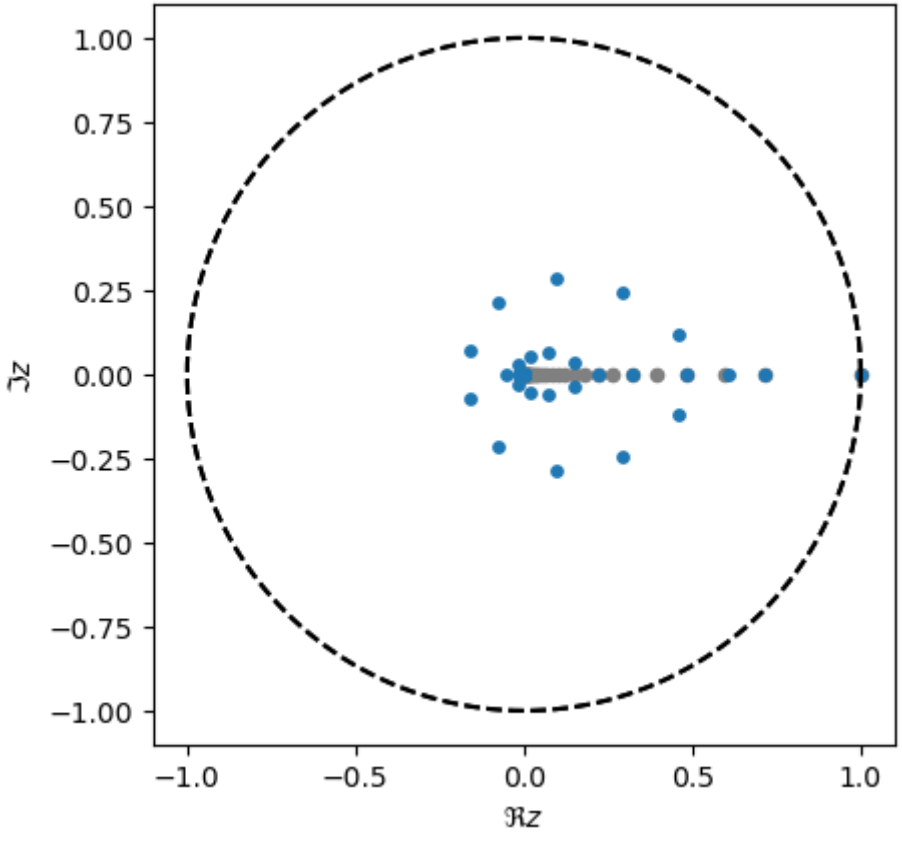
Now Ψ_0, Ψ_1 are matrices and we can say:

In [343]:
```
Koop =  Psi0 \ Psi1; # = (Psi0' * Psi0) \ (Psi0' * Psi1)
```

Now Ψ_0, Ψ_1 are matrices and we can say:

In [343]:
```
Koop =  Psi0 \ Psi1; # = (Psi0' * Psi0) \ (Psi0' * Psi1)
```

In [341]:
```
using LinearAlgebra, PyPlot
spectrumplot(true_eigs,c="grey",s=20) # true eigenvalues
spectrumplot(eigvals(Koop),s=15)
```



Out[341]:

```
PyObject Text(24.156249999999986, 0.5, '$\\Im z$')
```

If we want to sample from the physical measure ρ of our system, we could also set $x_n = f^n(x_0)$:

If we want to sample from the physical measure ρ of our system, we could also set $x_n = f^n(x_0)$:

In [302]:
```
N = 10^3
y = rand()
for i = 1:10000 # spinup so that x_0 ~ ϱ
    y = f(y)
end
x = Array{Float64}(undef,N+1) #initialise an empty array
for n = 1:N+1
    y = f(y)
    x[n] = y
end
x
```

Out[302]:
```
1001-element Vector{Float64}:
 -0.8837234610509348
 -0.8291007053653247
 -0.7432170461674767
 -0.600941373407094
 -0.3457692502950017
  0.17273377390273606
 -0.5687943618408953
 -0.2847491251491329
  0.3083015131761724
 -0.2554459595579824
  0.37499208810194484
 -0.10939201061837747
  0.7227607395844974
```

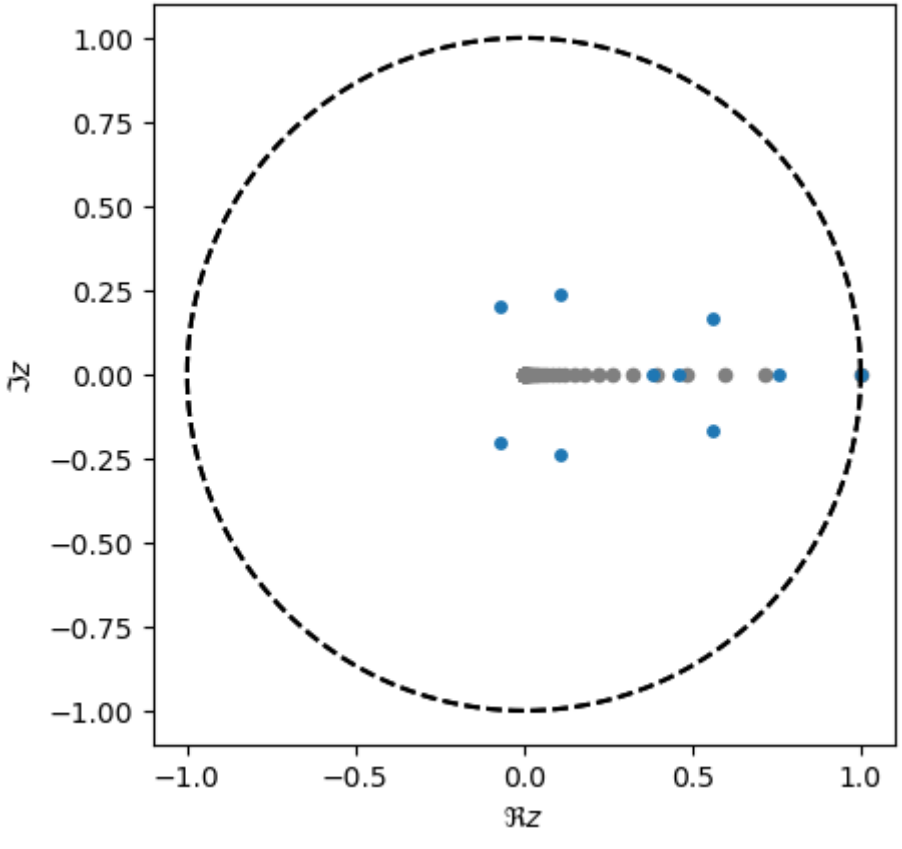
```
 ⋮
-0.3803078717629952
 0.09797997981115181
-0.7510120983572816
-0.6142199526015277
-0.3706121866596783
 0.11882035042523731
-0.6995380542994887
-0.5251868475308019
-0.1999930686884799
 0.5040163579240133
 0.15802303716944238
-0.6041228715252028
```

