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

April 15, 2022

Contents

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

2 Hydrodynamic equations 2
2.1 General equations 2
2.2 Non-dimensionalization 3
2.3 Component-wise expansions 4
2.3.1 Cartesian coordinate components 4

3 Physical parameters and phenomenology 6

4 Dedalus implementation 6
4.1 Basis, domain, fields 6
4.2 Data input 8
4.3 Problem and solver classes 8
4.4 Timestepping 9
4.4.1 Adaptive timestep via the `CFL` class 9
4.4.2 Timesteppers 10
4.5 Running the solver 10
4.6 Data output 10
4.7 Postprocessing 11
4.8 Dedalus parameters 11
4.9 Miscellaneous 12

5 Simulation details: PRL paper [6] 12
5.1 Physical parameters 12
5.2 Dedalus parameters 12
5.2.1 Parameter conversion 13

1 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.)

2 Hydrodynamic equations

2.1 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. [4], a passive elastic stress $\boldsymbol{\Pi}^{\text{elastic}}$ is included in the momentum equation. Here, we drop this term entirely.
- Note 2: The simulations in our PRL paper ([6]) 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 [3]. 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, 5]). 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

$$f = \frac{A}{2}Q^i{}_kQ^k{}_i + \frac{C}{3}Q^i{}_kQ^k{}_jQ^j{}_i + \frac{Ab}{4}(Q^i{}_kQ^k{}_i)^2 + \frac{K}{2}(\nabla^kQ^{ij})(\nabla_kQ_{ji}) \tag{10}$$

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

2.2 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 $\mathbf{Q}$-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 $\mathbf{u}$-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:

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

2.3 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:

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

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

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

2.3.1 Cartesian coordinate components

We will denote the Cartesian coordinate components of **u** and **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

$$\frac{\partial u^i}{\partial t} \quad\to\quad \frac{\partial U}{\partial t} \tag{26}$$

$$u^j\nabla_j u^i \quad\to\quad U\partial_x U + V\partial_y U \tag{27}$$

$$-\nabla^i p \quad\to\quad -\partial_x p \tag{28}$$

$$2\nabla_j E^{ji} = \partial_j(\partial^i u^j + \partial^j u^i) \quad\to\quad 2\partial_x^2 U + \partial_x\partial_y V + \partial_y^2 U \tag{29}$$

$$\nabla_j Q^{ji} \quad\to\quad \partial_x QAA + \partial_y QAB \tag{30}$$

$$\tag{31}$$

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

$$\frac{\partial u^i}{\partial t} \rightarrow \frac{\partial V}{\partial t} \tag{32}$$

$$u^j \nabla_j u^i \rightarrow U\partial_x V + V\partial_y V \tag{33}$$

$$-\nabla^i p \rightarrow -\partial_y p \tag{34}$$

$$2\nabla_j E^{ji} = \partial_j(\partial^i u^j + \partial^j u^i) \rightarrow \partial_x\partial_y V + \partial_x^2 V + 2\partial_y^2 V \tag{35}$$

$$\nabla_j Q^{ji} \rightarrow \partial_x QAB - \partial_y QAA \tag{36}$$

$$\tag{37}$$

and for the xx component of equation (24):

$$\frac{\partial Q^{ij}}{\partial t} \rightarrow \frac{\partial QAA}{\partial t} \tag{38}$$

$$u^k \nabla_k Q^{ij} \rightarrow U(\partial_x QAA) + V(\partial_y QAA) \tag{39}$$

$$W^{ik}Q_k^{\ j} = \frac{1}{2}(\nabla^i v^k - \nabla^k v^i)Q_k^{\ j} \rightarrow \frac{QAB}{2}\cdot(\partial_x V - \partial_y U) \tag{40}$$

$$Q^i_{\ k}W^{kj} = \frac{1}{2}Q^i_{\ k}(\nabla^k v^j - \nabla^j v^k) \rightarrow \frac{QAB}{2}\cdot(\partial_y U - \partial_x V) \tag{41}$$

$$W^{ik}Q_k^{\ j} - Q^i_{\ k}W^{kj} \rightarrow QAB\cdot(\partial_x V - \partial_y U) \tag{42}$$

$$\lambda E^{ij} \rightarrow \lambda\partial_x U \tag{43}$$

$$b\,Q_{k\ell}Q^{\ell k}Q^{ij} \rightarrow 2b\left[(QAA)^2 + (QAB)^2\right]QAA \tag{44}$$

$$\nabla^k\nabla_k Q^{ij} \rightarrow (\partial_x^2 + \partial_y^2)QAA \tag{45}$$

$$\tag{46}$$

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

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

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

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

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

$$W^{ik}Q_k^{\ j} - Q^i_{\ k}W^{kj} \rightarrow QAA\cdot(\partial_y U - \partial_x V) \tag{51}$$

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

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

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

The yx and yy components of **Q** do not need to be considered because they are fixed by the fact that **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 end 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.

3 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 $\langle V^2\rangle = 0 \longleftrightarrow \langle V^2\rangle > 0$
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.

