ECSAct : Exact Coherent Structures in Active matter
V1.0 Documentation

Caleb G. Wagner* and Piyush Grover†

Dynamical Systems Lab, University of Nebraska-Lincoln, Nebraska, USA

May 1, 2023

Contents

*c.g.wagner23@gmail.com
†piyush.grover@unl.edu

1 Acknowledgement

This material is partly based upon work supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences under award number DE-SC0022280. This work was completed utilizing the Holland Computing Center of the University of Nebraska, which receives support from the Nebraska Research Initiative.

Please cite as : Wagner, Caleb G., et al. "Exact Coherent Structures and Phase Space Geometry of Preturbulent 2D Active Nematic Channel Flow." Physical Review Letters 128.2 (2022): 028003.

2 How to run ECSAct

In ECSAct V1.0, the general philosophy is to treat each type of ECS as a RPO. This is done for two reasons. First, this avoids code duplication and associated bugs. Second, it eliminates the need for explicit preconditioning of the linear systems that are involved in the search for equilibrium and travelling wave ECS. Nevertheless, the details of theory and implementation in Sections 6-8 is provided for each ECS separately. ECSAct V1.0 contains the following three solvers for computing ECS:

1. RPO Solver (/dedalus/'nematics2_-RPO.py'): This is the preferred solver for computing all types of ECS. The general theory and implementation details are given in Sections 7-9. Note that explicit preconditioning is not required for this solver, since it is handled implicitly by Dedalus.

2. Direct Equilibria solver (/dedalus/'nematics2_-GMRES_EQ.py') : This is a non-RPO code that solves for equilibrium directly, and implements the required preconditioning. The associated theory and implementation details are in Sections 8.1 and 9.4.

3. Inbuilt Dedalus NLBVP solver (/dedalus/'nematics2_-NLBVP.py') : For 1D problems (and problems that can be reduced to 1D), the inbuilt NLBVP code maybe used. See Section 9.4.3 for more details. This solver uses the standard Newton's method without the hookstep or Kylov subspace reduction.

Additionally, ECSACT V1.0 also contains the code for performing time-dependent simulations.

2.1 Finding Relative Periodic Orbit ECS via RPO Solver

Consider a dynamical system $\partial_t \mathbf{X} = \mathbf{F}(\mathbf{X})$, with time-t flowmap $\Phi(\mathbf{X}, t)$ that maps a point $\mathbf{X}$ to its image under the flow of the dynamical system. A point $\mathbf{X}_{RPO}$ on a RPO satisfies the equation $\tau_d \, \Phi(\mathbf{X}_{\mathrm{RPO}}, T) - \mathbf{X}_{\mathrm{RPO}} = 0$, where τ_d is the shift operator that shifts the solution by d units, and T is the time-period of the RPO. Copy the RPO code script (/dedalus/'nematics2_-RPO.py') in a new folder, and run it in the default mode, i.e.,

```
# --------> RPO MODE (DEFAULT) <---------- #
 T_guess = np.array(input_file.get('/T')) # Initial guess for the period 'T'
 d_guess = np.array(input_file.get('/d')) # Initial guess for the shift 'd'
# #---------------------------------------- #
```

The initial guess and parameters should be in a file named 'solution_init.h5' in the same folder as the code. For more information on file format, see Section 6.

Example: In the /examples/RPO_from_RPO_Code folder, running the RPO code 'nematics2_-RPO.py' will compute a RPO at parameter value $Ra = 2.51$, starting from an initial guess of a point on a RPO at parameter value $Ra = 2.5$.

2.2 Finding Periodic Orbit ECS via RPO Solver

A periodic orbit is a RPO with $d = 0$ and fixed T. To find a PO, copy the RPO code (/dedalus/nematics2_RPO.py) into a new folder. Run it with the RPO mode commented out, and the PO mode enabled as follows:

```
# --------> RPO MODE (DEFAULT) <---------- #
#T_guess = np.array(input_file.get('/T')) # Initial guess for the period 'T'
#d_guess = np.array(input_file.get('/d')) # Initial guess for the shift 'd'
# #---------------------------------------- #

# --------> PO (PERIODIC ORBIT) MODE <---------- #
# For POs, set the shift to something close to 0. Strictly speaking, it should be exactly 0, but
# the current code requires a finite value to avoid division by 0.
T_guess = np.array(input_file.get('/T')) # Initial guess for the period 'T'
d_guess = 1e-12
# #---------------------------------------- #
```

The initial guess and parameters should be in a file named 'solution_init.h5' in the same folder as the code.

Example: In the /examples/PO_from_RPO_Code folder, running the RPO code 'nematics2_RPO.py' will compute a PO at parameter value $Ra = 0.915$, starting from an initial guess of a point on a PO at parameter value $Ra = 0.910$.

2.3 Finding Travelling-wave ECS via RPO Solver

A travelling wave is a RPO with $d = cT$ being the shift, c is the wave speed. and T is arbitrary. A guess for c is required. To find a TW, Copy the RPO code script (/dedalus/nematics2_RPO.py) in a new folder and modify it as follows:

```
# --------> TW (TRAVELING WAVE) MODE <---------- #
# For TWs, there is 1 unknown parameter in addition to the field data: namely,
# the wave speed 'c'. To calculate TWs in this script, the initial value for 'T' is
# taken to be 1.0 -- about 100 timesteps -- and 'd' is initialized to 'c*T'.
# The value for 'T' is meant to be "small but not too small". If it is too large, then
# the solver may leave the vicinity of the TW during the time integration (for unstable TWs).
# If it is too small, then the Newton solver will seek the trivial fixed point T = 0.
T_guess = 1.0
c_guess = c_guess # Input initial guess for the wave speed here
d_guess = c_guess*T_guess
#---------------------------------------- #
```

The initial guess and parameters should be in a file named 'solution_init.h5' in the same folder as the code.

Example: In the /examples/TW_from_RPO_Code folder, running the RPO code 'nematics2_RPO.py' will compute a TW at parameter value $Ra = 1.5025$, starting from an initial guess of a point on a TW at parameter value $Ra = 1.5$. The guess of c is taken from the TW at $Ra = 1.5$.

2.4 Finding Equilibria ECS via RPO Solver

An equilbrium point $\mathbf{X}_{EQ}$ is a RPO with $d = 0$ and arbitrary T. To compute an equilibrium, copy the RPO code script (/dedalus/'nematics2_RPO.py') in a new folder, and run it after enabling the EQ mode by uncommenting the corresponding code as follows:

```
# --------> EQ (EQUILIBRIA) MODE <---------- #
# Equilibria are treated similarly to traveling waves, except that 'd' is initialized as for a PO,
# that is, approximately zero within working precision.)
T_guess = 1.0
d_guess = 1e-12
#---------------------------------------- #
```

The initial guess and parameters should be in a file named 'solution_init.h5' in the same folder as the code.

Example: In the /examples/EQ_from_RPO_Code folder, running the RPO code 'nematics2__RPO.py' will compute an equilibrium solution at parameter value $Ra = 1.005$, starting from an initial guess of an exact equilibrium solution at parameter value $Ra = 1$.

2.5 Finding Equilibria ECS via the direct EQ Solver

Simply copy the file '/dedalus/nematics2__GMRES_EQ.py' into a new folder and run it. The initial guess and parameters should be in a file named 'solution_init.h5' in the same folder as the code. For more information on file format, see Section 6.

Example: In the /examples/EQ_Direct folder, running the code 'nematics2__GMRES_EQ.py' will compute an equilibrium solution at parameter value $Ra = 1.005$, starting from an initial guess of an exact equilibrium solution at parameter value $Ra = 1$.

2.6 Time-Dependent Simulations

The time-dependent simulations are performed by running 'nematics2_td.py'. See section 6 for more details.

Example: Folders /examples/lambda=0 and /examples/lambda=1 contain two examples of time-dependent simulations starting from an initial condition. These correspond to the flow-alignment parameter $\lambda = 0$ and $\lambda = 1$ in the active nematic channel flow.

3 Q-tensor in 2D: Definitions and quick reference

For a generic active nematic, we can define a probability density function (PDF) $f(\mathbf{r}, \hat{\mathbf{u}}, t)$ that quantifies the probability of finding a particle at position $\mathbf{r}$ and with orientation vector $\hat{\mathbf{u}} = (\cos\theta, \sin\theta)$. We assume normalization

$$\int f(\mathbf{r}, \theta, t) d\mathbf{r} d\theta = 1, \qquad \text{for all } t \text{ for which } f \text{ is defined} \tag{1}$$

Define the tensor $\mathbf{T}$:

$$\mathbf{T} \equiv \hat{\mathbf{u}} \otimes \hat{\mathbf{u}} - \frac{\mathbf{I}}{2} = \frac{1}{2} \begin{pmatrix} \cos 2\theta & \sin 2\theta \\ \sin 2\theta & -\cos 2\theta \end{pmatrix} \tag{2}$$

The local average of $\mathbf{T}$ is also a tensor, which we denote $\mathbf{Q}$:

$$\mathbf{Q}(\mathbf{r}, t) \equiv \langle \mathbf{T} \rangle \equiv \frac{1}{\rho(\mathbf{r}, t)} \int \mathbf{T} f(\mathbf{r}, \theta, t) d\theta \tag{3}$$

where $\rho(\mathbf{r}, t = \int f(\mathbf{r}, \theta, t) d\theta$ is the density.

The order parameter q is the defined as twice the positive eigenvalue of $\mathbf{Q}$, and the director $\hat{\mathbf{n}}$ is defined as the corresponding normalized eigenvector. As long as $q > 0$, $\hat{\mathbf{n}}$ is unique up to a sign, and $\mathbf{Q}$ can be decomposed as follows:

$$\mathbf{Q} = q \left(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}} - \frac{\mathbf{I}}{2} \right) \tag{4}$$

Note that the matrix components of $\mathbf{T}$ (in the standard basis) are bounded between $-1/2$ and $1/2$, which means the same is true for the matrix components of $\mathbf{Q}$. Moreover, it can be shown that the eigenvalues of $\mathbf{Q}$ are similarly bounded, so that q is always between 0 and 1. (I am not aware of a rigorous proof that these bounds are preserved during time evolution under the nematohydrodynamic equations. However, I have not seen evidence to the contrary in my own simulations.)

4 Hydrodynamic equations

4.1 General equations

The equations of motion are

$$\begin{aligned} \rho\left(\partial_t + \mathbf{u}\cdot\boldsymbol{\nabla}\right)\mathbf{u} &= -\boldsymbol{\nabla}p + 2\eta(\boldsymbol{\nabla}\cdot\mathbf{E}) - \alpha(\boldsymbol{\nabla}\cdot\mathbf{Q}), \\ \left(\partial_t + \mathbf{u}\cdot\boldsymbol{\nabla}\right)\mathbf{Q} + \mathbf{W}\cdot\mathbf{Q} - \mathbf{Q}\cdot\mathbf{W} &= \lambda\mathbf{E} + \gamma^{-1}\mathbf{H}, \\ \boldsymbol{\nabla}\cdot\mathbf{u} &= 0. \end{aligned} \tag{5}$$

where

$$E_{ij} = \frac{1}{2}\left(\partial_i u_j + \partial_j u_i\right); \qquad W_{ij} = \frac{1}{2}\left(\partial_i u_j - \partial_j u_i\right) \tag{6}$$

$$\mathbf{H} = A\left[\mathbf{Q} - b\mathrm{Tr}\left(\mathbf{Q}^2\right)\mathbf{Q}\right] + K\nabla^2\mathbf{Q} \tag{7}$$

$$\hat{\mathbf{Q}} = \mathbf{Q} + \frac{\mathbf{I}}{2} \tag{8}$$

- Note 1: In some papers, such as Ref. [5], a passive elastic stress $\mathbf{\Pi}^{\mathrm{elastic}}$ is included in the momentum equation. Here, we drop this term entirely.
- Note 2: Ref. [8] defined the vorticity tensor as the transpose of $\mathbf{W}$. This convention amounts to changing the sign on $\mathbf{W}$ when the equations are written in tensorial form. Component-wise, of course, the equations are the same.

The response of the material to activity is contained in the active stress $-\alpha\mathbf{Q}$, which models an active particle as a force dipole [4]. In the $\mathbf{Q}$-equation, the l.h.s. is the convective derivative of $\mathbf{Q}$ with respect to $\mathbf{v}$. In addition the usual $(\mathbf{u}\cdot\boldsymbol{\nabla})$ contribution, there is also the commutator product $\mathbf{W}\cdot\mathbf{Q} - \mathbf{Q}\cdot\mathbf{W}$, which accounts for rotations of $\mathbf{Q}$ in the convected coordinates. $\mathbf{H}$ derives from an effective free energy functional that penalizes unfavorable configurations, such as those with large elastic stresses. It is usually of the Landau-de Gennes type–that is, consisting of a low-order polynomial expansion in invariants of $\mathbf{Q}$ and its gradients ([1, 6]). Our choice for $\mathbf{H}$ in equation (7) derives from a free energy density f:

$$\mathbf{H} = -\left(\frac{\delta f}{\delta \mathbf{Q}} - \frac{\mathbf{I}}{2}\mathrm{Tr}\frac{\delta f}{\delta \mathbf{Q}}\right) \tag{9}$$

where

$$f = \frac{A}{2}Q^i{}_k Q^k{}_i + \frac{C}{3}Q^i{}_k Q^k{}_j Q^j{}_i + \frac{Ab}{4}(Q^i{}_k Q^k{}_i)^2 + \frac{K}{2}(\nabla^k Q^{ij})(\nabla_k Q_{ji}) \tag{10}$$

In 2D, the cubic term is exactly 0, so it does not appear in (7).

4.2 Non-dimensionalization

The length and time scales ℓ, τ for non-dimensionalization are

$$\ell = \frac{K^{1/2}}{A^{1/2}}, \qquad \tau = \frac{\gamma}{A} \tag{11}$$

The corresponding velocity scale v_0 is

$$v_0 = \frac{\ell}{\tau} = \frac{K^{1/2}}{A^{1/2}}\cdot\frac{A}{\gamma} = \frac{K^{1/2}A^{1/2}}{\gamma} \tag{12}$$

Define:

$$\mathbf{r}' = \mathbf{r}/\ell; \qquad t' = t/\tau; \qquad \mathbf{u}' = \mathbf{u}/v_0 \tag{13}$$

Transforming the **Q**-equation into these new variables gives

$$\left(\partial_{t'} + \mathbf{u'} \cdot \boldsymbol{\nabla'}\right) \mathbf{Q} + \mathbf{W'} \cdot \mathbf{Q} - \mathbf{Q} \cdot \mathbf{W'} = \lambda \mathbf{E'} + \mathbf{H'} \tag{14}$$

where

$$\mathbf{H'} = \mathbf{Q} - b\mathrm{Tr}\left(\mathbf{Q}^2\right)\mathbf{Q} + \nabla'^2\mathbf{Q} \tag{15}$$

Similarly, transforming the **u**-equation gives

$$\frac{\rho K^{1/2} A^{3/2}}{\gamma^2}\left(\partial_{t'} + \mathbf{u'} \cdot \boldsymbol{\nabla'}\right)\mathbf{u'} = -\frac{A^{1/2}}{K^{1/2}}\boldsymbol{\nabla'} p + \frac{2\eta A^{3/2}}{K^{1/2}\gamma}(\boldsymbol{\nabla'} \cdot \mathbf{E}) - \frac{\alpha A^{1/2}}{K^{1/2}}(\boldsymbol{\nabla'} \cdot \mathbf{Q}) \tag{16}$$

Multiply both sides by $\frac{\gamma K^{1/2}}{A^{3/2}\eta}$:

$$\frac{\rho v_0 \ell}{\eta}\left(\partial_{t'} + \mathbf{u'} \cdot \boldsymbol{\nabla'}\right)\mathbf{u'} = -\boldsymbol{\nabla'}\left(\frac{\tau}{\eta} p\right) + 2(\boldsymbol{\nabla'} \cdot \mathbf{E}) - \frac{\gamma\alpha}{A\eta}(\boldsymbol{\nabla'} \cdot \mathbf{Q}) \tag{17}$$

Now, define $p' = (\tau/\eta)p$ and drop primes on everything (including p'):

$$\frac{\rho v_0 \ell}{\eta}\left(\partial_t + \mathbf{u} \cdot \boldsymbol{\nabla}\right)\mathbf{u} = -\boldsymbol{\nabla} p + 2(\boldsymbol{\nabla} \cdot \mathbf{E}) - \frac{\gamma\alpha}{\eta A}\left(\boldsymbol{\nabla} \cdot \mathbf{Q}\right) \tag{18}$$

Borrowing the notation from Ref. [2], we define

$$\mathrm{Re_n} = \frac{\rho v_0 \ell}{\eta} \tag{19}$$

$$\mathrm{Er} = \frac{\eta v_0 \ell}{K} = \frac{\eta}{K} \cdot \frac{K}{\gamma} = \frac{\eta}{\gamma} \tag{20}$$

$$\mathrm{R_a} = \frac{K}{A} \cdot \frac{\alpha}{K} = \frac{\alpha}{A} \tag{21}$$

Making these substitutions gives the final result for the non-dimensionalized equations of motion:

$$\begin{aligned} \mathrm{Re_n}\left(\partial_t + \mathbf{u} \cdot \boldsymbol{\nabla}\right)\mathbf{u} &= -\boldsymbol{\nabla} p + 2(\boldsymbol{\nabla} \cdot \mathbf{E}) - \frac{\mathrm{R_a}}{\mathrm{Er}}\left(\boldsymbol{\nabla} \cdot \mathbf{Q}\right), \\ \left(\partial_t + \mathbf{u} \cdot \boldsymbol{\nabla}\right)\mathbf{Q} + \mathbf{W} \cdot \mathbf{Q} - \mathbf{Q} \cdot \mathbf{W} &= \lambda \mathbf{E} + \mathbf{H}, \\ \boldsymbol{\nabla} \cdot \mathbf{u} &= 0. \end{aligned} \tag{22}$$

4.3 Component-wise expansions

Dedalus requires that equations (22) be expressed in component form. To do so, it is convenient to switch to index notation. Here, we will write the upper and lower indices explicitly. Even though it does not affect the results in Cartesian coordinates, we may later on want to work with curvilinear coordinates, where the distinction will matter. The contravariant form of the equations are the following:

$$\mathrm{Re_n}\left(\frac{\partial u^i}{\partial t} + u^j \nabla_j u^i\right) = -\nabla^i p + 2\nabla_j E^{ji} - \frac{\mathrm{R_a}}{\mathrm{Er}}\nabla_j Q^{ji}, \tag{23}$$

$$\frac{\partial Q^{ij}}{\partial t} + u^k \nabla_k Q^{ij} + W^{ik} Q_k{}^j - Q^i{}_k W^{kj} = \lambda E^{ij} + Q^{ij} - b\, Q_{k\ell} Q^{\ell k} Q^{ij} + \nabla^k \nabla_k Q^{ij}, \tag{24}$$

$$\nabla_i u^i = 0. \tag{25}$$

4.3.1 Cartesian coordinate components

We will denote the Cartesian coordinate components of $\mathbf{u}$ and $\mathbf{Q}$ as follows:

$$\mathbf{u} = U\,\hat{\mathbf{x}} + V\,\hat{\mathbf{y}}$$

$$\mathbf{Q} = QAA\,(\hat{\mathbf{x}}\otimes\hat{\mathbf{x}}) + QAB\,(\hat{\mathbf{x}}\otimes\hat{\mathbf{y}}) + QAB\,(\hat{\mathbf{y}}\otimes\hat{\mathbf{x}}) - QAA\,(\hat{\mathbf{y}}\otimes\hat{\mathbf{y}}) = \begin{pmatrix} QAA & QAB \\ QAB & -QAA \end{pmatrix}$$

Then, the terms in the x component of equation (23) can be expanded as

$$\begin{aligned}
\frac{\partial u^i}{\partial t} &\rightarrow \frac{\partial U}{\partial t} && (26)\\
u^j\nabla_j u^i &\rightarrow U\partial_x U + V\partial_y U && (27)\\
-\nabla^i p &\rightarrow -\partial_x p && (28)\\
2\nabla_j E^{ji} = \partial_j(\partial^i u^j + \partial^j u^i) &\rightarrow 2\partial_x^2 U + \partial_x\partial_y V + \partial_y^2 U && (29)\\
\nabla_j Q^{ji} &\rightarrow \partial_x QAA + \partial_y QAB && (30)\\
& && (31)
\end{aligned}$$

Similarly, for the y component of equation (23), we can write:

$$\begin{aligned}
\frac{\partial u^i}{\partial t} &\rightarrow \frac{\partial V}{\partial t} && (32)\\
u^j\nabla_j u^i &\rightarrow U\partial_x V + V\partial_y V && (33)\\
-\nabla^i p &\rightarrow -\partial_y p && (34)\\
2\nabla_j E^{ji} = \partial_j(\partial^i u^j + \partial^j u^i) &\rightarrow \partial_x\partial_y V + \partial_x^2 V + 2\partial_y^2 V && (35)\\
\nabla_j Q^{ji} &\rightarrow \partial_x QAB - \partial_y QAA && (36)\\
& && (37)
\end{aligned}$$

and for the xx component of equation (24):

$$\begin{aligned}
\frac{\partial Q^{ij}}{\partial t} &\rightarrow \frac{\partial QAA}{\partial t} && (38)\\
u^k\nabla_k Q^{ij} &\rightarrow U(\partial_x QAA) + V(\partial_y QAA) && (39)\\
W^{ik}Q_k{}^j = \frac{1}{2}(\nabla^i v^k - \nabla^k v^i)Q_k{}^j &\rightarrow \frac{QAB}{2}\cdot(\partial_x V - \partial_y U) && (40)\\
Q^i{}_k W^{kj} = \frac{1}{2}Q^i{}_k(\nabla^k v^j - \nabla^j v^k) &\rightarrow \frac{QAB}{2}\cdot(\partial_y U - \partial_x V) && (41)\\
W^{ik}Q_k{}^j - Q^i{}_k W^{kj} &\rightarrow QAB\cdot(\partial_x V - \partial_y U) && (42)\\
\lambda E^{ij} &\rightarrow \lambda\partial_x U && (43)\\
b\,Q_{k\ell}Q^{\ell k}Q^{ij} &\rightarrow 2b\left[(QAA)^2 + (QAB)^2\right]QAA && (44)\\
\nabla^k\nabla_k Q^{ij} &\rightarrow (\partial_x^2 + \partial_y^2)QAA && (45)\\
& && (46)
\end{aligned}$$

and finally, the xy component of equation (24):

$$\frac{\partial Q^{ij}}{\partial t} \quad \to \quad \frac{\partial QAB}{\partial t} \tag{47}$$

$$u^k \nabla_k Q^{ij} \quad \to \quad U(\partial_x QAB) + V(\partial_y QAB) \tag{48}$$

$$W^{ik} Q_k{}^j = \frac{1}{2}(\nabla^i v^k - \nabla^k v^i) Q_k{}^j \quad \to \quad -\frac{QAA}{2} \cdot (\partial_x V - \partial_y U) \tag{49}$$

$$Q^i{}_k W^{kj} = \frac{1}{2} Q^i{}_k (\nabla^k v^j - \nabla^j v^k) \quad \to \quad \frac{QAA}{2} \cdot (\partial_x V - \partial_y U) \tag{50}$$

$$W^{ik} Q_k{}^j - Q^i{}_k W^{kj} \quad \to \quad QAA \cdot (\partial_y U - \partial_x V) \tag{51}$$

$$\lambda E^{ij} \quad \to \quad \frac{\lambda}{2} (\partial_x V + \partial_y U) \tag{52}$$

$$b\, Q_{k\ell} Q^{\ell k} Q^{ij} \quad \to \quad 2b \left[(QAA)^2 + (QAB)^2 \right] QAB \tag{53}$$

$$\nabla^k \nabla_k Q^{ij} \quad \to \quad (\partial_x^2 + \partial_y^2) QAB \tag{54}$$

The yx and yy components of $\mathbf{Q}$ do not need to be considered because they are fixed by the fact that $\mathbf{Q}$ is traceless and symmetric. In fact, they give the same equations as for the xy and xx components, as they must for consistency.

