Eigentools Documentation

Dedalus Collaboration

Jun 22, 2021

CONTENTS

Eigentools is a set of tools for studying linear eigenvalue problems. The underlying eigenproblems are solved using Dedalus, which provides a domain-specific language for partial differential equations. Each entry in the following list of features links to a Jupyter notebook giving an example of its use.

- *automatic rejection of unresolved eigenvalues*
- *simple plotting of drift ratios (both ordinal and nearest) to evaluate tolerance for eigenvalue rejection*
- *simple plotting of specified eigenmodes*
- *simple plotting of spectra*
- *computation of pseudospectra for any Differential-Algebraic Equations* with *user-specifiable norms*
- *tools to find critical parameters for linear stability analysis* with *user-specifiable definitions of growth and stability*
- *ability to project eigenmode onto 2- or 3-D domain for visualization*
- *ability to output projected eigenmodes as Dedalus-formatted HDF5 file to be used as initial conditions for Initial Value Problems*

CHAPTER

ONE

CONTENTS

1.1 Installing eigentools

eigentools requires Dedalus, which you can install via any of the methods found in the Dedalus installation instructions.

Once Dedalus is installed, eigentools is *pip* installable:

```
pip install eigentools
```

If you would like the development version, you can clone the repository and install locally:

```
git clone https://github.com/DedalusProject/eigentools.git
pip install -e eigentools
```

1.2 Getting Started

eigentools comes with several examples to get you started, but let's outline some basics in a very simple problem, the 1-D wave equation with $u = 0$ at both ends. This is not quite as trivial a problem as it might seem, because we are expanding the solution in Chebyshev polynomials, but the eigenmodes are sines and cosines.

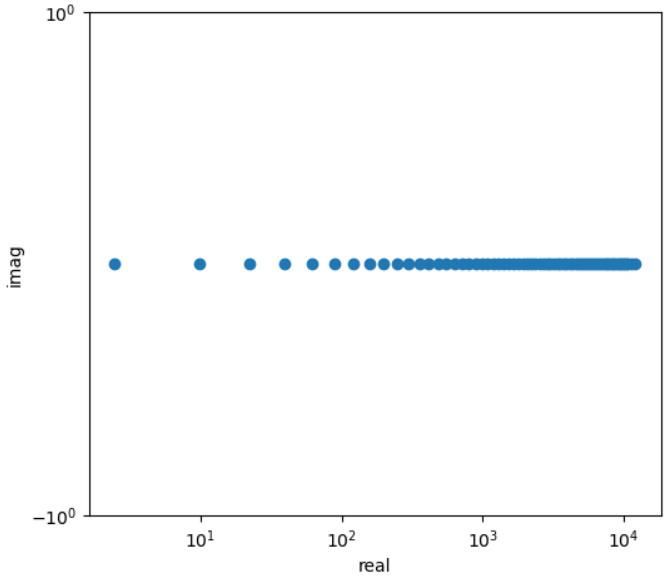
```
from eigentools import Eigenproblem
import dedalus.public as de

Nx = 128
x = de.Chebyshev('x',Nx, interval=(-1, 1))
d = de.Domain([x])

string = de.EVP(d, ['u','u_x'], eigenvalue='omega')
string.add_equation("omega*u + dx(u_x) = 0")
string.add_equation("u_x - dx(u) = 0")
string.add_bc("left(u) = 0")
string.add_bc("right(u) = 0")

EP = Eigenproblem(string)
EP.solve(sparse=False)
ax = EP.plot_spectrum()
print("there are {} good eigenvalues.".format(len(EP.evalues)))
ax.set_ylim(-1,1)
ax.figure.savefig('waves_spectrum.png')
```

(continues on next page)

(continued from previous page)

```
ax = EP.plot_drift_ratios()
ax.figure.savefig('waves_drift_ratios.png')
```

That code takes about 10 seconds to run on a 2020 Core-i7 laptop, produces about 68 "good" eigenvalues, and produces the following output:



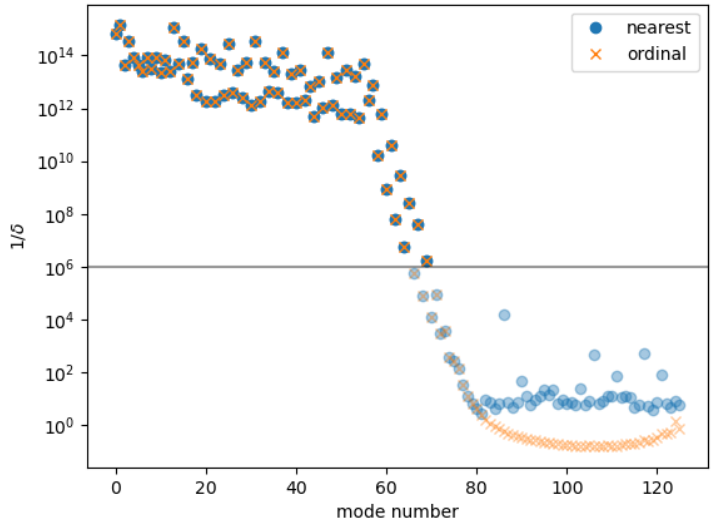
eigentools has taken a Dedalus eigenvalue problem, automatically run it at 1.5 times the specified resolution, rejected any eigenvalues that do not agree to a default precision of one part in 10^{-6} and plotted a spectrum in six extra lines of code!

Most of the plotting functions in eigentools return a *matplotlib axes* object, making it easy to modify the plot defaults. Here, we set the y-limits manually, because the eigenvalues of a string are real. Try removing the *ax.set_ylim(-1,1)* line and see what happens.

1.2.1 Mode Rejection

One of the most important tasks eigentools performs is spurious mode rejection. It does so by computing the "drift ratio" [Boyd2000] between the eigenvalues at the given resolution and a higher resolution problem that eigentools automatically assembles. By default, the "high" resolution case is 1.5 times the given resolution, though this is user configurable via the *factor* keyword option to *Eigenproblem()*.

The drift ratio δ is calculated using either the **ordinal** (e.g. first mode of low resolution to first mode of high resolution) or **nearest** (mode with smallest difference between a given high mode and all low modes). In order to visualize this, *EP.plot_drift_ratios()* in the above code returns an *axes* object making a plot of the *inverse drift ratio* $(1/\delta)$,



Good modes are those *above* the horizontal line at 10^6; bad modes are also grayed out. In this case, the **nearest** and **ordinal** methods produce identical results. If the problem contains more than one wave *family*, **nearest** typically fails. For an example, see the MRI example script. Note that **nearest** is the default criterion used by eigentools.

1.3 Upgrading eigentools scripts

Version 2 of eigentools has made significant changes to the API and will necessitate some changes (for the better, we hope) to the user experience. The guiding principle behind the new API is that one should no longer need to touch the Dedalus `EVP` object that defines the eigenvalue problem at hand.

Most importantly, no changes need to be made to the underlying Dedalus `EVP` object.

1.3.1 Basic `eigenproblem` usage

Choosing a sparse or dense solve is no longer done when instantiating `Eigenproblem` objects. Instead, this is a choice at *solve* time:

```
EP = Eigenproblem(string, reject=True)
EP.solve(sparse=False)
```

Also, notice that rejection of spurious modes is now done automatically with `EP.solve` if `reject=True` is selected at instantiation time. Note that although in the above code, we explicitly set `reject=True`, this is **unnecessary**, as it is the default. The `EP.reject_spurious()` function has been removed

In addition, solving again with different parameters has been greatly simplified from the previous version. You now simply *pass a dictionary* with the parameters you wish to change to solve itself. Let's revisit the simple waves-on-a-string problem from *the getting started page*, but add a parameter, `c2`, the wave speed squared.

Here, we solve twice, once with `c1 = 1` and once with `c2 = 2`. Given the dispersion relation for this problem is $\omega^2 = c^2 k$ and our eigenvalue `omega` is really ω^2, we expect the eigenvalues for the second solve to be twice those for the first.

```
import numpy as np
from eigentools import Eigenproblem
import dedalus.public as de

Nx = 128
x = de.Chebyshev('x',Nx, interval=(-1, 1))
d = de.Domain([x])

string = de.EVP(d, ['u','u_x'], eigenvalue='omega')
string.parameters['c2'] = 1
string.add_equation("omega*u + c2*dx(u_x) = 0")
string.add_equation("u_x - dx(u) = 0")
string.add_bc("left(u) = 0")
string.add_bc("right(u) = 0")
EP = Eigenproblem(string)
EP.solve(sparse=False)
evals_c1 = EP.evalues
EP.solve(sparse=False, parameters={'c2':2})
evals_c2 = EP.evalues

print(np.allclose(evals_c2, 2*evals_c1))
```

Getting eigenmodes

Getting eigenmodes has also been simplified and significantly extended. Previously, getting an eigenmode corresponding to an eigenvalue required using the `set_state()` method on the underlying `EVP` object. In keeping with the principle of not needing to manipulate the `EVP`, we provide a new `.eigenmode(index)`, where `index` is the mode number corresponding to the eigenvalue index in `EP.evalues`. By default, with mode rejection on, these are the "good" eigenmodes. *.eigenmode(index)* returns a Dedalus `FieldSystem` object, with a Dedalus `Field` for each field in the eigenmode:

```
emode = EP.eigenmode(0)
print([f.name for f in emode.fields])
u = emode.fields[0]
u_x = emode.fields[1]
```

1.3.2 Finding critical parameters

This has been considerably cleaned up. The two major things to note are that

1. one no longer needs to create a shim function to translate between an x-y grid and the parameter names within the `EVP`.
2. The parameter grid is no longer defined inside `CriticalFinder`, but is instead created by the user and passed in

For example, here are the relevant changes necessary for the MRI test problem.