In [303]:

```
Psi0 = [cheby(x[n],j-1) for n = 1:N, j = 1:K]
Psi1 = [cheby(x[n+1],j-1) for n = 1:N, j=1:K];
```

In [303]:
```
Psi0 = [cheby(x[n],j-1) for n = 1:N, j = 1:K]
Psi1 = [cheby(x[n+1],j-1) for n = 1:N, j=1:K];
```

In [304]:
```
Koop = Psi0\Psi1
spectrumplot(true_eigs,c="grey",s=20) # true eigenvalues
spectrumplot(eigvals(Koop),s=15);
```



Example 2: something a bit wilder

Here is the Lozi map, a piecewise(!) uniformly hyperbolic map of the circle:

$$l(x_1, x_2) = (1 - 1.7|x_1| + 0.5x_2, x_1)$$

In [354]:
```
lozi(x) = (1 - 1.7abs(x[1])+0.5x[2],x[1])
```

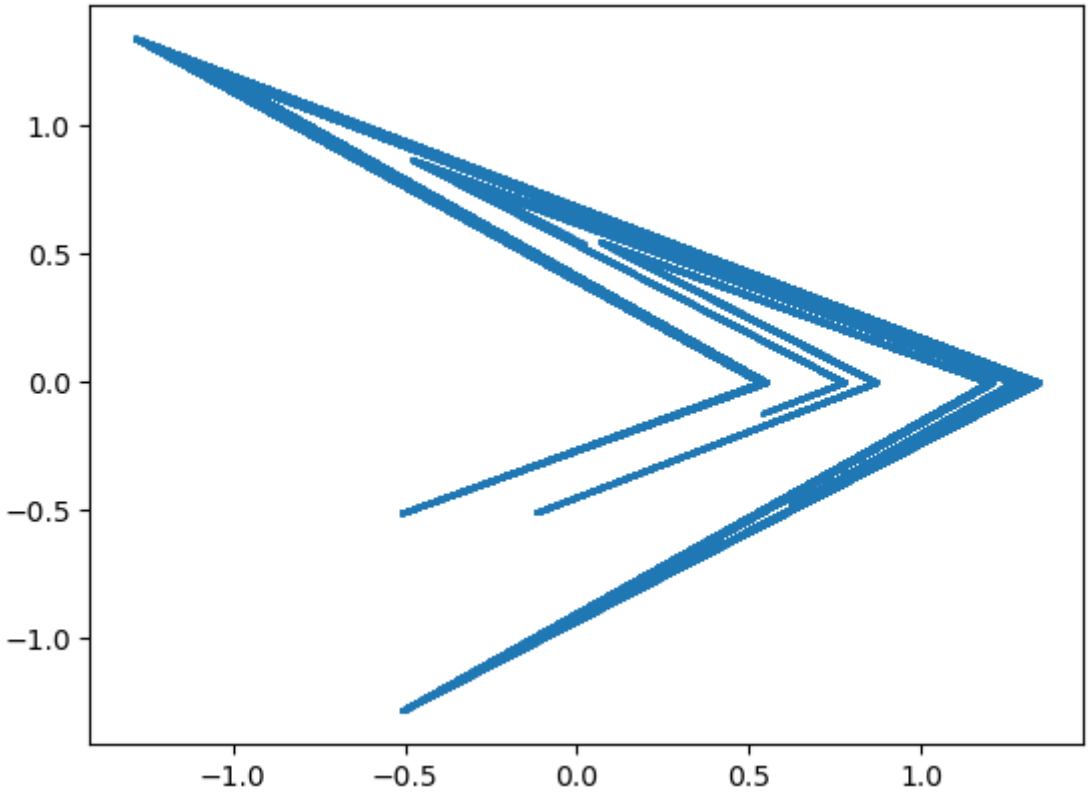
Out[354]:
```
lozi (generic function with 1 method)
```

In [354]:
```
lozi(x) = (1 - 1.7abs(x[1])+0.5x[2],x[1])
```

Out[354]:
```
lozi (generic function with 1 method)
```

In [355]:
```
N = 10^6
y = (rand(),rand())
for i = 1:10000 # spinup so that x_0 ~ ϱ
    y = lozi(y)
end
x = Array{typeof(y)}(undef,N+1) #initialise an empty array
for n = 1:N+1
    y = lozi(y)
    x[n] = y
end

scatter(getindex.(x,1),getindex.(x,2),s=0.5);
# xlim(0,1);ylim(0,1)
```



Out[355]:

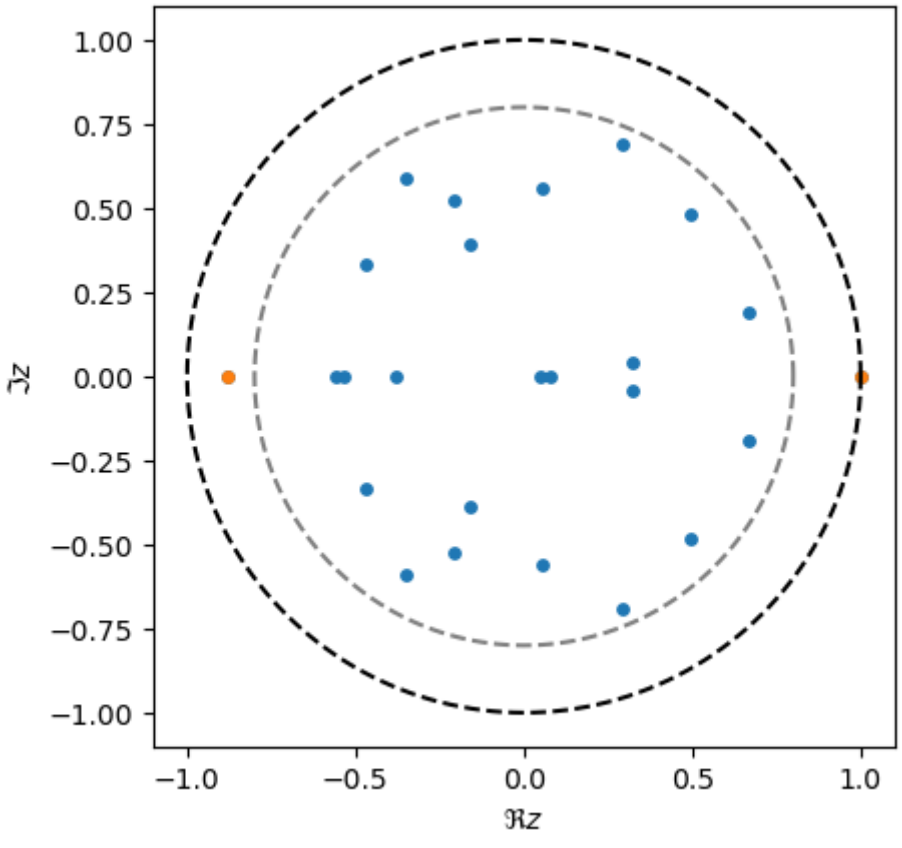
```
PyObject <matplotlib.collections.PathCollection object at 0x7f9
660164e80>
```

In [356]:

```
sqrtK = 5;
cheby2D_01(x,j,k) = cheby(x[1]/1.5,j-1) * cheby(x[2]/1.5,k-1) # our bas

Psi0 = reshape([cheby2D_01(x[n],j,k) for n = 1:N, j = 1:sqrtK, k = 1:sq
Psi1 = reshape([cheby2D_01(x[n+1],j,k) for n = 1:N, j = 1:sqrtK, k = 1:
Koop = Psi0 \ Psi1
Koopvals, Koopvecs = eigen(Koop)

spectrumplot(Koopvals,s=15)
scatter(reim(Koopvals[[1,end]])...,s=15,c="C1");
plot(0.8cos.((0:0.01:2)*pi),0.8sin.((0:0.01:2)*pi),"--",c="grey")
```



Out[356]:

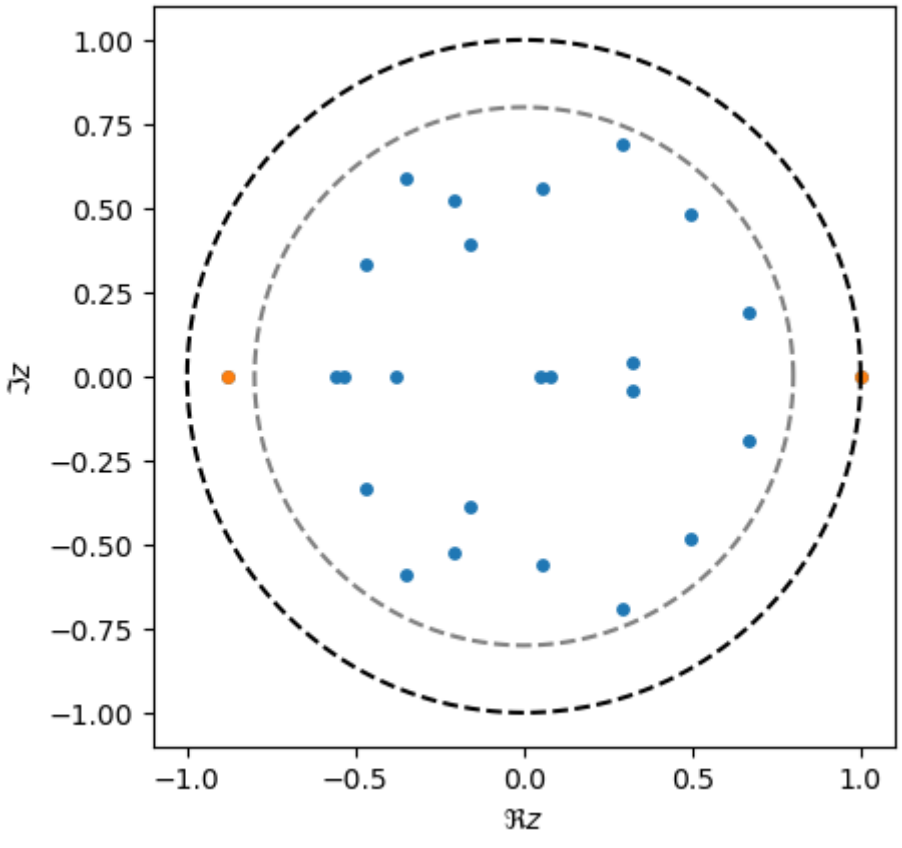
```
1-element Vector{PyCall.PyObject}:
 PyObject <matplotlib.lines.Line2D object at 0x7f965f3a5be0>
```