Dedalus formatting

As a final check, we will write out the full componentwise equations in Dedalus formatting. Because Dedalus requires higher-order derivative equations be converted to an equivalent first-order system, the first-order derivatives in the active nematic equations are considered to be independent field components. So these get their own labels: QAAx, QAAy, etc. On the other hand, the second-order derivatives are written in terms of the actual derivative operator, as dx(QAAx), dy(QAAx), and so on. The result is

```
dt(QAA)-dx(QAAx)-dy(QAAy)-QAA-lamb*Ux=-2*b*QAA*QAA*QAA-2*b*QAA*QAB*QAB-QAAx*U-QAAy*V+QAB*Uy-QAB*Vx
dt(QAB)-dx(QABx)-dy(QABy)-QAB-0.5*lamb*Uy-0.5*lamb*Vx=-2*b*QAA*QAA*QAB-2*b*QAB*QAB*QAB-QAA*Uy+QAA*Vx-QABx*U-QABy*V
Re*dt(U)+dx(p)-2*dx(Ux)-dy(Uy)-dx(Vy)+RaEr*QAAx+RaEr*QABy=-Re*U*Ux-Re*Uy*V
Re*dt(V)+dy(p)-2*dy(Vy)-dx(Uy)-dx(Vx)-RaEr*QAAy+RaEr*QABx=-Re*U*Vx-Re*V*Vy
```

where `RaEr` is equal to Ra/Er.

5 Physical parameters and phenomenology

Figure 1 summarizes the results of a parameter scan of the pre-turbulent phase diagram, for $\lambda = 0.5$. (The phase diagrams for $\lambda = 0$ and $\lambda = 1$ can be found in the `figures` folder.) Each colored circle represents a single, time-dependent simulation, which is initialized (quenched) from a high-activity state. In most cases, the activity was chosen such that this initial state was turbulent. The exceptions are some of the channels with small heights, where (even at very high activity) we found a vortex lattice rather than turbulence.

The channel width `w` was chosen to be 20x the height `h` in most cases, the intent being to more closely approximate the behavior of an infinite channel. Further details of these simulations will be provided elsewhere. Here, we will focus on the phenomenology. To distinguish the transitions (in parameter space) between qualitatively distinct flow regimes, we computed certain channel-averaged quantities over the asymptotic portion of each trajectory. Two of the most useful are the x-velocity average $\langle U \rangle$ and the squared y-velocity average $\langle V^2 \rangle$.

Neglecting hysteresis, plotting these in the (Ra, h) parameter space will indicate the following phase boundaries:

1. Zero-flow $\longleftrightarrow$ Unidirectional, indicated by $\langle U \rangle = 0 \longleftrightarrow \langle U \rangle \neq 0$
2. Unidirectional $\longleftrightarrow$ Traveling wave / oscillating RPOs, indicated by $\langle V^2 \rangle = 0 \longleftrightarrow \langle V^2 \rangle > 0$

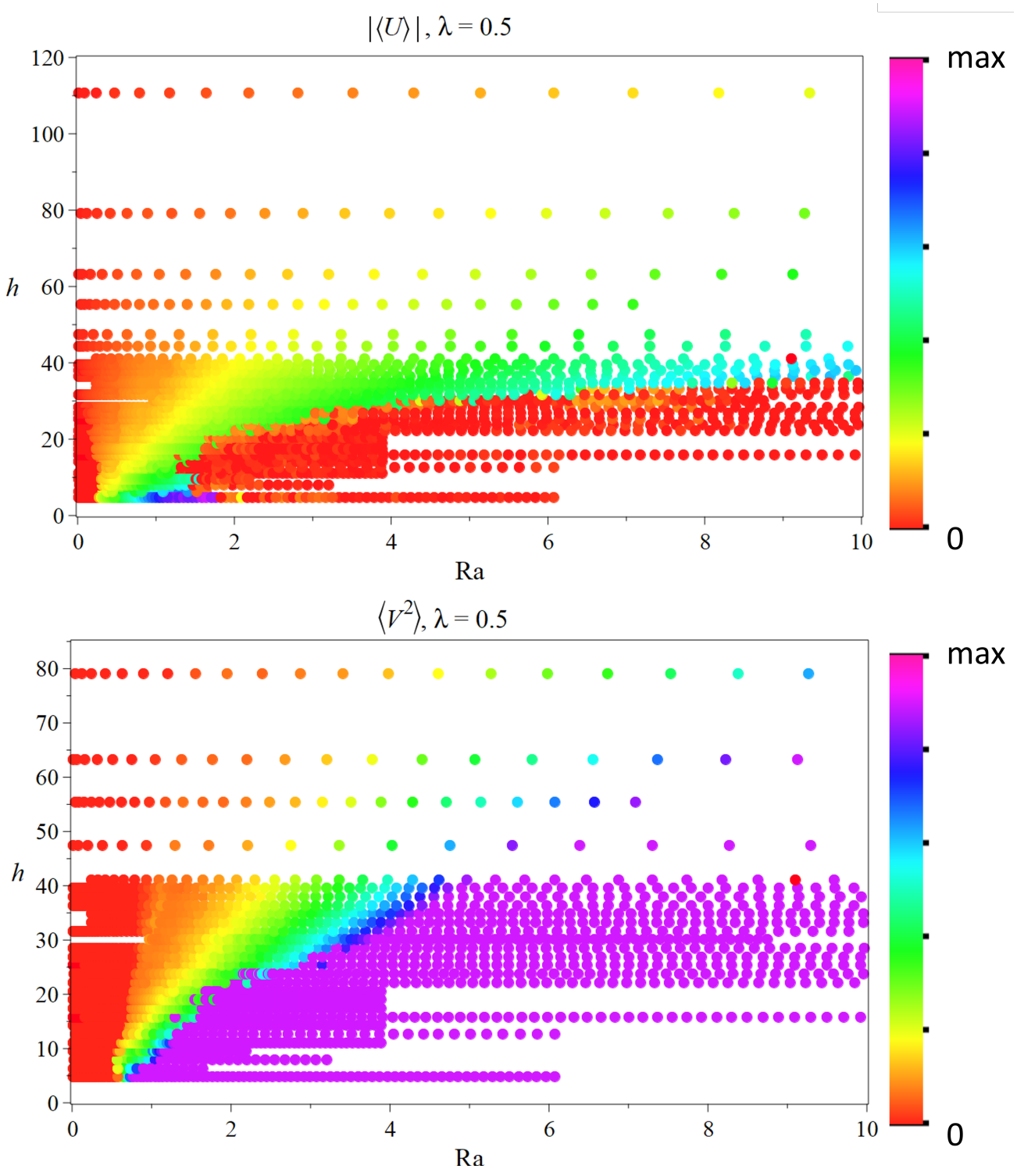

Figure 1: Parameter scan of the pre-turbulent phase diagram for $\lambda = 0.5$. Each colored circle represents a single, time-dependent simulation. The colors represent the values of spatiotemporal averages over the asymptotic portion of each trajectory ($|\langle U \rangle|$ on top and $\langle V^2 \rangle$ >) on bottom, and can be used to infer approximate phase boundaries separating qualitatively distinct flow regimes. See section 5 for more information.

3. Traveling wave / oscillating RPOs $\longleftrightarrow$ vortex lattice, indicated by $\langle U \rangle \neq 0 \longleftrightarrow \langle U \rangle \approx 0$, together with $\langle V^2 \rangle > 0$.

Note: the blue to magenta boundary in the $\langle V^2 \rangle$ diagram is not an actual phase boundary. It appears that way only because the color gradient for $\langle V^2 \rangle$ was capped at a threshold, so that the variations in the lower activity regime would be more visible.

6 Time-dependent simulations in Dedalus

6.1 Basis, domain, fields

The `basis` and `domain` classes provide a setting not only for the Dedalus solvers, but also for defining and manipulating functions in the pseudospectral representation. A typical initialization sequence looks like the following:

```
from dedalus import public as de
x_basis = de.Fourier('x', NX, interval=(-0.5*width, 0.5*width), dealias = 2)
y_basis = de.Chebyshev('y', NY, interval=(0, height), dealias = 2)
domain = de.Domain([x_basis, y_basis], grid_dtype = np.float64)
xg = domain.grid(0)
yg = domain.grid(1)
```

A function with respect to this basis and domain is described by an object from the `field` class. Using this, one could obtain the Fourier/Chebyshev coefficients of a function such as $f(x, y) = y(y - h)\cos^2(4x)$:

```
f_field = domain.new_field()
f_field['g'] = yg*(yg-h)*np.cos(4*xg)*np.cos(4*xg)
f_field.require_coeff_space()
f_coefficients = np.copy(f_field.data)
```

See the Dedalus tutorials for more information on the relevant commands; the above examples are given mainly to help cross-reference with the active nematics code.

6.2 Data input

It is convenient to read in $\mathbf{u}$ and $\mathbf{Q}$ directly in terms of their grid space or coefficient space representation. In these representations, the data is stored as numpy arrays, which can be read straightforwardly from an HDF5 file. Sometimes, it is preferable to switch from one representation to the other after loading the data. For example, changing grid resolution is easiest in coefficient space, but to save disk space I sometimes only output the solution fields in grid space.

```
dataQAA_init = np.array(input_file.get('/QAA_coeff'))
QAA_data_init_coeff = np.zeros((NXH, NY), dtype = np.complex128)
NXH_m = min(NXH,NXH_input)
NY_m = min(NY,NY_input)
QAA_data_init_coeff[0:NXH_m,0:NY_m] = dataQAA_init[0:NXH_m,0:NY_m]
QAA_state_init = solver.state['QAA']
QAA_state_init['c'] = np.copy(dataQAA_init)
```

It is also easy to restart simulations from a previous output file. See section 6.6 for instructions on how to tell Dedalus to write the solver state to a file. Assuming this has been done, then the following line will load the data and initialize the new solver:

```
write, last_dt = solver.load_state('restart.h5', -1)
```

where `restart.h5` is the previous data file. It is important to remain aware of the `sim_time` parameter, as the simulation time does not restart from 0. Also, I do not recommend changing the timestep abruptly from its most recent value during the previous run (in the above line of code, this is stored in `last_dt`). If you want a different timestep, then it would be best to change it gradually over several iterations.

Note also that multistep time integrators will run reduced-order for the first few iterations of a restart, so in that case it is probably especially important not to increase the timestep abruptly from the previous simulation state.

6.3 Problem and solver classes

The pseudospectral basis and domain can be constructed and utilized without defining a PDE problem; see section 6.1 above. To solve a PDE problem on a domain, `problem` and `solver` objects must be instantiated. The `problem` object comes first, as the `solver` object is defined with respect to it. For time-dependent problems, we create an `InitialValueProblem` object, named `problem_example` as follows:

```
from dedalus import public as de
problem_example = de.IVP(domain,
    variables = ['p','U','V','Ux','Uy','Vx','Vy','QAA','QAAx','QAAy','QAB','QABx','QABy'])
```

The above code constructs a `problem` object in terms of a `domain` object and the names of the problem variables. Additional properties of the problem object can be specified using the `meta` property. In the active nematics code, we typically just add the following line:

```
problem_example.meta[:]['y']['dirichlet'] = True
```

which tells the internal Dedalus routines that the boundary conditions in y are Dirichlet. This information speeds up the solver but is not necessary.

The main aspects of instantiating the `problem` object are to give the equations and boundary conditions. These are entered as Python strings. The string labels for the problem variables were already defined when the object was constructed; it remains to specify the labels of any parameters. For active nematics, the basic format is:

```
problem_example.parameters['width'] = width
problem_example.parameters['height'] = height
problem_example.parameters['RaEr'] = Ra/Er
problem_example.parameters['Re'] = Re
```

Once all the variables have been defined, the equations can be entered. First, we have to define the first-derivative functions of the problem variables, as Dedalus requires the equations to be entered as an equivalent first-order system:

```
problem_example.add_equation("Ux - dx(U) = 0")
problem_example.add_equation("Uy - dy(U) = 0")
problem_example.add_equation("Vx - dx(V) = 0")
problem_example.add_equation("Vy - dy(V) = 0")
problem_example.add_equation("QAAx - dx(QAA) = 0")
problem_example.add_equation("QAAy - dy(QAA) = 0")
problem_example.add_equation("QABx - dx(QAB) = 0")
problem_example.add_equation("QABy - dy(QAB) = 0")
```

Now, the actual hydrodynamics can be entered. For example, the x-velocity equation is specified as:

```
problem_example.add_equation("Re*dt(U)+dx(p)-2*dx(Ux)-dy(Uy)-dx(Vy)+RaEr*QAAx+RaEr*QABy=-Re*U*Ux-Re*Uy*V")
```

and similarly for the other equations. Last, the boundary conditions in y must be specified. (The periodic boundary conditions in the x direction are implicitly accounted for by the Fourier basis.) Typical usage is:

```
problem_example.add_bc("left(U) = 0")
problem_example.add_bc("left(V) = 0")
problem_example.add_bc("left(QAA) = -0.5")
problem_example.add_bc("left(QAB) = 0")
problem_example.add_bc("right(U) = 0")
problem_example.add_bc("right(V) = 0", condition="(nx != 0)")
problem_example.add_bc("left(p) = 0", condition="(nx == 0)")
problem_example.add_bc("right(QAA) = -0.5")
problem_example.add_bc("right(QAB) = 0")
```

Due to the $\nabla \cdot \mathbf{v} = 0$ constraint, the boundary condition on the zeroth Fourier mode of `V` must be replaced with a boundary condition on the pressure `p`. See the Dedalus documentation and examples for further information.

Once the `problem` object is constructed, the corresponding `solver` object can be constructed and initialized in just a few lines of code. For time-dependent simulations, it requires specification of the timestepper (see below for more details on available timestepping routines). For example:

```
solver_example = problem.build_solver(de.timesteppers.RK222)
logger.info('Solver built')
QAA_state_init = solver_example.state['QAA']
QAA_state_init['c'] = dataQAA_init
```

The third line defines the field `QAA_state_init`, which can be used to write to or read from the solver state. The fourth line writes to the solver state in coefficient space using the 2D numpy array `dataQAA_init`.

6.4 Timestepping

6.4.1 Adaptive timestep via the `CFL` class

For computing ECS, we use a fixed timestep. In time-dependent simulations, however, an adaptive timestep is often preferable. This is especially true if one is primarily concerned with the average, long-time behavior, which should be less sensitive to the timestep. Typical usage is the following:

```
from dedalus.extras import flow_tools
CFL = flow_tools.CFL(solver_example, initial_dt=0.001, cadence=10,
                     max_change=1.5, safety=0.5, threshold=0.1, max_dt=0.5)
CFL.add_velocities(('U', 'V'))
dt = CFL.compute_dt()
```

Here, a `CFL` object is created for `solver_example`, with initial time step `initial_dt` and maximum timestep `max_dt`. The other parameters are related to the technical details of the CFL implementation; see section 6.8 below and the Dedalus documentation for more information.

6.4.2 Timesteppers

The four timesteppers I use most frequently are described below.

- `de.timesteppers.SBDF1` – Fastest but least accurate. I sometimes use this for exploratory time-dependent simulations in which I am more interested in the average, asymptotic behavior versus the exact time-dependence.
- `de.timesteppers.RK111` – Like `SBDF1`, this timestepper is first-order and should only be used for exploratory runs.
- `de.timesteppers.RK222` – Second-order, two-stage Runga-Kutta. For computationally intensive tasks such as computing ECS, I have found this to strike a good balance between speed and accuracy. It is roughly 2x slower than `SBDF1` and `RK111`.
- `de.timesteppers.RK443` – Third-order, four-stage Runga-Kutta. This is the slowest and most accurate of the timesteppers listed here: 2x as slow as `RK222` and 4x as slow as `SBDF1`. I recommend it for all production runs.

6.5 Running the solver

Iterating the time-dependent solver is simple. First, one specifies stopping conditions, e.g.

```
solver.stop_sim_time = np.inf
solver.stop_wall_time = np.inf
solver.stop_iteration = np.inf
```

These can be set to infinity, as above, in which case they will not act as stopping conditions. Otherwise, `stop_sim_time` gives the stop time in simulation units, `stop_wall_time` gives the stop time in seconds of real (wall) time, and `stop_iteration` is the iteration number at which to stop the solver.

The final timestepping loop takes up just a few lines of code:

```
while solver.ok:
        solver.step(dt)
        if solver.iteration % 10 == 0:
                logger.info('Iteration: %i, Time: %e, dt: %e' %(solver.iteration, solver.sim_time, dt))
        dt = CFL.compute_dt()
```

The above example prints logging information every 10 iterations, and the last line updates the timestep via the `CFL` object, assuming one was created as described in section 6.4.

6.6 Data output

HDF5 file format.— The default output mode for Dedalus is the Hierarchical Data Format (HDF5). My experience with this format has been positive: both MATLAB and Python have straightforward APIs for reading data. In my simulations, I have been using `.h5` as the extension for HDF5 files. Below is a list of resources:

- Wikipedia article – The usual straightforward overview
- HDF View – Software with a GUI for browsing the metadata of HDF5 files. This is easier and faster than typing out all the MATLAB or Python commands to move through the metadata tree. I use this regularly.
- Python quick start – Quick start guide for `h5py`, the Python package for reading HDF5 files.

Specifying tasks.— It is straightforward to manually write data to an HDF5 file. I do so often, especially in the code for computing ECS. At the same time, Dedalus has powerful output capabilities built on top of the `solver` class, and I use these extensively as well. They allow one to calculate functions of the solution fields, including integrals over the domain, and output the data to an HDF5 file. I will defer to the Dedalus documentation for a comprehensive description, but example usage for active nematics is shown below:

```
order_params = solver.evaluator.add_file_handler('order_params', iter = 1, max_size = np.inf)
order_params.add_task("integ(U)/(height*width)", layout='g', name='u_int')
order_params.add_task("integ(2*sqrt(QAA*QAA+QAB*QAB))/(height*width)", layout='g', name='S_int')

full_solution = solver.evaluator.add_file_handler('full_solution', iter = 5, max_size = np.inf)
full_solution.add_task('QAA', layout='g', name='QAA')
full_solution.add_task('QAB', layout='g', name='QAB')
full_solution.add_task('U', layout='g', name='U')
full_solution.add_task('V', layout='g', name='V')
```

Assuming the code is run in serial and there is no preexisting output, the first three lines instruct the Dedalus solver to create a file `order_params_s1_p0.h5` under the directory `./order_params/order_params_s1`. Every iteration, Dedalus writes two data arrays to the file, containing the average x-velocity and average order parameter in the channel. These each receive a key–`'/tasks/u_int'` and `'/tasks/S_int'`, respectively–that label the data in the HDF5 file.