First, replace

```
EP = Eigenproblem(mri, sparse=True)
```

(continues on next page)

(continued from previous page)

```
# create a shim function to translate (x, y) to the parameters for the eigenvalue␣
↪problem:
def shim(x,y):
    iRm = 1/x
    iRe = (iRm*Pm)
    print("Rm = {}; Re = {}; Pm = {}".format(1/iRm, 1/iRe, Pm))
    gr, indx, freq = EP.growth_rate({"Q":y,"iRm":iRm,"iR":iRe})
    ret = gr+1j*freq
    return ret

cf = CriticalFinder(shim, comm)
```

with

```
EP = Eigenproblem(mri)

cf = CriticalFinder(EP, ("Q", "Rm"), comm, find_freq=False)
```

Important: note that `find_freq` is specified at instantiation rather than when calling `cf.crit_finder` later.

Once this is done, the grid generation changes from

```
mins = np.array((4.6, 0.5))
maxs = np.array((5.5, 1.5))
ns   = np.array((10,10))
logs = np.array((False, False))

cf.grid_generator(mins, maxs, ns, logs=logs)
```

to

```
nx = 20
ny = 20
xpoints = np.linspace(0.5, 1.5, nx)
ypoints = np.linspace(4.6, 5.5, ny)

cf.grid_generator((xpoints, ypoints), sparse=True)
```

1.4 Example notebooks

1.4.1 Orr Sommerfeld pseudospectra

Let's use the classic Orr-Sommerfeld problem to explore *pseudospectra* with Dedalus.

The Orr-Sommerfeld problem explores the stability of flow down a rectangular channel driven by a pressure gradient. The equilibrium solution is known analytically as plane Poiseuille flow, and is given by

$$U(z) = 1 - z^2.$$

This shear flow becomes unstable despite having no inflection point in the flow. This is in contradiction to Rayleigh's inflection point theorem, which states that a *necessary* condition (for a *sufficient* condition, see Balmforth and Morrison

(1998)). for a shear flow in an *inviscid* fluid to become unstable, it must have a point z_i with

$$\frac{d^2U}{dz^2}(z_i) = 0.$$

This means that for plane Poiseuille flow, the viscosity itself is the source of the instability.

If we write the problem in canonical Dedalus form

$$\mathcal{M}\frac{\partial \mathbf{X}}{\partial t} + \mathcal{L}\mathbf{X} = N(\mathbf{X}),$$

linearize about $U(z)$ such that $N(\mathbf{X}) = 0$, and take the time dependence of $\mathbf{X} \propto e^{-ict}$, we have a generalized eigenvalue problem.

This is a very interesting problem because the linear operator for the Orr-Sommerfeld problem $\mathcal{L}_{OS}$ is non-normal, that is $\mathcal{L}_{OS}^\dagger\mathcal{L}_{OS} \neq \mathcal{L}_{OS}\mathcal{L}_{OS}^\dagger$. Non-normal operators can drive transient growth: even systems in which *all* eigenmodes are linearly stable and will decay to zero, arbitrary initial conditions can be amplified by large factors.

Here, we assume perturbations take the form $u(x,z,t) = \hat{u}e^{i\alpha(x-ct)}$, where $c = \omega/\alpha$ is the wave speed. One must take care to note that $c = c_r + ic_i$. If a perturbation has $c_i > 0$, it will linearly grow. This occurs at a Reynolds number $\mathrm{Re} \simeq 5772$, a result you can also find using `CriticalFinder` (it's actually a test problem for `eigentools`).

However, as we will see below, even at $\mathrm{Re} = 10000$, the growth rate is tiny: $c_i \approx 3.7 \times 10^{-3}$! Yet at those Reynolds numbers, channel flow is clearly unstable. How can this be?

The answer is that the *spectrum* of the Orr-Sommerfeld operator is quite misleading. Because $\mathcal{L}_{OS}$ is non-normal, its eigenfuntions are not orthogonal. As they decay at different rates, the total amplitude of some arbitrary initial condition can **increase**!

This observation was first noted in the pioneering paper by Reddy, Schmid, and Henningson (1993). For a detailed discussion of the Orr-Sommerfeld problem including a thorough discussion of transient growth, see Schmid and Henningson (2001).

[1]:
```
import matplotlib.pyplot as plt
from mpi4py import MPI
from eigentools import Eigenproblem, CriticalFinder
import dedalus.public as de
import numpy as np
import time
```

Orr-Sommerfeld problem

First, we'll define the Orr-Sommerfeld equation in the usual way (following Orszag 1971), in which the Navier-Stokes equations are reduced to a single fourth-order differential equation for the wall-normal velocity w. Because of the first order form of dedalus, this becomes four first order equations.

Then we create an `Eigenproblem` object.

[2]:
```
z = de.Chebyshev('z', 128)
d = de.Domain([z],comm=MPI.COMM_SELF)

alpha = 1.
Re = 10000

orr_somerfeld = de.EVP(d,['w','wz','wzz','wzzz'],'c')
orr_somerfeld.parameters['alpha'] = alpha
```

(continues on next page)

(continued from previous page)

```
orr_somerfeld.parameters['Re'] = Re
orr_somerfeld.substitutions['umean']= '1 - z**2'
orr_somerfeld.substitutions['umeanzz']= '-2'

orr_somerfeld.add_equation('dz(wzzz) - 2*alpha**2*wzz + alpha**4*w -1j*alpha*Re*((umean-
↪c)*(wzz - alpha**2*w) - umeanzz*w) = 0')
orr_somerfeld.add_equation('dz(w)-wz = 0')
orr_somerfeld.add_equation('dz(wz)-wzz = 0')
orr_somerfeld.add_equation('dz(wzz)-wzzz = 0')

orr_somerfeld.add_bc('left(w) = 0')
orr_somerfeld.add_bc('right(w) = 0')
orr_somerfeld.add_bc('left(wz) = 0')
orr_somerfeld.add_bc('right(wz) = 0')

# create an Eigenproblem object
EP = Eigenproblem(orr_somerfeld)
```

```
2021-02-01 17:21:04,593 problems 0/1 INFO :: Solving EVP with homogeneity tolerance of 1.
↪000e-10
2021-02-01 17:21:04,613 problems 0/1 INFO :: Solving EVP with homogeneity tolerance of 1.
↪000e-10
```

Eigenmode rejection

Calling the `solve` method on the `Eigenproblem` object will solve the eigenvalue problem with 128 modes and then solve it again with $3/2 \cdot 128 = 192$ modes. Using the rejection algorithm detailed in Chapter 7 of Boyd (1989), `eigentools` creates `EP.evalues` which holds only the good eigenvalues: those that are the same (to some specified tolerance) between both solves.

Of course, `EP.evalues_low` and `EP.evalues_high` store the low and high resolution eigenvalues, respectively, if you need them.

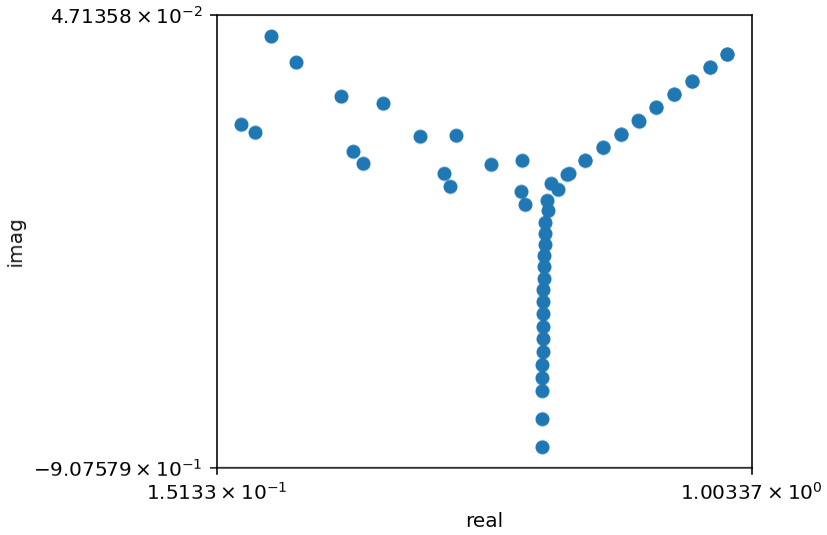
The `sparse=False` option tells Dedalus to use the dense eigenvalue solver, which will retrieve all the eigenmodes (as many as there are modes in the problem). Of course, about half of them will be unusably inaccurate; the rejection methods in `eigentools` hides those from us.