In [356]:

```
sqrtK = 5;
cheby2D_01(x,j,k) = cheby(x[1]/1.5,j-1) * cheby(x[2]/1.5,k-1) # our bas

Psi0 = reshape([cheby2D_01(x[n],j,k) for n = 1:N, j = 1:sqrtK, k = 1:sq
Psi1 = reshape([cheby2D_01(x[n+1],j,k) for n = 1:N, j = 1:sqrtK, k = 1:
Koop = Psi0 \ Psi1
Koopvals, Koopvecs = eigen(Koop)

spectrumplot(Koopvals,s=15)
scatter(reim(Koopvals[[1,end]])...,s=15,c="C1");
plot(0.8cos.((0:0.01:2)*pi),0.8sin.((0:0.01:2)*pi),"--",c="grey")
```



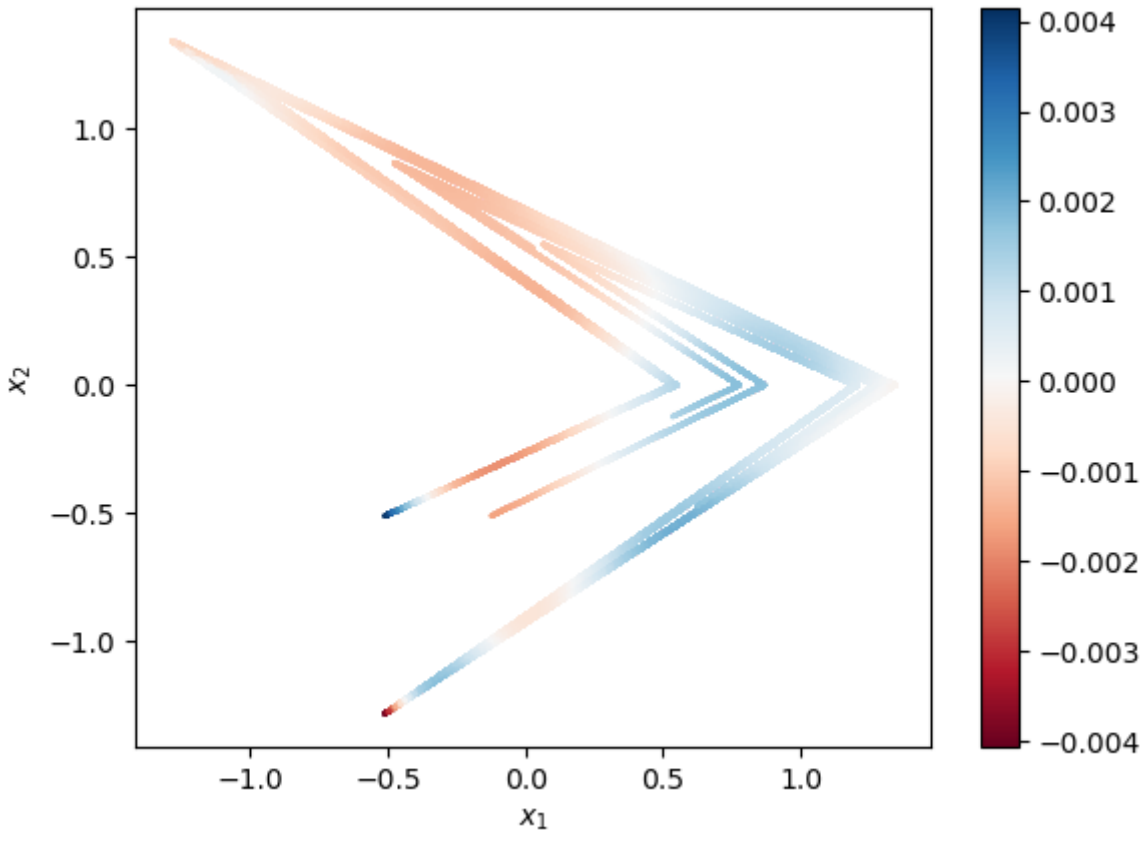
Out[356]:
```
1-element Vector{PyCall.PyObject}:
 PyObject <matplotlib.lines.Line2D object at 0x7f965f3a5be0>
```

The blue eigenvalues are probably essential spectrum artefacts.

However, the leftmost eigenvalue tells us something nice about almost periodic subsets for the map:

However, the leftmost eigenvalue tells us something nice about almost periodic subsets for the map:

In [349]:
```
eigval_index = 1
xlabel("\$x_1\$"); ylabel("\$x_2\$")
scatter(getindex.(x[1:N],1),getindex.(x[1:N],2),s=0.5, c=Psi0*Koopvecs[
colorbar()
Koopvals[eigval_index]
```




```
/home/caro/.julia/conda/3/lib/python3.9/site-packages/matplotli
b/axes/_axes.py:4193: ComplexWarning: Casting complex values to
real discards the imaginary part
  c = np.asanyarray(c, dtype=float)
```