The last five lines output the full solution fields `QAA, QAB, U, V` to `full_solution_s1_p0.h5` every five iterations. The keyword assignment `layout = 'g'` tells Dedalus to output in grid space; `layout = 'c'` would output the fields in coefficient space.

6.7 Postprocessing

To create snapshots and videos, see MATLAB scripts `snapshot.m` and `video.m` and the documentation therein.

6.8 Dedalus parameters

- File handler max size `dedalus.core.evaluator.max_size`
  This sets a maximum size, in bytes, for individual files. Once a file exceeds the maximum size, it is closed and a new file opened to continue the output. I generally set this to infinity since there has not been much risk of accidentally making too large of a file. But when running large simulations, it may make sense to build in a safety here because extremely large HDF5 files may overload RAM space in postprocessing.
  Example usage:

  ```
  fh = solver.evaluator.add_file_handler('name', iter=20, max_size = np.inf)
  ```

- CFL Parameters
  - Cadence (`int`, optional) – Iteration cadence for computing new timestep (default: 1)

- Threshold (`float`, optional) – Fractional change threshold for changing timestep (default: 0)
- Every time the timestep is adjusted, the Dedalus solver has to recalculate certain coefficient matrices. So increasing cadence or threshold will usually speed up the simulation (unless, of course, they are too large and lead to an instability). Although I do not use the CFL class for calculating ECS and connections, when I have used it for general time-dependent simulations, I have chosen cadence = 10 and threshold = 0.1. These values are based on existing Dedalus example scripts.

6.9 Miscellaneous

- The time integration has better stability properties if all the linear terms are put on the left hand side of the equations. The reason is that Dedalus integrates the l.h.s. implicitly and the r.h.s. explicitly (even if the r.h.s. contains linear terms).
- It is possible to use periodic boundary conditions with the Chebyshev basis; simply use:

  ```
  problem.add_bc("left(f) - right(f) = 0")
  ```

  in all the problem variables, including any extra variables added to cast the problem into a first-order system.

7 Newton-Krylov-Hookstep: general theory

We will first explain the Newton-Krylov-Hookstep method in the context of computing equilibria. Section 7.4 extends the method to traveling waves, periodic orbits, and relative periodic orbits.

Let $\mathbf{X} = \{\mathbf{v}, \mathbf{Q}\}$ denote the set of solution fields and $\mathbf{F}(\mathbf{X})$ the right hand side of the equations of motion. Then,

$$\partial_t \mathbf{X} = \mathbf{F}(\mathbf{X}) \tag{55}$$

Equilibria satisfy $\mathbf{F}(\mathbf{X}) = 0$. If we expand X in a spectral representation (e.g. Fourier series or some other functional basis), then $\mathbf{F}(\mathbf{X}) = 0$ becomes a countably infinite set of nonlinear algebraic equations in the spectral coefficients. Our task is to solve this system of equations. Unlike linear systems, however, there is no general, closed-form solution for nonlinear systems, and iterative methods are usually required. Here, we use a version of Newton's method, which iteratively solves a sequence of linear problems in the hope of converging to a root of the nonlinear system.

7.1 Newton's method (elementary version)

In this section, we state the elementary version of Newton's method, i.e., without the Krylov and hookstep modifications. Let $\mathbf{X} = (X_1, X_2, X_3, \ldots)^T$ be the components of $\mathbf{X}$ in some functional basis, and $\mathbf{F} = (F_1(\mathbf{X}), F_2(\mathbf{X}), \ldots)^T$ be the corresponding components of $\mathbf{F}$. In Dedalus, for instance, the components $(X_1, X_2, X_3, \ldots)^T$ will be the Fourier and Chebyshev expansion coefficients of $\mathbf{X}$. Next, suppose $\mathbf{F}(\mathbf{X})$ is differentiable in the neighborhood of a root $\mathbf{X}^*$, so that we can define the Jacobian matrix $\mathbf{J}(\mathbf{X})$ with elements

$$[\mathbf{J}(\mathbf{X})]_{ij} = \frac{\partial F_i(\mathbf{X})}{\partial X_j} \tag{56}$$

Any version of Newton's method requires an initial guess $\mathbf{X}^{(0)}$. Assuming $\mathbf{X}^{(0)}$ is known, the elementary version of Newton's method generates a sequence of iterates $\mathbf{X}^{(1)}, \mathbf{X}^{(2)}, \ldots$ by carrying out the following steps for $k = 1, 2, \ldots$:

1. Solve $\mathbf{J}\left[\mathbf{X}^{(k)}\right] \mathbf{Y}^{(k)} = -\mathbf{F}[\mathbf{X}^{(k)}]$ for $\mathbf{Y}^{(k)}$.

2. Set $\mathbf{X}^{(k+1)} \leftarrow \mathbf{X}^{(k)} + \mathbf{Y}^{(k)}$. If the residual $\epsilon_{k+1} = ||\mathbf{X}^{(k+1)}||$ is less than the target threshold, exit the algorithm. Otherwise, take $k \leftarrow k+1$ and go to step 1.

Convergence means that $||\mathbf{X}^{(k)} - \mathbf{X}^*||$ as $k \to \infty$.

7.1.1 Motivation and visualization

The elementary version of Newton's method can be thought of as a sequence of linear approximations to $\mathbf{F}(\mathbf{X}) = 0$. Specifically, suppose $\mathbf{X}^{(k)}$ is close to the solution $\mathbf{X}^*$, i.e.

$$\mathbf{X}^{(k)} + \epsilon\,\mathbf{Y} = \mathbf{X}^* \tag{57}$$

where $\frac{||\mathbf{Y}||}{||\mathbf{X}^*||} = \mathcal{O}(1)$ and $\epsilon \ll 1$. Then, $\mathbf{F}[\mathbf{X}^{(k)} + \epsilon\,\mathbf{Y}] = 0$, and the task is to solve for $\mathbf{Y}$. Of course, this equation is only a reformulation of the original problem and is no easier to solve. However, if ϵ is small, then we can approximate $\mathbf{F}$ by its Taylor series about $\mathbf{X}^{(k)}$:

$$\mathbf{F}[\mathbf{X}^{(k)} + \epsilon\,\mathbf{Y}] \approx \mathbf{F}[\mathbf{X}^{(k)}] + \epsilon\,\mathbf{J}[\mathbf{X}^{(k)}]\mathbf{Y} \tag{58}$$

Setting the above to 0 and solving for $\mathbf{Y}$ is equivalent to step 1 in the algorithm. In 1D, $\mathbf{Y}$ is the point of intersection between the x-axis and the tangent line at $\mathbf{F}[\mathbf{X}^{(k)}]$.

This derivation gives insight into the performance of the method: it will not work well if $\mathbf{F}$ is ill-behaved in the neighborhood of $\mathbf{X}^*$, or if $\mathbf{X}^{(k)}$ is too far from $\mathbf{X}^*$, in which case the linearization does not adequately capture the behavior of $\mathbf{F}$ near $\mathbf{X}^*$.

7.1.2 Convergence

Assuming $\mathbf{J}(\mathbf{X}^*)$ is nonsingular and the initial guess $\mathbf{X}^{(0)}$ is sufficiently close to $\mathbf{X}^*$, then there exists $k^* > 0$ such that, for $k > k^*$, Newton's method converges quadratically. This means there exists a constant $C > 0$ such that $||\mathbf{X}^{(k+1)} - \mathbf{X}^*|| < C||\mathbf{X}^{(k)} - \mathbf{X}^*||^2$ for all $k > k^*$. This type of convergence is very fast. For example, if the error at the k^{th} iteration is $\mathcal{O}(10^{-5})$, the the error at the next iteration is $\mathcal{O}(10^{-10})$, assuming $C = \mathcal{O}(1)$ for simplicity.

Quadratic convergence is an idealization that may not be realized in typical applications. For example, even if $\mathbf{J}(\mathbf{X}^*)$ is nonsingular, the region of quadratic convergence may be extremely small. Modifications of the elementary Newton's method (such as Krylov subspace methods) may also destroy quadratic convergence. For these reasons, the algorithm we use for computing ECS usually exhibits linear convergence at best – that is, each iteration decreases the error by a constant factor.

7.2 Hookstep modification

While Newton's method performs well locally, its global convergence is unreliable. For computing ECS, global convergence is important because we often are unable to construct initial guesses sufficiently close to an ECS. One modification of Newton's method that improves global performance is the *hookstep*, which constrains the step size of each iteration. The idea is that one can choose the maximum step size small enough that the linearization remains valid in the region of interest, in which case the Newton iteration can navigate the local functional landscape and find a direction that reliably decreases the value of objective function. In the typical case that the local topography is unknown, one must try different values for the maximum step size and see which one decreases the objective function past some threshold.

It is crucial to recognize the difference between the hookstep and the damped Newton's method. The latter solves the full linear system as usual, and then "damps" the step by multiplying $\mathbf{Y}^{(k)}$ by some factor $\mu < 1$. This approach performs poorly compared to the hookstep because the linear system is solved assuming it is valid all the way to the root $\mathbf{Y}^{(k)}$. Unless we are close to the solution, this assumption generally fails. As a result, $\mathbf{Y}^{(k)}$ likely points in the wrong direction, and multiplying it by a damping factor $\mu < 1$ will not change this fact. In one or two dimensions, it is easy to overlook the importance of

the orientation of $\mathbf{Y}^{(k)}$, but in high-dimensional spaces the majority of degrees of freedom are contained in the orientation, so it is especially important to compute accurately.

The step size constraint is imposed when solving the linear system at each iteration. The resulting system is overconstrained, so a least-squares approach or similar is necessary. The least-squares formulation is equivalent to a quadratic minimization with quadratic constraint, which can be solved efficiently using standard techniques.

Denote the maximum step size by ρ. This is also called the trust radius. Then, the Newton-Hookstep algorithm consists of the following steps for $k = 1, 2, \ldots$:

1. Initialize the trust radius $\rho = \rho_0$, where ρ_0 is guessed or taken from the previous iteration.
2. Solve $\mathbf{J}\left[\mathbf{X}^{(k)}\right]\mathbf{Y}^{(k)} = -\mathbf{F}[\mathbf{X}^{(k)}]$ for $\mathbf{Y}^{(k)}$, subject to the hookstep constraint $||\mathbf{Y}^{(k)}|| < \rho$.
3. Evaluate the residual $\epsilon_{k+1} = ||F[\mathbf{X}^{(k)} + \mathbf{Y}^{(k)}]||$. If $\epsilon_{k+1}/\epsilon_k$ is less than some threshold h, proceed to step 4. Otherwise, decreases the step size, e.g., set $\rho \leftarrow \rho/2$ and return to step 2.
4. Set $\mathbf{X}^{(k+1)} \leftarrow \mathbf{X}^{(k)} + \mathbf{Y}^{(k)}$. If ϵ_{k+1} is less than the target threshold, exit the algorithm. Otherwise, take $k \leftarrow k + 1$ and go to step 1.

There are a few knobs to tune in the hookstep algorithm, such as varying the threshold h on $\epsilon_{k+1}/\epsilon_k$ for accepting the hookstep, or changing how the trust radius is updated if the acceptance criterion is not met. For now, we note that the simple choice $h = 0.99$ and trust radius update $r \leftarrow r/2$ work well for computing periodic orbits in active nematic channel flow.

7.3 Solving the linear system: Krylov subspace methods

7.3.1 Background and motivation

The number of degrees of freedom required to resolve physically relevant flows can be extremely large– $\approx 10^5$ in turbulent 2D flows and $> 10^6$ in turbulent 3D flows. As a result, the linear system in Newton's method may be intractably large.

Fortunately, solving the full system to machine precision may not be necessary, because Newton's method can still function well even if the linear system is only solved approximately. Intuitively, this makes sense: the linearization is approximate to begin with, so adding a small error from a different source should not damage convergence too severely. In fact, this notion is implicit in the hookstep modification from section 7.2.

This reasoning underlies a class of *inexact* variants of Newton's method [3], in which various approximation schemes are applied to the linear system. Compared with the exact version of Newton's method, these variants generally require more iterations to converge. However, for high-dimensional problems the benefit in approximating the linear system may substantially outweigh the cost of performing more iterations. (Although quadratic convergence may be lost, inexact methods generally do preserve linear or superlinear convergence.)

In our case, we employ Krylov subspace methods, which have been successful in a variety of contexts involving large linear systems. The following sections describe the formulation and implementation.

7.3.2 Krylov subspace approximations

Suppose $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{u}$ is an n-dimensional vector. Then, the Krylov subspace $\mathcal{K}_m(\mathbf{A}, \mathbf{u})$ is defined as [3]:

$$\mathcal{K}_m(\mathbf{A}, \mathbf{u}) = \text{span}\{\mathbf{u}, \mathbf{A}\mathbf{u}, \ldots, \mathbf{A}^{m-1}\mathbf{u}\} \tag{59}$$

Note that $\mathcal{K}_m(\mathbf{A}, \mathbf{u})$ is not necessarily m-dimensional, as it is possible that one or more vectors in the span are linearly dependent on the others.

Krylov subspaces are widely used to approximate large linear systems. The strategy is to look for $\mathbf{A}$ and $\mathbf{u}$ such that the low-dimensional Krylov subspace $\mathcal{K}_m(\mathbf{A}, \mathbf{u})$, with $m \ll n$, overlaps with the most important components of the original $n \times n$ system. Then, by projecting onto $\mathcal{K}_m(\mathbf{A}, \mathbf{u})$, one obtains a low-dimensional approximation to the original system, which can be solved directly. The success of this approach depends on finding $\mathbf{A}$ and $\mathbf{u}$ such that the subspace approximation is good even for small m. Often, $\mathbf{A}$ and $\mathbf{u}$ are constructed using the matrices and vectors of the original system, as illustrated below. One also needs to decide how to perform the subspace projection, as there is not a unique way of doing so. Different choices of the projection, possibly in combination with additional assumptions on $\mathbf{A}$, give rise to various well-known approximation methods, such as Conjugate Gradients (CG), Lanczos, bi-CG, SYMMLQ, and GMRES.

Intuitive justification

Why use Krylov subspaces for approximating linear systems? One answer is to consider the following Neumann series solution of a linear system:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \rightarrow [\mathbf{I} - (\mathbf{I} - \mathbf{A})]\,\mathbf{x} = \mathbf{b} \tag{60}$$

$$\rightarrow \mathbf{x} = [\mathbf{I} - (\mathbf{I} - \mathbf{A})]^{-1}\,\mathbf{b} \tag{61}$$

$$\rightarrow \mathbf{x} = \left(\sum_{i=0}^{\infty} (\mathbf{I} - \mathbf{A})^i\right)\mathbf{b} \tag{62}$$

This sum converges if $||\mathbf{I} - \mathbf{A}|| < 1$ in some matrix norm. For sake of argument, suppose $||\mathbf{I} - \mathbf{A}||$ is small and $\mathbf{b}$ is "typical", so that a low-order truncation up to $i = m \ll n$ is a good approximation in the l^2 norm:

$$\mathbf{x} \approx \mathbf{x}_m \equiv \left(\sum_{i=0}^{m} (\mathbf{I} - \mathbf{A})^i\right)\mathbf{b} \tag{63}$$

$$\in \text{span}\{\mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}^2\mathbf{b}, \ldots, \mathbf{A}^m\mathbf{b}\} \tag{64}$$

In other words, there exists $\mathbf{x}_m \in \mathcal{K}_{m+1}(\mathbf{A}, \mathbf{b})$ satisfying $||\mathbf{A}\mathbf{x}_m - \mathbf{b}|| \ll ||\mathbf{b}||$. It is therefore reasonable to expect that $\mathbf{x}$ could be approximated by solving some optimality condition restricted to $\mathcal{K}_{m+1}(\mathbf{A}, \mathbf{b})$. This is the approach taken by Krylov subspace methods. For example, minimizing $||\mathbf{A}\mathbf{x} - \mathbf{b}||$ over $\mathbf{x} \in \mathcal{K}_{m+1}(\mathbf{A}, \mathbf{b})$ leads to GMRES, mentioned above.

This explanation is not completely satisfactory because $||\mathbf{I} - \mathbf{A}||$ could easily be greater than 1. Then the Neumann series would no longer converge, and we would need a different method for proving that there exists $\mathbf{x}_m \in \mathcal{K}_{m+1}(\mathbf{A}, \mathbf{b})$ that satisfies $||\mathbf{A}\mathbf{x}_m - \mathbf{b}||$. Nevertheless, the span defining $\mathcal{K}_{m+1}(\mathbf{A}, \mathbf{b})$ exists regardless, and the special case $||\mathbf{I} - \mathbf{A}|| < 1$ suggests that it may be related to the dominant eigenspace of $\mathbf{A}$.

To investigate this possibility further, we will ignore mathematical precision and make a rough qualitative argument. First, let us drop the condition $||\mathbf{I} - \mathbf{A}|| < 1$ but impose three new conditions:

1. $\mathbf{A}$ is diagonalizable and has n distinct eigenvalues, such that there is a orthonormal eigenbasis $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_m\}$. Note: distinct eigenvalues is not required for the arguments that follow, but it simplifies the presentation.

2. The spectrum of $\mathbf{A}$ splits into two distinct sets – namely, there is a cluster of $n - m$ eigenvalues with magnitude close to ϵ, whereas the remaining $m \ll n$ eigenvalues are spaced apart and have magnitude much larger than ϵ. Denote the corresponding eigenspaces by $\mathscr{E}_{n-m}$ and $\mathscr{E}_m$.

3. The orthogonal projection of $\mathbf{b}$ onto $\mathscr{E}_{n-m}$ is small.

Since we are considering norms and projections with respect to the l^2 inner product, it is not useful to work with the representation $\mathcal{K}_m(\mathbf{A}, \mathbf{b}) = \text{span}\{\mathbf{b}, \mathbf{A}\mathbf{b}, \ldots, \mathbf{A}^{m-1}\mathbf{b}\}$ because the vectors in the span are

highly non-orthonormal. However, it is easy to generate an orthonormal basis for $\mathcal{K}_m(\mathbf{A}, \mathbf{b})$ by performing the Gram-Schmidt process. Doing so leads to a crucial result: under the assumptions stated above, the orthonormal Gram-Schmidt vectors approximate the dominant eigenvectors $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_m\}$ of $\mathbf{A}$, so that $\mathcal{K}_m(\mathbf{A}, \mathbf{b}) \approx \mathscr{E}_m$ (details left for later). If, like $\mathbf{b}$, the orthogonal projection of $\mathbf{x}$ onto $\mathscr{E}_{n-m}$ is small, then $\mathbf{x}$ approximately lies in $\mathscr{E}_m$ and, by extension, in $\mathcal{K}_m(\mathbf{A}, \mathbf{b})$.

In summary, we have considered two scenarios for the system $\mathbf{Ax} = \mathbf{b}$. In the first, $||\mathbf{I} - \mathbf{A}|| < 1$ and the usefulness of Krylov subspaces can be explained using a truncated Neumann series. In the second, $||\mathbf{I}-\mathbf{A}||$ is unconstrained, but $\mathbf{A}$ and $\mathbf{b}$ must satisfy other conditions. The intuition behind Krylov subspace approximations in this case is that $\mathcal{K}_m(\mathbf{A}, \mathbf{b})$ is roughly equivalent to the dominant eigenspace $\mathscr{E}_m$.

The second scenario is typical of the linear systems encountered when solving for active nematic ECS. In particular, for computing periodic orbits, the dissipative nature of the dynamics naturally preconditions $\mathbf{A}$ so that the eigenvalues asymptotically cluster around 0.

7.3.3 Implementation

Choice of subspace

The arguments from the previous section suggest using the Krylov subspace $\mathcal{K}_m(\mathbf{A}, \mathbf{b})$ for the system $\mathbf{Ax} = \mathbf{b}$. Depending on the context, it may be necessary to precondition $\mathbf{A}$, i.e. solve the modified system

$$\mathbf{P}^{-1}\mathbf{Ax} = \mathbf{P}^{-1}\mathbf{b} \tag{65}$$

in the subspace $\mathcal{K}_m(\mathbf{P}^{-1}\mathbf{A}, \mathbf{P}^{-1}\mathbf{b})$. Ideally, we would choose $\mathbf{P}$ such that $\mathbf{P}^{-1}\mathbf{A} \approx \mathbf{I}$, but for Krylov subspace methods it may suffice to cluster the eigenvalues as described in the previous section. We will see later that integrating the active nematics equations in time already has this effect, so no preconditioning is necessary. For equilibria and traveling waves, however, the eigenvalues of the un-preconditioned linear operator asymptotically diverge, so we do need to perform preconditioning.