[3]:
```
EP.solve(sparse=False)
```

Plotting Spectra

`eigentools` has the ability to make simple spectra automatically:

[4]:
```
ax = EP.plot_spectrum()
```

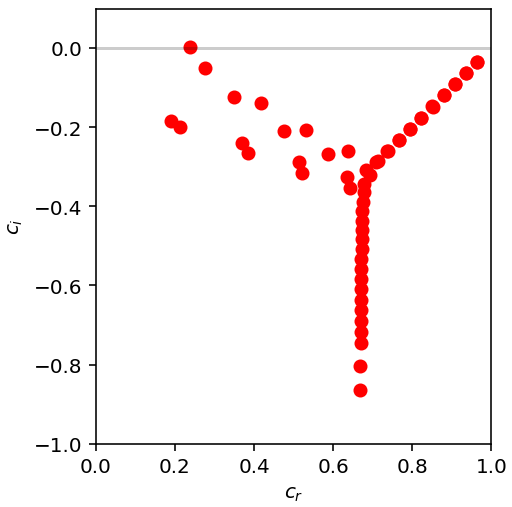


By default, this plotting tool uses double-log plotting (that is, it logs positive and negative values separately); while it is highly configurable, sometimes it's easiest to simply make a plot manually using the real and imaginary parts of the eigenvalues themselves. Because we're going to overplot pseudospectra later, it's easier to use this approach here.

Below, we use standard `matplotlib` plotting tools to construct a simple spectrum with a horizontal line at the stability threshold.

[5]:
```
plt.scatter(EP.evalues.real, EP.evalues.imag,color='red')
plt.axis('square')
plt.axhline(0,color='k',alpha=0.2)
plt.xlim(0,1)
plt.ylim(-1,0.1)
plt.xlabel(r"$c_r$")
plt.ylabel(r"$c_i$")
```

[5]:
```
Text(0, 0.5, '$c_i$')
```



We can find the growth rate also:

[6]:
```
print(r"fastest growth rate: max(c_i) = {:5.3e}".format(np.max(EP.evalues.imag)))
```

```
fastest growth rate: max(c_i) = 3.740e-03
```

Pseudospectra

The `Eigenproblem` object also provides a simple method to calculate pseudospectra, `calc_ps()`. Pseudospectra were originally defined for regular eigenvalue problems. In order to calculate pseudospectra for the differential-algebraic equation systems Dedalus typically provides, we use the algorithm developed by Embree and Keeler (2017). This uses the sparse eigenvalue solver to construct an invariant subspace to approximate a regular eigenvalue problem. Typically, it requires a rather high number of modes, k. Here we set $k = 100$. After we construct the approximate matrix, we feed it to a standard pseudospectrum routine. This requres a set of points in the complex plane at which the pseudospectrum will be computed. We do that by choosing a set of real and imaginary points, and feeding them to the `EP.calc_ps()` method.

[7]:
```
k = 100

psize = 100
real_points = np.linspace(0,1, psize)
imag_points = np.linspace(-1,0.1, psize)
```

Choosing an inner product and norm

The concept of non-normality is inherently linked to the norm used to evaluate orthogonality. When calculating pseudospectra, one must specify the norm to be used.

`eigentools` allows users to pass a norm function to the pseudospectrum calculation. If you do not pass a norm, it will simply use the standard vector 2-norm in the coffiecient space. **This is almost assuredly unphysical**. However, it is simple to write a function to use the energy norm (or anything else).

Here, we will use the energy inner product for the Orr-Sommerfeld form of the equations,

$$< \mathbf{X}_1 | \mathbf{X}_2 >_E = \int_{-1}^{1} w_1^\dagger w_2 + \frac{(\partial_z w_1)^\dagger \partial_z w_2}{\alpha^2} dz,$$

with $\mathbf{X}_{1,2}$ state vectors containing w and its first three derivatives. This inner product induces a norm $||\mathbf{X}||_E$ that is equal to the energy of the perturbation.

The input to the functions will be Dedalus `FieldSystem` objects which will be populated by `EP.calc_ps()`. Note that the function returns the energy itself, that is it calls `field['g'][0]` to return the integral in grid space.

[8]:
```
def energy_norm(X1, X2):
    u1 = X1['wz']/alpha
    w1 = X1['w']
    u2 = X2['wz']/alpha
    w2 = X2['w']

    field = (np.conj(u1)*u2 + np.conj(w1)*w2).evaluate().integrate()
    return field['g'][0]
```

Now we can call `calc_ps` with the points we defined above and `energy_norm`:

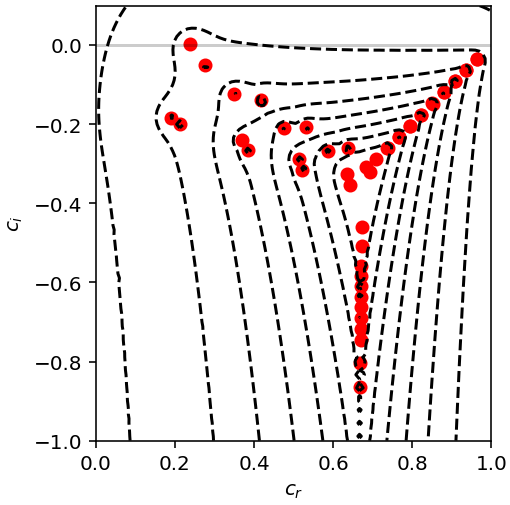
[9]:
```
EP.calc_ps(k, (real_points, imag_points), inner_product=energy_norm)
```

This adds data attributes `EP.pseudospectrum`, `EP.ps_real`, `EP.ps_imag` to the `EP` object, storing the pseudospectrum contours, and the real and imaginary grid for conveninece.

Traditionally, one plots pseudospectra as logarithmic contours over top of the spectrum. Here, the contours are from -1 to -8 from outside in. This means that a perturbation of amplitude $\epsilon \simeq 10^{-1}$ to $\epsilon \simeq 10^{-8}$ will cause an $\mathcal{O}(1)$ change in the eigenvalues within that contour. Importantly, transient growth is possible within all of the contours that cross $c_i > 0$ with amplification factors proportional to $1/\epsilon$. Here, we can clearly see why channel flow is turbulent even when the only unstable mode has such a feeble growth rate!

[10]:
```
plt.scatter(EP.evalues.real, EP.evalues.imag,color='red')
plt.contour(EP.ps_real,EP.ps_imag, np.log10(EP.pseudospectrum),levels=np.arange(-8,0),
↪colors='k')
plt.axis('square')
plt.axhline(0,color='k',alpha=0.2)

plt.xlim(0,1)
plt.ylim(-1,0.1)
plt.xlabel(r"$c_r$")
plt.ylabel(r"$c_i$")
```

[10]:
```
Text(0, 0.5, '$c_i$')
```



Primitive Form

One of the great advantages of Dedalus is that we can study problems without having to manipulate them into special forms, as in the case of the Orr-Sommerfeld operator. After all, this is nothing more than a special form of the 2-D Navier-Stokes equation linearized around $U(z)$.

We should be able to enter the "primitive" form of the equations and get equivalent results. Here, we define a new Dedalus `EVP`, and then feed it to `eigentools`.

[11]:
```
primitive = de.EVP(d,['u','w','uz','wz', 'p'],'c')
primitive.parameters['alpha'] = alpha
primitive.parameters['Re'] = Re

primitive.substitutions['umean'] = '1 - z**2'
primitive.substitutions['umeanz'] = '-2*z'
```

(continues on next page)

(continued from previous page)

```
primitive.substitutions['dx(A)'] = '1j*alpha*A'
primitive.substitutions['dt(A)'] = '-1j*alpha*c*A'
primitive.substitutions['Lap(A,Az)'] = 'dx(dx(A)) + dz(Az)'

primitive.add_equation('dt(u) + umean*dx(u) + w*umeanz + dx(p) - Lap(u, uz)/Re = 0')
primitive.add_equation('dt(w) + umean*dx(w) + dz(p) - Lap(w, wz)/Re = 0')
primitive.add_equation('dx(u) + wz = 0')
primitive.add_equation('uz - dz(u) = 0')
primitive.add_equation('wz - dz(w) = 0')
primitive.add_bc('left(w) = 0')
primitive.add_bc('right(w) = 0')
primitive.add_bc('left(u) = 0')
primitive.add_bc('right(u) = 0')


prim_EP = Eigenproblem(primitive)
```

```
2021-02-01 17:21:40,598 problems 0/1 INFO :: Solving EVP with homogeneity tolerance of 1.
↪000e-10
2021-02-01 17:21:40,648 problems 0/1 INFO :: Solving EVP with homogeneity tolerance of 1.
↪000e-10
```

Again, we solve the dense eigenproblem just to get the full spectrum.

[12]:
```
prim_EP.solve(sparse=False)
```

And we recalculate the pseudospectrum, covering the same part of the complex plane.

However, the energy norm in primitive variables is slightly different:

$$< \mathbf{X}_1 | \mathbf{X}_2 >_E = \int_{-1}^{1} u_1^\dagger u_2 + w_1^\dagger w_2 dz,$$

so we write another `energy_norm` function and then call `prim_EP.calc_ps` with the same arguments as before.

[13]:
```
def energy_norm(Q1, Q2):
    u1 = Q1['u']
    w1 = Q1['w']
    u2 = Q2['u']
    w2 = Q2['w']

    field = (np.conj(u1)*u2 + np.conj(w1)*w2).evaluate().integrate()
    return field['g'][0]
```

[14]:
```
prim_EP.calc_ps(k, (real_points, imag_points), inner_product=energy_norm)
```

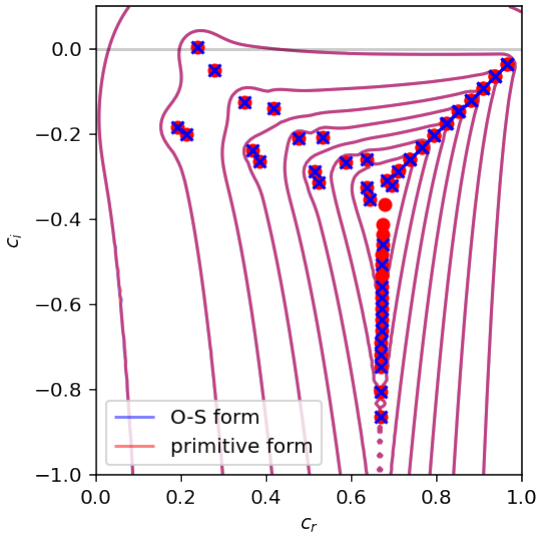
Now, let's plot both the Orr-Sommerfeld form and the primitive form.