4 Dedalus implementation

4.1 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)
```

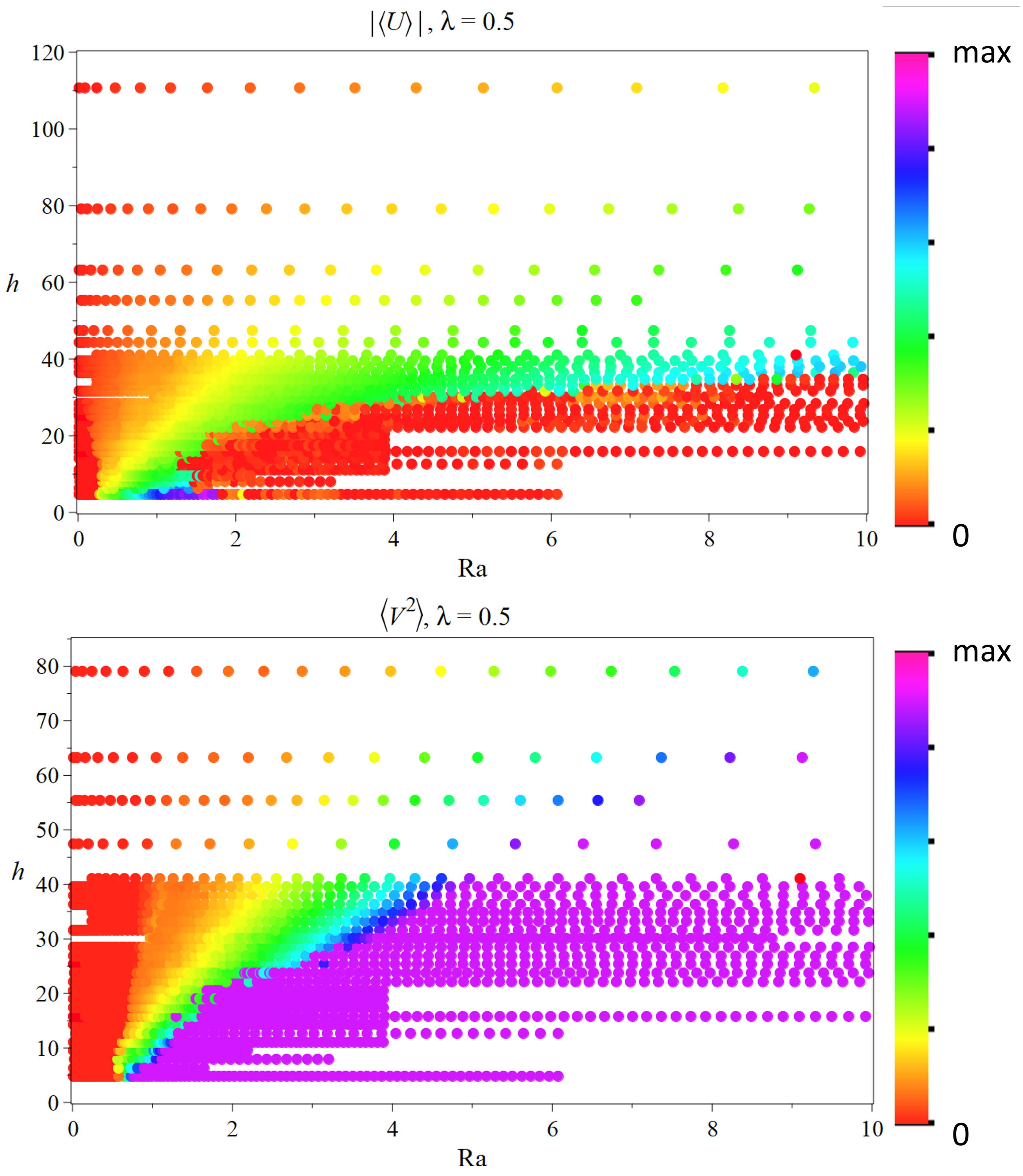

Figure 1: Parameter scan of the pre-turbulent phase diagram for $\lambda = 0.5$. Each colored circle represents a single, time-dependent simulation. The colors represent the values of spatiotemporal averages over the asymptotic portion of each trajectory ($|\langle U\rangle|$ on top and $\langle V^2\rangle >$) on bottom, and can be used to infer approximate phase boundaries separating qualitatively distinct flow regimes. See section 3 for more information.

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.

4.2 Data input

It is convenient to read in **u** and **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 4.6 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.

4.3 Problem and solver classes

The pseudospectral basis and domain can be constructed and utilized without defining a PDE problem; see section 4.1 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` accessor. 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`.

4.4 Timestepping

4.4.1 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 4.8 below and the Dedalus documentation for more information.

4.4.2 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.

4.5 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 4.4.

4.6 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.

4.7 Postprocessing

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

4.8 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.

4.9 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.

5 Simulation details: PRL paper [6]

5.1 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.

5.2 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 $\approx$ 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 $\approx$ 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

5.2.1 Parameter conversion

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

$$\mathrm{Re_n} = 0.0136 \tag{55}$$
$$\mathrm{Er} = 0.34 \tag{56}$$
$$\mathrm{R_a} = 10\alpha \tag{57}$$

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 $[X]_{\mathrm{PRL}} = (250/17)\sqrt{10}\,[X]_{\mathrm{new}} \simeq 46.5041\,[X]_{\mathrm{new}}$

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] 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.

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

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

[6] 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.