Orthonormal basis

The previous section also suggests the importance of using an orthonormal basis for the Krylov subspace. Even without the detailed arguments given there, it should be apparent that a basis constructed from the set $\{\mathbf{b}, \mathbf{Ab}, \ldots, \mathbf{A}^{m-1}\mathbf{b}\}$ (excluding any linearly dependent vectors) is undesirable from a numerical standpoint. As m increases, the last few elements in the set become nearly parallel, which raises concerns about round-off error. A common solution to this issue is to step-by-step orthonormalize the elements as the Krylov subspace is constructed, usually via the modified Gram-Schmidt process. This approach is called the Arnoldi procedure. For the m^{th} Krylov subspace $\mathcal{K}_m$, denote the resulting orthonormal basis as

$$\mathscr{V}_m = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\} \tag{66}$$

In addition, let $\mathbf{V}_m$ be $n \times m$ matrix whose columns are the basis vectors:

$$\mathbf{V}_m = \begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_m \end{pmatrix} \tag{67}$$

This is a unitary transformation from $\mathcal{K}_m$ to $\mathbb{R}^n$, that is, $\mathbf{V}_m\mathbf{V}_m^\dagger = \mathbf{I}_n$. Therefore, for vector $\mathbf{y} \in \mathcal{K}_m$,

$$||\mathbf{V}_m\mathbf{y}||^2 = (\mathbf{V}_m\mathbf{y})^\dagger \mathbf{V}_m\mathbf{y} = \mathbf{y}^\dagger\mathbf{V}_m^\dagger\mathbf{V}_m\mathbf{y} = ||\mathbf{y}||^2 \tag{68}$$

We will use this result in the next section.

Projection onto the Krylov subspace

Last, we need to project the full $n \times n$ system $\mathbf{Ax} = \mathbf{b}$ onto $\mathcal{K}_m(\mathbf{A}, \mathbf{b})$. This can be done in different ways, leading to different categories of Krylov subspace methods. In ECSAct, we adopt the *minimal residual norm approach*, which minimizes the residual $r_m = ||\mathbf{Ax}_m - \mathbf{b}||$ over $\mathbf{x}_m \in \mathcal{K}_m(\mathbf{A}, \mathbf{b})$. With a good choice of subspace, we hope that $\mathbf{x}_m \approx \mathbf{x}$.

The minimum residual norm approach can be formulated as a least-squares problem entirely within the Krylov subspace, as follows [7]. First, the Arnoldi procedure applied to $\mathcal{K}_{m+1}(\mathbf{A}, \mathbf{b})$ leads to an $(m+1) \times m$ upper Hessenberg matrix $\mathbf{H}_{m+1,m}$ that satisfies

$$\mathbf{AV}_m = \mathbf{V}_{m+1}\mathbf{H}_{m+1,m} \tag{69}$$

For $\mathbf{x}_m \in \mathcal{K}_m$, we can write $\mathbf{x}_m = \mathbf{V}_m\mathbf{y}$, where $\mathbf{y}$ is an $m \times 1$ matrix. Then,

$$\begin{aligned} \rho_m = ||\mathbf{Ax}_m - \mathbf{b}|| &= ||\mathbf{AV}_m\mathbf{y} - \mathbf{b}|| && (70)\\ &= ||\mathbf{V}_{m+1}\mathbf{H}_{m+1,m}\mathbf{y} - r_0\mathbf{V}_{m+1}\mathbf{e}_1|| && (71)\\ &= ||\mathbf{H}_{m+1,m}\mathbf{y} - r_0\mathbf{e}_1|| && (72) \end{aligned}$$

where $\mathbf{e}_1$ is the $(m+1)$-dimensional unit vector with first element 1 and all others 0. In going from (71) to (72), we used equation 68. The minimization of r_m over $\mathcal{K}_m$ is therefore equivalent to an $(m+1) \times m$ least-squares problem. For many practical problems, this reduction from a n dimensions to m is an enormous achievement. For example, in turbulent 2D active nematics, $n = \mathcal{O}(10^5)$, whereas $m = \mathcal{O}(10^2)$ (or less) gives accurate approximations to the original $n \times n$ system.

The modification to include the hookstep is straightforward: we simply add a quadratic constraint $||\mathbf{y}||^2 < \rho^2$ to the least-squares problem in (72).

7.3.4 Convergence

It is useful to think of Krylov subspace methods as iterative procedures in the order of the subspace. Let $\mathbf{x}_{i+1}$ be the vector obtained by approximately solving the $n \times n$ system $\mathbf{Ax} = \mathbf{b}$ within $\mathcal{K}_{i+1}(\mathbf{A}, \mathbf{b})$. Because computing $\mathcal{K}_{i+1}$ implicitly involves the construction of $\mathcal{K}_i$, and $\mathcal{K}_i \subseteq \mathcal{K}_{i+1}$, it makes sense to think of $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_n$ as a sequence of successive approximations to $\mathbf{Ax} = \mathbf{b}$. Moreover, if there are no invariant subspaces of $\mathbf{A}$ with respect to $\mathbf{b}$, then $\mathcal{K}_n(\mathbf{A}, \mathbf{b})$ is n-dimensional. Therefore, in exact arithmetic, the iterative procedure converges in at most n steps.

For $i < n$, however, the convergence may be nonuniform, and in worst-case scenarios $\mathbf{x}_i$ is a poor approximation for all $i < n$. However, in many applications convergence is (empirically) observed to be much better, and adequate approximations are obtained for $i \ll n$. If $\mathbf{A}$ and $\mathbf{b}$ satisfy certain assumptions, then the convergence behavior can be analyzed rigorously. For example, if $\mathbf{A}$ is Hermitian and positive-definite, then the conjugate gradients method (CG) satisfies convergence bounds depending only on the condition number of $\mathbf{A}$.

For general $\mathbf{A}$ and $\mathbf{b}$ (e.g. non-symmetric or indefinite $\mathbf{A}$), then convergence is much harder to analyze. For active nematics problems, this situation is typical because $\mathbf{A}$ is generally non-symmetric and indefinite. For the subspace approximation method presented in the previous section, not even complete knowledge of the spectrum of $\mathbf{A}$ fully characterizes the convergence behavior – that is, there exist matrices with the same spectrum but different convergence behavior. Nevertheless, some rules of thumb provide useful guidelines. Based partly on a rigorous result for the Hermitian, positive-definite case, clustering eigenvalues seems to improve convergence (so long as $\mathbf{A}$ is not too non-normal). Monitoring the eigenvalues of $\mathbf{H}_{m,m}$ (c.f. equation (69)) can also provide real-time feedback on the convergence behavior as m varies. The current version of ECSAct does not utilize information from $\mathbf{H}_{m,m}$ in this way, although the eigenvalues of $\mathbf{H}_{m,m}$ are calculated in the linear stability code, because for increasing m they are increasingly accurate approximations of the dominate eigenvalues of $\mathbf{A}$.

7.4 Extension to periodic orbits

We have described the Newton-Krylov-Hookstep algorithm in the context of equilibria of the system $\partial_t \mathbf{X} = F(\mathbf{X})$, i.e. $F(\mathbf{X}) = 0$. The same algorithm carries over straightfowardly to other fixed point problems such as traveling waves and (relative) periodic orbits, the main difference being the form of the linear operator. Here, we formulate the problem for periodic orbits. Traveling waves and relative periodic orbits depend on the spatial geometry, so we leave them for later.

Define the flow map $\Phi(\mathbf{X}_0, t) = \mathbf{X}_0 + \int_0^t F\left[\mathbf{X}(s)\right] ds$. A point $\mathbf{X}_{\mathrm{PO}}$ on periodic orbit (PO) satisfies

$$\Phi(\mathbf{X}_{\mathrm{PO}}, T) - \mathbf{X}_{\mathrm{PO}} = 0 \qquad \text{(periodic orbit)}, \tag{73}$$

where T is the period. It is useful to define a linear operator $\mathcal{L}$ that maps a small perturbation $\epsilon \frac{\delta \mathbf{X}}{||\delta \mathbf{X}||}$ about $\mathbf{X}$ (not necessary on a periodic orbit) onto the corresponding perturbation about $\Phi(\mathbf{X}, T)$:

$$\mathcal{L}\left(\mathbf{X}, T\right) \cdot \left(\epsilon \frac{\delta \mathbf{X}}{||\delta \mathbf{X}||}\right) = \Phi\left(\mathbf{X} + \epsilon \frac{\delta \mathbf{X}}{||\delta \mathbf{X}||}, T\right) - \Phi(\mathbf{X}, T) + \mathcal{O}\left(\epsilon^2\right) \tag{74}$$

$$\rightarrow \mathcal{L}\left(\mathbf{X}, T\right) \cdot \delta \mathbf{X} = \lim_{\epsilon \to 0} \frac{\Phi\left(\mathbf{X} + \epsilon \frac{\delta \mathbf{X}}{||\delta \mathbf{X}||}, T\right) - \Phi(\mathbf{X}, T)}{\epsilon / ||\delta \mathbf{X}||} \tag{75}$$

Then, expanding (73) to linear order about a base state $\mathbf{X}_0$ gives the linear system

$$\mathcal{L}\left(\mathbf{X}_0, T\right) \cdot \delta \mathbf{X} - \delta \mathbf{X} = -\Phi(\mathbf{X}_0, T) + \mathbf{X}_0 \tag{76}$$

If T is known, and we want to find a point on the PO starting from $\mathbf{X}_0$, then the algorithms from the preceding sections carry over with the new linear system (76). In the typical case that T is unknown, we must include it as a dependent variable and expand the linear system. Specifically, we replace (76) with

$$\begin{pmatrix} 0 & \left.\dfrac{\partial \Phi(\mathbf{X}_0, t)}{\partial t}\right|_{t=0} \\ \left.\dfrac{\partial \Phi(\mathbf{X}_0, t)}{\partial t}\right|_{t=T_0} & \mathcal{L}\left(\mathbf{X}_0, T_0\right) - \mathbf{I} \end{pmatrix} \begin{pmatrix} \delta T \\ \delta \mathbf{X} \end{pmatrix} = \begin{pmatrix} 0 \\ -\Phi(\mathbf{X}_0, T_0) + \mathbf{X}_0 \end{pmatrix} \tag{77}$$

where T_0 is the base value for the period. The bottom row of the 2×2 block matrix (n rows when expanded) is the Taylor series of (73) about $(\mathbf{X}_0, T_0)$. The top row is added to produce a square $(n+1) \times (n+1)$ system: it requires that $\delta \mathbf{X}$ is orthogonal to a change in $\mathbf{X}_0$ stemming from a shift of the time origin. In other words, we exclude redundant perturbations that shift $\mathbf{X}_0$ parallel to the trajectory.

8 Newton-Krylov-Hookstep: Formulation for active nematic ECS

ECSAct can compute four types of ECS in active nematic channel flow: equilibria, traveling waves, periodic orbits, and relative periodic orbits. In this section, we formulate the fixed-point equations for each, and derive corresponding linear operators to use in the Newton-Krylov-Hookstep algorithm. The next section focuses on the code and Dedalus implementation.

The phase space representation $\partial_t \mathbf{X} = \mathbf{F}(\mathbf{X})$, (equation (55)), of the active nematic equations corresponds to

$$\mathbf{X} = \begin{pmatrix} Q_{xx} \\ Q_{xy} \\ U \\ V \end{pmatrix} \tag{78}$$

Here, $\mathbf{X}$ should be understood as a 4×1 block matrix, where each block contains the spectral coefficients (Fourier and Chebyshev) of the corresponding field. Expanded out, $\mathbf{X}$ becomes a $4N_xN_y$-dimensional vector, where N_x and N_y are the number of Fourier and Chebyshev modes of each of the four fields. As usual, the pressure p is determined from the constraint $\partial_x U + \partial_y V = 0$, and we do not consider it as an independent field. The nonlinear operator $\mathbf{F}(\mathbf{X})$ comes directly from the hydrodynamic equations, and has the same dimension as $\mathbf{X}$. As explained in section 9, $\mathbf{F}(\mathbf{X})$ is easy to evaluate in vector form using the field and operator classes in Dedalus.

8.1 Equilibria

Equilibria satisfy $\mathbf{F}(\mathbf{X}) = 0$. In ECSAct, two methods are available for solving this equation, which use the same linear operator but differ in their choice of preconditioner. Section 8.1.2 discusses computation of the unpreconditioned linear operator, and section 8.1.3 describes the two preconditioners.

8.1.1 Nonlinear operator

For equilibria, the nonlinear operator is the right-hand-side of the equations of motion, which we have been calling $\mathbf{F}(\mathbf{X})$ in general notation. Similar to $\mathbf{X}$, it is convenient to split $\mathbf{F}$ into a 4×1 block matrix corresponding to each field component in $\mathbf{X}$ (see equation 78).

8.1.2 Linear operator

To apply Newton's method to $\mathbf{F}(\mathbf{X}) = 0$, we need to compute the Jacobian matrix $\mathbf{J}(\mathbf{X})$ defined in equation (56). Before giving its expression, we note that the terminology *vector* and *matrix* is an oversimplification because we are technically dealing with members of a functional space. The Jacobian $\mathbf{J}(\mathbf{X})$ is really a linear operator defined on the appropriate functional space. However, for the systems we consider, a finite-dimensional truncation is a permissible approximation, and we can think of the Jacobian as a very large but finite-dimensional matrix.

Keeping this in mind, we first define $\mathbf{J}(\mathbf{X})$ in the abstract sense of a linear operator, i.e. in terms of its action on an nonzero element $\mathbf{Y}$ of the vector space:

$$\mathbf{J}(\mathbf{X}) \cdot \mathbf{Y} = \lim_{\epsilon \to 0} \frac{\mathbf{F}(\mathbf{X} + \epsilon \mathbf{Y}) - \mathbf{F}(\mathbf{X})}{\epsilon} \tag{79}$$

In the finite-dimensional approximation, we can substitute $\mathbf{Y} = \mathbf{e}_j$ (the j^{th} unit vector) and extract the matrix components of $\mathbf{J}(\mathbf{X})$:

$$[\mathbf{J}(\mathbf{X})]_{ij} = \lim_{\epsilon \to 0} \frac{\mathbf{F}_i(\mathbf{X} + \epsilon\, \mathbf{e}_j) - \mathbf{F}(\mathbf{X})}{\epsilon} \tag{80}$$

$$= \frac{\partial F_i(\mathbf{X})}{\partial X_j} \tag{81}$$

The last line is the same as (56).

For computing equilibria using the Newton-Krylov-Hookstep method, we do not require the full Jacobian matrix, only its action on certain vectors $\mathbf{Y}$ related to the Krylov subspace iterations. For this purpose, equation (79) is most appropriate. In fact, we can use the equations of motion to explicitly evaluate (79), dispensing with any finite-differencing and limit-taking. Let $\mathbf{X}^{(0)}$ be the base state and $\mathbf{X}^{(1)}$ the perturbation, with 4×1 block matrix representations as follows:

$$\mathbf{X}^{(0)} = \begin{pmatrix} Q_{xx}^{(0)} \\ Q_{xy}^{(0)} \\ U^{(0)} \\ V^{(0)} \end{pmatrix}; \quad \mathbf{X}^{(1)} = \begin{pmatrix} Q_{xx}^{(1)} \\ Q_{xy}^{(1)} \\ U^{(1)} \\ V^{(1)} \end{pmatrix} \tag{82}$$

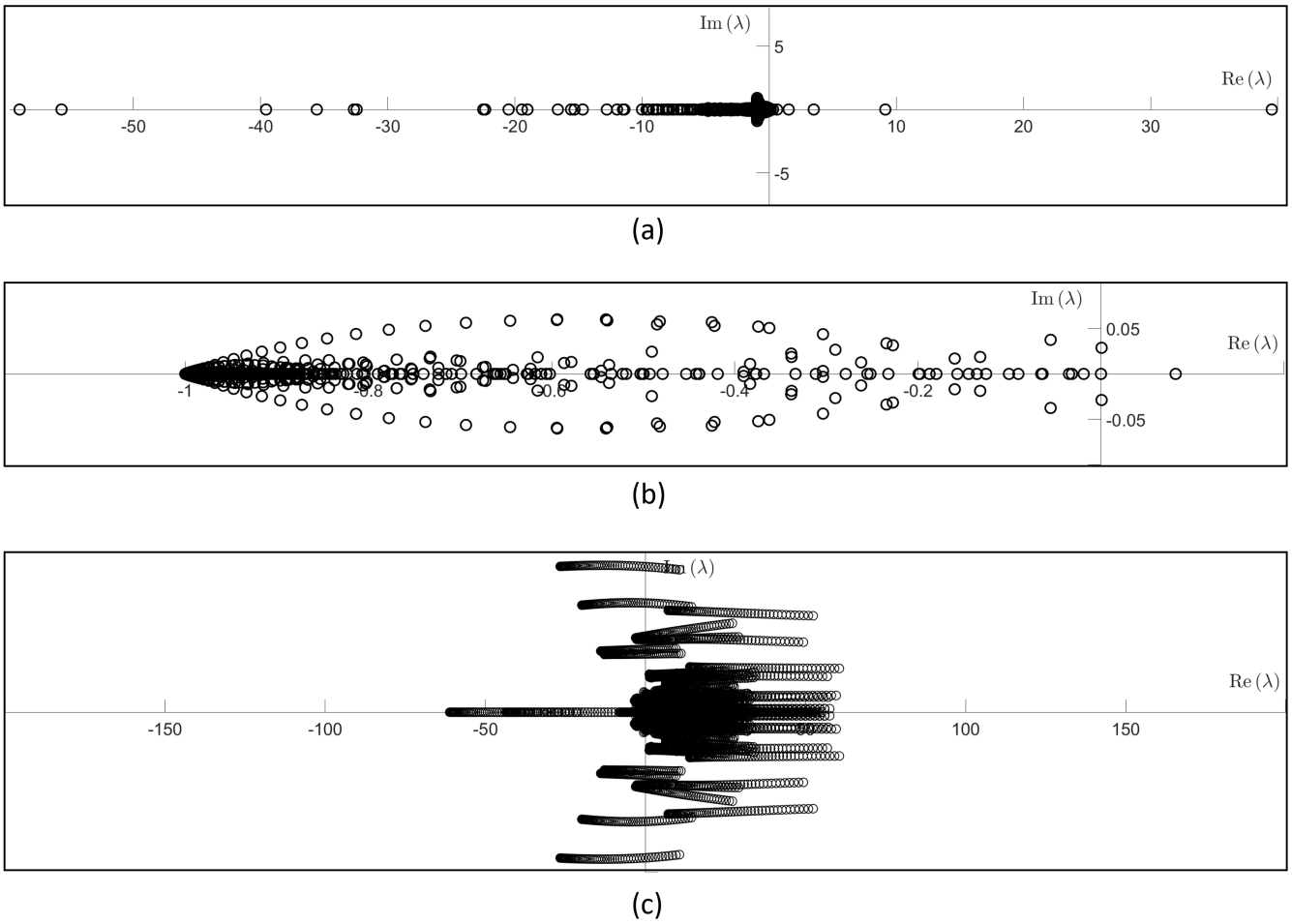

Figure 2: Eigenvalue distributions of the linear operator around a representative equilibrium, for different preconditioning schemes. (a). Inverse Laplacian (scheme 1); (b) Time-integration (scheme 2); (c) No preconditioning. The equilibrium state consists of two counter-rotating vortices in an 80×20 channel at $\mathrm{Ra} = 0.7$. In the truncated Fourier-Chebyshev basis (128 Fourier modes and 32 Chebyshev modes), the linear operator is a 16384×16384 matrix.

Then, in equation (79), we substitute $\mathbf{X}^{(0)}$ for $\mathbf{X}$ and $\mathbf{X}^{(1)}$ for $\mathbf{Y}$, expand to order ϵ, and finally take the limit $\epsilon \to 0$. The resulting expression can be easily evaluated in Dedalus using the field and operator classes; see sections 9.4.1 and 9.4.2.

8.1.3 Preconditioning

The Jacobian from the previous section is not suitable for direct use in Krylov subspace methods because its eigenvalue distribution does not isolate the physically relevant, small wavenumber modes (see figure 2(c)). In other words, it will take many, many iterations for the Krylov subspace to "see" the relevant modes. In addition, the boundary conditions and incompressibility constraint are not automatically satisfied, i.e. $\mathbf{JX}$ does not satisfy the constraints even if $\mathbf{X}$ does. Finally, the repeated application of numerical derivatives raises concerns regarding error accumulation.

The goal of preconditioning is to generate a better-behaved spectrum by multiplying $\mathbf{A}$ with a preconditioning matrix $\mathbf{P}^{-1}$. Instead of solving the linear system $\mathbf{Ax} = \mathbf{b}$, we solve the equivalent equation $\mathbf{P}^{-1}\mathbf{Ax} = \mathbf{P}^{-1}\mathbf{b}$, where $\mathbf{P}^{-1}\mathbf{A}$ hopefully has better spectral properties than $\mathbf{A}$.

In the present context, we want the spectrum of the preconditioned Jacobian $\mathbf{P}^{-1}\mathbf{J}$ to bounded, with the eigenvalues of the high wavenumber modes accumulating (clustering) at a single point, and the eigenvalues of the small wavenumber modes isolated on the periphery.