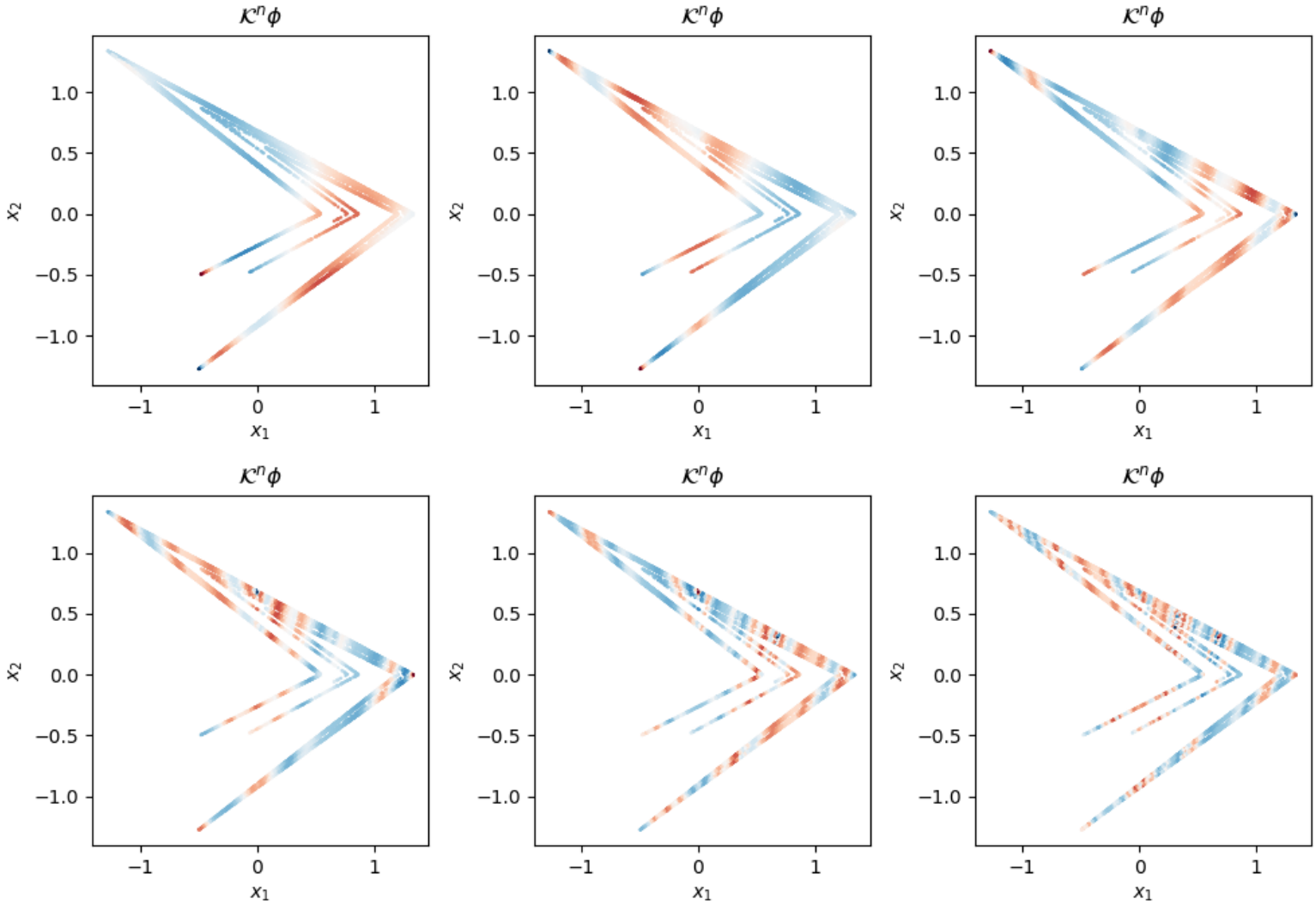
Out[349]:

```
-0.8793423882248114 + 0.0im
```

In [358]:

```
eigvalgraph_slides
```

Out[358]:


$\mathcal{K}^n\phi$
$\mathcal{K}^n\phi$
$\mathcal{K}^n\phi$
$\mathcal{K}^n\phi$
$\mathcal{K}^n\phi$
$\mathcal{K}^n\phi$
x_1
x_2
1.0
0.5
0.0
−0.5
−1.0
−1
0
1

We could also try the Henon map:

We could also try the Henon map:

In [316]:
```
henon(x) = (1 - 1.4x[1]^2+0.1x[2],x[1])
```

Out[316]:
```
henon (generic function with 1 method)
```

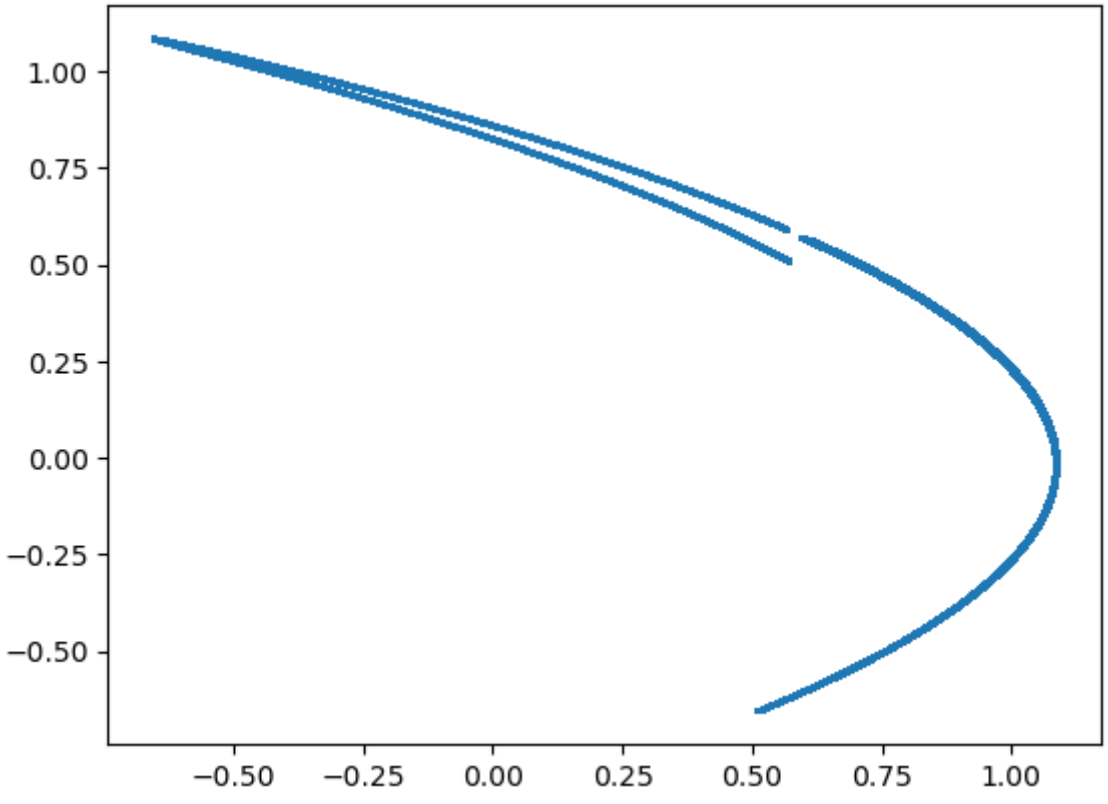
We could also try the Henon map:

In [316]:
```
henon(x) = (1 - 1.4x[1]^2+0.1x[2],x[1])
```

Out[316]:
```
henon (generic function with 1 method)
```

In [317]:
```
N = 10^6
y = (rand(),rand())
for i = 1:10000 # spinup so that x_0 ~ ϱ
    y = henon(y)
end
x = Array{typeof(y)}(undef,N+1) #initialise an empty array
for n = 1:N+1
    y = henon(y)
    x[n] = y
end

scatter(getindex.(x,1),getindex.(x,2),s=0.5);
# xlim(0,1);ylim(0,1)
```



Out[317]:

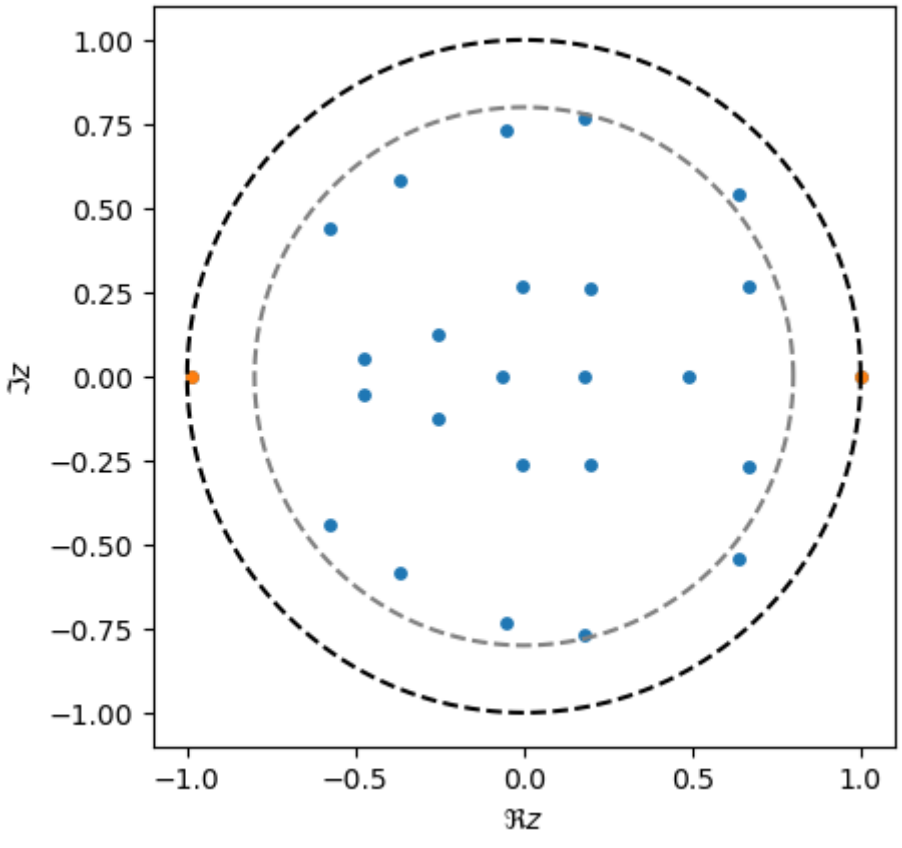
```
PyObject <matplotlib.collections.PathCollection object at 0x7f9
5feee9940>
```

In [318]:

```
sqrtK = 5;
cheby2D_01(x,j,k) = cheby(x[1]/1.5,j-1) * cheby(x[2]/1.5,k-1) # our bas

Psi0 = reshape([cheby2D_01(x[n],j,k) for n = 1:N, j = 1:sqrtK, k = 1:sq
Psi1 = reshape([cheby2D_01(x[n+1],j,k) for n = 1:N, j = 1:sqrtK, k = 1:
Koop = Psi0 \ Psi1
Koopvals, Koopvecs = eigen(Koop)

spectrumplot(Koopvals,s=15)
scatter(reim(Koopvals[[1,end]])...,s=15,c="C1");
plot(0.8cos.((0:0.01:2)*pi),0.8sin.((0:0.01:2)*pi),"--",c="grey")
```



Out[318]:

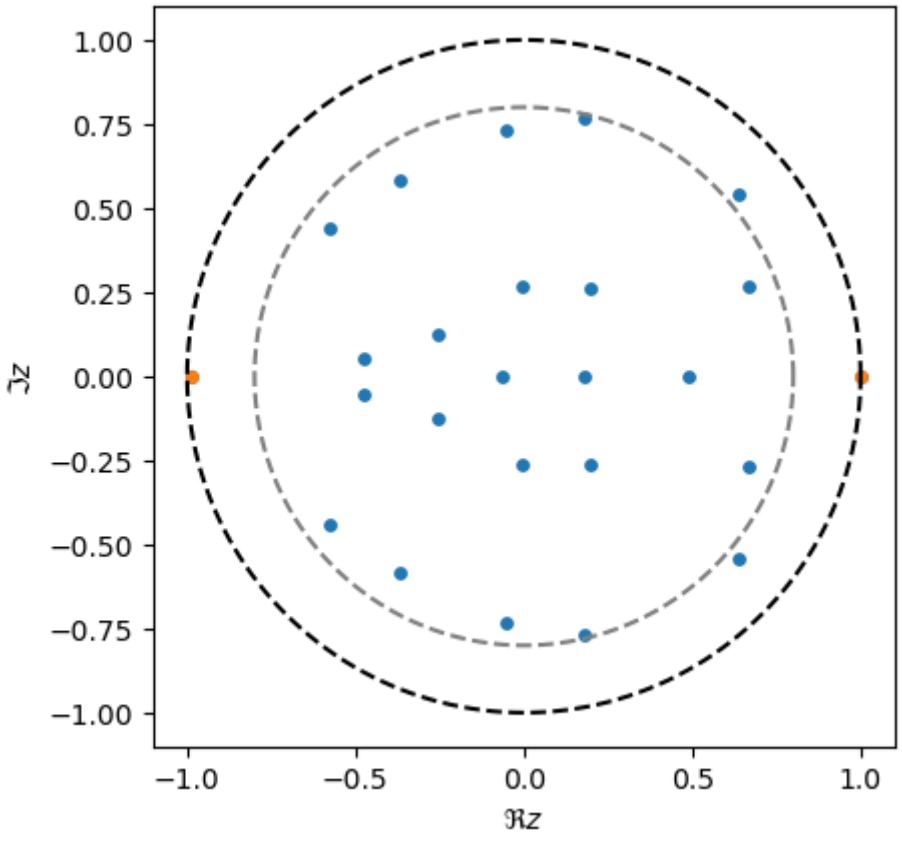
```
1-element Vector{PyCall.PyObject}:
 PyObject <matplotlib.lines.Line2D object at 0x7f95fc0eea30>
```