[15]:
```
clevels = np.arange(-8,0)
OS_CS = plt.contour(EP.ps_real,EP.ps_imag, np.log10(EP.pseudospectrum),levels=clevels,
↪colors='blue',linestyles='solid', alpha=0.5)
P_CS = plt.contour(prim_EP.ps_real,prim_EP.ps_imag, np.log10(prim_EP.pseudospectrum),
↪levels=clevels,colors='red',linestyles='solid', alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
plt.scatter(prim_EP.evalues.real, prim_EP.evalues.imag,color='red',zorder=3)
plt.scatter(EP.evalues.real, EP.evalues.imag,color='blue',marker='x',zorder=4)

lines = [OS_CS.collections[0],P_CS.collections[0]]
labels = ['O-S form', 'primitive form']

plt.axis('square')
plt.xlim(0,1)
plt.ylim(-1,0.1)
plt.axhline(0,color='k',alpha=0.2)
plt.xlabel(r"$c_r$")
plt.ylabel(r"$c_i$")
plt.legend(lines, labels)
plt.tight_layout()
plt.savefig("OS_pseudospectra.png", dpi=300)
```



We see that the on the whole, there is very good agreement. The primitive form has more good eigenvalues (red dots with no corresponding blue crosses; none of the converse). Furthermore, the pseudospectra contours are indistinguishable except in the lower branch around $(c_r, c_i) = (0.65, -0.8)$.

1.4.2 Mixed Layer Instability

This notebook implements the eigenvalue problem posed in

Boccaletti, Ferrari, & Fox-Kemper, 2007: Journal of Physical Oceanography, 37, 2228-2250.

This demonstrates the use of the `grow_func` option to Eigenproblem to specify a custom calculation method for determining the growth rate of a given eigenvalue.

This script also gives an example of using `project_mode` to compute a 2D visualization of the most unstable eigenmode.

[1]:
```
import numpy as np
import matplotlib.pylab as plt
```

(continues on next page)

(continued from previous page)

```
from eigentools import Eigenproblem, CriticalFinder
import dedalus.public as de
from dedalus.extras import plot_tools
```

[2]:
```
# problem parameters
kx = 0.8  # x-wavelength
ky = 0    # y-wavelength
 = 0.1   # Prandtl ratio f/N
Ri = 2    # Richardson number
L = 1     # Characteristic length scale along front
Hy = 0.05 # Vertical scale
B = 10   # Buoyancy jump at bottom of mixed layer

# discretization parameters
nz = 32
nx = 32 # only necessary for 2D visualization
```

[3]:
```
# Only need the z direction to compute eigenvalues
z = de.Chebyshev('z',nz, interval=[-1,0])
d = de.Domain([z], grid_dtype=np.complex128)

evp = de.EVP(d, ['u','v','w','p','b'],'sigma')

evp.parameters['Ri'] = Ri
evp.parameters[''] =
evp.parameters['L'] = L
evp.parameters['Hy'] = Hy
evp.parameters['B'] = B
evp.parameters['kx'] = kx
evp.parameters['ky'] = ky
evp.substitutions['dt(A)'] = '1j*sigma*A'
evp.substitutions['dx(A)'] = '1j*kx*A'
evp.substitutions['dy(A)'] = '1j*ky*A'
evp.substitutions['U'] = 'z + L'

evp.add_equation('dt(u) + U*dx(u) + w - v + Ri*dx(p) = 0')
evp.add_equation('dt(v) + U*dx(v)     + u + Ri*dy(p) = 0')
evp.add_equation('Ri***2*(dt(w) + U*dx(w)) - Ri*b  + Ri*dz(p) = 0')
evp.add_equation('dt(b) + U*dx(b) - v/Ri + w = 0')
evp.add_equation('dx(u) + dy(v) + dz(w) = 0')

evp.add_bc('right(w) = 0')
evp.add_bc('left(w + dt(p/B) - Hy*v) = 0')
```

```
2021-02-01 17:21:18,102 problems 0/1 INFO :: Solving EVP with homogeneity tolerance of 1.
↪000e-10
```

Specifying a definition of stability

Because the notion of *stability* depends on the assumed time dependence and there are several widely used conventions, `eigentools` allows users to define what "growth" and "frequency" mean in the context of their problem. This is done by passing `grow_func` and `freq_func` to `Eigenproblem`. These functions take a complex number z and return the "growth" and "frequency" corresponding to it.

In this problem the time dependence is given by

$$e^{i\sigma t},$$

so the growth rate is $-Im(\sigma)$ and the frequency is $Re(\sigma)$.

[4]:
```
g_func = lambda z: -z.imag
f_func = lambda z: z.real
EP = Eigenproblem(evp, grow_func=g_func, freq_func=f_func)
```

```
2021-02-01 17:21:18,137 problems 0/1 INFO :: Solving EVP with homogeneity tolerance of 1.
↪000e-10
```

The `EP.growth_rate()` method will provide the eigenvalue corresponding to the fastest growing mode, as defined by `grow_func`. Here, we use `sparse=False` because this is an ideal problem with many exactly zero eigenvalues, the fastest growing mode is never closest to zero. Thus, we must either provide a guess or use the dense solver.

[5]:
```
rate, indx, freq = EP.growth_rate(parameters={'Hy':Hy, 'kx': kx}, sparse=False)
print("fastest growing mode: {} @ freq {}".format(rate, freq))
```

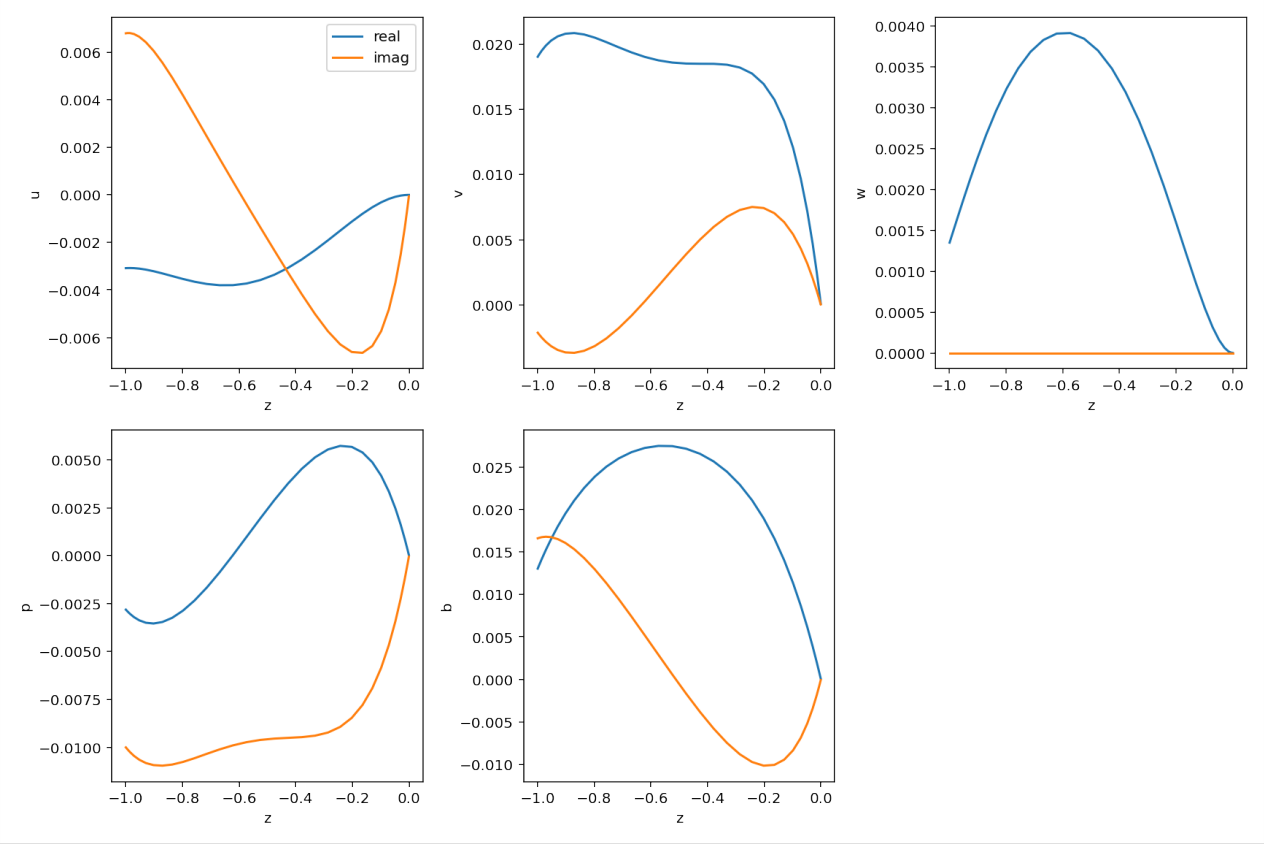
```
fastest growing mode: 0.19931003960491872 @ freq -0.33556838187371807
```

`EP.growth_rate` also provides `indx`, which is the index to the most unstable eigenmode. We can then use that to access that eigenmode for visualization.

Plotting eigenmodes

Because the eigenvalue problems solved by Dedalus are one-dimensional, `eigentools` provides a simple method to visualize all of the modes. In general, these solutions are complex, with the real and imaginary parts of each variable determining their phase relationships. Because eigenmodes are only defined to a relative phase, `EP.plot_mode` allows the user to specify a variable to normalize against. All variables are multiplied by the complex conjugate of the chosen mode. Here, we have chosen the vertical velocity `w`. Note that its plot shows no imaginary part.

[6]:
```
fig = EP.plot_mode(indx, norm_var='w')
fig.savefig('mixed_layer_unstable_1D.png')
```



Projection onto higher dimensional domains

However, because this is a 2-D problem with e^{ikx} dependence in x, we often want to visualize this mode in 2-D also. In order to do so, we define a 2-D domain using Dedalus with the x direction added. Note that *this* domain has a grid dtype of `np.float64` because once we project against the complex exponential in x, the various fields in the eigenmode become real variables.

[7]:
```
# define a 2D domain to plot the eigenmode
x = de.Fourier('x',nx)
d_2d = de.Domain([x,z], grid_dtype=np.float64)
```

Next we will produce a 2-D plot of the most unstable eigenmode shown in figure 7 from Boccaletti, Ferrari, and Fox-Kemper (2007).

First we must *project* the eigenmode against the 2-D domain using `EP.project_mode`. Again we specify the mode index using `indx` and then specify the transverse mode; here we choose 1 so that we do not need to scale L_x.

[8]:
```
fs = EP.project_mode(indx, d_2d, (1,))
```

`EP.project_mode()` returns a Dedalus `FieldSystem` object holding all of the variables in the problem. It can then be accessed using the familiar `fs['u']['g']` interface to get the grid space representation of the x-velocity u.

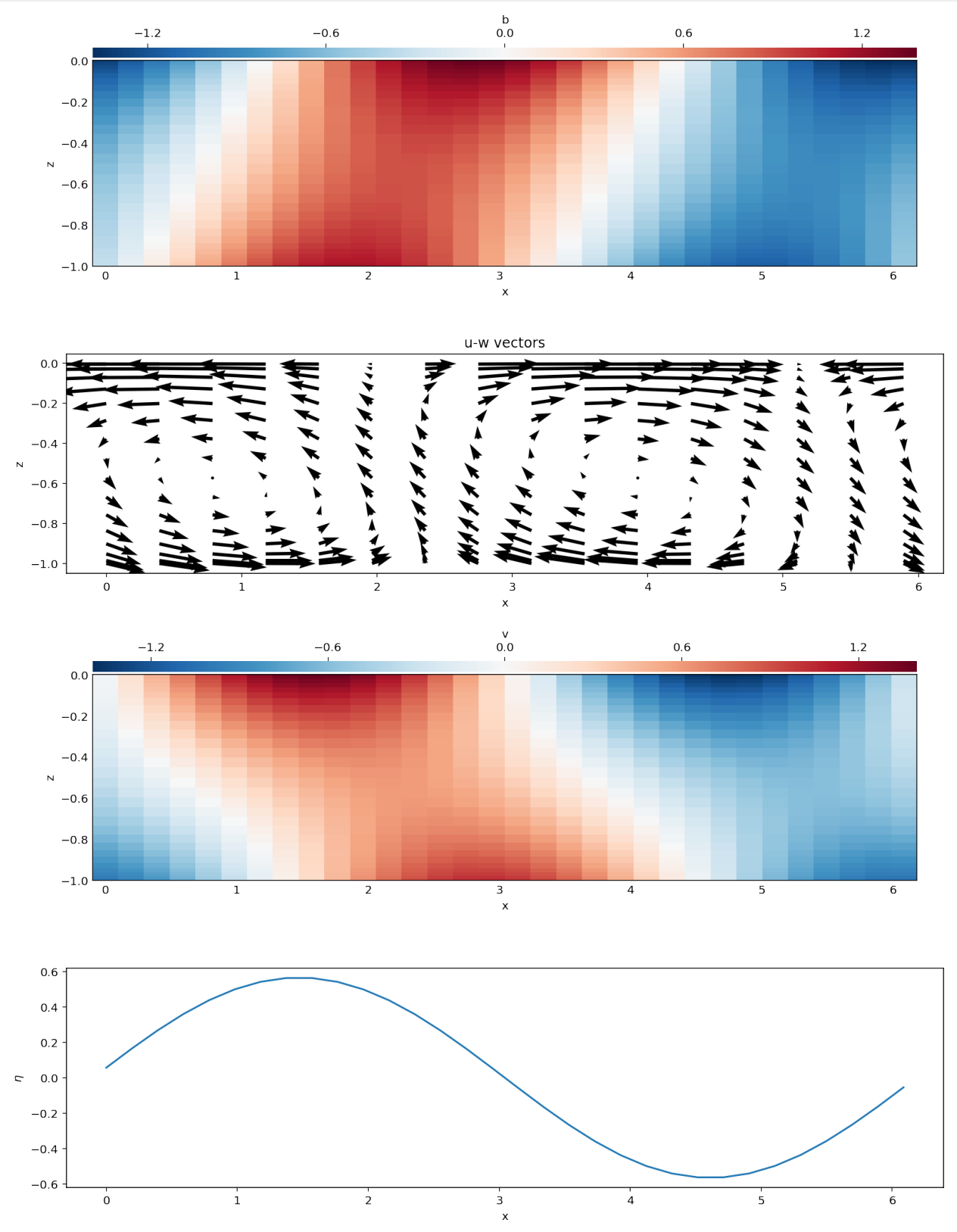
Below, we use Dedalus's `plot_tools` to help generate the plots.

[9]:
```
scale=2.5
nrows = 4
```

(continues on next page)

(continued from previous page)

```
ncols = 1
image = plot_tools.Box(4, 1)
pad = plot_tools.Frame(0.2, 0.2, 0.1, 0.1)
margin = plot_tools.Frame(0.3, 0.2, 0.1, 0.1)
mfig = plot_tools.MultiFigure(nrows, ncols, image, pad, margin, scale)
fig = mfig.figure

for var in ['b','u','v','w','p']:
    fs[var]['g']

# b
axes = mfig.add_axes(0,0, [0,0,1,1])
plot_tools.plot_bot_2d(fs['b'], axes=axes)

# u,w
axes = mfig.add_axes(1,0, [0,0,1,1])
data_slices = (slice(None), slice(None))
xx,zz = d_2d.grids()
xx,zz = np.meshgrid(xx,zz)
axes.quiver(xx[::2,::2],zz[::2,::2], fs['u']['g'][::2,::2].T, fs['w']['g'][::2,::2].T,
↪zorder=10)
axes.set_xlabel('x')
axes.set_ylabel('z')
axes.set_title('u-w vectors')

# v
axes = mfig.add_axes(2,0, [0,0,1,1])
plot_tools.plot_bot_2d(fs['v'], axes=axes)

# eta
axes = mfig.add_axes(3,0, [0,0,1,1])
axes.plot(xx[0,:],-fs['p']['g'][:,0])
axes.set_xlabel('x')
axes.set_ylabel(r'$\eta$')

fig.savefig('mixed_layer_unstable_2D.png',dpi=300)
```


b
−1.2
−0.6
0.0
0.6
1.2
z
x
u-w vectors
v
η

Writing Dedalus HDF5 files

One might want to use a single eigenmode as an initial condition for an initial value problem. `eigentools` makes this very simple:

[10]:
```
EP.write_global_domain(fs, base_name="mixed_layer_output")
```

```
2021-02-01 17:21:22,949 post 0/1 INFO :: Merging files from mixed_layer_output
```

This will create a directory named with the value of `base_name`, in which a Dedalus `.h5` file is found. This file is equivalent to a merged Dedalus initial value problem snapshot and can be used to start an IVP script. An example of this process can be found in the eigenvalue tests directory

Finding critical parameters

Next, we will look at the stability portrait for these parameters as a function of k_x and H_y. Eigentools provides `CriticalFinder` as an interface for doing this. We are only going to use the grid generation function, which runs a number of eigenvalue problems at different parameters.

First, we define a set of points in each dimension, k_x and H_y using `np.linspace`. We then use `cf.grid_generator` to run an eigenvalue problem at each specified k_x, H_y point and return its growth rate. This can be run in parallel via MPI; here we'll do a very coarse grid of just 10 by 10. This takes approximately 1 minute on a single core of a 2020 Core i7.

Because the this is a time consuming process, but plotting the data is fast and may require iteration, `cf.save_grid` saves the data to an HDF5 file. There is a corresponding `cf.load_grid` method that will load in such a file.

[11]:
```
cf = CriticalFinder(EP, ("kx","Hy"), find_freq=False)
nx = 10
ny = 10
xpoints = np.linspace(0.01, 1.6, nx)
ypoints = np.linspace(-0.05,0.05,ny)

cf.grid_generator((xpoints,ypoints))
cf.save_grid('mixed_layer_grid')
```

```
2021-02-01 17:21:23,036 criticalfinder 0/1 INFO :: Solving Local EVP 1/100
2021-02-01 17:21:23,703 criticalfinder 0/1 INFO :: Solving Local EVP 2/100
2021-02-01 17:21:24,562 criticalfinder 0/1 INFO :: Solving Local EVP 3/100
2021-02-01 17:21:25,388 criticalfinder 0/1 INFO :: Solving Local EVP 4/100
2021-02-01 17:21:26,028 criticalfinder 0/1 INFO :: Solving Local EVP 5/100
2021-02-01 17:21:26,736 criticalfinder 0/1 INFO :: Solving Local EVP 6/100
2021-02-01 17:21:27,635 criticalfinder 0/1 INFO :: Solving Local EVP 7/100
2021-02-01 17:21:28,903 criticalfinder 0/1 INFO :: Solving Local EVP 8/100
2021-02-01 17:21:29,904 criticalfinder 0/1 INFO :: Solving Local EVP 9/100
2021-02-01 17:21:30,679 criticalfinder 0/1 INFO :: Solving Local EVP 10/100
2021-02-01 17:21:31,638 criticalfinder 0/1 INFO :: Solving Local EVP 11/100
2021-02-01 17:21:32,656 criticalfinder 0/1 INFO :: Solving Local EVP 12/100
2021-02-01 17:21:33,673 criticalfinder 0/1 INFO :: Solving Local EVP 13/100
2021-02-01 17:21:34,497 criticalfinder 0/1 INFO :: Solving Local EVP 14/100
2021-02-01 17:21:35,299 criticalfinder 0/1 INFO :: Solving Local EVP 15/100
2021-02-01 17:21:35,973 criticalfinder 0/1 INFO :: Solving Local EVP 16/100
2021-02-01 17:21:36,640 criticalfinder 0/1 INFO :: Solving Local EVP 17/100
2021-02-01 17:21:37,451 criticalfinder 0/1 INFO :: Solving Local EVP 18/100
2021-02-01 17:21:38,191 criticalfinder 0/1 INFO :: Solving Local EVP 19/100
```

(continues on next page)

(continued from previous page)

```
2021-02-01 17:21:38,959 criticalfinder 0/1 INFO :: Solving Local EVP 20/100
2021-02-01 17:21:39,724 criticalfinder 0/1 INFO :: Solving Local EVP 21/100
2021-02-01 17:21:40,869 criticalfinder 0/1 INFO :: Solving Local EVP 22/100
2021-02-01 17:21:42,007 criticalfinder 0/1 INFO :: Solving Local EVP 23/100
2021-02-01 17:21:42,740 criticalfinder 0/1 INFO :: Solving Local EVP 24/100
2021-02-01 17:21:43,585 criticalfinder 0/1 INFO :: Solving Local EVP 25/100
2021-02-01 17:21:44,313 criticalfinder 0/1 INFO :: Solving Local EVP 26/100
2021-02-01 17:21:44,956 criticalfinder 0/1 INFO :: Solving Local EVP 27/100
2021-02-01 17:21:45,817 criticalfinder 0/1 INFO :: Solving Local EVP 28/100
2021-02-01 17:21:46,656 criticalfinder 0/1 INFO :: Solving Local EVP 29/100
2021-02-01 17:21:48,253 criticalfinder 0/1 INFO :: Solving Local EVP 30/100
2021-02-01 17:21:49,108 criticalfinder 0/1 INFO :: Solving Local EVP 31/100
2021-02-01 17:21:50,212 criticalfinder 0/1 INFO :: Solving Local EVP 32/100
2021-02-01 17:21:51,367 criticalfinder 0/1 INFO :: Solving Local EVP 33/100
2021-02-01 17:21:52,552 criticalfinder 0/1 INFO :: Solving Local EVP 34/100
2021-02-01 17:21:53,652 criticalfinder 0/1 INFO :: Solving Local EVP 35/100
2021-02-01 17:21:54,715 criticalfinder 0/1 INFO :: Solving Local EVP 36/100
2021-02-01 17:21:55,696 criticalfinder 0/1 INFO :: Solving Local EVP 37/100
2021-02-01 17:21:56,581 criticalfinder 0/1 INFO :: Solving Local EVP 38/100
2021-02-01 17:21:57,319 criticalfinder 0/1 INFO :: Solving Local EVP 39/100
2021-02-01 17:21:58,032 criticalfinder 0/1 INFO :: Solving Local EVP 40/100
2021-02-01 17:21:58,678 criticalfinder 0/1 INFO :: Solving Local EVP 41/100
2021-02-01 17:21:59,432 criticalfinder 0/1 INFO :: Solving Local EVP 42/100
2021-02-01 17:22:00,120 criticalfinder 0/1 INFO :: Solving Local EVP 43/100
2021-02-01 17:22:00,928 criticalfinder 0/1 INFO :: Solving Local EVP 44/100
2021-02-01 17:22:01,584 criticalfinder 0/1 INFO :: Solving Local EVP 45/100
2021-02-01 17:22:02,268 criticalfinder 0/1 INFO :: Solving Local EVP 46/100
2021-02-01 17:22:03,015 criticalfinder 0/1 INFO :: Solving Local EVP 47/100
2021-02-01 17:22:03,816 criticalfinder 0/1 INFO :: Solving Local EVP 48/100
2021-02-01 17:22:04,511 criticalfinder 0/1 INFO :: Solving Local EVP 49/100
2021-02-01 17:22:05,241 criticalfinder 0/1 INFO :: Solving Local EVP 50/100
2021-02-01 17:22:06,061 criticalfinder 0/1 INFO :: Solving Local EVP 51/100
2021-02-01 17:22:06,903 criticalfinder 0/1 INFO :: Solving Local EVP 52/100
2021-02-01 17:22:07,797 criticalfinder 0/1 INFO :: Solving Local EVP 53/100
2021-02-01 17:22:08,579 criticalfinder 0/1 INFO :: Solving Local EVP 54/100
2021-02-01 17:22:09,647 criticalfinder 0/1 INFO :: Solving Local EVP 55/100
2021-02-01 17:22:11,107 criticalfinder 0/1 INFO :: Solving Local EVP 56/100
2021-02-01 17:22:12,370 criticalfinder 0/1 INFO :: Solving Local EVP 57/100
2021-02-01 17:22:14,074 criticalfinder 0/1 INFO :: Solving Local EVP 58/100
2021-02-01 17:22:15,182 criticalfinder 0/1 INFO :: Solving Local EVP 59/100
2021-02-01 17:22:16,132 criticalfinder 0/1 INFO :: Solving Local EVP 60/100
2021-02-01 17:22:16,914 criticalfinder 0/1 INFO :: Solving Local EVP 61/100
2021-02-01 17:22:17,628 criticalfinder 0/1 INFO :: Solving Local EVP 62/100
2021-02-01 17:22:18,385 criticalfinder 0/1 INFO :: Solving Local EVP 63/100
2021-02-01 17:22:19,116 criticalfinder 0/1 INFO :: Solving Local EVP 64/100
2021-02-01 17:22:19,870 criticalfinder 0/1 INFO :: Solving Local EVP 65/100
2021-02-01 17:22:20,858 criticalfinder 0/1 INFO :: Solving Local EVP 66/100
2021-02-01 17:22:21,823 criticalfinder 0/1 INFO :: Solving Local EVP 67/100
2021-02-01 17:22:22,602 criticalfinder 0/1 INFO :: Solving Local EVP 68/100
2021-02-01 17:22:23,660 criticalfinder 0/1 INFO :: Solving Local EVP 69/100
2021-02-01 17:22:24,525 criticalfinder 0/1 INFO :: Solving Local EVP 70/100
2021-02-01 17:22:25,514 criticalfinder 0/1 INFO :: Solving Local EVP 71/100
```

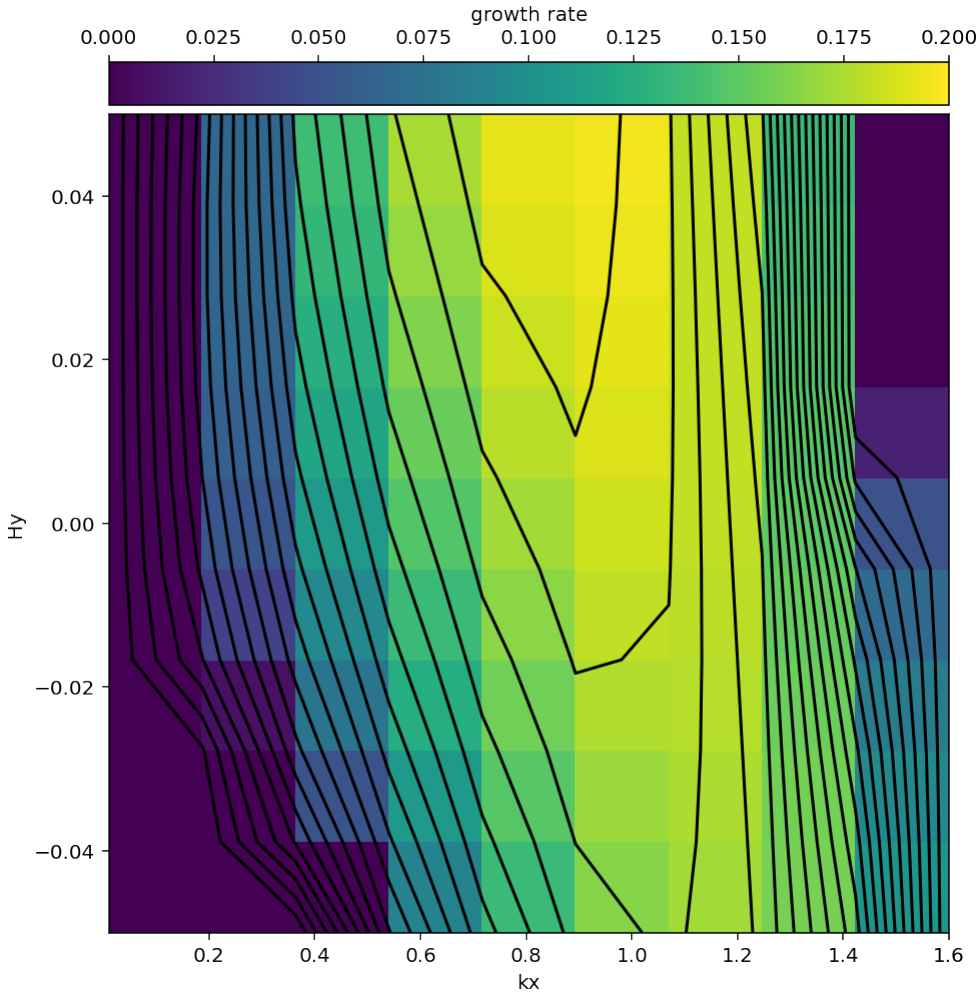
(continues on next page)

(continued from previous page)

```
2021-02-01 17:22:26,515 criticalfinder 0/1 INFO :: Solving Local EVP 72/100
2021-02-01 17:22:27,298 criticalfinder 0/1 INFO :: Solving Local EVP 73/100
2021-02-01 17:22:28,001 criticalfinder 0/1 INFO :: Solving Local EVP 74/100
2021-02-01 17:22:28,686 criticalfinder 0/1 INFO :: Solving Local EVP 75/100
2021-02-01 17:22:29,308 criticalfinder 0/1 INFO :: Solving Local EVP 76/100
2021-02-01 17:22:29,997 criticalfinder 0/1 INFO :: Solving Local EVP 77/100
2021-02-01 17:22:30,666 criticalfinder 0/1 INFO :: Solving Local EVP 78/100
2021-02-01 17:22:31,340 criticalfinder 0/1 INFO :: Solving Local EVP 79/100
2021-02-01 17:22:32,074 criticalfinder 0/1 INFO :: Solving Local EVP 80/100
2021-02-01 17:22:32,827 criticalfinder 0/1 INFO :: Solving Local EVP 81/100
2021-02-01 17:22:33,418 criticalfinder 0/1 INFO :: Solving Local EVP 82/100
2021-02-01 17:22:34,105 criticalfinder 0/1 INFO :: Solving Local EVP 83/100
2021-02-01 17:22:34,854 criticalfinder 0/1 INFO :: Solving Local EVP 84/100
2021-02-01 17:22:35,509 criticalfinder 0/1 INFO :: Solving Local EVP 85/100
2021-02-01 17:22:36,269 criticalfinder 0/1 INFO :: Solving Local EVP 86/100
2021-02-01 17:22:37,080 criticalfinder 0/1 INFO :: Solving Local EVP 87/100
2021-02-01 17:22:37,947 criticalfinder 0/1 INFO :: Solving Local EVP 88/100
2021-02-01 17:22:38,731 criticalfinder 0/1 INFO :: Solving Local EVP 89/100
2021-02-01 17:22:39,591 criticalfinder 0/1 INFO :: Solving Local EVP 90/100
2021-02-01 17:22:40,567 criticalfinder 0/1 INFO :: Solving Local EVP 91/100
2021-02-01 17:22:41,374 criticalfinder 0/1 INFO :: Solving Local EVP 92/100
2021-02-01 17:22:41,926 criticalfinder 0/1 INFO :: Solving Local EVP 93/100
2021-02-01 17:22:42,579 criticalfinder 0/1 INFO :: Solving Local EVP 94/100
2021-02-01 17:22:43,411 criticalfinder 0/1 INFO :: Solving Local EVP 95/100
2021-02-01 17:22:44,064 criticalfinder 0/1 INFO :: Solving Local EVP 96/100
2021-02-01 17:22:44,751 criticalfinder 0/1 INFO :: Solving Local EVP 97/100
2021-02-01 17:22:45,445 criticalfinder 0/1 INFO :: Solving Local EVP 98/100
2021-02-01 17:22:46,123 criticalfinder 0/1 INFO :: Solving Local EVP 99/100
2021-02-01 17:22:47,000 criticalfinder 0/1 INFO :: Solving Local EVP 100/100
```

Finally, we produce a plot comparable to the left panel of figure 6 in Boccaletti, Ferrari, & Fox-Kemper (2007).

[12]:
```
pax,cax = cf.plot_crit()
pax.collections[0].set_clim(0,0.2)
cax.xaxis.set_ticks_position('top')
cax.xaxis.set_label_position('top')
contours = np.linspace(0,0.2,20,endpoint=False)
pax.contour(cf.parameter_grids[0], cf.parameter_grids[1],cf.evalue_grid.real,␣
↪levels=contours, colors='k')
pax.figure.savefig('mixed_layer_growth_rates.png',dpi=300)
```



[ ]:

1.5 API reference

1.5.1 eigentools

Submodules

eigentools.criticalfinder

Module Contents

logger

class CriticalFinder(*eigenproblem*, *param_names*, *comm=MPI.COMM_WORLD*, *find_freq=False*)
finds critical parameters for eigenvalue problems.

This class provides simple tools for finding the critical parameters for the linear (in)stability of a given flow. The parameter space must be 2D; typically this will be (k, Re), where k is a wavenumber and Re is some control parameter (e. g. Reynolds or Rayleigh). The parameters are defined by the underlying Eigenproblem object.

Parameters

- **eigenproblem** (*Eigenproblem*) – An eigentools eigenproblem object over which to find critical parameters
- **param_names** (*tuple of str*) – The names of parameters to search over
- **comm** (*mpi4py.MPI.Intracomm, optional*) – The MPI comm group to share jobs across (default: MPI.COMM_WORLD)
- **find_freq** (*bool, optional*) – If True, also find frequency at critical point

Variables

- **parameter_grids** – NumPy mesh grids containing the parameter values for the EVP
- **evalue_grid** – NumPy array of complex values, containing the maximum growth rates of the EVP for the corresponding input values.
- **roots** (*ndarray*) – Array of roots along axis 1 of parameter_grid

grid_generator(*self*, *points*, *sparse=False*)

Generates a grid of eigenvalues over the specified parameter space of an eigenvalue problem.

Parameters **points** (*tuple of ndarray*) – The parameter values over which to find the critical value

load_grid(*self*, *filename*)

Load a grid file, in the format as created in save_grid.

Parameters **filename** (*str*) – The name of the .h5 file containing the grid data

save_grid(*self*, *filename*)

Saves the grids of all input parameters as well as the growth rate grid that has been solved for.

Parameters **filename** (*str*) – A file stem, which DOES NOT include the file type extension. The grid will be saved to a file called filen.h5

crit_finder(*self*, *polish_roots=False*, *polish_sparse=True*, *tol=0.001*, *method='Powell'*, *maxiter=200*, ***kwargs*)

returns parameters at which critical eigenvalue occurs and optionally frequency at that value.

The critical parameter is defined as the absolute minimum of the growth rate, defined in the Eigenproblem via its grow_func. If frequency is to be found also, returns the frequnecy defined in the Eigenproblem via its freq_func.

If find_freq is True, returns (critical parameter 1, critical parameter 2, frequency); otherwise returns (critical parameter 1, critical parameter 2)

Parameters

- **polish_roots** (*bool, optional*) – If true, use optimization routines to polish critical value (default: False)
- **polish_sparse** (*bool, optional*) – If true, use the sparse solver when polishing roots (default: True)
- **tol** (*float, optional*) – Tolerance for polishing routine (default: 1e-3)
- **method** (*str, optional*) – Method for scipy.optimize used for polishing (default: Powell)
- **maxiter** (*int, optional*) – Maximum number of optimization iterations used for polishing (default: 200)

Returns

Return type tuple

`critical_polisher`(*self*, *guess*, *sparse=True*, *tol=0.001*, *method='Powell'*, *maxiter=200*, ***kwargs*)
Polishes a guess for the critical value using scipy's optimization routines to find a more precise location of the critical value.

Parameters

- **guess** (*complex*) – Initial guess for optimization routines
- **sparse** (*bool, optional*) – If true, use the sparse solver when polishing roots (default: True)
- **tol** (*float, optional*) – Tolerance for polishing routine (default: 1e-3)
- **method** (*str, optional*) – Method for scipy.optimize used for polishing (default: Powell)
- **maxiter** (*int, optional*) – Maximum number of optimization iterations used for polishing (default: 200)

`plot_crit`(*self*, *axes=None*, *transpose=False*, *xlabel=None*, *ylabel=None*, *zlabel='growth rate'*, *cmap='viridis'*)
Create a 2D colormap of the grid of growth rates.

If available, the root values that have been found will be plotted over the colormap.

Parameters

- **transpose** (*bool, optional*) – If True, plot dim 0 on the y axis and dim 1 on the x axis.
- **xlabel** (*str, optional*) – If not None, the x-label of the plot. Otherwise, use parameter name from EVP
- **ylabel** (*str, optional*) – If not None, the y-label of the plot. Otherwise, use parameter name from EVP
- **zlabel** (*str, optional*) – Label for the colorbar. (default: growth rate)
- **cmp** (*str, optional*) – matplotlib colormap name (default: viridis)

eigentools.eigenproblem

Module Contents

`logger`

`class Eigenproblem`(*EVP*, *reject=True*, *factor=1.5*, *scales=1*, *drift_threshold=1000000.0*, *use_ordinal=False*, *grow_func=lambda x: ...*, *freq_func=lambda x: ...*)
An object for feature-rich eigenvalue analysis.

Eigenproblem provides support for common tasks in eigenvalue analysis. Dedalus EVP objects compute raw eigenvalues and eigenvectors for a given problem; Eigenproblem provides support for numerous common tasks required for scientific use of those solutions. This includes rejection of inaccurate eigenvalues and analysis of those rejection criteria, plotting of eigenmodes and spectra, and projection of 1-D eigenvectors onto 2- or 3-D domains for use as initial conditions in subsequent initial value problems.

Additionally, Eigenproblems can compute epsilon-pseudospectra for arbitrary Dedalus differential-algebraic equations.

Parameters

- **EVP** (*dedalus.core.problems.EigenvalueProblem*) – The Dedalus EVP object containing the equations to be solved
- **reject** (*bool, optional*) – whether or not to reject spurious eigenvalues (default: True)
- **factor** (*float, optional*) – The factor by which to multiply the resolution. NB: this must be a rational number such that factor times the resolution of EVP is an integer. (default: 1.5)
- **scales** (*float, optional*) – A multiple for setting the grid resolution. (default: 1)
- **drift_threshold** (*float, optional*) – Inverse drift ratio threshold for keeping eigenvalues during rejection (default: 1e6)
- **use_ordinal** (*bool, optional*) – If true, use ordinal method from Boyd (1989); otherwise use nearest (default: False)
- **grow_func** (*func*) – A function that takes a complex input and returns the growth rate as defined by the EVP (default: uses real part)
- **freq_func** (*func*) – A function that takes a complex input and returns the frequency as defined by the EVP (default: uses imaginary part)

Variables

- **evalues** (*ndarray*) – Lists "good" eigenvalues
- **evalues_low** (*ndarray*) – Lists eigenvalues from low resolution solver (i.e. the resolution of the specified EVP)
- **evalues_high** (*ndarray*) – Lists eigenvalues from high resolution solver (i.e. factor times specified EVP resolution)
- **pseudospectrum** (*ndarray*) – epsilon-pseudospectrum computed at specified points in the complex plane
- **ps_real** (*ndarray*) – real coordinates for epsilon-pseudospectrum
- **ps_imag** (*ndarray*) – imaginary coordinates for epsilon-pseudospectrum

Notes

See references for algorithms in individual method docstrings.

grid(*self*)
get grid points for eigenvectors.

solve(*self*, *sparse=False*, *parameters=None*, *pencil=0*, *N=15*, *target=0*, ***kwargs*)
solve underlying eigenvalue problem.

Parameters

- **sparse** (*bool, optional*) – If true, use sparse solver, otherwise use dense solver (default: False)
- **parameters** (*dict, optional*) – A dict giving parameter names and values to the EVP. If None, use values specified at EVP construction time. (default: None)
- **pencil** (*int, optional*) – The EVP pencil to be solved. (default: 0)
- **N** (*int, optional*) – The number of eigenvalues to find if using a sparse solver (default: 15)
- **target** (*complex, optional*) – The target value to search for when using sparse solver (default: 0+0j)

`eigenmode`(*self*, *index*, *scales=None*, *all_modes=False*)
Returns Dedalus FieldSystem object containing the eigenmode given by index.

Parameters

- **index** (*int*) – index of eigenvalue corresponding to desired eigenvector
- **scales** (*float*) – A multiple for setting the grid resolution. If not None, will overwrite self.scales. (default: None)
- **all_modes** (*bool, optional*) – If True, index specifies the unsorted index of the low-resolution EVP; otherwise it is the index corresponding to the self.evalues order (default: False)

`growth_rate`(*self*, *parameters=None*, ***kwargs*)
returns the maximum growth rate, defined by self.grow_func(), the index of the maximal mode, and the frequency of that mode. If there is no growing mode, returns the slowest decay rate.

also returns the index of the fastest growing mode. If there are no good eigenvalues, returns np.nan for all three quantities.

Returns **growth_rate, index, freqency**

Return type tuple of ints

`plot_mode`(*self*, *index*, *fig_height=8*, *norm_var=None*, *scales=None*, *all_modes=False*)
plots eigenvector corresponding to specified index.

By default, the plot will show the real and complex parts of the unnormalized components of the eigenmode. If a norm_var is specified, all components will be scaled such that variable chosen is purely real and has unit amplitude.

Parameters

- **index** (*int*) – index of eigenvalue corresponding to desired eigenvector
- **fig_height** (*float, optional*) – Height of constructed figure (default: 8)
- **norm_var** (*str*) – If not None, selects the field in the eigenmode with which to normalize. Otherwise, plots the unnormalized eigenmode. (default: None)
- **scales** (*float*) – A multiple for setting the grid resolution. If not None, will overwrite self.scales. (default: None)
- **all_modes** (*bool, optional*) – If True, index specifies the unsorted index of the low-resolution EVP; otherwise it is the index corresponding to the self.evalues order (default: False)

Returns

Return type matplotlib.figure.Figure

`project_mode`(*self*, *index*, *domain*, *transverse_modes*, *all_modes=False*)
projects a mode specified by index onto a domain of higher dimension.

Parameters

- **index** – an integer giving the eigenmode to project
- **domain** – a domain to project onto
- **transverse_modes** – a tuple of mode numbers for the transverse directions

Returns

Return type dedalus.core.system.FieldSystem

`write_global_domain`(*self*, *field_system*, *base_name='IVP_output'*)
Given a field system, writes a Dedalus HDF5 file.

Typically, one would use this to write a field system constructed by project_mode.

Parameters

- **field_system** (*dedalus.core.system.FieldSystem*) – A field system containing the data to be written
- **base_name** (*str, optional*) – The base filename of the resulting HDF5 file. (default: IVP_output)

`calc_ps`(*self*, *k*, *zgrid*, *mu=0.0*, *pencil=0*, *inner_product=None*, *norm=- 2*, *maxiter=10*, *rtol=0.001*)
computes epsilon-pseudospectrum for the eigenproblem.

Uses the algorithm described in section 5 of

Embree & Keeler (2017). SIAM J. Matrix Anal. Appl. 38, 3: 1028-1054.

to enable the approximation of epsilon-pseudospectra for arbitrary differential-algebraic equation systems.

k [int] number of eigenmodes in invariant subspace

zgrid [tuple] (real, imag) points

mu [complex] center point for pseudospectrum.

pencil [int] pencil holding EVP

inner_product [function] a function that takes two field systems and computes their inner product

`compute_mass_matrix`(*self*, *Q*, *inner_product*)
Compute the mass matrix M using a given inner product

M must be hermitian, so we compute only half the inner products.

Parameters

- **Q** (*ndarray*) – Matrix of eigenvectors
- **inner_product** (*function*) – a function that takes two field systems and computes their inner product

Returns

Return type ndarray

`set_state`(*self*, *system*, *evector*)
Set system to given evector

Parameters

- **system** (*FieldSystem*) – system to fill in
- **evector** (*ndarray*) – eigenvector

`plot_spectrum`(*self*, *axes=None*, *spectype='good'*, *xlog=True*, *ylog=True*, *real_label='real'*, *imag_label='imag'*)
Plots the spectrum.

The spectrum plots real parts on the x axis and imaginary parts on the y axis.

Parameters

- **spectype** (*{'good', 'low', 'high'}, optional*) – specifies whether to use good, low, or high eigenvalues

- **xlog** (*bool, optional*) – Use symlog on x axis
- **ylog** (*bool, optional*) – Use symlog on y axis
- **real_label** (*str, optional*) – Label to be applied to the real axis
- **imag_label** (*str, optional*) – Label to be applied to the imaginary axis

plot_drift_ratios(*self*, *axes=None*)

Plot drift ratios (both ordinal and nearest) vs. mode number.

The drift ratios give a measure of how good a given eigenmode is; this can help set thresholds.

Returns

Return type matplotlib.figure.Figure

eigentools.tools

Module Contents

bases_register

update_EVP_params(*EVP*, *key*, *value*)

basis_from_basis(*basis*, *factor*)

duplicates input basis with number of modes multiplied by input factor.

the new number of modes will be cast to an integer

basis : a dedalus basis factor : a float that will multiply the grid size by basis

CHAPTER

TWO

DEVELOPERS

The core development team consists of

- Jeff Oishi (<jsoishi@gmail.com>)
- Keaton Burns (<keaton.burns@gmail.com>)
- Susan Clark (<susanclark19@gmail.com>)
- Evan Anders (<evan.anders@northwestern.edu>)
- Ben Brown (<bpbrown@gmail.com>)
- Geoff Vasil (<geoffrey.m.vasil@gmail.com>)
- Daniel Lecoanet (<daniel.lecoanet@northwestern.edu>)

CHAPTER

THREE

SUPPORT

Eigentools was developed with support from the Research Corporation under award Scialog Collaborative Award (TDA) ID# 24231.

BIBLIOGRAPHY

[Boyd2000] Boyd, J (2000). “Chebyshev and Fourier Spectral Methods.” Dover. http://www-personal.umich.edu/~jpboyd/aaabook_9500toc.pdf

PYTHON MODULE INDEX

e

INDEX