Preconditioning scheme 1

In the first preconditioning scheme, we take $\mathbf{P}^{-1} = \nabla^{-2}$ defined with respect to homogeneous boundary conditions and divergence free velocity. To evaluate the action of $\nabla^{-2}\mathbf{J}$ on $\mathbf{X}$, we solve the boundary value problem

$$\nabla^2 \mathbf{Q} = x \tag{83}$$

$$\nabla^2 \mathbf{u} + \nabla p = x \tag{84}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{85}$$

$$\mathbf{u} = 0 \text{ and } \mathbf{Q} = 0 \text{ at the boundaries} \tag{86}$$

The inverse Laplacian has a smoothing effect on the unpreconditioned output $\mathbf{J} \cdot \mathbf{X}$, so that the eigenvalues of the high wavenumber modes cluster near -1, as desired. By contrast, we hope that the small wavenumber modes are isolated on the periphery. Figure 2(a) demonstrates this aspect for a representative equilibrium. There are about 20 eigenvalues clearly separated from the rest, and visual inspection of the corresponding eigenvectors indicate that these are indeed the low wavenumber modes.

Preconditioning scheme 2

The second preconditioning scheme is applied indirectly by reformulating the fixed-point equation for equilibria. Rather than solve $\mathbf{F}(\mathbf{X}) = 0$, we integrate the equations of motion over a small, fixed time interval T_0 and look for solutions of

$$\Phi(\mathbf{X}, T_0) - \mathbf{X} = 0 \tag{87}$$

The linear operator then becomes

$$\mathbf{J} = \mathcal{L}\left(\mathbf{X}, T_0\right) - \mathbf{I} \tag{88}$$

where $\mathcal{L}$ is defined as in equation (75).

The dissipative nature of the dynamics suggests that $\mathcal{L}\left(\mathbf{X}, T_0\right)$ goes to 0 asymptotically as the wavenumber of $\mathbf{X}$ increases. Intuitively, the time integration should kill off any large wavenumber perturbations to $\mathbf{X}$, so long as T_0 is not too small. By contrast, we expect that the small wavenumber perturbations are preserved or even amplified by the time integration.

As shown in figure 2(b), direct computation of the spectrum for a representative equilibrium confirms these hypotheses. In practice, the value of T_0 should be large enough to produce the desired smoothing effect, but smaller than the smallest period of any coexisting periodic orbits.

8.1.4 Hookstep

In preconditioning scheme 1, the residual of the nonlinear system can be computed relatively quickly as it does not require any integration in time. For this reason, it is feasible to optimize the hookstep by computing the nonlinear residual for a dense set of trust radii (trial hooksteps) and choosing the global minimum. By contrast, computing the nonlinear residual for periodic orbits requires an expensive time integration, which limits the number of trial hooksteps one can generate.

8.2 Traveling waves

8.2.1 Fixed point equation

A traveling wave is a static flow profile that translates in a fixed direction with constant speed. It is straightforward in principle to describe traveling waves in general geometries, but for simplicity we restrict

attention to a 2D periodic channel, as described in Ref. [8]. In this case, traveling waves can only move in the $\pm x$ directions, parallel to the channel walls.

Suppose such a traveling wave is moving in the positive x direction with speed c, and let $\mathbf{Y}_c(x, y)$ be the (time-independent) field profiles in the moving reference frame. Then, the full, time-dependent state $\mathbf{X}(x, y, t)$ is given by

$$\mathbf{X}(x, y, t) = \mathbf{Y}_c(x - ct, y) \tag{89}$$

The fact that the (x, t) dependence enters only through the combination $w \equiv x - ct$ allows one to eliminate time derivatives and obtain a time-independent fixed point equation in (w, y):

$$-c\frac{\partial \mathbf{Y}_c(w, y)}{\partial w} = \mathbf{F}(\mathbf{Y}_c(w, y)) \tag{90}$$

For active nematics, this results in the following system of equations for the physical fields $\mathbf{u}(w, y)$ and $\mathbf{Q}(w, y)$:

$$\begin{aligned} \mathrm{Re_n}\left(-c\frac{\partial}{\partial w} + \mathbf{u}\cdot\boldsymbol{\nabla}\right)\mathbf{u} &= -\boldsymbol{\nabla}p + 2(\boldsymbol{\nabla}\cdot\mathbf{E}) - \frac{\mathrm{R_a}}{\mathrm{Er}}(\boldsymbol{\nabla}\cdot\mathbf{Q}), \\ \left(-c\frac{\partial}{\partial w} + \mathbf{u}\cdot\boldsymbol{\nabla}\right)\mathbf{Q} + \mathbf{W}\cdot\mathbf{Q} - \mathbf{Q}\cdot\mathbf{W} &= \lambda\mathbf{E} + \mathbf{H}, \\ \boldsymbol{\nabla}\cdot\mathbf{u} &= 0. \end{aligned} \tag{91}$$

8.2.2 Linear operator

The state-dependent linear operator $\mathcal{L}_X$ can be calculated analytically as was done for equilibria in section 8.1.2. Typically, the wave speed c is unknown. Similarly to a periodic orbit with unknown period, c must be included as an independent variable and the linear system extended accordingly:

$$\begin{pmatrix} 0 & \left.\dfrac{\partial(\tau_z\mathbf{Y}_c)}{\partial z}\right|_{z=0} \\ -\dfrac{\partial\mathbf{Y}_c}{\partial w} & \mathcal{L}_X(\mathbf{Y}_c) \end{pmatrix}\begin{pmatrix} \Delta c \\ \delta\mathbf{Y}_c \end{pmatrix} = \begin{pmatrix} 0 \\ c\dfrac{\partial\mathbf{Y}_c}{\partial w} + \mathbf{F}(\mathbf{Y}_c) \end{pmatrix} \tag{92}$$

Here, τ_z denotes translation by z units in the positive x-direction. Therefore, the top row of the system imposes the requirement that the state perturbation $\delta\mathbf{Y}_c$ is orthogonal to a trivial shift of the origin.

8.2.3 Preconditioning

The preconditioning schemes for traveling waves are analogous to the ones used for equilibria. The first applies the same inverse Laplacian as equations (83)–(86). In the second, we again precondition the system indirectly by performing a time integration, but this time we include the wave speed as an extra variable. The implementation is similar to the process for computing relative periodic orbits; see section 8.4 below.

8.3 Periodic orbits

8.3.1 Fixed point equation

The general fixed point equation for POs was already given in equation (73), and its application to active nematic channel flow is straightforward.

8.3.2 Linear operator

Similarly, the linear system for a generic PO is given in equation (77).

8.4 Relative periodic orbits

Relative periodic orbits (RPOs) can occur when a system possesses both periodic orbits and a continuous translational symmetry. For active nematic channel flow (as considered here), the latter is translation in the x (streamwise) coordinate, while in an annulus, it would be translation in the azimuthal (polar) angle.

8.4.1 Fixed point equation

Let τ_z denote translation in the positive x direction by z units. Then, the fixed point equation is

$$\tau_\ell \, \Phi(\mathbf{X}_{\mathrm{RPO}}, T) - \mathbf{X}_{\mathrm{RPO}} = 0 \qquad \text{(relative periodic orbit)}, \tag{93}$$

Here, $\mathbf{X}_{\mathrm{RPO}}$ is a single reference point along the orbit, and ℓ (a constant) is the *shift*.

8.4.2 Linear operator

The linear system for a RPO is a natural generalization of (77) for POs. The main additional ingredient is the shift ℓ, which, like the period T, is usually unknown. Therefore, it must be included as an additional unknown in the state vector, resulting in a $(4N_xN_y + 2)$-dimensional system:

$$L_{\mathrm{RPO}} \begin{pmatrix} \delta T \\ \delta \ell \\ \delta \mathbf{X} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -\tau_{\ell_0} \, \Phi(\mathbf{X}_0, T_0) + \mathbf{X}_0 \end{pmatrix} \tag{94}$$

where

$$L_{\mathrm{RPO}} = \begin{pmatrix} 0 & 0 & \left.\dfrac{\partial \Phi(\mathbf{X}_0, t)}{\partial t}\right|_{t=0} \\ 0 & 0 & \left.\dfrac{\partial \left(\tau_z \mathbf{X}_0\right)}{\partial z}\right|_{z=0} \\ \left.\dfrac{\partial \left(\tau_{\ell_0} \Phi(\mathbf{X}_0, t)\right)}{\partial t}\right|_{t=T_0} & \left.\dfrac{\partial \left(\tau_{z+\ell_0} \Phi(\mathbf{X}_0, T_0)\right)}{\partial z}\right|_{z=0} & \mathcal{L}\left(\mathbf{X}_0, T_0, \ell_0\right) - \mathbf{I} \end{pmatrix} \tag{95}$$

9 Newton-Krylov-Hookstep: Dedalus implementation

The four types of ECS described in section 8 have different linear and nonlinear operators. However, they use the same implementation of the Arnoldi procedure and the GMRES-hookstep – the only exception being the more rigorous hookstep acceptance criteria implemented for equilibria and traveling waves. Hence, subsections 9.2 and 9.3 below discuss the implementation of Arnoldi procedure and GMRES hookstep in general terms, with the subsections thereafter focusing on each ECS type.

9.1 Notation and state representation

To ensure clear translation between mathematical symbols and python code, this section defines a standard notation for representing the state of 2D active nematics.

9.1.1 Math

Recall that section 8 defined a phase space representation of active nematics, in which the equations of motion take the form

$$\partial_t \mathbf{X} = \mathbf{F}(\mathbf{X}) \tag{96}$$

Here, the state is given by $\mathbf{X}$, which is a 4×1 matrix whose elements are *functions* of space and time, corresponding to the four physical field components:

$$\mathbf{X} = \begin{pmatrix} Q_{xx} \\ Q_{xy} \\ U \\ V \end{pmatrix} \tag{97}$$

Boundary conditions and incompressibility are then considered as constraints, which ensure the system is not underdetermined.

These functions also have a Fourier-Chebyshev spectral representation, which on a bounded domain consists of a countably infinite set of spectral coefficients. In numerical applications, this representation must be truncated to a finite number of modes, say N_x Fourier and N_y Chebyshev modes for each field. Therefore, in addition to (97) we will also use the approximate representation of $\mathbf{X}$ as a $4N_xN_y$-dimensional vector, or equivalently, a 4×1 block matrix:

$$\mathbf{X} = \begin{pmatrix} Q_{xx}^{(c)} \\ Q_{xy}^{(c)} \\ U^{(c)} \\ V^{(c)} \end{pmatrix} \tag{98}$$

where $Q_{xx}^{(c)}$ is a vector with N_xN_y components, and so on. In most cases, it will not be necessary to distinguish (97) from (98), so we will not use the (c) superscript elsewhere.

9.1.2 Code: State representation

There are two roughly equivalent representations of $\mathbf{X}$ in Python/Dedalus. In the first, called the *grid space representation*, each of the four components of (97) is represented by a real-valued, two-dimensional, $N_x \times N_y$ `numpy array`, which contains function values at different points on the problem domain. In the second, called the *coefficient space representation*, each component of $\mathbf{X}$ is represented by its truncated Fourier-Chebyshev coefficients (97). By default, Dedalus packages this data in a complex-valued, two-dimensional `numpy array` with shape $\lfloor (N_x/2) \rfloor \times N_y$. However, the most robust implementation of the Newton-Krylov-Hookstep algorithm works with a real-valued vector representation of this data, that is, a real-valued, one-dimensional `numpy array` with N_xN_y elements.

All functions in Newton-GMRES-Hookstep implementation input and output data in the real-valued vector format. The functions `GridToVector` and `VectorToGrid` facilitate the conversion between this format and the corresponding 2D complex-valued `array`:

```
def VectorToGrid(array_in):
        array_out = np.zeros((NXH, NY), dtype = np.complex128)
        for i in range(NXH):
                ih = NXH+i
                array_out[i,:] = array_in[(i*NY):(i+1)*NY] + 1j*array_in[(ih*NY):(ih+1)*NY]
        return array_out

def GridToVector(array_in):
        array_out = np.zeros(NX*NY, dtype = np.float64)
        for i in range(NXH):
                ih = NXH+i
                array_out[(i*NY):(i+1)*NY] = array_in[i,:].real
                array_out[(ih*NY):(ih+1)*NY] = array_in[i,:].imag
        return array_out
```

In Dedalus, the `field` class also allows one to combine the two representations (grid and coefficient space) into a single `field` object. Suppose `A` is a `field` defined on the `domain` object `D`:

```
A = D.new_field()
```

Data can be assigned in grid or coefficient space, as follows:

```
A['g'] = A_data_grid # grid space
A['c'] = A_data_coeff # coefficient space
```

where the right hand side in each case is a 2D numpy array with the appropriate shape and data type. Similarly, data can be read using the following commands:

```
# Grid space
A.require_grid_space()
A_data_grid_output = A.data

# Coefficient space
A.require_coeff_space()
A_data_coeff_output = A.data
```

Note, however, the the coefficient data is always formatted in a complex-valued, two-dimensional `numpy array`, so to convert the real-valued vector representation, we would require

```
A.require_coeff_space()
A_coeffdata_vector = GridToVector(A.data)
```

9.1.3 Code: Notation

Next, we describe the Python/Dedalus notation used for these data representations. Typically, we use the following keywords as the "base" for all associated variable names:

$$\begin{aligned} \texttt{QAA} &\longleftrightarrow Q_{xx} \\ \texttt{QAB} &\longleftrightarrow Q_{xy} \\ \texttt{U} &\longleftrightarrow U \\ \texttt{V} &\longleftrightarrow V \end{aligned} \tag{99}$$

to which prefixes and suffixes may be added. For instance, a `field` object corresponding to Q_{xx} might be called `QAA` or `QAA_field`. The grid and coefficient space representations might then be called `QAA_data_grid` and `QAA_data_coeff`. These are just examples; the name itself as well as the surrounding code are intended to make clear what object type a variable name refers to.

A (code $\longleftrightarrow$ math $\longleftrightarrow$ English) dictionary for a few miscellaneous but important quantities is

$$\texttt{NX} \longleftrightarrow N_x \longleftrightarrow \text{Number of Fourier modes}$$
$$\texttt{NY} \longleftrightarrow N_y \longleftrightarrow \text{Number of Chebyshev modes}$$
$$\texttt{NXH} \longleftrightarrow \lfloor N_x/2 \rfloor \longleftrightarrow \text{Number of complex conjugate pairs in the Fourier expansion}$$
$$\texttt{MY} \longleftrightarrow 4N_xN_y \longleftrightarrow \text{Number of spectral coefficients summed over all four fields}$$
$$\texttt{f or x} \longleftrightarrow \mathbf{X} \longleftrightarrow 4N_xN_y\text{-component array/vector, Fourier-Chebyshev coefficients of all four fields}$$

As a final illustration, the following code reads in real-space (grid) values of Q_{xx}, Q_{xy}, U, and V, and transforms the data into a single, real-valued `numpy array`, named `f`, with $4N_xN_y$ components:

```
QAA_field = domain.new_field()
QAB_field = domain.new_field()
U_field = domain.new_field()
V_field = domain.new_field()

QAA_field['g'] = QAA_data_grid
QAB_field['g'] = QAB_data_grid
U_field['g'] = U_data_grid
V_field['g'] = V_data_grid

QAA_field.require_coeff_space()
QAB_field.require_coeff_space()
U_field.require_coeff_space()
V_field.require_coeff_space()

f = np.zeros(MY)
qaa_begin = 0
qaa_end = qaa_begin + (NX*NY)
qab_begin = qaa_end
qab_end = qab_begin + (NX*NY)
u_begin = qab_end
u_end = u_begin + (NX*NY)
v_begin = u_end
v_end = v_begin + (NX*NY)
f[qaa_begin:qaa_end] = GridToVector(QAA_field.data)
f[qab_begin:qab_end] = GridToVector(QAB_field.data)
f[u_begin:u_end] = GridToVector(U_field.data)
f[v_begin:v_end] = GridToVector(V_field.data)
```

9.2 Arnoldi procedure

The core of the Arnoldi procedure is a function that takes an orthonormal basis for the $(m-1)^{th}$ Krylov subspace as input, and returns a vector that extends the basis to the m^{th} subspace. That is, the input basis elements together with the new vector are an orthonormal basis for the m^{th} subspace. This function is implemented as follows:

```
def arnoldi_iteration(x_base, Q, k:int):
        Qk = applyLinearOperator(x_base, Q[:, k - 1])
        Hk = np.zeros(k+1)
        for j in range(0, k):
                Hk[j] = np.matmul(np.conj(Q[:,j]), Qk)
                Qk = Qk - Hk[j]*Q[:,j]
        Hk[k] = np.linalg.norm(Qk)
        Qk = Qk/Hk[k]

        return Qk, Hk
```

Here, `x_base` is the base state of the linear operator, and is not intrinsic to the Arnoldi procedure. `Q` is an $n \times (k-1)$ matrix whose columns are the orthonormal basis vectors of the $(k-1)^{th}$ Krylov subspace. `Qk` is the "next" basis element for the k^{th} subspace. Prior to orthonormalizaton, it is given by the action of the linear operator on the $k-1$ basis element. Finally, `Hk` is the $k-1$ column of the upper Hessenberg matrix $\mathbf{H}$ from section 7.3.3. The Gram-Schmidt orthonormalization happens in the inner loop (`for j in range(0, k)`).

Applied iteratively, this function can reproduce the full Arnoldi procedure – that is, generate an orthonormal basis for the k^{th} Krylov subspace starting only from the zeroth-order subspace vector, $\mathbf{A}^0\,\mathbf{b} = \mathbf{b}$. For linear stability calculations, it is convenient to package the full Arnoldi procedure in a single function. For periodic orbits, for example, this takes the form

```
def arnoldi(x_base, T, r, n:int):
        Q = np.zeros((r.size, n+1))
        H = np.zeros((n+1, n))
        Q[:,0] = r/np.linalg.norm(r)
        for k in range(1, n + 1):
                Q[:,k] = Dphi_prod(x_base, Q[:, k - 1], T)
                for j in range(0, k):
                        H[j, k-1] = np.matmul(np.conj(Q[:,j]), Q[:,k])
                        Q[:,k] = Q[:,k] - H[j, k-1]*Q[:,j]
                H[k, k-1] = np.linalg.norm(Q[:,k])
                Q[:,k] = Q[:,k]/H[k, k-1]

        return Q, H
```

where `n` is the subspace dimension, `T` is the period, and `Dphi_prod` computes the matrix vector product $\mathcal{L}\cdot\mathbf{X}$ described in section 7.4. In this case, $\mathbf{X}$ corresponds to the column vector `Q[:,k-1]`.

When using GMRES in combination with Newton's method, it is desirable to check the accuracy of the subspace approximation for a range of subspace dimensions, up to some maximum $k_{\max}$. Often, a sufficiently accurate result is found for $k < k_{\max}$. Because Arnoldi generates the subspace basis elements iteratively, it is redundant to execute the full Arnold procedure for each k tested. Therefore, the GMRES Hookstep function discussed in the next section only calls `arnoldi_iteration` externally, while maintaining a local copy of the basis elements.

9.3 GMRES-Hookstep

As described in section 7.3.3, The goal the GMRES-Hookstep algorithm is to approximate the solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ in the m^{th} Krylov subspace $\mathcal{K}_m(\mathbf{A}, \mathbf{b})$ by solving the following constrained least-squares problem:

$$\text{Minimize } ||\mathbf{H}_{m+1,m}\mathbf{y} - r_0\mathbf{e}_1|| \text{ with respect to } \mathbf{y} \text{ and subject to } ||\mathbf{y}|| < \rho$$

Then, $\mathbf{x}_m = \mathbf{V}_m\mathbf{y}$ is the m^{th} order estimate for $\mathbf{x}$. We reiterate the meaning of the symbols, assuming for simplicity that the dimension of the m^{th} Krylov subspace is m.

$$\begin{aligned}
\mathbf{y}: &\quad m \times 1 \text{ matrix} \\
\mathbf{H}_{m+1,m}: &\quad (m+1)\times m \text{ upper Hessenberg matrix, generated by the Arnoldi procedure} \\
\mathbf{V}_m: &\quad n \times m \text{ matrix, whose columns are the othonormal basis vectors for } \mathcal{K}_m \\
\mathbf{e}_1: &\quad \text{the } (m+1)\text{-dimensional unit vector with first element 1 and all others 0} \\
\rho: &\quad \text{trust radius} \\
r_0: &\quad \text{norm of the right-hand-side vector } \mathbf{b}, \text{ i.e. } r_0 \equiv ||\mathbf{b}||
\end{aligned} \tag{100}$$

In ECSAct, this recipe is implemented using a combination of two functions. The first, `Hookstep(H, r_0, m, rho)`, carries out the least squares minimization in $\mathcal{K}_m$, given $\mathbf{H}_{m+1,m}$, r_0, and ρ (= `rho`):

```
def Hookstep(H_, r_0, m, rho):
        e1 = np.zeros(m+1)
        e1[0] = r_0
        def fun(x_, F):
                r = np.matmul(F, x_) + e1
                return np.matmul(r, r)
        def Jacobian(x_, F):
                return 2*np.matmul(np.matmul(np.transpose(F), F), x_) + 2*np.matmul(np.transpose(F), e1)
        def constraint(x_):
                return rho*rho - np.matmul(np.transpose(x_), x_)
        def constraintJac(x_):
                return -2*x_
```

```
    ineq_cons = {'type': 'ineq','fun' : constraint,'jac' : constraintJac}

    y_init = np.zeros(m)
    y_init[0] = 1e-3

    result = scipy.optimize.minimize(fun, y_init, args=(H[0:m+1,0:m]), method='SLSQP', jac = Jacobian,
            constraints=(ineq_cons), options={'ftol': 1e-34, 'disp': False, 'maxiter': 100000000}, bounds=None)
    return result
```

The minimization is done using the `optimize` module in SciPy. The return data type is an `OptimizeResult` object whose attributes include the minimum function value (`fun`) and the function argument at the minimum (`x`).

The second function, `GMRES(x_base, b, kmax, rho)` oversees the construction of the Krylov subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{b})$ by iteratively calling the function `arnoldi_iteration` (section 9.2), up to a fixed maximum subspace dimension `kmax`. It also decides which values to use for the trust radius `kmax`. Given this information (Krylov subspace and trust radius), it calls `Hookstep`, which carries out the least squares minimization and returns a candidate solution $\mathbf{x}_k(\rho)$.

If some set of criteria are met, then `GMRES` exits and returns $\mathbf{x}_{\text{base}} + \mathbf{x}_k(\rho)$. For example, `GMRES` may test the accuracy of $\mathbf{x}_k(\rho)$ by evaluating the residual of the full nonlinear operator: $R = ||\mathbf{F}\left[\mathbf{x}_{\text{base}} + \mathbf{x}_k(\rho)\right]||$. If the criteria are not met, it goes back and reruns the hookstep with new k and ρ, though the details depend on the type of ECS being computed.

For example, the scripts for POs and RPOs use the `GMRES` function shown below. Initially, the function computes the hookstep for all Krylov subspaces from $k = 1$ to $k = k_{\max}$, and singles out the solution with the smallest residual in the linear system. For this single value of k, `GMRES` computes the nonlinear residual and checks whether it improves on the previous Newton iteration, i.e. whether it is less than $||b||$. If it is, then `GMRES` exits. Otherwise, the function passes into a `while` loop that repeats the hookstep with smaller and smaller values of the trust radius (but k fixed at $k_{\max}$), until either the nonlinear residual is less than $0.99||b||$, or the trust radius becomes impractically small, e.g. 1e-10.

```
def GMRES(x_base, b, kmax, rho):
        Q = np.zeros((x_base.size, kmax+1))
        H = np.zeros((kmax+1, kmax))
        r_0 = np.linalg.norm(b)

        linear_res = np.inf
        nonlinear_res = np.inf
        xk = np.zeros(x_base.size)

        Q[:,0] = b/np.linalg.norm(b)
        for k in range(1, kmax):
                Q[:,k], H[:k+1,k-1] = arnoldi_iteration(x_base, Q[:,0:k], k)
                result = Hookstep(H, r_0, k, rho)
                if np.linalg.norm(result.fun) < linear_res:
                        linear_res = np.linalg.norm(result.fun)
                        xk = np.matmul(Q[:,0:k], result.x)
        nonlinear_res = np.linalg.norm(applyNonLinearOperator(np.copy(x_base)+xk))
        if nonlinear_res < r_0:
            return xk, linear_res, rho
        else:
            rho_local = rho
            while nonlinear_res > 0.99*r_0 and rho_local > 1e-10:
                    rho_local = 0.5*rho_local
                    result = Hookstep(H, r_0, kmax - 1, rho_local)
                    xk = np.matmul(Q[:,0:(kmax-1)], result.x)
                    linear_res = result.fun
                    nonlinear_res = np.linalg.norm(applyNonLinearOperator(np.copy(x_base)+xk))
    return xk, linear_res, rho_local
```

9.4 Equilibria

Having discussed the Krylov subspace and hookstep algorithms, we now focus on each of the four ECS types implemented in ECSAct. The main items to focus on are the linear and nonlinear operators for each case. We begin with equilibria.

9.4.1 Nonlinear operator

In math notation, the nonlinear operator for equilibria is the right hand side $\mathbf{F}(\mathbf{X})$ of the equations of motionIn Dedalus, both the state $\mathbf{X}$ and $\mathbf{F}(\mathbf{X})$ can be represented in terms of four `field` objects. For $\mathbf{X}$, we write $\mathbf{X} = (Q_{xx}, Q_{xy}, U, V)^T$ and assign the following labels:

$$\begin{aligned} \texttt{QAA} &\leftarrow Q_{xx} \\ \texttt{QAB} &\leftarrow Q_{xy} \\ \texttt{U} &\leftarrow U \\ \texttt{V} &\leftarrow V \end{aligned}$$

Similarly, for $\mathbf{F}(\mathbf{X})$, we write $\mathbf{F}(\mathbf{X}) = (F_{\mathrm{Qxx}}, F_{\mathrm{Qxy}}, F_{\mathrm{U}}, F_{\mathrm{V}})^T$ and assign `field` objects as follows:

$$\begin{aligned} \texttt{QAA\_op} &\leftarrow F_{\mathrm{Qxx}} \\ \texttt{QAB\_op} &\leftarrow F_{\mathrm{Qxy}} \\ \texttt{U\_op} &\leftarrow F_{\mathrm{U}} \\ \texttt{V\_op} &\leftarrow F_{\mathrm{V}} \end{aligned}$$

If `array_in` is Fourier-Chebyshev coefficient data in the real-valued vector format (section 9.1.2), then the field objects for $\mathbf{X}$ are created as follows:

```
QAA = domain.new_field()
QAB = domain.new_field()
U = domain.new_field()
V = domain.new_field()

QAA['c'] = VectorToGrid(array_in[qaa_begin:qaa_end])
QAB['c'] = VectorToGrid(array_in[qab_begin:qab_end])
U['c'] = VectorToGrid(array_in[u_begin:u_end])
V['c'] = VectorToGrid(array_in[v_begin:v_end])
```

where `qaa_begin`, etc. are shorthand labels for the indices that partition the data into four parts:

```
qaa_begin = 0
qaa_end = qaa_begin + (NX*NY)
qab_begin = qaa_end
qab_end = qab_begin + (NX*NY)
u_begin = qab_end
u_end = u_begin + (NX*NY)
v_begin = u_end
v_end = v_begin + (NX*NY)
```

In Dedalus, we can conveniently calculate $\mathbf{F}(\mathbf{X})$ by treating the `field` objects as algebraic entities that can be differentiated, multiplied, and added. For example, we can create two new fields for the first and second derivatives of Q_{xx} with respect to x:

```
QAAx = QAA.differentiate(x = 1)
QAAxx = QAA.differentiate(x = 2)
```

Suppose we repeat the above lines for all the derivatives appearing in $\mathbf{F}(\mathbf{X})$. Then,

```
QAA_op = (QAAxx+QAAyy+QAA-10*QAA*QAA*QAA-10*QAA*QAB*QAB-QAAx*U-QAAy*V+QAB*Uy-QAB*Vx).evaluate()
QAB_op = (QABxx+QAByy+QAB-10*QAA*QAA*QAB-10*QAB*QAB*QAB-QAA*Uy+QAA*Vx-QABx*U-QABy*V).evaluate()
U_op = (2*Uxx+Uyy+Vxy-RaEr*QAAx-RaEr*QABy-Re*U*Ux-Re*Uy*V).evaluate()
V_op = (2*Vyy+Uxy+Vxx+RaEr*QAAy-RaEr*QABx-Re*U*Vx-Re*V*Vy).evaluate()
```

Technically, an algebraic combination of fields is by default represented by an `operator` object, which differs from a `field` object in that its evaluation is deferred. This feature may be useful in some cases, but for our purposes we immediately transform the `operator` into a `field` using the in-place function `evaluate()`.

Preconditioning is performed using the `LBVP` (Linear Boundary Value Problem) problem and solver classes in Dedalus:

```
problemL = de.LBVP(domainL, variables=['p','U','V','Ux','Uy','Vx','Vy','QAA','QAAx','QAAy','QAB','QABx','QABy'])
problemL.meta[:]['y']['dirichlet'] = True

problemL.parameters['QAA_rhs'] = QAA_op
problemL.parameters['QAB_rhs'] = QAB_op
problemL.parameters['U_rhs'] = U_op
problemL.parameters['V_rhs'] = V_op
problemL.parameters['Re'] = Re

problemL.add_equation("Ux - dx(U) = 0")
problemL.add_equation("Uy - dy(U) = 0")
problemL.add_equation("Vx - dx(V) = 0")
problemL.add_equation("Vy - dy(V) = 0")
problemL.add_equation("QAAx - dx(QAA) = 0")
problemL.add_equation("QAAy - dy(QAA) = 0")
problemL.add_equation("QABx - dx(QAB) = 0")
problemL.add_equation("QABy - dy(QAB) = 0")
problemL.add_equation("dx(QAAx)+dy(QAAy)=QAA_rhs")
problemL.add_equation("dx(QABx)+dy(QABy)=QAB_rhs")
problemL.add_equation("dx(Ux)+dy(Uy)+dx(p)=U_rhs")
problemL.add_equation("dx(Vx)+dy(Vy)+dy(p)=V_rhs")
problemL.add_equation("Ux+Vy=0")
problemL.add_bc("left(U) = 0")
problemL.add_bc("left(V) = 0")
problemL.add_bc("left(QAA) = 0")
problemL.add_bc("left(QAB) = 0")
problemL.add_bc("right(U) = 0")
problemL.add_bc("right(V) = 0", condition="(nx != 0)")
problemL.add_bc("left(p) = 0", condition="(nx == 0)")
problemL.add_bc("right(QAA) = 0")
problemL.add_bc("right(QAB) = 0")

solverL = problemL.build_solver()
solverL.solve()
```

Finally, we need to convert the preconditioned output into the standard, real-valued vector format:

```
QAA_out = solverL.state['QAA']
QAB_out = solverL.state['QAB']
U_out = solverL.state['U']
V_out = solverL.state['V']

QAA_out.require_coeff_space()
QAB_out.require_coeff_space()
U_out.require_coeff_space()
V_out.require_coeff_space()

array_out = np.zeros(MY)
array_out[qaa_begin:qaa_end] = GridToVector(np.copy(QAA_out.data))
array_out[qab_begin:qab_end] = GridToVector(np.copy(QAB_out.data))
array_out[u_begin:u_end] = GridToVector(np.copy(U_out.data))
array_out[v_begin:v_end] = GridToVector(np.copy(V_out.data))
return array_out
```

9.4.2 Linear operator

Evaluation of the linear operator follows the same template as for the nonlinear operator. The main difference is that both the base state and perturbation are involved. The `field` objects constituting the perturbation retain the default naming `QAA, QAB, U, V`, whereas the base state `fields` are called `QAAeq, QABeq, Ueq, Veq`.

Like the nonlinear operator, the linear operator is evaluated using its analytical expression:

```
QAA_op = (-(QAAxx+QAAyy+QAA-30*QAA*QAAeq*QAAeq-10*QAA*QABeq*QABeq-20*QAAeq*QAB*QABeq-QAAeqx*U
           -QAAeqy*V-QAAx*Ueq-QAAy*Veq+QAB*Ueqy-QAB*Veqx+QABeq*Uy-QABeq*Vx)).evaluate()
QAB_op = (-(QABxx+QAByy+QAB-20*QAA*QAAeq*QABeq-10*QAAeq*QAAeq*QAB-30*QAB*QABeq*QABeq-QAA*Ueqy
           +QAA*Veqx-QAAeq*Uy+QAAeq*Vx-QABeqx*U-QABeqy*V-QABx*Ueq-QABy*Veq)).evaluate()
U_op = (-(+2*Uxx+Uyy+Vxy-RaEr*QAAx-RaEr*QABy-Re*U*Ueqx-Re*Ueq*Ux-Re*Ueqy*V-Re*Uy*Veq)).evaluate()
V_op = (-(Uxy+Vxx+2*Vyy+RaEr*QAAy-RaEr*QABx-Re*U*Veqx-Re*Ueq*Vx-Re*V*Veqy-Re*Veq*Vy)).evaluate()
```

9.4.3 Built-in Dedalus solver for 1D problems

Dedalus has a built-in solver for nonlinear boundary value problems (NLBVPs): `dedalus.core.solvers.NonlinearBoundaryValueSolver`. ECSAct does not make heavy use of this solver because it only implements the elementary version of Newton's method (without the hookstep or Krylov subspace approximations), and is restricted to problems in one spatial dimension. Nevertheless, it is fast and can be useful for computing equilibria that depend only on the wall-normal coordinate y.

ECSAct implements this solver in the script `nematics2__NLBVP.py`. The setup is simple. Many of the user-specified parameters are the same as the other ECS solvers. On the other hand, any parameters involving the streamwise x coordinate are dropped, and there is no hookstep or Krylov subspace dimension. There are also two new parameters: `tolerance` and `damping`, discussed below.

Restriction to the wall-normal coordinate y leads to the following code for creating and defining the associated NLBVP `problem` object:

```
problem = de.NLBVP(domain, variables=['QAA','QAAy','QAB','QABy','U','Uy'])
problem.meta[:]['y']['dirichlet'] = True
problem.parameters['RaEr'] = Ra/Er
problem.parameters['Re'] = Re
problem.parameters['lamb'] = lamb
problem.parameters['Erinv'] = 1/Er
problem.add_equation("QAAy - dy(QAA) = 0")
problem.add_equation("QABy - dy(QAB) = 0")
problem.add_equation("Uy - dy(U) = 0")
problem.add_equation("-dy(QAAy)-QAA=-10*QAA*QAA*QAA-10*QAA*QAB*QAB+QAB*Uy")
problem.add_equation("-dy(QABy)-QAB-0.5*lamb*Uy=-10*QAA*QAA*QAB-10*QAB*QAB*QAB-QAA*Uy")
problem.add_equation("-dy(Uy)+RaEr*QABy=0")
problem.add_bc("left(U) = 0")
problem.add_bc("left(QAA) = -0.5")
problem.add_bc("left(QAB) = 0")
problem.add_bc("right(U) = 0")
problem.add_bc("right(QAA) = -0.5")
problem.add_bc("right(QAB) = 0")
```

The 1D reduction involves (1) dropping any x-derivatives, (2) setting the y-velocity component `V` to 0, (3) dropping the dynamical equation for `V`, (4) dropping the divergence-free constraint. This reduction is also valid also for 1D time-dependent problems.

To understand the steps in the reduction, note that if $\mathbf{u} = (U, V)$ depends only on y and satisfies $\nabla \cdot \mathbf{u} = 0$, then V is necessarily 0. If $V = 0$, then the dynamical equation for V becomes a constraint (no time-derivatives), equivalent to a first-order ODE for the pressure $p(y, t)$ at time t. We can therefore drop this equation from the problem specification, knowing that we can always find $p(y, t)$ to satisfy it. In code notation, there are six remaining problem variables: `QAA`, `QAAy`, `QAB`, `QABy`, `U`, `Uy`. For the NLBVP solver, of course, there is also no time dependence.

Creation and initialization of the solver object looks the same as for other Dedalus solvers:

```
solver = problem.build_solver()
QAA = solver.state['QAA']
QAB = solver.state['QAB']
U = solver.state['U']
QAA['c'] = np.copy(QAA_data_init_coeff)
QAB['c'] = np.copy(QAB_data_init_coeff)
U['c'] = np.copy(U_data_init_coeff)
```

Here, of course, *initialization* means something different from the time-dependent solver. Rather than specifying the initial condition, we are giving Dedalus the initial guess to use with Newton's method. Finally, the following commands run the solver:

```
pert = solver.perturbations.data
pert.fill(1+tolerance)
while np.sum(np.abs(pert)) > tolerance:
    solver.newton_iteration(damping)
    print("Perturbation norm = " + str(np.sum(np.abs(pert))))
```

Each iteration of the `while` loop corresponds to a single iteration of Newton's method. The parameter `damping` is a user-specified and allows for damped Newton iterations, as explained in section 7.2. There is

usually little benefit to changing it from the default of 1. Next, `tolerance` is a user-specified parameter that is usually set near machine precision, e.g. 10^{-14}. It can be used to track convergence and set a stopping criterion. In the above code, the solver is terminated once the residual `np.sum(np.abs(pert))` is less than `tolerance`. This residual corresponds to a preconditioned version of the fixed point equation and may be much smaller than the direct residual. To verify convergence, the script also computes the direct residual as follows (lines xx-xx):

```
QAA_nonlinear_op = (QAAyy+QAA-10*QAA*QAA*QAA-10*QAA*QAB*QAB+QAB*Uy).evaluate()
QAB_nonlinear_op = (QAByy+QAB-10*QAA*QAA*QAB-10*QAB*QAB*QAB-QAA*Uy).evaluate()
U_nonlinear_op = (-Uyy+(Ra/Er)*QABy).evaluate()

QAA_nonlinear_op.require_coeff_space()
QAB_nonlinear_op.require_coeff_space()
U_nonlinear_op.require_coeff_space()

direct_residual = np.linalg.norm(QAA_nonlinear_op.data)
                  + np.linalg.norm(QAB_nonlinear_op.data)
                  + np.linalg.norm(U_nonlinear_op.data)
```

where the y-derivatives were computed on lines xx-xx. As an example, consider the unidirectional state at the following parameter set:

```
Ra = 1.0
Er = 1.0
Re = 0.0136
height = 20.0
width = 80.0
lambd = 0
tolerance = 5e-15
damping = 1.0
```

With resolution $N_y = 32$, the direct residual is 0.00035, even though `tolerance` is set to 5e-15. However, using the finer grid $N_y = 64$ we find a direct residual of 1.8e-08, and with $N_y = 128$ this decreases further to 6.6e-12. This behavior with respect to increasing grid resolution is typical of both 1D and 2D equilibria, and reflects the fact that the unpreconditioned operator is more sensitive to errors in the large wavenumber modes.

9.5 Traveling waves

Traveling waves are handled similarly to equilibria, the main differences being the following:

1. To include the unknown wave speed c, the dimension of the linear system is increased by 1.
2. The linear and nonlinear operators receive extra terms resulting from a coordinate transformation $w \equiv x - c\,t$ to the comoving frame.
3. An additional, c-dependent term is added to the preconditioning operator.

9.5.1 Nonlinear operator

The nonlinear operator function, called `applyNonLinearOperator` takes an $(4N_xN_y + 1) = (M_y + 1)$-component, real-valued array as input. We will call it `array_in` The first element is the wave speed divided `c` by a normalizing factor `cfac`. The remaining elements contain the coefficient data for the physical fields. We will call this M_y-component array `array_temp`. Then, the data input sequence starts as

```
x_basisL = de.Fourier('x', NX, interval=(-0.5*width, 0.5*width), dealias = dealias_fac)
y_basisL = de.Chebyshev('y', NY, interval=(0, height), dealias = dealias_fac)
domainL = de.Domain([x_basisL, y_basisL], grid_dtype=np.float64)

c = cfac*array_in[0]
array_temp = np.copy(array_in[1:])

QAA = domainL.new_field()
QAA['c'] = VectorToGrid(array_temp[qaa_begin:qaa_end])
QAAx = QAA.differentiate(x = 1)
```

with the remaining fields and derivatives computed in the same way as `QAA` and `QAAx`.

The dimension of the nonlinear operator itself is also $M_y + 1$. The first component is exactly 0 (c.f. equation 92) and the remaining M_y components (before preconditioning) are encapsulated in the following `field` objects:

```
QAA_op = (c*QAAx+QAAxx+QAAyy+QAA-10*QAA*QAA*QAA-10*QAA*QAB*QAB-QAAx*U-QAAy*V+QAB*Uy-QAB*Vx).evaluate()
QAB_op = (c*QABx+QABxx+QAByy+QAB-10*QAA*QAA*QAB-10*QAB*QAB*QAB-QAA*Uy+QAA*Vx-QABx*U-QABy*V).evaluate()
U_op = (c*Re*Ux+2*Uxx+Uyy+Vxy-RaEr*QAAx-RaEr*QABy-Re*U*Ux-Re*Uy*V).evaluate()
V_op = (c*Re*Vx+2*Vyy+Uxy+Vxx+RaEr*QAAy-RaEr*QABx-Re*U*Vx-Re*V*Vy).evaluate()
```

Preconditioning and output formatting are largely the same as for equilibria, with the exception of a c-dependent term in the preconditioning operator:

```
problemL = de.LBVP(domainL, variables=['p','U','V','Ux','Uy','Vx','Vy','QAA','QAAx','QAAy','QAB','QABx','QABy'])
problemL.meta[:]['y']['dirichlet'] = True

problemL.parameters['QAA_rhs'] = QAA_op
problemL.parameters['QAB_rhs'] = QAB_op
problemL.parameters['U_rhs'] = U_op
problemL.parameters['V_rhs'] = V_op
problemL.parameters['c'] = c
problemL.parameters['Re'] = Re

problemL.add_equation("Ux - dx(U) = 0")
problemL.add_equation("Uy - dy(U) = 0")
problemL.add_equation("Vx - dx(V) = 0")
problemL.add_equation("Vy - dy(V) = 0")
problemL.add_equation("QAAx - dx(QAA) = 0")
problemL.add_equation("QAAy - dy(QAA) = 0")
problemL.add_equation("QABx - dx(QAB) = 0")
problemL.add_equation("QABy - dy(QAB) = 0")
problemL.add_equation("dx(QAAx)+dy(QAAy)+c*QAAx=QAA_rhs")
problemL.add_equation("dx(QABx)+dy(QABy)+c*QABx=QAB_rhs")
problemL.add_equation("dx(Ux)+dy(Uy)+dx(p)+c*Re*Ux=U_rhs")
problemL.add_equation("dx(Vx)+dy(Vy)+dy(p)+c*Re*Vx=V_rhs")
problemL.add_equation("Ux+Vy=0")
problemL.add_bc("left(U) = 0")
problemL.add_bc("left(V) = 0")
problemL.add_bc("left(QAA) = 0")
problemL.add_bc("left(QAB) = 0")
problemL.add_bc("right(U) = 0")
problemL.add_bc("right(V) = 0", condition="(nx != 0)")
problemL.add_bc("left(p) = 0", condition="(nx == 0)")
problemL.add_bc("right(QAA) = 0")
problemL.add_bc("right(QAB) = 0")

solverL = problemL.build_solver()
solverL.solve()

QAA = solverL.state['QAA']
QAB = solverL.state['QAB']
U = solverL.state['U']
V = solverL.state['V']

QAA.require_coeff_space()
QAB.require_coeff_space()
U.require_coeff_space()
V.require_coeff_space()

array_out = np.zeros(MY+1)
array_out[(qaa_begin+1):(qaa_end+1)] = GridToVector(np.copy(QAA.data))
array_out[(qab_begin+1):(qab_end+1)] = GridToVector(np.copy(QAB.data))
array_out[(u_begin+1):(u_end+1)] = GridToVector(np.copy(U.data))
array_out[(v_begin+1):(v_end+1)] = GridToVector(np.copy(V.data))
return array_out
```

9.5.2 Linear operator

Computation of the linear operator follows essentially the same format as the nonlinear operator, with the corresponding mathematical expression given in equation (92) as a 2×2 block matrix. The bottom row

of this block matrix is easy to compute directly using the Dedalus field and operator classes. The top row requires evaluation of $\frac{\partial(\tau_z \mathbf{Y}_c)}{\partial z}\Big|_{z=0}$, which we accomplish using the following functions:

```
def TransformG(array_in, d):
        data_temp_c = np.copy(array_in)
        for i in range(NXH):
                data_temp_c[i,:] = np.exp(2*1j*i*np.pi*(d/width))*data_temp_c[i,:]
        return GridToVector(data_temp_c)

def dTransform(array_in):
        array_temp = np.copy(array_in)

        QAA_data_temp = VectorToGrid(array_temp[qaa_begin:qaa_end])
        QAB_data_temp = VectorToGrid(array_temp[qab_begin:qab_end])
        U_data_temp = VectorToGrid(array_temp[u_begin:u_end])
        V_data_temp = VectorToGrid(array_temp[v_begin:v_end])

        array_out = np.zeros(MY)
        array_out[qaa_begin:qaa_end] = (TransformG(QAA_data_temp, d_tol_trans) - array_temp[qaa_begin:qaa_end])/d_tol_trans
        array_out[qab_begin:qab_end] = (TransformG(QAB_data_temp, d_tol_trans) - array_temp[qab_begin:qab_end])/d_tol_trans
        array_out[u_begin:u_end] = (TransformG(U_data_temp, d_tol_trans) - array_temp[u_begin:u_end])/d_tol_trans
        array_out[v_begin:v_end] = (TransformG(V_data_temp, d_tol_trans) - array_temp[v_begin:v_end])/d_tol_trans
        return array_out
```

where `array_in` is a real-valued array with $4N_xN_y$ components, and defines the state vector $\mathbf{Y}_c$.

In summary, if the preconditioned output of the linear operator is contained in the fields `QAA, QAB, U, V`, then the final output sequence looks like the following:

```
        array_out = np.zeros(MY+1)
        array_out[0] = np.matmul(np.conj(dTransform(x_base)), delta_x)
        array_out[(qaa_begin+1):(qaa_end+1)] = GridToVector(np.copy(QAA.data))
        array_out[(qab_begin+1):(qab_end+1)] = GridToVector(np.copy(QAB.data))
        array_out[(u_begin+1):(u_end+1)] = GridToVector(np.copy(U.data))
        array_out[(v_begin+1):(v_end+1)] = GridToVector(np.copy(V.data))
        return array_out
```

9.6 Periodic orbits

The core ingredient of the linear/nonlinear operator implementation for periodic orbits (POs) is the flow map $\phi(\mathbf{X}, T)$:

```
def phi(array_in, T):
        x_basis_phi = de.Fourier('x', NX, interval=(-0.5*width, 0.5*width), dealias = dealias_fac)
        y_basis_phi = de.Chebyshev('y', NY, interval=(0, height), dealias = dealias_fac)
        domain_phi = de.Domain([x_basis_phi, y_basis_phi], grid_dtype = np.float64)
        problem_phi = de.IVP(domain_phi, variables=['p','U','V','Ux','Uy','Vx','Vy','QAA','QAAx','QAAy','QAB','QABx','QABy'])
        problem_phi.meta[:]['y']['dirichlet'] = True
        problem_phi.parameters['width'] = width
        problem_phi.parameters['height'] = height
        problem_phi.parameters['RaEr'] = Ra/Er
        problem_phi.parameters['Re'] = Re
        problem_phi.add_equation("Ux - dx(U) = 0")
        problem_phi.add_equation("Uy - dy(U) = 0")
        problem_phi.add_equation("Vx - dx(V) = 0")
        problem_phi.add_equation("Vy - dy(V) = 0")
        problem_phi.add_equation("QAAx - dx(QAA) = 0")
        problem_phi.add_equation("QAAy - dy(QAA) = 0")
        problem_phi.add_equation("QABx - dx(QAB) = 0")
        problem_phi.add_equation("QABy - dy(QAB) = 0")
        problem_phi.add_equation("dt(QAA)-dx(QAAx)-dy(QAAy)-QAA=-10*QAA*QAA*QAA-10*QAA*QAB*QAB-QAAx*U-QAAy*V+QAB*Uy-QAB*Vx")
        problem_phi.add_equation("dt(QAB)-dx(QABx)-dy(QABy)-QAB=-10*QAA*QAA*QAB-10*QAB*QAB*QAB-QAA*Uy+QAA*Vx-QABx*U-QABy*V")
        problem_phi.add_equation("Re*dt(U)+dx(p)-2*dx(Ux)-dy(Uy)-dx(Vy)+RaEr*QAAx+RaEr*QABy=-Re*U*Ux-Re*Uy*V")
        problem_phi.add_equation("Re*dt(V)+dy(p)-2*dy(Vy)-dx(Uy)-dx(Vx)-RaEr*QAAy+RaEr*QABx=-Re*U*Vx-Re*V*Vy")
        problem_phi.add_equation("Ux + Vy = 0")
        problem_phi.add_bc("left(U) = 0")
        problem_phi.add_bc("left(V) = 0")
        problem_phi.add_bc("left(QAA) = -0.5")
        problem_phi.add_bc("left(QAB) = 0")
        problem_phi.add_bc("right(U) = 0")
```

```
problem_phi.add_bc("right(V) = 0", condition="(nx != 0)")
problem_phi.add_bc("left(p) = 0", condition="(nx == 0)")
problem_phi.add_bc("right(QAA) = -0.5")
problem_phi.add_bc("right(QAB) = 0")
solver_phi = problem_phi.build_solver(timestepper)

QAA = solver_phi.state['QAA']
QAB = solver_phi.state['QAB']
U = solver_phi.state['U']
V = solver_phi.state['V']

QAA['c'] = VectorToGrid(array_in[qaa_begin:qaa_end])
QAB['c'] = VectorToGrid(array_in[qab_begin:qab_end])
U['c'] = VectorToGrid(array_in[u_begin:u_end])
V['c'] = VectorToGrid(array_in[v_begin:v_end])

num_steps = int(T/dt_nominal)
dt = T/num_steps
for i in range(num_steps):
        solver_phi.step(dt)

array_out = np.zeros(MY)
array_out[qaa_begin:qaa_end] = np.copy(QAA.data)
array_out[qab_begin:qab_end] = np.copy(QAB.data)
array_out[u_begin:u_end] = np.copy(U.data)
array_out[v_begin:v_end] = np.copy(V.data)
return array_out
```

Note that the actual timestep `dt` is slightly different from the nominal timestep `dt_nominal`. This is done so that an integer number of timesteps exactly add up to `T`.

Important note: In a single Newton iteration, the quantity $\phi(\mathbf{X}_0, T_0) \longleftrightarrow$ `phi(x_base,T0)` is used more than once. Rather than recalculate it each time, we compute it once in the main Newton solver loop and reuse it as a global variable called `phi_base`. For clarity, however, some of the code excerpts below use `phi(x_base, T_0)` instead of `phi_base`, though the actual code uses the latter.

9.6.1 Nonlinear operator

The nonlinear operator has a simple expression in terms of `phi`:

```
def applyNonLinearOperator(array_in):
        array_out = np.zeros(MY+1)
        array_out[0] = 0
        array_out[1:] = -phi(array_in[1:], (array_in[0]*tfac)) + array_in[1:]
        return array_out
```

This function should be compared with the right hand side of equation (77). The first element of `array_in` is equal to the period `T` divided by a normalization factor `tfac`, which is set earlier in the code so that `array_in[0]` is $\mathcal{O}(1)$.

9.6.2 Linear operator

The task is to evaluate the matrix in the linear system from section 7.4:

$$\begin{pmatrix} 0 & \left.\dfrac{\partial \Phi(\mathbf{X}_0, t)}{\partial t}\right|_{t=0} \\ \left.\dfrac{\partial \Phi(\mathbf{X}_0, t)}{\partial t}\right|_{t=T_0} & \mathcal{L}\left(\mathbf{X}_0, T_0\right) - \mathbf{I} \end{pmatrix} \begin{pmatrix} \delta T \\ \delta \mathbf{X} \end{pmatrix} = \begin{pmatrix} 0 \\ -\Phi(\mathbf{X}_0, T_0) + \mathbf{X}_0 \end{pmatrix} \tag{101}$$

We use a finite-difference approximation to compute the time-derivatives. These are formally equivalent to the right-hand-side of the equations of motion, so the corresponding Python function is called `RHS`. The single argument of the function is an array containing the data for $\Phi(\mathbf{X}_0, t_0)$, where t_0 is the time at which the derivative should be evaluated.

```
def RHS(array_in):
        x_basis_phi = de.Fourier('x', NX, interval = (-0.5*width, 0.5*width), dealias = dealias_fac)
        y_basis_phi = de.Chebyshev('y', NY, interval = (0, height), dealias = dealias_fac)
        domain_phi = de.Domain([x_basis_phi, y_basis_phi], grid_dtype = np.float64)
        problem_phi = de.IVP(domain_phi, variables=['p','U','V','Ux','Uy','Vx','Vy','QAA','QAAx','QAAy','QAB','QABx','QABy'])
        problem_phi.meta[:]['y']['dirichlet'] = True
        problem_phi.parameters['width'] = width
        problem_phi.parameters['height'] = height
        problem_phi.parameters['RaEr'] = Ra/Er
        problem_phi.parameters['Re'] = Re
        problem_phi.add_equation("Ux - dx(U) = 0")
        problem_phi.add_equation("Uy - dy(U) = 0")
        problem_phi.add_equation("Vx - dx(V) = 0")
        problem_phi.add_equation("Vy - dy(V) = 0")
        problem_phi.add_equation("QAAx - dx(QAA) = 0")
        problem_phi.add_equation("QAAy - dy(QAA) = 0")
        problem_phi.add_equation("QABx - dx(QAB) = 0")
        problem_phi.add_equation("QABy - dy(QAB) = 0")
        problem_phi.add_equation("dt(QAA)-dx(QAAx)-dy(QAAy)-QAA=-10*QAA*QAA*QAA-10*QAA*QAB*QAB-QAAx*U-QAAy*V+QAB*Uy-QAB*Vx")
        problem_phi.add_equation("dt(QAB)-dx(QABx)-dy(QABy)-QAB=-10*QAA*QAA*QAB-10*QAB*QAB*QAB-QAA*Uy+QAA*Vx-QABx*U-QABy*V")
        problem_phi.add_equation("Re*dt(U)+dx(p)-2*dx(Ux)-dy(Uy)-dx(Vy)+RaEr*QAAx+RaEr*QABy=-Re*U*Ux-Re*Uy*V")
        problem_phi.add_equation("Re*dt(V)+dy(p)-2*dy(Vy)-dx(Uy)-dx(Vx)-RaEr*QAAy+RaEr*QABx=-Re*U*Vx-Re*V*Vy")
        problem_phi.add_equation("Ux + Vy = 0")
        problem_phi.add_bc("left(U) = 0")
        problem_phi.add_bc("left(V) = 0")
        problem_phi.add_bc("left(QAA) = -0.5")
        problem_phi.add_bc("left(QAB) = 0")
        problem_phi.add_bc("right(U) = 0")
        problem_phi.add_bc("right(V) = 0", condition="(nx != 0)")
        problem_phi.add_bc("left(p) = 0", condition="(nx == 0)")
        problem_phi.add_bc("right(QAA) = -0.5")
        problem_phi.add_bc("right(QAB) = 0")
        solver_phi = problem_phi.build_solver(de.timesteppers.SBDF1)

        QAA = solver_phi.state['QAA']
        QAB = solver_phi.state['QAB']
        U = solver_phi.state['U']
        V = solver_phi.state['V']

        QAA['c'] = VectorToGrid(array_in[qaa_begin:qaa_end])
        QAB['c'] = VectorToGrid(array_in[qab_begin:qab_end])
        U['c'] = VectorToGrid(array_in[u_begin:u_end])
        V['c'] = VectorToGrid(array_in[v_begin:v_end])

        QAA_data_initial = np.copy(QAA.data)
        QAB_data_initial = np.copy(QAB.data)
        U_data_initial = np.copy(U.data)
        V_data_initial = np.copy(V.data)

        for i in range(n_timesteps):
                solver_phi.step(delta)

        array_out = np.zeros(MY)
        array_out[qaa_begin:qaa_end] = GridToVector((np.copy(QAA.data) - QAA_data_initial)/(n_timesteps*delta))
        array_out[qab_begin:qab_end] = GridToVector((np.copy(QAB.data) - QAB_data_initial)/(n_timesteps*delta))
        array_out[u_begin:u_end] = GridToVector((np.copy(U.data) - U_data_initial)/(n_timesteps*delta))
        array_out[v_begin:v_end] = GridToVector((np.copy(V.data) - V_data_initial)/(n_timesteps*delta))
        return array_out
```

Here, `delta` is the nominally small timestep, 1e-7 by default. A total of `n_timesteps` ($= 8$ by default) are carried out to further condition the output. In terms of `RHS`, the time derivatives in (101) are calculated as

$$\texttt{RHS(x\_base)} \longleftrightarrow \left.\frac{\partial \Phi(\mathbf{X}_0, t)}{\partial t}\right|_{t=0}$$

$$\texttt{RHS(phi(x\_base, T\_0))} \longleftrightarrow \left.\frac{\partial \Phi(\mathbf{X}_0, t)}{\partial t}\right|_{t=T_0}$$

where $\mathbf{X}_0$ is represented by `x_base`.

Next, we need the matrix-vector product $\mathcal{L}(\mathbf{X}_0, T_0) \cdot \delta\mathbf{X}$. Recall equation (75):

$$\mathcal{L}(\mathbf{X}, T) \cdot \delta\mathbf{X} = \lim_{\epsilon \to 0} \frac{\Phi\left(\mathbf{X} + \epsilon \frac{\delta\mathbf{X}}{||\delta\mathbf{X}||}, T\right) - \Phi(\mathbf{X}, T)}{\epsilon / ||\delta\mathbf{X}||} \tag{102}$$

The function `Dphi_prod` uses this expression to compute $\mathcal{L}(\mathbf{X}_0, T_0) \cdot \delta\mathbf{X}$:

```
def Dphi_prod(array_base, array_pert, T):
        if np.linalg.norm(array_pert) == 0:
                return np.zeros(MY)
        else:
                epsilon = d_tol/np.linalg.norm(array_pert)
        array_init = array_base + epsilon*array_pert
        array_final = phi(array_init, T)
        array_final_base = np.copy(phi_base) # = phi(array_base, T)
        array_out = (array_final-array_final_base)/epsilon

        return np.copy(array_out)
```

where `d_tol` takes the default value 1e-7. Putting the pieces together, the full linear operator can be computed using the following function:

```
def applyLinearOperator(array_base, array_pert):
        T0 = array_base[0]*tfac
        delta_T = array_pert[0]*tfac

        x_base = np.copy(array_base[1:])
        delta_x = np.copy(array_pert[1:])

        array_out = np.zeros(MY+1)
        array_out[0] = np.matmul(np.conj(RHS(x_base)), delta_x)
        array_out[1:] = Dphi_prod(x_base, delta_x, T0) - delta_x + RHS(np.copy(phi_base))*delta_T
        return array_out
```

9.7 Relative periodic orbits

The linear and nonlinear operators for RPOs are essentially the same as for POs, but with the shift ℓ as an additional variable. Therefore, we need a function that translates a state by a certain amount d.

```
def TransformG(array_in, d):
        data_temp_c = np.copy(array_in)
        for i in range(NXH):
                data_temp_c[i,:] = np.exp(2*1j*i*np.pi*(d/width))*data_temp_c[i,:]
        return GridToVector(data_temp_c)
```

which shifts a single field by d units, not the entire state vector, which consists of four independent fields. The input data format is a 2D, complex-valued array, containing the coefficient space representation of the field. The output format is the 1D, real-valued array representation.

10 Linear stability

Computing the linear stability of ECS is closely related to solving the linear system during an iteration of Newton's method, as both involve the linear operator with respect to a base state. In particular, linear stability is characterized by the spectrum of the linear operator with respect to a base state that lies on the ECS. As in Newton's method, the high dimensionality of the matrix representation of the linear operator calls for Krylov subspace methods.

10.1 General formulation

Suppose $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{b}$ is an n-dimensional vector, and consider the Krylov subspace $\mathcal{K}_m(\mathbf{A}, \mathbf{b})$:

$$\mathcal{K}_m(\mathbf{A}, \mathbf{b}) = \text{span}\{\mathbf{b}, \mathbf{A}\mathbf{b}, \ldots, \mathbf{A}^{m-1}\mathbf{b}\} \tag{103}$$

The main idea of Krylov subspace methods is that projecting an n-dimensional vector $\mathbf{x}$ onto $\mathcal{K}_m$ concentrates the most important information in $\mathbf{x}$ into a low-dimensional representation, namely an m-dimensional vector $\mathbf{y}$. The explicit connection is expressed in terms of the $m \times n$ matrix $\mathbf{V}_m^\dagger$, defined in equation 67, which is a (pseudo-) projection of $\mathbb{R}^n$ onto $\mathcal{K}_m$:

$$\mathbf{y} = \mathbf{V}_m^\dagger \mathbf{x} \tag{104}$$

The same idea applies to an $n \times n$ matrix $\mathbf{B}$ – namely, the $m \times m$ dimensional projection of $\mathbf{B}$ onto $\mathcal{K}_m$ in some sense preserves the most important information contained in $\mathbf{B}$. The explicit connection is again provided by the matrix $\mathbf{V}_m^\dagger$. If $\mathbf{C}_{m,m}$ is the projection of $\mathbf{B}$ onto $\mathcal{K}_m$, then

$$\mathbf{C}_{m,m} = \mathbf{V}_m^\dagger \mathbf{B} \mathbf{V}_m \tag{105}$$

For the special case $\mathbf{B} = \mathbf{A}$, the projection is an upper-Hessenberg matrix $\mathbf{H}_{m,m}$:

$$\mathbf{H}_{m,m} = \mathbf{V}_m^\dagger \mathbf{A} \mathbf{V}_m \tag{106}$$
$$[\mathbf{V}_m^\dagger \text{ defined with respect to Krylov subspace } \mathcal{K}_m(\mathbf{A}, \mathbf{b})]$$

The eigenvalues of $\mathbf{H}_{m,m}$ are called Ritz values and feature in many discussions of Krylov subspace methods. It is reasonable to expect that they approximate the first $p < m$ eigenvalues of $\mathbf{A}$, ordered by decreasing magnitude. Indeed, there are rigorous theorems expressing this connection, though detailed analysis of the convergence behavior is complicated for non-normal matrices.

This approach does not work for non-extremal eigenvalues, e.g. the q^{th} eigenvalue sorted by decreasing magnitude, with q much larger than a practical Krylov subspace dimension. These are also called *interior eigenvalues*. An alternate approach is to modify the Krylov subspace to concentrate on eigenvalues near some μ, for example using $\mathcal{K}\left((\mathbf{A} - \mu\mathbf{I})^{-1}, \mathbf{b}\right)$. However, the implementation becomes complicated because it is usually impractical to invert $\mathbf{A} - \mu\mathbf{I}$ directly. In ECSAct, we have not implemented any utilities for calculating non-extremal eigenvalues. An exception is when there is an invariant subspace of the dynamics, in which case explicit symmetry reduction can access a subset of the interior eigenvalues.

10.2 Equilibria

Linear stability of an equilibrium $\mathbf{X}_{\text{eq}}$ is defined by the linear system

$$\partial_t \mathbf{X} = \mathbf{J}\left(\mathbf{X}_{\text{eq}}\right) \cdot \mathbf{X} \tag{107}$$

where $\mathbf{J}\left(\mathbf{X}_{\text{eq}}\right)$ is the linearization of the right hand side of the equations of motion about $\mathbf{X}_{\text{eq}}$. See equation (79) for a definition in terms of a directional derivative. The spectrum of $\mathbf{J}\left(\mathbf{X}_{\text{eq}}\right)$ characterizes linear stability: an eigenvalue with positive (negative) real part indicates an unstable (stable) direction.

If $\mathbf{X}_{\text{eq}}$ is effectively one-dimensional, i.e. independent of the streamwise coordinate x, then the spectrum can be computed using the `EVP problem` and `solver` classes in Dedalus.

Ordinarily, one would treat the general case using Krylov subspace methods. However, for equilibria, the subspace iterations with respect to $\mathbf{J}$ do not converge for any realistic subspace dimension. As explained in section 8.1.3, this failure can be traced to the spectral properties of $\mathbf{J}\left(\mathbf{X}_{\text{eq}}\right)$. Preconditioning cannot solve this problem because the spectrum of the preconditioned operator may be difficult to relate to the original spectrum.

A more flexible alternative is to integrate the equations of motion over a short time interval T_0 (during which the linearization is a good approximation) and use this information to extract information on $\mathbf{J}\left(\mathbf{X}_{\text{eq}}\right)$. From (107), the time integration is equivalent to preconditioning by exponentiation:

$$\mathbf{P}^{-1}\mathbf{J} = e^{\mathbf{J}T_0} \tag{108}$$

where $\mathbf{P}^{-1}$ is the preconditioner. With this explicit representation, the original spectrum is easy to recover – namely, if μ is an eigenvalue of the preconditioned operator, then the corresponding eigenvalue λ of $\mathbf{J}(\mathbf{X}_{\text{eq}})$ is given by $\mu = e^{\lambda T_0}$.

This method essentially approximates the equilibrium as a periodic orbit with period T_0, such that μ is the Floquet multiplier and λ is the Floquet exponent. Therefore, the Dedalus implementation is essentially the same as for periodic orbits (section 10.3).

10.3 Periodic orbits

In section 7.4, we defined the linear operator $\mathcal{L}$ that maps a small perturbation $\epsilon\frac{\delta\mathbf{X}}{||\delta\mathbf{X}||}$ about $\mathbf{X}$ onto the corresponding perturbation about $\Phi(\mathbf{X}, T)$:

$$\mathcal{L}(\mathbf{X}, T) \cdot \left(\epsilon\frac{\delta\mathbf{X}}{||\delta\mathbf{X}||}\right) = \Phi\left(\mathbf{X} + \epsilon\frac{\delta\mathbf{X}}{||\delta\mathbf{X}||}, T\right) - \Phi(\mathbf{X}, T) + \mathcal{O}\left(\epsilon^2\right) \tag{109}$$

$$\rightarrow \mathcal{L}(\mathbf{X}, T) \cdot \delta\mathbf{X} = \lim_{\epsilon\to 0} \frac{\Phi\left(\mathbf{X} + \epsilon\frac{\delta\mathbf{X}}{||\delta\mathbf{X}||}, T\right) - \Phi(\mathbf{X}, T)}{\epsilon/||\delta\mathbf{X}||} \tag{110}$$

If the base state $\mathbf{X}$ is on a PO, then $\mathcal{L}$ defines the linear stability of the PO. Specifically, comparing $\delta\mathbf{X}$ with $\Phi(\mathbf{X} + \delta\mathbf{X}, T) - \Phi(\mathbf{X}, T)$ can tell us whether a generic perturbation will grow or decay upon repeated circumnavigation of the orbit. Similarly to equilibria, this information is encapsulated in the spectrum of the linear operator $\mathcal{L}$ (equation (75)). The eigenvalues μ of $\mathcal{L}$ are called Floquet multipliers. If $|\mu| > 1$ then the corresponding eigenvector represents an unstable direction, and likewise $|\mu| < 1$ represents a stable direction. The case $\mu = 1$ corresponds to a neutral direction. There is always at least one neutral eigenvalue for a periodic orbit, corresponding to a shift parallel to the orbit.

Last, the eigenvectors depend on the base state $\mathbf{X}$, but the eigenvalues do not. Nevertheless, the eigenvectors at different points on the orbit are related by a similarity transform. That is, if $\mathbf{U}(\mathbf{X}_1)$ and $\mathbf{U}(\mathbf{X}_2)$ are the matrices of the eigenvectors at $\mathbf{X}_1$ and $\mathbf{X}_2$, then $\mathbf{U}(\mathbf{X}_1) = \mathbf{S}\,\mathbf{U}(\mathbf{X}_2)\,\mathbf{S}^{-1}$ for some $\mathbf{S}$.

10.3.1 Dedalus implementation

Our goal is to approximate the spectrum of $\mathcal{L}$ in the m^{th} Krylov subspace $\mathcal{K}_m(\mathcal{L}, \mathbf{b})$. Following equation (106), we first compute the $m \times m$ projection of $\mathcal{L}$ onto $\mathcal{K}_m$:

$$\mathbf{H}_{m,m} = \mathbf{V}_m^\dagger \mathcal{L} \mathbf{V}_m \tag{111}$$

where $\mathbf{V}_m$ is $n \times m$ matrix whose columns are orthonormal basis vectors of $\mathcal{K}_m$. The spectrum of $\mathbf{H}_{m,m}$ then (presumably) approximates the leading $p < m$ eigenmodes of $\mathcal{L}$. In practice, convergence can be checked (and p estimated) by repeating the calculation with varying values of m. In ECSAct, the default value is $m = 60$.

In the Python code, the function for computing $\mathbf{H}_{m,m}$ is just `arnoldi`, introduced in section 9.2 and reproduced below:

```
def arnoldi(x_base, T, r, n:int):
        Q = np.zeros((r.size, n+1))
        H = np.zeros((n+1, n))
        Q[:,0] = r/np.linalg.norm(r)
        for k in range(1, n + 1):
                Q[:,k] = Dphi_prod(x_base, Q[:, k - 1], T)
                for j in range(0, k):
                        H[j, k-1] = np.matmul(np.conj(Q[:,j]), Q[:,k])
                        Q[:,k] = Q[:,k] - H[j, k-1]*Q[:,j]
                H[k, k-1] = np.linalg.norm(Q[:,k])
                Q[:,k] = Q[:,k]/H[k, k-1]

        return Q, H
```

Here, `r` is interpreted as the vector $\mathbf{b}$ appearing in the subspace definition $\mathcal{K}_m(\mathcal{L}, \mathbf{b})$. For solving linear systems, `r` was the residual of the linear system for some initial iteration $\mathbf{x}$. For linear stability calculations, it is not given *a priori*, and the only requirement is that it does not fall in an invariant subspace of $\mathcal{L}$. Even in the latter case, the subspace iterations may still converge due to round-off error, though more slowly than otherwise. In any case, using a vector with random components should be sufficiently robust choice for $\mathbf{b}$ / `r`. The overall computation, including output formatting, is implemented as follows:

```
phi_base = phi(f[1:], f[0]*tfac)
x0 = np.random.rand(MY)
Q, H_ = arnoldi_iteration(f[1:], f[0]*tfac, x0, 60)
H = H_[0:-1,:]

scipy.io.mmwrite('./stability/H.mtx', H)
w, vr_ = scipy.linalg.eig(H)
w_abs = np.abs(w)
vr = np.matmul(Q[:,0:-1], vr_)
scipy.io.mmwrite('./stability/w.mtx', [w])

h5f = h5py.File('vu.h5', 'w')
h5f.create_dataset('/eigenvectors', data = vr)
h5f.create_dataset('/eigenvalues', data = w)
h5f.close()

dim_w = w.size
counter = 0
for i in range(dim_w):
        if w_abs[i] > 1.01:
                counter = counter + 1

outputFile = open('./stability/N_unstable.txt', 'w')
outputFile.writelines(str(counter))
outputFile.close()
```

where `f` is a real-valued array representing a point on the PO. The number of unstable directions is approximated by counting all eigenvalues with magnitude greater than 1.01. This threshold is chosen greater than 1 because, in exact arithmetic, there is always at least one eigenvalue equal to unity, whereas in practice this eigenvalue may be represented as slightly less than or equal to 1 (due to the subspace approximation and round-off error). This is the situation for POs. For equilibria, there is no such requirement, while for RPOs there are always at least two eigenvalues equal to unity.

11 Symmetry subspaces

Symmetry subspaces play an important role in many of the ECS computations described through this documentation. The AN equations of motion are equivariant under the one-parameter group of x translations, $\tau_x(\ell)$, as well as the following x and y reflections, denoted σ_x and σ_y:

$$\sigma_x[u, v, Q_{11}, Q_{12}](x, y) = [u, -v, Q_{11}, -Q_{12}](x, h - y),$$
$$\sigma_y[u, v, Q_{11}, Q_{12}](x, y) = [-u, v, Q_{11}, -Q_{12}](L - x, y).$$

If a state initially possesses any symmetries derived from these group operations, then it will retain the same symmetries under time evolution. Some ECS and heteroclinic connections fall into such invariant subspaces, while others possess no symmetries at all. For those objects that do fall in a subspace, the effective phase space dimension can be reduced by ignoring all symmetry-violating components. Doing so speeds up computations and in some cases also improves convergence of the Newton solver.

11.1 Translations

Pure translational symmetries can be enforced simply by reducing the domain size, in this case the channel width. The idea is the following: because the dynamics is equivariant under translations in x, a k-fold ECS can be decomposed into k unit cells with identical time evolution, such that the dynamics of a single unit

cell on a domain of width w/k completely characterizes the full, k-fold ECS. For example, if an ECS has the $\tau_{L/2}$ symmetry, then it consists of two identical unit cells filling the channel, and it suffices to simulate just one of them in a channel of width $L/2$. This interpretation is especially useful because it does not require modifying existing code: to enforce a k-fold symmetry, one need only change the input parameter for the channel width.

One application is parameter continuation in the channel width, which allows one to explore families of k-fold ECS, where k corresponds to a range of integers. This works as follows. Given a k-fold symmetric ECS, one first constructs the unit cell representation and a restricted domain of width w/k. Then, if one can use parameter continuation to shrink or grow the domain and obtain corresponding ECS with width $w/(k+1)$ or $w/(k-1)$, these can be stitched back together to obtain $(k \pm 1)$-fold ECS in the full space.

11.2 Reflections

In the version of Dedalus used by ECSAct, a parity basis is available only in a periodic (Fourier) direction. Its API reference is `dedalus.core.basis.SinCos`, and it consists of the appropriate sine or cosine modes. (Note: it appears that the most recent version of Dedalus replaces this with `dedalus.core.basis.RealFourier`.) The `SinCos` basis can be used to enforce the σ_y symmetry exactly. However, ECSAct makes little use of this basis, for two reasons. First, except at low activity, there are few dynamically relevant ECS which have the σ_y symmetry. Second, the small benefit of using a parity basis seems to be outweighed by the added complexity in programming a separate basis for one particular symmetry.

Rather, symmetries involving reflections are enforced explicitly by setting symmetry violating components to 0 where appropriate. Mathematically, this is a projection onto the symmetry subspace. In a time-dependent simulation, for instance, the state vector is projected onto the symmetry subspace is every few timesteps (the optimal frequency depends on the situation).

In the code, these projections are implemented as functions `ProjectXXX(array_in, sgn)`, where `XXX` is replaced by a keyword for the symmetry. The first input parameter `array_in` is a complex-valued `numpy` array containing the coefficients of a *single field*, e.g. `QAA`. The second input parameter `sgn` is the sign of that field's parity: $+1$ for even and -1 for odd. For example, in a state satisfying the σ_x symmetry, `QAA` and `U` take `sgn = +1`, and `QAB` and `V` take `sgn = -1`.

This nomenclature applies to combined reflection and translation symmetries. For instance, the $\sigma_y\tau_{L/2}$ symmetry is prevalent in preturbulent channel flow, and its projection is given as

```
def ProjectS1T2(array_in, sgn):
        if sgn == 1:
                for i in range(NXH):
                        for j in range(NY):
                                if i % 2 == 1 and j % 2 == 0:
                                        array_in[i,j] = 0
                                elif i % 2 == 0 and j % 2 == 1:
                                        array_in[i,j] = 0
        elif sgn == -1:
                for i in range(NXH):
                        for j in range(NY):
                                if i % 2 == 0 and j % 2 == 0:
                                        array_in[i,j] = 0
                                elif i % 2 == 1 and j % 2 == 1:
                                        array_in[i,j] = 0
        return array_in
```

References

[1] P. M. Chaikin and T. C. Lubensky. *Principles of Condensed Matter Physics*. Cambridge University Press, 1995.

[2] Colin-Marius Koch and Michael Wilczek. Role of advective inertia in active nematic turbulence. *Phys. Rev. Lett.*, 127:268005, Dec 2021.

[3] Marcos Raydan. Nonlinear equations and newton's method. In Nicholas J. Higham, editor, *The Princeton Companion to Applied Mathematics*, pages 120–122. The Princeton University Press, Princeton, NJ, 2015.

[4] M. Reza Shaebani, Adam Wysocki, Roland G. Winkler, Gerhard Gompper, and Heiko Rieger. Computational models for active matter. *Nature Reviews Physics*, 2(4):181–199, March 2020.

[5] Tyler N Shendruk, Amin Doostmohammadi, Kristian Thijssen, and Julia M Yeomans. Dancing disclinations in confined active nematics. *Soft Matter*, 13(21):3853–3862, 2017.

[6] Andre M. Sonnet and Epifanio G. Virga. *Dissipative Ordered Fluids*. Springer US, 2012.

[7] Henk A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2003.

[8] Caleb G. Wagner, Michael M. Norton, Jae Sung Park, and Piyush Grover. Exact coherent structures and phase space geometry of preturbulent 2d active nematic channel flow. *Phys. Rev. Lett.*, 128:028003, Jan 2022.

Appendix A: PRL paper simulations [8]

Physical parameters

In the following, the nondimensionalization is implicit.

- height = 11
- width = 50
- $K = 0.04$ (elastic energy constant, aka Frank free energy constant in the 1-constant approximation)
- $\mathscr{A} = -0.1$ (coefficient of quadratic term in free energy; denoted as Roman `A` in Python code
- $\mathscr{C} = 0.5$ (coefficient of quartic term in free energy; denoted as Roman `C` in Python code
- $\lambda = 0$ (flow alignment)
- $\eta = 1$ (viscosity)
- $\rho = 1$ (density)
- diffusion constant $\Gamma = 0.34$ (denoted as `G` in the Dedalus code)
- $\alpha = KA^2/\text{height}^2$ (A is the dimensionless 'activity number')

Note that the activity number A is defined as $A = \sqrt{\frac{\alpha}{K}} \cdot \text{height}$. The above expression for α is obtained from this definition. In the Python code, the activity number is denoted as `Activity_num` to avoid ambiguity with $\mathscr{A}$, which itself is denoted by `A` in the code.

Dedalus parameters

1. Exploratory runs (generally acceptable in terms of accuracy)
    - Number of Fourier modes (NX) = 128
    - Number of Chebyshev modes (NY) = 32
    - State space dimension ≈ 16384
    - Timestepper = Various; sometimes first-order backward Euler (`SBDF1`), other times a first or second order Runga-Kutta (`RK111` or `RK222`). `SBDF1` is the fastest but least accurate
    - Timestep = 0.25 (the CFL threshold is typically between 1 and 10).
    - Dealias factor = 3/2
2. Primary runs
    - Number of Fourier modes (NX) = 256
    - Number of Chebyshev modes (NY) = 64
    - State space dimension ≈ 65536
    - Timestepper = 2nd-order 2-stage explicit/implicit Runga-Kutta (`RK222` in Dedalus)
    - Timestep = 0.05 (the CFL threshold is typically between 1 and 10). With this value, the period of an ECS is about 10000-20000 timesteps
    - Dealias factor = 3/2
3. Validation runs

- Number of Fourier modes (NX) = 512
- Number of Chebyshev modes (NY) = 128
- State space dimension ≈ 262144
- Timestepper = 3rd-order 4-stage Runga-Kutta (`RK443` in Dedalus). This is more accurate than `RK222`, but roughly 2x slower.
- Timestep = 0.01
- Dealias factor = 3/2

Parameter conversion

The simulations in the PRL paper effectively use the following values for the dimensionless constants:

$$\mathrm{Re_n} = 0.0136 \tag{112}$$
$$\mathrm{Er} = 0.34 \tag{113}$$
$$\mathrm{R_a} = 10\alpha \tag{114}$$

Also, the system dimensions in units of ℓ are $w \simeq 79.057$ and $h \simeq 17.393$.

Suppose X is a quantity we know in the PRL units, and we want to convert it to new units. Let $[X]_{\mathrm{PRL}}$ and $[X]_{\mathrm{new}}$ be the values in the PRL unit system and the new non-dimensionalization.

- If X is a length, then $[X]_{\mathrm{PRL}} = \sqrt{10}/5\,[X]_{\mathrm{new}} \simeq 0.63246\,[X]_{\mathrm{new}}$
- If X is a time, then $[X]_{\mathrm{PRL}} = (500/17)\,[X]_{\mathrm{new}} \simeq 29.4118\,[X]_{\mathrm{new}}$
- If X is a velocity, then $[X]_{\mathrm{PRL}} = (250/17)\sqrt{10}\,[X]_{\mathrm{new}} \simeq 46.5041\,[X]_{\mathrm{new}}$

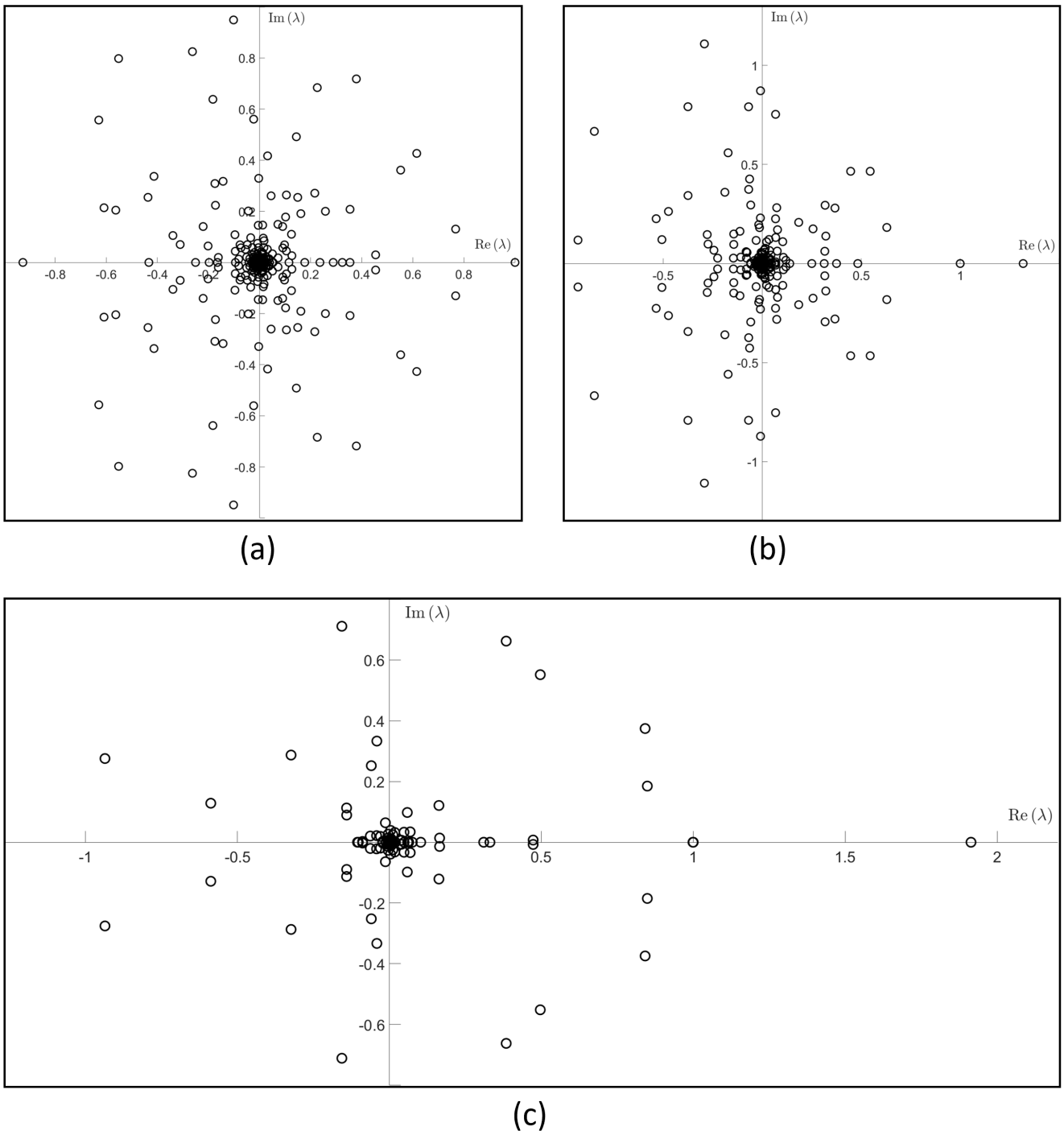

Figure 3: Eigenvalue distributions of the linear operator around a representative relative periodic orbits (RPOs). In the truncated Fourier-Chebyshev basis (128 Fourier modes and 32 Chebyshev modes), the linear operator is a 16384×16384 matrix.