In [318]:

```
sqrtK = 5;
cheby2D_01(x,j,k) = cheby(x[1]/1.5,j-1) * cheby(x[2]/1.5,k-1) # our bas

Psi0 = reshape([cheby2D_01(x[n],j,k) for n = 1:N, j = 1:sqrtK, k = 1:sq
Psi1 = reshape([cheby2D_01(x[n+1],j,k) for n = 1:N, j = 1:sqrtK, k = 1:
Koop = Psi0 \ Psi1
Koopvals, Koopvecs = eigen(Koop)

spectrumplot(Koopvals,s=15)
scatter(reim(Koopvals[[1,end]])...,s=15,c="C1");
plot(0.8cos.((0:0.01:2)*pi),0.8sin.((0:0.01:2)*pi),"--",c="grey")
```



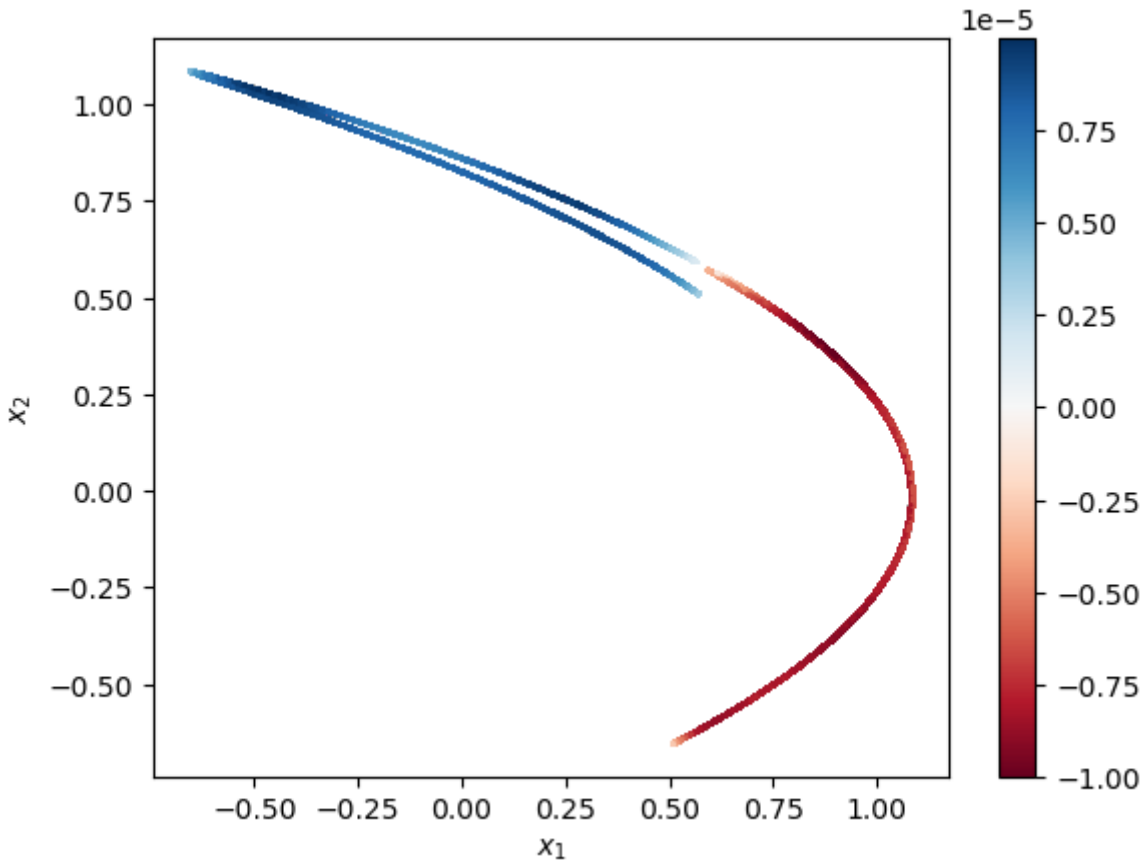
Out[318]:
```
1-element Vector{PyCall.PyObject}:
 PyObject <matplotlib.lines.Line2D object at 0x7f95fc0eea30>
```

The blue eigenvalues are probably essential spectrum artefacts.

From the left orange eigenvalue (almost on the unit circle) we see two mixing components:

In [319]:

```
eigval_index = 1
xlabel("\$x_1\$"); ylabel("\$x_2\$")
scatter(getindex.(x[1:N],1),getindex.(x[1:N],2),s=0.5, c=Psi0*Koopvecs[
colorbar()
Koopvals[eigval_index]
```




```
/home/caro/.julia/conda/3/lib/python3.9/site-packages/matplotli
b/axes/_axes.py:4193: ComplexWarning: Casting complex values to
```

```
real discards the imaginary part
  c = np.asanyarray(c, dtype=float)
```

Out[319]:

```
-0.9853943567902816 + 0.0im
```

We have been looking at various (data-based) Galerkin discretisations of the Koopman operator $\mathcal{K}\phi = \phi \circ f$ (using nice basis functions).

This known as

Extended Dynamical Mode Decomposition

This is big for engineers and scientists.

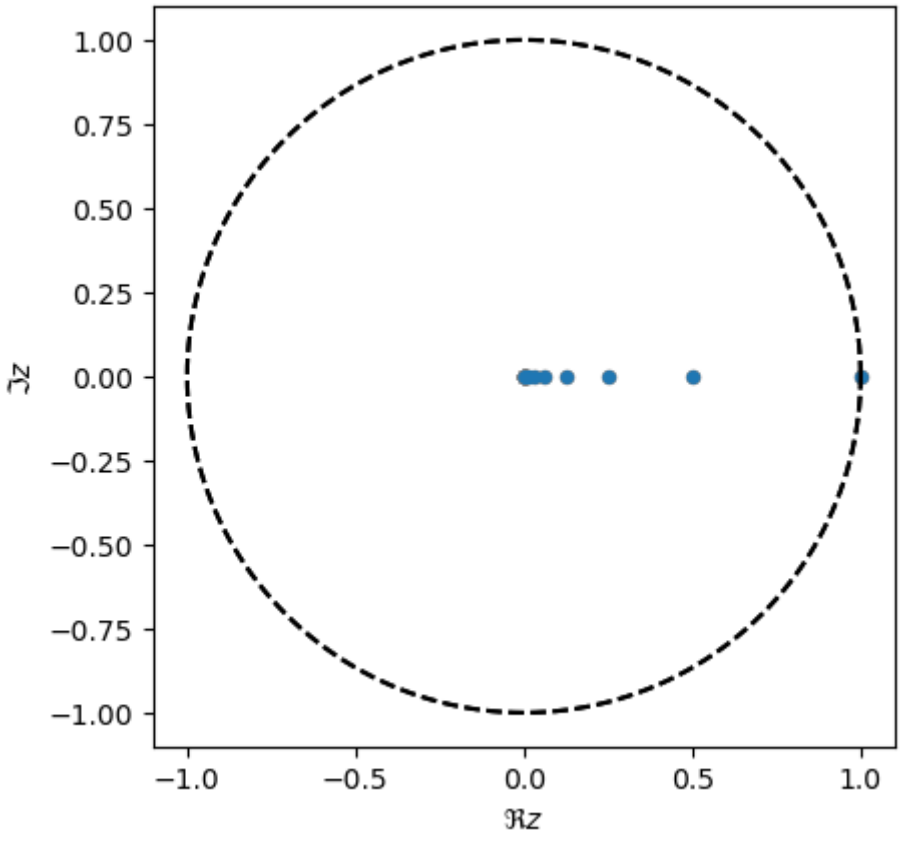
It generalises regular DMD, which only uses ψ_k linear.

We can try discretising other operators in a similar fashion.

In [321]:

```
N = 10^3
x = rand(N)
Psi0 = [cheby(x[n],j-1) for n = 1:N, j = 1:K]
Psi1 = [cheby((x[n]+1)/2,j-1)/2 + cheby((x[n]-1)/2,j-1)/2 # transfer op
            for n = 1:N, j=1:K]

Transfer = Psi0 \ Psi1
spectrumplot(0.5.^(0:1:20),c="grey",s=20) # true eigenvalues
spectrumplot(eigvals(Transfer),s=15);
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0
ℜz
ℑz

Many common transfer operator discretisation algorithms are Galerkin algorithms.

Name	operator discretised	ψ_k	μ	μ_N
Ulam's method	$\mathcal{K}$ †	characteristic functions $\{1_E\}_{E \in P}$	Lebesgue	varies
Higher-order Ulam's method	$\mathcal{L}$	C^k bump functions	Lebesgue	
Lagrange-Chebyshev	$\mathcal{L}$	Chebyshev polys on $[-1, 1]$	$\frac{\mathrm{d}x}{\sqrt{1-x^2}}$	Chebyshev nodes $\cos \pi \frac{2n-1}{2N}, n = 1, \ldots, N$
Lagrange-Fourier	$\mathcal{L}$	complex unit circle	Lebesgue	Evenly spaced notes
Dynamical Mode Decomposition	$\mathcal{K}$	linear functions	phys. measure †	empirical measure of a time series †
Extended DMD	$\mathcal{K}$		phys. measure †	empirical measure of a time series †

† = usually

Many common transfer operator discretisation algorithms are Galerkin algorithms.

Name	operator discretised	ψ_k	μ	μ_N
Ulam's method	$\mathcal{K}$ †	characteristic functions $\{1_E\}_{E \in P}$	Lebesgue	varies
Higher-order Ulam's method	$\mathcal{L}$	C^k bump functions	Lebesgue	
Lagrange-Chebyshev	$\mathcal{L}$	Chebyshev polys on $[-1, 1]$	$\frac{\mathrm{d}x}{\sqrt{1-x^2}}$	Chebyshev nodes $\cos \pi \frac{2n-1}{2N}, n = 1, \ldots, N$
Lagrange-Fourier	$\mathcal{L}$	complex unit circle	Lebesgue	Evenly spaced notes
Dynamical Mode Decomposition	$\mathcal{K}$	linear functions	phys. measure †	empirical measure of a time series †
Extended DMD	$\mathcal{K}$		phys. measure †	empirical measure of a time series †

† = usually

Often these special methods have some nice structure that makes things algorithmically, e.g. $\Psi_0^*\Psi_0$ is diagonal, or $\Psi_1^*\Psi_0$ is sparse, or...

In [ ]: